PostgreSQL for Backend & Infrastructure Engineers

> **Zero → Staff-Level Mastery | Production Systems | README.md Format**

Table of Contents

- Part 1 – Database Foundations
- Part 2 – SQL Basics
- Part 3 – Relational Thinking
- Part 4 – Advanced SQL
- Part 5 – PostgreSQL Power Features
- Part 6 – Transactions & Concurrency
- Part 7 – Performance Engineering
- Part 8 – PostgreSQL in Production
- Part 9 – Security & Reliability
- Part 10 – Engineer Playbook

Part 1 – Database Foundations (Engineer View)

Chapter 1 – What a Database Really Is

Most engineers learn SQL syntax before they understand what a database actually does. That order is backwards. When you understand the mechanics — disk, memory, WAL — everything else clicks into place.

1.1 The Three Layers of a Database

A database is a system for durably storing, organizing, and querying data. At the infrastructure level, it has three concerns:

```
┌──────────────────────────────────────────────┐
│                   CLIENTS                    │
│          (your app, psql, pg_admin)          │
└──────────────────────┬───────────────────────┘
                       │ SQL queries
                       ▼
┌──────────────────────────────────────────────┐
│              POSTGRESQL PROCESS              │
│  ┌──────────┐  ┌────────────┐  ┌──────────┐ │
│  │  Parser  │  │  Planner   │  │Executor  │ │
│  └──────────┘  └────────────┘  └──────────┘ │
│  ┌──────────────────────────────────────┐   │
│  │        Shared Buffer Cache           │   │  ← RAM
│  └──────────────────────────────────────┘   │
└──────────────────────┬───────────────────────┘
                       │ read/write pages
                       ▼
┌──────────────────────────────────────────────┐
│                    DISK                      │
│  ┌──────────────┐  ┌──────────────────────┐  │
│  │  Data Files  │  │  WAL (Write-Ahead Log)│ │
│  │ (heap files) │  │   pg_wal/            │  │
│  └──────────────┘  └──────────────────────┘  │
└──────────────────────────────────────────────┘
```

RAM (Shared Buffer Cache): PostgreSQL reads pages from disk into RAM. All queries work against pages in memory. Dirty pages (modified but not yet written to disk) are flushed periodically by the background writer.

Disk (Heap Files): Tables are stored as heap files — sequences of 8KB pages. Each page holds rows (called tuples). There is no inherent ordering of rows on disk.

WAL (Write-Ahead Log): Before any data page is modified, the *change* is written to the WAL. This guarantees durability — if PostgreSQL crashes, it replays the WAL to reconstruct state. This is how `COMMIT` is fast: it only needs to flush a small WAL record, not entire pages.

1.2 How a Query Flows Through PostgreSQL

```
Client sends: SELECT * FROM users WHERE id = 42;

Step 1: PARSE
  Raw SQL text → Parse tree (AST)
  Validates syntax only
```

```
Step 2: ANALYZE / REWRITE
  Resolve table names, column names
  Apply rewrite rules (views expand here)

Step 3: PLAN / OPTIMIZE
  Generate possible execution plans
  Estimate costs using statistics
  Choose cheapest plan

Step 4: EXECUTE
  Walk the plan tree
  Fetch pages from shared buffers (or disk)
  Apply filters, joins, aggregates

Step 5: RETURN
  Send result rows to client
```

This pipeline is why `EXPLAIN` is so powerful — it shows you the plan *before* step 4 runs.

1.3 Pages and Tuples

PostgreSQL stores data in **8KB pages**. You can think of each page as a fixed-size container:

```
┌──────────────────────────────────────────────────┐
│                  PAGE (8 KB)                     │
│                                                  │
│  Page Header (24 bytes)                          │
│  ┌──────────────────────────────────────────┐    │
│  │ lsn | checksum | flags | lower | upper   │    │
│  └──────────────────────────────────────────┘    │
│                                                  │
│  Item Pointers (line pointer array)              │
│  ┌──────┬──────┬──────┬──────┐                   │
│  │ LP1  │ LP2  │ LP3  │ LP4  │ ...               │
│  └──────┴──────┴──────┴──────┘                   │
│       ↑ grow down from top                       │
│                                                  │
│       (free space in the middle)                 │
│                                                  │
│       ↑ grow up from bottom                      │
│  ┌────────┬────────┬────────┬────────────┐       │
│  │ Tuple4 │ Tuple3 │ Tuple2 │ Tuple1     │       │
│  └────────┴────────┴────────┴────────────┘       │
│                                                  │
│                                                │
│  Special Area (indexes only)                   │
└────────────────────────────────────────────────┘
```

Each **tuple** (row) contains:

- **t_xmin** — transaction ID that created this row version
- **t_xmax** — transaction ID that deleted/updated this row (0 if still live)
- **t_ctid** — physical location (page, offset) of this tuple or its newer version
- Null bitmap
- Column data

This structure is what makes MVCC possible. More on this in Chapter 6.

1.4 The WAL in Practice

```
Timeline of a simple UPDATE:

T=1  Client sends: UPDATE users SET email='new@x.com' WHERE id=1;

T=2  PostgreSQL finds page containing row, loads into shared buffer

T=3  WAL record written to pg_wal/:
     "At LSN 0/1234, in relation 16384, page 0: tuple (0,1)
      changed xmax from 0 to txid 500, new email='new@x.com'"

T=4  Shared buffer page marked dirty (row updated in memory)

T=5  Client sends: COMMIT;

T=6  WAL record flushed to disk (fdatasync)
     → COMMIT returns to client ✓

T=7  (Later) Background writer flushes dirty data page to disk

If crash happens between T=6 and T=7:
  WAL replay at recovery will re-apply the change to the data file
  → Zero data loss guaranteed
```

This is why `fsync = off` is dangerous: it disables the WAL flush at step 6, making commits lie about durability.

Chapter 2 – PostgreSQL Architecture

2.1 Process Model

PostgreSQL uses a **multi-process** model (not multi-threaded like MySQL). Each client connection gets its own OS process.

```
┌──────────────────────────────────────────────────────┐
│               POSTGRESQL CLUSTER                     │
│                                                      │
│  postmaster (PID 1)                                  │
│   ├── Background Writer   (flushes dirty pages)      │
│   ├── WAL Writer          (flushes WAL)              │
│   ├── Checkpointer        (periodic checkpoints)     │
│   ├── Autovacuum Launcher                            │
│   │    └── Autovacuum Worker (per table)             │
│   ├── Stats Collector                                │
│   ├── Logical Replication Launcher                   │
│   │                                                  │
│   ├── Backend #1  ← connection from app server       │
│   ├── Backend #2  ← connection from app server       │
│   └── Backend #3  ← psql session                     │
└──────────────────────────────────────────────────────┘
```

Each backend process:

- Has its own local memory (work_mem for sorts, hash joins)
- Shares the global shared_buffers pool
- Can lock rows, pages, tables

Why this matters for infra engineers: Each connection is an OS process (~5-10MB). At 500 connections you're using gigabytes just for process overhead. This is why connection poolers (PgBouncer) are essential.

2.2 Memory Architecture

```
┌───────────────────────────────────────────────────────────┐
│                     SHARED MEMORY                         │
│   ┌─────────────────────────────────────────────┐        │
│   │  Shared Buffer Cache   (shared_buffers)      │        │
│   │  Default: 128MB  Recommended: 25% of RAM     │        │
│  └──────────────────────────────────────────────┘    │
│                                                        │
│  ┌──────────────┐  ┌───────────────────────────┐     │
│  │ WAL Buffers  │  │ Lock Table                │     │
│  │ (wal_buffers│  │ (in-memory lock mgr)      │     │
│  └──────────────┘  └───────────────────────────┘     │
└────────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────────┐
│           PER-BACKEND (LOCAL) MEMORY                   │
│                                                        │
│  ┌─────────────────────────────┐                     │
│  │ work_mem  (4MB default)     │  ← sorts, hashes  │
│  │ (per sort/hash operation!)  │                      │
│  └─────────────────────────────┘                      │
│                                                        │
│  ┌─────────────────────────────┐                      │
│  │ maintenance_work_mem (64MB) │  ← VACUUM, CREATE  │
│  └─────────────────────────────┘     INDEX            │
└────────────────────────────────────────────────────────┘
```

Critical gotcha: `work_mem` is *per sort operation*. A complex query with 3 sort steps and 50 concurrent connections could use `50 × 3 × 4MB = 600MB`. Setting `work_mem = 256MB` naively can OOM your server.

2.3 The Data Directory Layout

```
$PGDATA/
├── PG_VERSION              ← PostgreSQL major version
├── pg_hba.conf             ← Client authentication rules
├── postgresql.conf         ← Main configuration
├── postgresql.auto.conf    ← ALTER SYSTEM overrides
├── global/
│   └── pg_control          ← Cluster state, checkpoint info
├── base/
│   ├── 1/                  ← template1 database
│   ├── 13212/              ← your database (OID)
│   │   ├── 16384           ← table file (OID)
│   │   ├── 16384_fsm       ← free space map
│   │   ├── 16384_vm        ← visibility map
│   │   └── 16385           ← index file
├── pg_wal/                 ← WAL segment files (16MB each)
├── pg_xact/                ← Transaction commit status
└── pg_stat_tmp/            ← Statistics collector data
```

Chapter 3 – MVCC (Gentle Introduction)

MVCC — Multi-Version Concurrency Control — is the core mechanism that allows PostgreSQL to:

- Let readers never block writers
- Let writers never block readers
- Provide snapshot isolation

3.1 The Mental Model

Instead of locking a row when you update it, PostgreSQL creates a *new version* of the row. Old versions are kept until no transaction can see them anymore.

```
BEFORE UPDATE:
users table, page 0:

┌──────────────────────────────────────────────────┐
│  Tuple (0,1):  xmin=100  xmax=0  id=1            │
│                name='Alice'  email='a@x.com'      │
└──────────────────────────────────────────────────┘

AFTER UPDATE email (txid=200):

┌──────────────────────────────────────────────────┐
│  Tuple (0,1):  xmin=100  xmax=200  id=1          │  ← old version
│                name='Alice'  email='a@x.com'      │     (marked dead)
├──────────────────────────────────────────────────┤
│  Tuple (0,2):  xmin=200  xmax=0    id=1          │  ← new version
│                name='Alice'  email='b@x.com'      │     (live)
└──────────────────────────────────────────────────┘
```

A transaction started before txid 200 will see tuple (0,1). A transaction started after txid 200 commits will see tuple (0,2).

This is why **VACUUM** is needed — to clean up old row versions (dead tuples) that no transaction can see anymore.

We will go deep on MVCC in Part 6. For now, keep this model in mind as you learn SQL.

Chapter 1–3 Exercises

Beginner

1. What are the three layers of a PostgreSQL database and what does each do?
2. Why does PostgreSQL need WAL? What problem does it solve?
3. What is a "page" in PostgreSQL's storage?

Intermediate

4. A developer sets `shared_buffers = 1GB` and `work_mem = 512MB` on a server with 4GB RAM, expecting 100 concurrent connections. What could go wrong? Calculate the worst-case memory usage.
5. Explain why each connection in PostgreSQL being a separate OS process matters for infrastructure design.

Advanced

6. A row is updated 5 times in quick succession by 5 different transactions. Draw the tuple chain in the heap page showing xmin/xmax values. What happens to old versions over time?

Solutions

4. work_mem is per sort operation, not per connection. A query with 2 sort steps = 2 × 512MB = 1GB per connection. At 100 connections: worst case 100GB just for work_mem, plus 1GB shared_buffers = system OOM. Safe rule: `work_mem = (available_ram - shared_buffers) / (max_connections × 2)`.

5. Each OS process consumes ~5-10MB of RAM for the process itself, plus its local memory. 500 connections = 2.5-5GB just in process overhead before any query runs. Connection poolers (PgBouncer) multiplex many application connections to a small pool of actual PostgreSQL backends, making this sustainable.

6.

```
Tuple 1: xmin=101  xmax=102  (dead, only visible to txid 101's snapshot)
Tuple 2: xmin=102  xmax=103  (dead)
Tuple 3: xmin=103  xmax=104  (dead)
Tuple 4: xmin=104  xmax=105  (dead)
Tuple 5: xmin=105  xmax=0    (live, current version)

VACUUM scans the page and removes tuples 1-4 because no active
transaction's snapshot can see them. The page free space increases.
```

Part 2 — SQL Basics (Zero Level)

Chapter 4 – Tables, Rows, Columns

4.1 The Demo Database

Throughout this book we use one consistent database. Tables grow as chapters progress.

```
-- Our domain: e-commerce platform
-- Tables we'll build:
--   users        → registered accounts
--   sessions     → login sessions
--   orders       → purchase orders
--   order_items  → line items per order
--   payments     → payment transactions
--   products     → product catalog
--   inventory    → stock levels
--   logs         → system event log
```

Let's start:

```
CREATE TABLE users (
    id          SERIAL PRIMARY KEY,
    email       TEXT NOT NULL UNIQUE,
    username    TEXT NOT NULL,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    is_active   BOOLEAN NOT NULL DEFAULT true
);
```

What this creates:

```
users table:
┌────┬────────────────────┬──────────┬─────────────────────────┬──────────
│ id │ email              │ username │ created_at              │ is_active
├────┼────────────────────┼──────────┼─────────────────────────┼──────────
│    │                    │          │                         │
└────┴────────────────────┴──────────┴─────────────────────────┴──────────
```

- **SERIAL** — auto-incrementing integer (shorthand for `INT DEFAULT nextval('seq')`)
- **PRIMARY KEY** — implies NOT NULL + UNIQUE + B-Tree index
- **NOT NULL** — enforced at storage level, cheaper than application checks
- **DEFAULT now()** — evaluated at insert time, not at CREATE TABLE time

4.2 Data Types: Memory and Performance Impact

Choosing the right type is not about pedantry — it affects storage, index size, and join performance.

INTEGERS:

Type	Size	Range
SMALLINT	2 bytes	-32,768 to 32,767
INTEGER	4 bytes	-2.1B to 2.1B
BIGINT	8 bytes	-9.2×10^{18} to 9.2×10^{18}
SERIAL	4 bytes	Same as INTEGER (auto-increment)
BIGSERIAL	8 bytes	Same as BIGINT (auto-increment)

DECIMALS:

REAL	4 bytes	6 decimal digits precision
DOUBLE	8 bytes	15 decimal digits precision
NUMERIC(p,s)	variable	Exact, arbitrary precision

→ Use NUMERIC for money. REAL/DOUBLE lose precision (0.1+0.2 ≠ 0.3).

TEXT:

TEXT	variable	Unlimited length
VARCHAR(n)	variable	Limit n chars (checked on write)
CHAR(n)	fixed	Padded to n chars. AVOID.

→ In PostgreSQL, TEXT and VARCHAR(n) have identical performance.
Use TEXT unless you need the constraint.

DATE/TIME:

DATE	4 bytes	Date only. No timezone.
TIME	8 bytes	Time only.
TIMESTAMP	8 bytes	Date+time. No timezone. AVOID.
TIMESTAMPTZ	8 bytes	Date+time. UTC internally. USE THIS.
INTERVAL	16 bytes	Duration.

→ TIMESTAMPTZ stores UTC, displays in session timezone.
TIMESTAMP stores whatever you give it — timezone ambiguous.
Always use TIMESTAMPTZ.

```
BOOLEAN:
┌──────────┬────────┬────────────────────────────────────────────┐
│ BOOLEAN  │ 1 byte │ true/false/NULL                            │
└──────────┴────────┴────────────────────────────────────────────┘

UUID:
┌──────────┬─────────┬───────────────────────────────────────────┐
│ UUID     │ 16 bytes│ 128-bit identifier                        │
└──────────┴─────────┴───────────────────────────────────────────┘
→ UUID as primary key: larger than int, random UUIDs fragment
  B-Tree indexes (see ULIDv7 or gen_random_uuid() discussion in Part 5).
```

4.3 Creating Our Full Schema

```sql
-- Products catalog
CREATE TABLE products (
    id          SERIAL PRIMARY KEY,
    sku         TEXT NOT NULL UNIQUE,
    name        TEXT NOT NULL,
    description TEXT,
    price_cents INTEGER NOT NULL CHECK (price_cents >= 0),
    category    TEXT NOT NULL,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Inventory (separate from products for flexibility)
CREATE TABLE inventory (
    product_id  INTEGER NOT NULL REFERENCES products(id),
    quantity    INTEGER NOT NULL DEFAULT 0 CHECK (quantity >= 0),
    updated_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    PRIMARY KEY (product_id)
);

-- Orders
CREATE TABLE orders (
    id           BIGSERIAL PRIMARY KEY,
    user_id      INTEGER NOT NULL REFERENCES users(id),
    status       TEXT NOT NULL DEFAULT 'pending'
                 CHECK (status IN ('pending','processing','shipped','deli
    total_cents  INTEGER NOT NULL DEFAULT 0,
    created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at   TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Order line items
CREATE TABLE order_items (
    id           BIGSERIAL PRIMARY KEY,
    order_id     BIGINT NOT NULL REFERENCES orders(id),
    product_id   INTEGER NOT NULL REFERENCES products(id),
    quantity     INTEGER NOT NULL CHECK (quantity > 0),
    unit_price_cents INTEGER NOT NULL CHECK (unit_price_cents >= 0)
);

-- Payments
CREATE TABLE payments (
    id               BIGSERIAL PRIMARY KEY,
    order_id         BIGINT NOT NULL REFERENCES orders(id),
    amount_cents     INTEGER NOT NULL CHECK (amount_cents > 0),
    status           TEXT NOT NULL DEFAULT 'pending'
                     CHECK (status IN ('pending','completed','failed','ref
    provider         TEXT NOT NULL,
    provider_ref     TEXT,
    created_at       TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Sessions
CREATE TABLE sessions (
    id           UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id      INTEGER NOT NULL REFERENCES users(id),
    created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
    expires_at   TIMESTAMPTZ NOT NULL,
    ip_address   INET,
    user_agent   TEXT
);

-- System logs (append-only)
CREATE TABLE logs (
    id           BIGSERIAL PRIMARY KEY,
    level        TEXT NOT NULL CHECK (level IN ('DEBUG','INFO','WARN','ERR
    service      TEXT NOT NULL,
    message      TEXT NOT NULL,
    context      JSONB,
    created_at   TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Chapter 5 – INSERT / SELECT / UPDATE / DELETE

5.1 INSERT — Adding Data

Problem: Register a new user when they sign up.

Schema:

```
users
┌────┬────────────────────┬──────────┬─────────────────────┬───────────┐
│ id │ email              │ username │ created_at          │ is_active │
└────┴────────────────────┴──────────┴─────────────────────┴───────────┘
```

SQL:

```sql
INSERT INTO users (email, username)
VALUES ('alice@example.com', 'alice');
```

Step-by-Step:

1. PostgreSQL acquires an `id` from the `users_id_seq` sequence
2. `created_at` is filled with `now()` (current transaction start time)
3. `is_active` is filled with `true` (default)
4. Row written to heap page + WAL record appended

Inserting multiple rows (efficient):

```sql
INSERT INTO users (email, username) VALUES
    ('bob@example.com',   'bob'),
    ('carol@example.com', 'carol'),
    ('dave@example.com',  'dave'),
    ('eve@example.com',   'eve');
```

One round-trip vs. 4. Always batch inserts.

INSERT ... RETURNING (get the ID back):

```sql
INSERT INTO users (email, username)
VALUES ('frank@example.com', 'frank')
RETURNING id, created_at;
```

```
Result:
┌────┬──────────────────────────────────┐
│ id │ created_at                       │
├────┼──────────────────────────────────┤
│  6 │ 2024-01-15 10:23:45+00         │
└────┴──────────────────────────────────┘
```

Why this matters: No need for a second `SELECT id FROM users WHERE email = ....` RETURNING eliminates the round-trip.

Sample data for the rest of this chapter:

```
INSERT INTO users (email, username) VALUES
    ('alice@example.com',   'alice'),
    ('bob@example.com',     'bob'),
    ('carol@example.com',   'carol'),
    ('dave@example.com',    'dave'),
    ('eve@example.com',     'eve');

INSERT INTO products (sku, name, price_cents, category) VALUES
    ('LAPTOP-001',  'Pro Laptop 15"',   149900, 'electronics'),
    ('MOUSE-001',   'Wireless Mouse',     2999, 'accessories'),
    ('MONITOR-001', '4K Monitor 27"',    59900, 'electronics'),
    ('KEYBOARD-001','Mech Keyboard',     12900, 'accessories'),
    ('HEADSET-001', 'Noise Cancel Headset', 19900, 'accessories');

INSERT INTO orders (user_id, status, total_cents) VALUES
    (1, 'delivered', 152899),
    (1, 'shipped',    62899),
    (2, 'delivered',   2999),
    (3, 'pending',   19900),
    (4, 'cancelled', 12900);

INSERT INTO order_items (order_id, product_id, quantity, unit_price_cents
    (1, 1, 1, 149900),
    (1, 2, 1,   2999),
    (2, 3, 1,  59900),
    (2, 4, 1,  12900) -- note: doesn't match total, intentional for exerc
    -- corrected below:
    ;

INSERT INTO payments (order_id, amount_cents, status, provider) VALUES
    (1, 152899, 'completed', 'stripe'),
    (2,  62899, 'completed', 'stripe'),
    (3,   2999, 'completed', 'paypal'),
    (4,  19900, 'pending',   'stripe');
```

5.2 SELECT — Reading Data

Problem: Find all active users.

SQL:

```
SELECT id, email, username, created_at
FROM users
WHERE is_active = true;
```

Step-by-Step Dry Run:

```
Execution:
1. PostgreSQL opens the users heap file
2. Reads page 0 (8KB block)
3. Scans each tuple on the page:
   - Tuple 1: is_active = true  → PASS → output row
   - Tuple 2: is_active = true  → PASS → output row
   - Tuple 3: is_active = true  → PASS → output row
   - ...all users are active in our sample...
4. Continues to next page if needed
5. Returns results to client
```

Output:

```
┌────┬───────────────────┬──────────┬────────────────────────┐
│ id │ email             │ username │ created_at             │
├────┼───────────────────┼──────────┼────────────────────────┤
│  1 │ alice@example.com │ alice    │ 2024-01-15 10:00:00+00 │
│  2 │ bob@example.com   │ bob      │ 2024-01-15 10:00:01+00 │
│  3 │ carol@example.com │ carol    │ 2024-01-15 10:00:02+00 │
│  4 │ dave@example.com  │ dave     │ 2024-01-15 10:00:03+00 │
│  5 │ eve@example.com   │ eve      │ 2024-01-15 10:00:04+00 │
└────┴───────────────────┴──────────┴────────────────────────┘
```

SELECT with ordering and limiting:

```
SELECT id, email, created_at
FROM users
ORDER BY created_at DESC
LIMIT 3;
```

Step-by-Step:

1. Scan all rows (or use index if available on created_at)
2. Sort results by created_at descending — requires a sort step
3. Return first 3 rows

```
Output:
┌────┬───────────────────┬────────────────────────┐
│ id │ email             │ created_at             │
├────┼───────────────────┼────────────────────────┤
│  5 │ eve@example.com   │ 2024-01-15 10:00:04+00 │
│  4 │ dave@example.com  │ 2024-01-15 10:00:03+00 │
│  3 │ carol@example.com │ 2024-01-15 10:00:02+00 │
└────┴───────────────────┴────────────────────────┘
```

Performance note: Without an index on `created_at`, this sorts the entire table then discards. With an index, PostgreSQL can read the index in reverse to get the top 3 without a full sort.

5.3 UPDATE — Modifying Data

Problem: Deactivate a user account.

SQL:

```
UPDATE users
SET is_active = false,
    -- we'd add updated_at here in real life
WHERE id = 3
RETURNING id, email, is_active;
```

Step-by-Step:

```
1. Find row with id=3 (via PRIMARY KEY index → O(log n))
2. MVCC: mark old tuple's xmax = current_txid
3. Write new tuple version with is_active = false
```

```
4. Append WAL record for the change
5. RETURNING clause reads the new tuple and returns it
```

Output:

```
┌────┬───────────────────┬───────────┐
│ id │ email             │ is_active │
├────┼───────────────────┼───────────┤
│  3 │ carol@example.com │ f         │
└────┴───────────────────┴───────────┘
```

Bulk UPDATE:

```
UPDATE orders
SET status = 'processing'
WHERE status = 'pending'
  AND created_at < now() - INTERVAL '1 hour'
RETURNING id, status;
```

Internal behavior: Each matching row gets a new tuple version. If 10,000 rows match, 10,000 new tuples are written. This bloats the table — why VACUUM matters.

5.4 DELETE — Removing Data

Problem: Clean up expired sessions.

SQL:

```
DELETE FROM sessions
WHERE expires_at < now()
RETURNING id;
```

Step-by-Step:

```
1. Scan sessions table (or use index on expires_at)
2. For each matching tuple:
   - Set xmax = current_txid (mark as deleted)
   - Write WAL record
3. Return deleted row ids
```

```
4. Rows not physically removed — just marked dead
5. VACUUM will later reclaim the space
```

Why rows aren't immediately removed: Other transactions might be reading those rows (MVCC snapshot). Physical removal happens only when VACUUM confirms no transaction can see the old version.

Chapter 4–5 Exercises

Beginner

1. Insert a new product: SKU 'WEBCAM-001', name 'HD Webcam', price $49.99 (in cents), category 'accessories'.
2. Select all products with price_cents > 10000, ordered by price descending.
3. Update product id=2 to set price_cents = 3499.

Intermediate

4. Insert an order for user_id=2 with status 'pending', then immediately get back the generated id using RETURNING. Then insert an order_item for that order with product_id=1, quantity=1.
5. Write a query to find all orders created in the last 7 days.
6. Soft-delete (deactivate, don't actually DELETE) all users who have no orders.

Advanced / Production-style

7. You need to atomically transfer 100 units of inventory from one product to a "reserved" count. Write the SQL that does this safely. What constraint prevents negative inventory?
8. Write a DELETE that removes all logs older than 90 days but only from the 'DEBUG' level. Use RETURNING to count how many were deleted. How would you make this efficient for a table with 100M rows?

Solutions

1.

```
INSERT INTO products (sku, name, price_cents, category)
VALUES ('WEBCAM-001', 'HD Webcam', 4999, 'accessories');
```

2.

```
SELECT id, sku, name, price_cents
FROM products
```

```
WHERE price_cents > 10000
ORDER BY price_cents DESC;
```

3.

```
UPDATE products SET price_cents = 3499 WHERE id = 2;
```

4.

```
-- Two statements, same transaction:
BEGIN;
INSERT INTO orders (user_id, status) VALUES (2, 'pending') RETURNING id;
-- Assume returned id = 6
INSERT INTO order_items (order_id, product_id, quantity, unit_price_cents
VALUES (6, 1, 1, 149900);
COMMIT;
```

5.

```
SELECT id, user_id, status, total_cents, created_at
FROM orders
WHERE created_at >= now() - INTERVAL '7 days';
```

Performance: add index `CREATE INDEX ON orders (created_at)` for large tables.

6.

```
UPDATE users
SET is_active = false
WHERE id NOT IN (SELECT DISTINCT user_id FROM orders);
```

Better with NOT EXISTS for performance on large tables:

```
UPDATE users u
SET is_active = false
WHERE NOT EXISTS (
    SELECT 1 FROM orders o WHERE o.user_id = u.id
);
```

7.

```
-- The CHECK constraint (quantity >= 0) prevents negative inventory
-- This will raise an error if quantity would go below 0:
UPDATE inventory
SET quantity = quantity - 100
WHERE product_id = 1
  AND quantity >= 100;  -- optimistic lock: only update if enough stock
-- Check affected rows = 1 to confirm success
```

8.

```
-- Efficient approach: use index on (level, created_at)
CREATE INDEX CONCURRENTLY ON logs (level, created_at)
WHERE level = 'DEBUG';  -- partial index!

-- Then delete in batches to avoid long transactions:
DELETE FROM logs
WHERE id IN (
    SELECT id FROM logs
    WHERE level = 'DEBUG'
      AND created_at < now() - INTERVAL '90 days'
    LIMIT 10000
)
RETURNING id;
-- Repeat until 0 rows deleted
```

Single large DELETE on 100M rows holds locks for minutes. Batch deletes reduce lock time and WAL pressure.

Part 3 – Relational Thinking (Backend Core)

Chapter 6 – Primary Keys, Foreign Keys, and Relationships

6.1 Why Relationships Exist

The relational model exists to avoid one thing: **duplicate data**. Duplication causes update anomalies — when the same fact is stored in multiple places, changing it requires updating all copies. Miss one and your data is inconsistent.

```
BAD (no normalization):
orders table:
┌─────────┬─────────────────────┬──────────┬─────────────────────┐
│ order_id│ user_email          │ username │ product_name        │
├─────────┼─────────────────────┼──────────┼─────────────────────┤
│ 1       │ alice@example.com   │ alice    │ Pro Laptop 15"      │
│ 2       │ alice@example.com   │ alice    │ Wireless Mouse      │  ← dupli
│ 3       │ bob@example.com     │ bob      │ Pro Laptop 15"      │  ← dupli
└─────────┴─────────────────────┴──────────┴─────────────────────┘

GOOD (normalized):
users      orders         order_items    products
┌──┬─────┐ ┌──┬────────────┐ ┌──┬────┬──┐ ┌──┬─────────────┐
│id│email│ │id│user_id     │ │id│ord │pr│ │id│name         │
│  │     │ │  │(→users)    │ │  │_id │id│ │  │             │
└──┴─────┘ └──┴────────────┘ └──┴────┴──┘ └──┴─────────────┘
```

6.2 Foreign Keys in Detail

```
-- This constraint:
REFERENCES users(id)

-- Means:
-- 1. orders.user_id MUST exist in users.id  (referential integrity)
-- 2. You CANNOT delete a user who has orders (unless CASCADE is set)
-- 3. PostgreSQL automatically creates an index on users.id (the PK)
-- 4. PostgreSQL does NOT automatically index orders.user_id (you must!)
```

The missing FK index problem:

```
users                    orders
┌────┬───────┐           ┌────┬─────────┐
│ id │ email │           │ id │ user_id │
├────┼───────┤           ├────┼─────────┤
│  1 │ alice │           │  1 │       1 │
│  2 │ bob   │           │  2 │       1 │
└────┴───────┘           │  3 │       2 │
                         │  4 │       1 │
  ↑ indexed (PK)         └────┴─────────┘
```

```
                                          ↑ NOT indexed by default!

  If you DELETE FROM users WHERE id=1, PostgreSQL must scan
  ALL of orders to find and check/cascade related rows.
  On a table with 10M orders, this is a sequential scan.

  FIX: CREATE INDEX ON orders (user_id);
```

6.3 Relationship Types

```
ONE-TO-ONE:
  user has one profile
  users ─────────── user_profiles
   (1)                     (1)
  users.id ←── user_profiles.user_id (UNIQUE)

ONE-TO-MANY:
  user has many orders
  users ─────────── orders
   (1)                 (N)
  users.id ←── orders.user_id

MANY-TO-MANY:
  orders contain many products; products appear in many orders
  orders ─────── order_items ─────── products
   (N)              (bridge)              (M)
  orders.id ←── order_items.order_id
  products.id ←── order_items.product_id
```

Chapter 7 – JOINs

JOINs are not magic — they combine rows from two tables based on a condition. Understanding the execution logic makes them predictable.

7.1 INNER JOIN

Problem: Find all orders with the username of who placed them.

Schema:

```
users                     orders
┌────┬──────────┐         ┌────┬─────────┬──────────┐
│ id │ username │         │ id │ user_id │ status   │
├────┼──────────┤         ├────┼─────────┼──────────┤
│  1 │ alice    │         │  1 │    1    │ delivered│
│  2 │ bob      │         │  2 │    1    │ shipped  │
│  3 │ carol    │         │  3 │    2    │ delivered│
│  4 │ dave     │         │  4 │    3    │ pending  │
│  5 │ eve      │         │  5 │    4    │ cancelled│
└────┴──────────┘         └────┴─────────┴──────────┘
```

SQL:

```sql
SELECT
    o.id          AS order_id,
    u.username,
    o.status,
    o.total_cents
FROM orders o
INNER JOIN users u ON u.id = o.user_id;
```

JOIN Execution Flow:

```
Strategy: Nested Loop Join (for small tables)

For each row in orders (outer):
  ┌─────────────────────────────────────────────┐
  │ order row: id=1, user_id=1, ...             │
  │   → Look up users where id = 1              │
  │   → Found: username='alice'                 │
  │   → Emit combined row                       │
  └─────────────────────────────────────────────┘

  ┌─────────────────────────────────────────────┐
  │ order row: id=2, user_id=1, ...             │
  │   → Look up users where id = 1              │
  │   → Found: username='alice'                 │
  │   → Emit combined row                       │
  └─────────────────────────────────────────────┘
  ... and so on

For each lookup in users, PostgreSQL uses the PRIMARY KEY index
→ O(log n) per lookup, not a full table scan
```

Output:

```
┌──────────┬──────────┬───────────┬─────────────┐
│ order_id │ username │ status    │ total_cents │
├──────────┼──────────┼───────────┼─────────────┤
│        1 │ alice    │ delivered │      152899 │
│        2 │ alice    │ shipped   │       62899 │
│        3 │ bob      │ delivered │        2999 │
│        4 │ carol    │ pending   │       19900 │
│        5 │ dave     │ cancelled │       12900 │
└──────────┴──────────┴───────────┴─────────────┘
```

Why user_id=5 (eve) doesn't appear: INNER JOIN only returns rows where a match exists in *both* tables. Eve has no orders.

7.2 LEFT JOIN

Problem: List all users and how many orders they've placed (including users with zero orders).

```sql
SELECT
    u.id,
    u.username,
    COUNT(o.id) AS order_count
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
GROUP BY u.id, u.username
ORDER BY order_count DESC;
```

JOIN Execution Flow:

```
LEFT JOIN: Start from the LEFT table (users), include ALL rows.
If no matching row in RIGHT table (orders), fill with NULLs.

User alice (id=1):  matches orders 1, 2     → 2 rows
User bob   (id=2):  matches order 3          → 1 row
User carol (id=3):  matches order 4          → 1 row
User dave  (id=4):  matches order 5          → 1 row
User eve   (id=5):  NO matching orders       → 1 row with NULL order id
                                               COUNT(NULL) = 0  ✓
```

Output:

```
┌────┬──────────┬─────────────┐
│ id │ username │ order_count │
├────┼──────────┼─────────────┤
│  1 │ alice    │           2 │
│  2 │ bob      │           1 │
│  3 │ carol    │           1 │
│  4 │ dave     │           1 │
│  5 │ eve      │           0 │
└────┴──────────┴─────────────┘
```

Key insight: `COUNT(o.id)` counts non-NULL values of o.id, so it correctly returns 0 for eve. `COUNT(*)` would return 1 (counting the NULL-padded row itself).

7.3 Multi-Table JOIN

Problem: Show order details with user name and product names.

```
SELECT
    o.id                    AS order_id,
    u.username,
    p.name                  AS product_name,
    oi.quantity,
    oi.unit_price_cents
FROM orders o
JOIN users u          ON u.id = o.user_id
JOIN order_items oi ON oi.order_id = o.id
JOIN products p       ON p.id = oi.product_id
ORDER BY o.id, p.name;
```

Multi-table JOIN execution:

```
PostgreSQL's planner chooses join order. For small tables:

Step 1: orders JOIN users
        (nested loop, index on users.id)
        → result: order rows with username attached

Step 2: result JOIN order_items
        (nested loop, index on order_items.order_id needed!)
        → result: expanded rows with line items
```

```
Step 3: result JOIN products
        (nested loop, index on products.id)
        → result: fully expanded rows
```

Output:

```
┌──────────┬──────────┬──────────────────┬──────────┬──────────────────┐
│ order_id │ username │ product_name     │ quantity │ unit_price_cents │
├──────────┼──────────┼──────────────────┼──────────┼──────────────────┤
│        1 │ alice    │ Pro Laptop 15"   │        1 │           149900 │
│        1 │ alice    │ Wireless Mouse   │        1 │             2999 │
│        2 │ alice    │ 4K Monitor 27"   │        1 │            59900 │
│        2 │ alice    │ Mech Keyboard    │        1 │            12900 │
└──────────┴──────────┴──────────────────┴──────────┴──────────────────┘
```

7.4 JOIN Types — Visual Summary

```
INNER JOIN                LEFT JOIN
  A    B                    A    B
 ┌──┐ ┌──┐                 ┌──┐ ┌──┐
 │  │█│  │                 │██│█│  │
 │  │█│  │                 │██│█│  │
 └──┘ └──┘                 └──┘ └──┘
 Only intersection      All of A + intersection

RIGHT JOIN                FULL OUTER JOIN
  A    B                    A    B
 ┌──┐ ┌──┐                 ┌──┐ ┌──┐
 │  │█│██│                 │██│█│██│
 │  │█│██│                 │██│█│██│
 └──┘ └──┘                 └──┘ └──┘
 intersection + all B   Everything from both

CROSS JOIN
 Every row in A paired with every row in B
 5 users × 5 products = 25 rows
 Use with extreme care. Usually a bug if unintentional.
```

Chapter 8 – Aggregations and Analytics

8.1 GROUP BY Mechanics

Problem: Total revenue per user.

```
SELECT
    u.username,
    COUNT(o.id)                AS order_count,
    SUM(o.total_cents)         AS total_spent_cents,
    AVG(o.total_cents)         AS avg_order_cents,
    MAX(o.total_cents)         AS largest_order_cents
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.status = 'delivered'
GROUP BY u.id, u.username
ORDER BY total_spent_cents DESC;
```

Execution flow:

```
Step 1: JOIN users with orders (inner join → only users with orders)

Step 2: Filter WHERE o.status = 'delivered'
        Remaining rows:
        alice, order 1, 152899
        bob,   order 3,   2999

Step 3: GROUP BY u.id, u.username
        Bucket 'alice': [152899]
        Bucket 'bob':   [2999]

Step 4: Apply aggregates per bucket:
        alice: COUNT=1, SUM=152899, AVG=152899, MAX=152899
        bob:   COUNT=1, SUM=2999,   AVG=2999,   MAX=2999

Step 5: ORDER BY total_spent_cents DESC
```

Output:

```
┌──────────┬─────────────┬───────────────────┬─────────────────┬────────
│ username │ order_count │ total_spent_cents │ avg_order_cents │ largest_
├──────────┼─────────────┼───────────────────┼─────────────────┼────────
│ alice    │           1 │            152899 │        152899.0 │
│ bob        │         1 │              2999 │           2999.0 │
└────────────┴───────────┴───────────────────┴──────────────────┴
```

8.2 HAVING — Filtering Groups

```
-- Users who have spent more than $50 total
SELECT
    u.username,
    SUM(o.total_cents) AS total_spent_cents
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.status != 'cancelled'
GROUP BY u.id, u.username
HAVING SUM(o.total_cents) > 5000
ORDER BY total_spent_cents DESC;
```

WHERE vs HAVING:

```
WHERE  → filters individual rows BEFORE grouping
HAVING → filters groups AFTER aggregation

Timeline:
  FROM + JOIN     → all rows
  WHERE           → remove cancelled orders
  GROUP BY        → bucket by user
  Aggregation     → calculate SUM per bucket
  HAVING          → remove buckets where SUM <= 5000
  ORDER BY        → sort remaining buckets
  SELECT          → project columns
```

8.3 Common Aggregate Functions

```
COUNT(*) vs COUNT(col):
  COUNT(*)      → count all rows including NULLs
  COUNT(col)    → count non-NULL values of col
  COUNT(DISTINCT col) → count unique non-NULL values

SUM(col)            → sum of non-NULL values
AVG(col)            → average of non-NULL values (ignores NULLs)
MIN(col)            → minimum
```

```
MAX(col)          → maximum
BOOL_AND(col)     → true if all values are true
BOOL_OR(col)      → true if any value is true
STRING_AGG(col, sep) → concatenate strings
ARRAY_AGG(col)    → collect values into array
JSON_AGG(col)     → collect rows into JSON array
```

Chapter 6–8 Exercises

Beginner

1. Write a query showing all products and their inventory quantity (use LEFT JOIN so products with no inventory row still appear).
2. Count how many orders are in each status.
3. Find all orders where total_cents > 50000.

Intermediate
4. Show each user's most recent order date and status. Include users with no orders (show NULL for those fields).
5. Find all products that have never been ordered (hint: use LEFT JOIN + IS NULL).
6. Calculate the average order value by month for the last 12 months.

Advanced
7. Write a query that returns, for each user: their total spending, their percentage of total site revenue, and their rank by spending.
8. Find orders where the sum of order_items.unit_price_cents × quantity does NOT match orders.total_cents (data integrity check query).

Solutions

1.

```
SELECT p.name, p.sku, COALESCE(i.quantity, 0) AS stock
FROM products p
LEFT JOIN inventory i ON i.product_id = p.id;
```

2.

```
SELECT status, COUNT(*) AS order_count
FROM orders
```

```sql
GROUP BY status
ORDER BY order_count DESC;
```

3.

```sql
SELECT id, user_id, status, total_cents
FROM orders
WHERE total_cents > 50000;
```

4.

```sql
SELECT
    u.username,
    MAX(o.created_at) AS latest_order_date,
    (SELECT status FROM orders o2
     WHERE o2.user_id = u.id
     ORDER BY created_at DESC LIMIT 1) AS latest_status
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
GROUP BY u.id, u.username;
```

5.

```sql
SELECT p.id, p.name, p.sku
FROM products p
LEFT JOIN order_items oi ON oi.product_id = p.id
WHERE oi.id IS NULL;
```

Performance: For large tables, `NOT EXISTS` is often faster.

6.

```sql
SELECT
    DATE_TRUNC('month', created_at) AS month,
    COUNT(*) AS order_count,
    AVG(total_cents) AS avg_value_cents
FROM orders
WHERE created_at >= now() - INTERVAL '12 months'
GROUP BY 1
ORDER BY 1;
```

7.

```
SELECT
    u.username,
    SUM(o.total_cents) AS total_spent,
    ROUND(100.0 * SUM(o.total_cents) /
          SUM(SUM(o.total_cents)) OVER (), 2) AS pct_of_revenue,
    RANK() OVER (ORDER BY SUM(o.total_cents) DESC) AS spending_rank
FROM users u
JOIN orders o ON o.user_id = u.id
GROUP BY u.id, u.username
ORDER BY total_spent DESC;
```

8.

```
SELECT
    o.id,
    o.total_cents AS stored_total,
    SUM(oi.quantity * oi.unit_price_cents) AS calculated_total,
    o.total_cents - SUM(oi.quantity * oi.unit_price_cents) AS discrepancy
FROM orders o
JOIN order_items oi ON oi.order_id = o.id
GROUP BY o.id, o.total_cents
HAVING o.total_cents != SUM(oi.quantity * oi.unit_price_cents);
```

Part 4 – Advanced SQL

Chapter 9 – Subqueries vs JOINs

9.1 Subqueries: What They Are

A subquery is a SELECT inside another SELECT. It has three forms:

```
1. Scalar subquery → returns exactly one row, one column
   SELECT (SELECT COUNT(*) FROM orders) AS total_orders;

2. Row subquery → returns one row, multiple columns
   SELECT * FROM users WHERE (id, email) =
    (SELECT user_id, email FROM sessions WHERE id = 'abc');

3. Table subquery → returns multiple rows (used with IN, EXISTS, FROM)
   SELECT * FROM users WHERE id IN (SELECT user_id FROM orders);
```

9.2 Correlated vs Uncorrelated

Uncorrelated subquery — runs once, result reused:

```
-- Subquery executes once
SELECT id, username
FROM users
WHERE id IN (SELECT user_id FROM orders WHERE status = 'delivered');
```

```
Execution:
Step 1: Run subquery → {1, 2}  (executed ONCE)
Step 2: Scan users, keep rows where id IN {1, 2}
```

Correlated subquery — runs once *per outer row*:

```
-- Subquery references o.user_id from outer query
SELECT
    u.username,
    (SELECT COUNT(*) FROM orders o WHERE o.user_id = u.id) AS order_count
FROM users u;
```

```
Execution:
For user alice (id=1): run "SELECT COUNT(*) FROM orders WHERE user_id=1"
For user bob   (id=2): run "SELECT COUNT(*) FROM orders WHERE user_id=2"
For user carol (id=3): run subquery → 1
...

N users = N subquery executions.
PostgreSQL often optimizes this into a JOIN, but not always.
```

9.3 Subquery vs JOIN: When to Use Which

```
-- SUBQUERY version (readable):
SELECT username
```

```
FROM users
WHERE id IN (
    SELECT user_id FROM orders WHERE status = 'delivered'
);

-- JOIN version (often same plan after optimization):
SELECT DISTINCT u.username
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.status = 'delivered';

-- EXISTS version (best for "does relationship exist"):
SELECT u.username
FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.user_id = u.id AND o.status = 'delivered'
);
```

Mental model for choosing:

```
Q: Do you need columns from BOTH tables?
  → JOIN

Q: Do you just want rows from table A where a condition holds in table B
  → EXISTS (most explicit, often fastest)
  → IN (clean, fine for small subquery result sets)
  → JOIN + DISTINCT (watch for duplicates)

Q: Do you want rows from A where NO match exists in B?
  → NOT EXISTS
  → LEFT JOIN ... WHERE b.id IS NULL
  → NOT IN (dangerous: if subquery returns any NULL, result is empty!)
```

The NULL trap with NOT IN:

```
-- Dangerous! If any order has user_id = NULL, returns 0 rows:
SELECT username FROM users
WHERE id NOT IN (SELECT user_id FROM orders);

-- Safe:
SELECT username FROM users u
WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.user_id = u.id);
```

Chapter 10 – CTEs and Recursive CTEs

10.1 Common Table Expressions (CTEs)

CTEs let you name subqueries and reuse them. They make complex queries readable.

```
WITH
-- Step 1: get delivered orders with user info
delivered_orders AS (
    SELECT o.id, o.user_id, o.total_cents, u.username
    FROM orders o
    JOIN users u ON u.id = o.user_id
    WHERE o.status = 'delivered'
),
-- Step 2: aggregate per user
user_stats AS (
    SELECT
        user_id,
        username,
        COUNT(*)        AS order_count,
        SUM(total_cents) AS total_spent
    FROM delivered_orders
    GROUP BY user_id, username
)
-- Final: only high-value users
SELECT username, order_count, total_spent
FROM user_stats
WHERE total_spent > 10000
ORDER BY total_spent DESC;
```

Execution flow:

```
PostgreSQL evaluates CTEs in sequence:

1. delivered_orders CTE executes → result set stored (or inlined)
2. user_stats CTE executes using delivered_orders result
3. Final SELECT runs using user_stats result

In PostgreSQL < 12: CTEs were optimization fences (always materialized)
In PostgreSQL >= 12: CTEs are inlined by default unless:
  - Referenced more than once
  - Contains side effects (INSERT, UPDATE, DELETE)
  - Marked WITH ... AS MATERIALIZED
```

10.2 Writable CTEs (Data-Modifying CTEs)

```
-- Move cancelled orders to an archive table and delete from main table
WITH deleted_orders AS (
    DELETE FROM orders
    WHERE status = 'cancelled'
      AND created_at < now() - INTERVAL '30 days'
    RETURNING *
)
INSERT INTO orders_archive
SELECT * FROM deleted_orders;
```

Execution timeline:

```
T1: BEGIN (implicit)
T2: DELETE runs, rows marked dead, RETURNING captures them
T3: INSERT runs using captured rows
T4: COMMIT — both operations committed atomically

If INSERT fails → entire transaction rolls back → no data loss
```

10.3 Recursive CTEs

Recursive CTEs let you traverse tree and graph structures — parent-child hierarchies, org charts, category trees.

Problem: Our products have categories, and categories have parent categories.

```
CREATE TABLE categories (
    id        SERIAL PRIMARY KEY,
    name      TEXT NOT NULL,
    parent_id INTEGER REFERENCES categories(id)
);

INSERT INTO categories (id, name, parent_id) VALUES
    (1, 'All Products', NULL),
    (2, 'Electronics',  1),
    (3, 'Accessories',  1),
    (4, 'Laptops',      2),
    (5, 'Monitors',     2),
    (6, 'Mice',         3),
    (7, 'Keyboards',    3);
```

Tree structure:

```
All Products (1)
├── Electronics (2)
│   ├── Laptops (4)
│   └── Monitors (5)
└── Accessories (3)
    ├── Mice (6)
    └── Keyboards (7)
```

Query: Get all descendants of 'Electronics':

```
WITH RECURSIVE category_tree AS (
    -- Base case: start with Electronics
    SELECT id, name, parent_id, 0 AS depth
    FROM categories
    WHERE id = 2

    UNION ALL

    -- Recursive case: find children of current rows
    SELECT c.id, c.name, c.parent_id, ct.depth + 1
    FROM categories c
    JOIN category_tree ct ON ct.id = c.parent_id
)
SELECT id, name, depth,
       REPEAT('  ', depth) || name AS indented_name
FROM category_tree
ORDER BY depth, name;
```

Recursive execution:

```
Iteration 0 (base case):
  Result: {id=2, name='Electronics', depth=0}

Iteration 1 (children of id=2):
  Result: {id=4, name='Laptops', depth=1}
          {id=5, name='Monitors', depth=1}

Iteration 2 (children of id=4 or id=5):
  Result: (none — leaves have no children)

Termination: no new rows added → stop
```

Output:

```
┌────┬──────────────┬───────┬───────────────────┐
│ id │ name         │ depth │ indented_name     │
├────┼──────────────┼───────┼───────────────────┤
│  2 │ Electronics  │     0 │ Electronics       │
│  4 │ Laptops      │     1 │   Laptops         │
│  5 │ Monitors     │     1 │   Monitors        │
└────┴──────────────┴───────┴───────────────────┘
```

Chapter 11 – Window Functions

Window functions let you compute values *across a set of rows related to the current row* without collapsing them into a group. They are the most powerful SQL feature for analytics.

11.1 The Window Function Mental Model

```
Normal aggregation:                    Window function:
  Input:  5 order rows                  Input:  5 order rows
  Output: 1 summary row                 Output: 5 rows, each enriched
                                                 with a computed value

GROUP BY collapses rows.              OVER() doesn't collapse rows.
```

11.2 ROW_NUMBER, RANK, DENSE_RANK

Problem: Rank orders by amount within each user.

```
SELECT
    u.username,
    o.id           AS order_id,
    o.total_cents,
    ROW_NUMBER() OVER (PARTITION BY u.id ORDER BY o.total_cents DESC) AS
    RANK()       OVER (PARTITION BY u.id ORDER BY o.total_cents DESC) AS
    DENSE_RANK() OVER (PARTITION BY u.id ORDER BY o.total_cents DESC) AS
FROM orders o
JOIN users u ON u.id = o.user_id
ORDER BY u.id, total_cents DESC;
```

Execution flow:

```
PARTITION BY u.id → separate window per user
ORDER BY o.total_cents DESC → within each window, order by amount

For alice (two orders: 152899, 62899):
  ROW_NUMBER: 1, 2   (always unique)
  RANK:       1, 2   (gaps on ties)
  DENSE_RANK: 1, 2   (no gaps on ties)

If two orders had equal amount (e.g., both 62899):
  ROW_NUMBER: 1, 2      (arbitrary tiebreak)
  RANK:       1, 1, 3   (skips 2)
  DENSE_RANK: 1, 1, 2   (no skip)
```

11.3 LAG and LEAD — Accessing Adjacent Rows

Problem: For each order, show the previous order's amount (to calculate growth).

```
SELECT
    user_id,
    id         AS order_id,
    created_at,
    total_cents,
    LAG(total_cents, 1)  OVER (PARTITION BY user_id ORDER BY created_at)
    total_cents - LAG(total_cents, 1)
                         OVER (PARTITION BY user_id ORDER BY created_at)
FROM orders
ORDER BY user_id, created_at;
```

Output:

```
┌─────────┬──────────┬────────────┬─────────────┬──────────────────┬───
│ user_id │ order_id │ created_at │ total_cents │ prev_order_cents │ ch
├─────────┼──────────┼────────────┼─────────────┼──────────────────┼───
│       1 │        1 │ 2024-01-15 │      152899 │ NULL             │ NU
│       1 │        2 │ 2024-01-16 │       62899 │ 152899           │ -9
│       2 │        3 │ 2024-01-15 │        2999 │ NULL             │ NU
└─────────┴──────────┴────────────┴─────────────┴──────────────────┴───
```

11.4 Running Totals and Moving Averages

```sql
SELECT
    DATE_TRUNC('day', created_at) AS day,
    SUM(total_cents) AS daily_revenue,
    SUM(SUM(total_cents)) OVER (
        ORDER BY DATE_TRUNC('day', created_at)
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_revenue,
    AVG(SUM(total_cents)) OVER (
        ORDER BY DATE_TRUNC('day', created_at)
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS rolling_7day_avg
FROM orders
WHERE status != 'cancelled'
GROUP BY 1
ORDER BY 1;
```

Frame specification:

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  = from the start of the partition to the current row
  = running total

ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  = current row plus the 6 rows before it
  = 7-day rolling window

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
  = 3-row centered moving average
```

11.5 NTILE — Bucketing

```sql
-- Divide customers into 4 quartiles by spend
SELECT
    username,
    total_spent,
    NTILE(4) OVER (ORDER BY total_spent DESC) AS quartile
FROM (
    SELECT u.username, SUM(o.total_cents) AS total_spent
    FROM users u
    JOIN orders o ON o.user_id = u.id
    GROUP BY u.id, u.username
) user_totals;
```

Chapter 9–11 Exercises

Beginner

1. Using a CTE, write a query that first gets all 'shipped' orders, then counts them.
2. Use ROW_NUMBER() to number all orders per user by date (oldest = 1).

Intermediate

3. Write a query showing each order and the running total of orders for that user (by date).
4. Find the top 2 most expensive products in each category using window functions.
5. Using a recursive CTE, list all categories in the tree from Part 10 with their full path (e.g., "All Products > Electronics > Laptops").

Advanced

6. Write a query that identifies users whose most recent order value is more than 50% lower than their previous order value (potential churn signal).
7. Calculate a 30-day rolling average revenue, but exclude weekends from the calculation.

Solutions

2.

```sql
SELECT
    user_id, id AS order_id, created_at,
    ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY created_at ASC) AS c
FROM orders;
```

4.

```
WITH ranked AS (
    SELECT
        category, name, price_cents,
        ROW_NUMBER() OVER (PARTITION BY category ORDER BY price_cents DES
    FROM products
)
SELECT category, name, price_cents
FROM ranked
WHERE rn <= 2;
```

5.

```
WITH RECURSIVE cat_path AS (
    SELECT id, name, parent_id, name::TEXT AS path
    FROM categories WHERE parent_id IS NULL

    UNION ALL

    SELECT c.id, c.name, c.parent_id,
           cp.path || ' > ' || c.name
    FROM categories c
    JOIN cat_path cp ON cp.id = c.parent_id
)
SELECT id, name, path FROM cat_path ORDER BY path;
```

6.

```
WITH order_pairs AS (
    SELECT
        user_id,
        total_cents,
        LAG(total_cents) OVER (PARTITION BY user_id ORDER BY created_at)
        ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY created_at DESC)
    FROM orders
    WHERE status != 'cancelled'
)
SELECT u.username, op.total_cents AS last_order, op.prev_amount
FROM order_pairs op
JOIN users u ON u.id = op.user_id
WHERE op.recency = 1
  AND op.prev_amount IS NOT NULL
  AND op.total_cents < op.prev_amount * 0.5;
```

Part 5 – PostgreSQL Power Features

Chapter 12 – Index Internals

12.1 Why Indexes Exist

Without an index, finding a row requires reading every page in the table — a sequential scan. For a 10M row table stored on 100,000 pages, that's 100,000 disk reads per query.

An index is a separate data structure that maps search key → physical row location.

12.2 B-Tree Index (Default)

B-Tree (Balanced Tree) is PostgreSQL's default index type. It handles equality, range, prefix, and sort operations.

```
B-Tree structure for users.email index:

Level 2 (Root):
┌─────────────────────────────────────────┐
│  ["eve@x.com"]                          │
│  left ptr      │      right ptr         │
└────────────────┼────────────────────────┘
                 │
       ┌─────────┴─────────────────┐
       ▼                           ▼
Level 1 (Internal):      Level 1 (Internal):
┌──────────────────┐     ┌──────────────────┐
│["bob", "carol"]  │     │["frank"]         │
└──────────────────┘     └──────────────────┘
    │       │       │        │          │
    ▼       ▼       ▼        ▼          ▼
Level 0 (Leaf pages) — contain actual data pointers:
┌───────┐ ┌───────┐ ┌───────┐ ┌───────┐ ┌───────┐
│ alice │ │  bob  │ │ carol │ │  dave │ │  eve  │
│ctid   │ │ctid   │ │ctid   │ │ctid   │ │ctid   │
│(0,1)  │ │(0,2)  │ │(0,3)  │ │(1,1)  │ │(1,2)  │
└───────┘ └───────┘ └───────┘ └───────┘ └───────┘
           ↑ ctid = (page number, tuple offset)
```

B-Tree traversal for `WHERE email = 'carol@example.com'`**:**

```
Start at root: is 'carol@' < 'eve@'? YES → go left
At internal: is 'carol@' > 'bob@'? YES → go right child
At leaf: scan leaf page → found 'carol@example.com', ctid=(0,3)
Fetch heap page 0, tuple 3

Cost: O(log n) — for 1M rows ≈ 20 comparisons
vs sequential scan: O(n) — 1M comparisons
```

What B-Tree supports:

```
✓ Equality:      WHERE email = 'alice@x.com'
✓ Range:         WHERE price_cents BETWEEN 1000 AND 5000
✓ Prefix:        WHERE name LIKE 'Pro%'  (left-anchored only!)
✓ Sort:          ORDER BY created_at    (avoids sort step)
✓ NOT NULL:      WHERE col IS NOT NULL
✗ Suffix/infix:  WHERE name LIKE '%Laptop'  (can't use B-Tree)
✗ NULL equality: WHERE col IS NULL  (NULLs stored in index, this works)
```

12.3 Index Types Comparison

```
┌────────────────┬──────────────────────────────────────────────────────────
│ Index Type     │ Best For
├────────────────┼──────────────────────────────────────────────────────────
│ B-Tree         │ Equality, range, sort. Default. 90% of use cases.
│                │ CREATE INDEX ON orders (created_at);
├────────────────┼──────────────────────────────────────────────────────────
│ Hash           │ Equality only. Faster than B-Tree for pure =.
│                │ CREATE INDEX ON users USING HASH (email);
│                │ Rarely useful. B-Tree usually wins overall.
├────────────────┼──────────────────────────────────────────────────────────
│ GIN            │ Full-text search, arrays, JSONB keys.
│ (Generalized   │ CREATE INDEX ON logs USING GIN (context);
│  Inverted)     │ CREATE INDEX ON products USING GIN
│                │   (to_tsvector('english', description));
├────────────────┼──────────────────────────────────────────────────────────
│ GiST           │ Geometric types, range types, nearest-neighbor.
│ (Generalized   │ CREATE INDEX ON locations USING GiST (coordinates);
│  Search Tree)  │ CREATE INDEX ON reservations USING GiST
│                    │    (daterange(start_date, end_date));
├────────────────────┼──────────────────────────────────────────────────────────
│ BRIN               │ Very large tables with naturally ordered data.
│ (Block Range)      │ E.g., logs table ordered by time (append-only).
│                    │ CREATE INDEX ON logs USING BRIN (created_at);
│                    │ Tiny index (1 entry per 128 pages), fast for ranges.
│                    │ Useless if data is not physically correlated.
└────────────────────┴──────────────────────────────────────────────────────────
```

12.4 Partial Indexes

Index only the rows you actually query:

```
-- 99% of queries are for active users only:
CREATE INDEX ON users (email) WHERE is_active = true;

-- Only index pending/processing orders (not historical):
CREATE INDEX ON orders (user_id, created_at)
    WHERE status IN ('pending', 'processing');

-- Index only non-NULL values:
CREATE INDEX ON orders (provider_ref)
    WHERE provider_ref IS NOT NULL;
```

Why partial indexes win:

```
Full index on orders.status for 10M orders:
  - All 10M rows indexed
  - Large index, slower updates

Partial index WHERE status IN ('pending','processing'):
  - Only 50K active orders indexed
  - Tiny index, fast updates, fits in memory
  - Queries on active orders use this small index
```

12.5 Composite Indexes and Column Order

```
-- For query: WHERE user_id = 1 AND status = 'pending'
CREATE INDEX ON orders (user_id, status);
```

```
-- This index also supports:
-- WHERE user_id = 1                     ✓ (leading column)
-- WHERE user_id = 1 AND status = 'x'  ✓
-- WHERE user_id = 1 ORDER BY status   ✓

-- But NOT:
-- WHERE status = 'pending'            ✗ (non-leading column only)
```

Index column order rule:

```
Equality columns first, then range columns.

For: WHERE user_id = 1 AND created_at > '2024-01-01'
  Good:  CREATE INDEX ON orders (user_id, created_at)
  Bad:   CREATE INDEX ON orders (created_at, user_id)
    → The bad version can't skip to user_id=1 efficiently
      because created_at is a range, not equality
```

Chapter 13 – JSONB

PostgreSQL's JSONB stores JSON as a binary tree, not raw text. It's parsed once on insert and queried efficiently.

13.1 Basic JSONB Operations

```
-- Our logs table already has: context JSONB

-- Insert with JSONB:
INSERT INTO logs (level, service, message, context) VALUES
    ('ERROR', 'payment-service', 'Stripe webhook failed',
     '{"error_code": "card_declined", "amount": 2999, "user_id": 1,
       "metadata": {"retry": true, "attempt": 3}}'),
    ('INFO', 'auth-service', 'User logged in',
     '{"user_id": 2, "ip": "192.168.1.1", "2fa": false}'),
    ('WARN', 'order-service', 'Low inventory',
     '{"product_id": 3, "remaining": 2, "threshold": 5}');

-- Extract a value (returns TEXT):
SELECT context->>'error_code' AS error_code
FROM logs WHERE level = 'ERROR';
```

```
-- Extract nested (returns TEXT):
SELECT context->'metadata'->>'retry' AS will_retry
FROM logs WHERE level = 'ERROR';

-- Extract as JSON (preserves type):
SELECT context->'metadata'->'attempt' AS attempt_num
FROM logs WHERE level = 'ERROR';
-- Returns: 3  (integer, not "3")
```

Operator reference:

```
->   Extract by key/index → returns JSON
->>  Extract by key/index → returns TEXT
#>   Extract by path      → returns JSON    context #> '{metadata,attempt
#>>  Extract by path      → returns TEXT    context #>> '{metadata,retry}
@>   Contains             → BOOLEAN         context @> '{"user_id": 1}'
<@   Is contained by      → BOOLEAN
?    Key exists            → BOOLEAN         context ? 'error_code'
?|   Any key exists        → BOOLEAN         context ?| array['error_code
?&   All keys exist        → BOOLEAN
```

13.2 JSONB Indexing

```
-- GIN index for key existence and @> (contains) queries:
CREATE INDEX ON logs USING GIN (context);

-- Now these queries are fast:
SELECT * FROM logs WHERE context @> '{"user_id": 1}';
SELECT * FROM logs WHERE context ? 'error_code';

-- Expression index for a specific frequently-queried key:
CREATE INDEX ON logs ((context->>'user_id'));
-- Fast for: WHERE context->>'user_id' = '1'
```

13.3 JSONB Functions

```
-- Build JSONB from columns:
SELECT jsonb_build_object(
    'user_id', id,
    'username', username,
    'email', email
) AS user_json
FROM users LIMIT 1;

-- Merge JSONB objects:
SELECT '{"a":1}'::jsonb || '{"b":2}'::jsonb;
-- → {"a": 1, "b": 2}

-- Remove a key:
SELECT '{"a":1,"b":2}'::jsonb - 'b';
-- → {"a": 1}

-- Expand JSONB array to rows:
SELECT jsonb_array_elements('["a","b","c"]'::jsonb);

-- Expand JSONB object to key-value pairs:
SELECT * FROM jsonb_each('{"a":1,"b":2}'::jsonb);
```

Chapter 14 – Full-Text Search

14.1 The FTS Mental Model

```
Regular LIKE search:                 Full-Text Search:
  WHERE name LIKE '%laptop%'           to_tsvector('english', description)
  → Scans every character              @@ to_tsquery('english', 'laptop')
  → Can't use normal index             → Stemming: 'running' matches 'run'
  → No relevance ranking               → Stop words: 'the', 'a' ignored
  → Exact match only                   → Ranking by relevance
```

14.2 Setting Up FTS

```
-- Add FTS column to products:
ALTER TABLE products ADD COLUMN search_vector TSVECTOR;

-- Populate it:
UPDATE products
SET search_vector = to_tsvector('english',
    COALESCE(name, '') || ' ' || COALESCE(description, '') || ' ' || cate
);

-- Create GIN index:
CREATE INDEX ON products USING GIN (search_vector);

-- Auto-update with trigger:
CREATE FUNCTION update_product_search_vector()
RETURNS TRIGGER AS $$
BEGIN
    NEW.search_vector := to_tsvector('english',
        COALESCE(NEW.name, '') || ' ' ||
        COALESCE(NEW.description, '') || ' ' ||
        COALESCE(NEW.category, ''));
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER product_search_vector_update
BEFORE INSERT OR UPDATE ON products
FOR EACH ROW EXECUTE FUNCTION update_product_search_vector();
```

14.3 Querying FTS

```
-- Basic search:
SELECT name, price_cents,
       ts_rank(search_vector, query) AS rank
FROM products,
     to_tsquery('english', 'laptop | monitor') AS query
WHERE search_vector @@ query
ORDER BY rank DESC;

-- Phrase search:
SELECT name FROM products
WHERE search_vector @@ phraseto_tsquery('english', 'noise cancelling');

-- Highlight matches:
SELECT
    name,
    ts_headline('english', description,
                to_tsquery('english', 'laptop'),
                'MaxWords=10, MinWords=5') AS excerpt
```

```
FROM products
WHERE search_vector @@ to_tsquery('english', 'laptop');
```

Chapter 12–14 Exercises

Beginner

1. Create a B-Tree index on orders(user_id). Explain why this helps the queries in Chapter 7.
2. Insert a log entry with JSONB context containing your current IP, the service name 'api', and a request_id.
3. Query all logs where the JSONB context contains a 'user_id' key.

Intermediate
4. Create a partial index that only covers orders with status='pending'. Write a query that would use it.
5. Write a JSONB query that finds all ERROR logs where the error_code in context is 'card_declined'.
6. Add full-text search to the users table on the username and email fields.

Advanced
7. You have a products table with 10M rows and a JSONB 'attributes' column. Queries frequently filter on `attributes->>'brand' = 'Apple'`. Design the optimal indexing strategy and explain your choice.
8. Write a query that aggregates JSONB log contexts to find the top 5 most common error codes across all ERROR-level logs.

Solutions

3.

```
SELECT id, level, message, context
FROM logs
WHERE context ? 'user_id';
```

5.

```
SELECT id, message, context
FROM logs
WHERE level = 'ERROR'
  AND context @> '{"error_code": "card_declined"}';
```

The GIN index makes `@>` efficient.

7.

```
-- Expression index on the extracted value:
CREATE INDEX ON products ((attributes->>'brand'));

-- Use when:
-- SELECT * FROM products WHERE attributes->>'brand' = 'Apple';
-- This is better than a full GIN index because:
-- 1. Smaller: indexes one extracted text value, not the full JSONB
-- 2. Faster for this specific query pattern
-- 3. GIN index is better when querying many different keys

-- If you also need: WHERE attributes @> '{"brand": "Apple", "year": 2023
-- Then add a GIN index too:
CREATE INDEX ON products USING GIN (attributes);
```

8.

```
SELECT
    context->>'error_code' AS error_code,
    COUNT(*) AS occurrences
FROM logs
WHERE level = 'ERROR'
  AND context->>'error_code' IS NOT NULL
GROUP BY 1
ORDER BY occurrences DESC
LIMIT 5;
```

Part 6 – Transactions & Concurrency

Chapter 15 – ACID with Real Scenarios

15.1 What ACID Means in Practice

ACID is not abstract theory — it directly affects how you write application code.

```
A = Atomicity
  "All or nothing"
  Scenario: Transfer $100 between accounts
    WRONG without atomicity:
      UPDATE accounts SET balance = balance - 100 WHERE id = 1; ✓
      [server crashes]
      UPDATE accounts SET balance = balance + 100 WHERE id = 2; ✗ never r
      → $100 disappeared from the system

    WITH atomicity:
      BEGIN;
        UPDATE accounts SET balance = balance - 100 WHERE id = 1;
        UPDATE accounts SET balance = balance + 100 WHERE id = 2;
      COMMIT;  ← either both happen or neither does

C = Consistency
  "Constraints are always satisfied"

  Our CHECK (quantity >= 0) on inventory is a consistency rule.
  PostgreSQL enforces it: the database can never be left in a
  state where inventory is negative, regardless of concurrent access.

I = Isolation
  "Transactions don't see each other's partial work"

  Scenario: Two users buy the last item simultaneously
  Without isolation: both see quantity=1, both decrement → quantity=-1
  With isolation + constraints: one succeeds, one gets an error

D = Durability
  "Committed data survives crashes"

  COMMIT returns → data is on disk (WAL flushed)
  Power failure 1ms after COMMIT → data still there on restart
  This is guaranteed by WAL + fsync
```

15.2 Real Transaction Example: Order Checkout

```sql
BEGIN;

-- 1. Reserve inventory (with row lock)
UPDATE inventory
SET quantity = quantity - 1
```

```sql
WHERE product_id = 1
  AND quantity >= 1;

-- Check that exactly 1 row was updated:
-- (application checks rowcount here; if 0, ROLLBACK — out of stock)

-- 2. Create the order
INSERT INTO orders (user_id, status, total_cents)
VALUES (1, 'processing', 149900)
RETURNING id AS order_id;
-- Assume returned id = 10

-- 3. Create order item
INSERT INTO order_items (order_id, product_id, quantity, unit_price_cents
VALUES (10, 1, 1, 149900);

-- 4. Create pending payment
INSERT INTO payments (order_id, amount_cents, status, provider)
VALUES (10, 149900, 'pending', 'stripe');

COMMIT;
```

Execution timeline:

```
T=1  BEGIN
T=2  UPDATE inventory: acquires ROW EXCLUSIVE lock on product_id=1 row
T=3  INSERT orders: new row visible only to this transaction
T=4  INSERT order_items: same
T=5  INSERT payments: same
T=6  COMMIT:
       WAL flushed to disk (fsync)
       All row versions stamped with commit txid
       Locks released
       Result returned to client

If crash at T=3: WAL incomplete → recovery sees incomplete transaction →
If crash at T=6 after WAL flush: recovery replays WAL → committed state r
```

Chapter 16 – MVCC Deep Dive

16.1 Transaction IDs and Visibility

Every transaction gets a transaction ID (txid). PostgreSQL uses txids to determine which row versions a transaction can see.

```
-- See the current transaction ID:
SELECT txid_current();
-- → 1234

-- See the current snapshot:
SELECT txid_current_snapshot();
-- → 1230:1235:1231,1232
-- Means:
--   xmin=1230 (all txids < 1230 are committed, visible)
--   xmax=1235 (all txids >= 1235 are in-progress, invisible)
--   xip=1231,1232 (these specific txids are in-progress, invisible)
```

MVCC visibility rules:

```
A tuple is visible to transaction T if:
  1. tuple.xmin is committed AND tuple.xmin < T's snapshot xmax
  2. AND tuple.xmin NOT in T's in-progress list
  3. AND (tuple.xmax = 0 OR tuple.xmax is NOT committed OR tuple.xmax > T

Simplified:
  → xmin must be a committed transaction T can see (row was created)
  → xmax must be empty or a future/uncommitted transaction (row not yet c
```

16.2 Concurrent Update Timeline

```
Session A (txid=100)              Session B (txid=101)
─────────────────────────────     ─────────────────────────────
T1: BEGIN
T2: SELECT * FROM users
    WHERE id=1;
    → sees: {id=1, email='a@x.com', xmin=50, xmax=0}

                                  T3: BEGIN
                                  T4: UPDATE users SET email='b@x.com'
                                      WHERE id=1;
                                      → Creates new tuple:
                                        {id=1, email='b@x.com', xmin=101,
                                        Old tuple: xmax set to 101
```

```
T5: SELECT * FROM users           T6: SELECT * FROM users
    WHERE id=1;                       WHERE id=1;
    → STILL sees old tuple            → sees NEW tuple
      xmin=50 committed,               xmin=101 = own txid,
      xmax=101 NOT yet committed        not committed yet but self-visibl
    → {email='a@x.com'}           → {email='b@x.com'}

                                  T7: COMMIT

T8: SELECT * FROM users
    WHERE id=1;
    → STILL sees old tuple!
      Snapshot was taken at T1.
      Even though 101 committed,
      T's snapshot xmax was 100.
    → {email='a@x.com'}

T9: COMMIT
```

This is **Snapshot Isolation** (PostgreSQL's default READ COMMITTED gives you a fresh snapshot per statement, REPEATABLE READ gives you one snapshot for the whole transaction).

16.3 Dead Tuples and Bloat

Every UPDATE creates a new tuple and marks the old one dead. DELETEs also create dead tuples. Over time, dead tuples accumulate — this is **table bloat**.

```
Before bulk UPDATE (all rows):
Page layout:
┌─────┬─────┬─────┬─────┬─────┐
│  T1 │  T2 │  T3 │  T4 │  T5 │  (all live)
└─────┴─────┴─────┴─────┴─────┘

After UPDATE all 5 rows:
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│ T1* │ T2* │ T3* │ T4* │ T5* │ T1' │ T2' │ T3' │ T4' │ T5' │
│dead │dead │dead │dead │dead │live │live │live │live │live │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
(double the space needed!)

After VACUUM:
┌─────┬─────┬─────┬─────┬─────┐
```

```
│ T1' │ T2' │ T3' │ T4' │ T5' │   (dead tuples reclaimed)
└─────┴─────┴─────┴─────┴─────┘
```

Chapter 17 – Locks and Deadlocks

17.1 Lock Hierarchy

```
PostgreSQL lock levels (weakest to strongest):

ACCESS SHARE           → taken by SELECT
ROW SHARE              → taken by SELECT FOR UPDATE
ROW EXCLUSIVE          → taken by INSERT, UPDATE, DELETE
SHARE UPDATE EXCLUSIVE → taken by VACUUM, CREATE INDEX CONCURRENTLY
SHARE                  → taken by CREATE INDEX (non-concurrent)
SHARE ROW EXCLUSIVE    → taken by triggers
EXCLUSIVE              → rarely used directly
ACCESS EXCLUSIVE       → taken by ALTER TABLE, DROP TABLE, TRUNCATE
                         Blocks EVERYTHING including SELECT
```

Conflict matrix (simplified):

```
                    Held lock →
                    AS  RS  RX  SUE  SH  SRX  EX  AEX
Requested lock ↓
ACCESS SHARE        .   .   .   .    .   .    .   X
ROW SHARE           .   .   .   .    .   .    X   X
ROW EXCLUSIVE       .   .   .   X    X   X    X   X
SHARE UPD EXCL      .   .   X   X    .   X    X   X
SHARE               .   .   X   X    .   X    X   X
SHARE ROW EXCL      .   .   X   X    X   X    X   X
EXCLUSIVE           .   X   X   X    X   X    X   X
ACCESS EXCLUSIVE    X   X   X   X    X   X    X   X

. = compatible (both can hold simultaneously)
X = conflict (second request must wait)
```

17.2 Row-Level Locking

```
-- Lock a specific row for update (prevents concurrent updates):
SELECT * FROM inventory
WHERE product_id = 1
FOR UPDATE;

-- Lock without waiting (fail immediately if locked):
SELECT * FROM inventory
WHERE product_id = 1
FOR UPDATE NOWAIT;

-- Skip locked rows (useful for job queues):
SELECT * FROM orders
WHERE status = 'pending'
ORDER BY created_at
LIMIT 10
FOR UPDATE SKIP LOCKED;
```

SKIP LOCKED for job queues:

```
Worker 1                          Worker 2
────────────────────────────      ────────────────────────────
SELECT id FROM orders             SELECT id FROM orders
WHERE status='pending'            WHERE status='pending'
LIMIT 1                           LIMIT 1
FOR UPDATE SKIP LOCKED;           FOR UPDATE SKIP LOCKED;

→ Gets order id=4                 → Gets order id=5 (skips locked 4)
→ Both workers work               No blocking! No double-processing!
  simultaneously
```

17.3 Deadlocks

A deadlock occurs when two transactions each hold a lock the other needs.

```
T1                                T2
────────────────────────────      ────────────────────────────
BEGIN;                            BEGIN;
UPDATE inventory                  UPDATE inventory
  SET qty=qty-1                     SET qty=qty-1
  WHERE product_id=1;               WHERE product_id=2;
→ Locks row product_id=1          → Locks row product_id=2
```

```
UPDATE inventory                  UPDATE inventory
  SET qty=qty-1                     SET qty=qty-1
  WHERE product_id=2;               WHERE product_id=1;
→ WAITING for T2's lock           → WAITING for T1's lock

Deadlock detected by PostgreSQL after deadlock_timeout (1s default)
→ One transaction is chosen as victim → ROLLBACK
→ Other transaction proceeds
```

Preventing deadlocks:

```
Strategy 1: Always lock rows in the same order
  Both T1 and T2 should lock product_id=1 before product_id=2
  → Impossible for circular dependency to form

Strategy 2: Use FOR UPDATE with explicit ordering
  SELECT * FROM inventory
  WHERE product_id IN (1, 2)
  ORDER BY product_id  -- ensures consistent lock order
  FOR UPDATE;

Strategy 3: Use NOWAIT and retry at application level
  BEGIN;
  SELECT ... FOR UPDATE NOWAIT;  -- fail fast instead of deadlock
  ... if exception: sleep + retry
```

Chapter 18 – Isolation Levels

18.1 The Four Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
READ UNCOMMITTED (not in PostgreSQL*)	Possible	Possible	Possible
READ COMMITTED (PostgreSQL default)	No	Possible	Possible

REPEATABLE READ	No	No	No**
SERIALIZABLE	No	No	No

```
* PostgreSQL treats READ UNCOMMITTED as READ COMMITTED
** PostgreSQL's REPEATABLE READ also prevents phantom reads (stronger tha
```

Definitions:

```
Dirty Read: Reading uncommitted changes from another transaction.
  T1 writes row, T2 reads it, T1 rolls back → T2 read garbage

Non-Repeatable Read: Reading the same row twice gets different values.
  T1 reads row, T2 updates+commits, T1 reads row again → different value

Phantom Read: Re-running a range query returns different rows.
  T1: SELECT COUNT(*) FROM orders WHERE status='pending' → 5
  T2: INSERT 3 new pending orders, COMMIT
  T1: SELECT COUNT(*) FROM orders WHERE status='pending' → 8 (phantom!)
```

18.2 READ COMMITTED (Default) — Step by Step

```
Setting: SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
(this is the default — you get this automatically)

Key behavior: Each STATEMENT gets a fresh snapshot.

T1                                T2
────────────────────────────      ────────────────────────────
BEGIN;
                                  BEGIN;
SELECT total_cents                UPDATE orders SET
FROM orders WHERE id=1;           total_cents=99999 WHERE id=1;
→ 152899                          COMMIT;

-- T2 committed between T1's statements!

SELECT total_cents
```

```
FROM orders WHERE id=1;
→ 99999  ← sees T2's change!
   (new snapshot taken for this statement)

COMMIT;
```

18.3 REPEATABLE READ — Step by Step

```
Setting: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

Key behavior: One snapshot for the entire transaction.

T1                              T2
────────────────────────────    ────────────────────────────
BEGIN;
-- Snapshot taken HERE
                                UPDATE orders SET
                                total_cents=99999 WHERE id=1;
                                COMMIT;

SELECT total_cents
FROM orders WHERE id=1;
→ 152899  ← sees original value
  (snapshot was taken before T2's update)

SELECT total_cents
FROM orders WHERE id=1;
→ 152899  ← same! Repeatable read guaranteed

COMMIT;
```

18.4 SERIALIZABLE — The Strongest Level

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
Serializable ensures transactions execute as if they ran one-at-a-time.
Implemented via Serializable Snapshot Isolation (SSI) in PostgreSQL.

Example scenario (serialization anomaly):
T1: SELECT SUM(amount) FROM accounts WHERE type='checking'; → 1000
```

```
T1: INSERT INTO accounts (type, amount) VALUES ('savings', 1000);
                              T2: SELECT SUM(amount) FROM accounts WHERE
                              T2: INSERT INTO accounts (type, amount) VAI

Each transaction read from one type and wrote to the other.
Under READ COMMITTED or REPEATABLE READ: both succeed.
Under SERIALIZABLE: one will be rolled back with serialization failure er
Application must retry the rolled-back transaction.
```

When to use each:

```
READ COMMITTED:    → Default. Correct for most OLTP workloads.
REPEATABLE READ:   → When a single transaction must see a consistent snaps
                     E.g., generating a report across multiple queries.
SERIALIZABLE:      → Financial transactions, anything requiring
                     absolute consistency guarantees.
                     Accept: ~5-10% more serialization failures to retry.
```

Chapter 15–18 Exercises

Beginner

1. Write a transaction that inserts an order and a payment atomically. Verify the transaction is atomic by intentionally causing an error mid-transaction.
2. What isolation level would you use for a nightly report that runs 50 queries and must see a consistent snapshot of the database? Why?

Intermediate
3. Two sessions try to update the same inventory row simultaneously. Write the SQL for each session and trace through what happens step by step.
4. Implement the job queue pattern using SKIP LOCKED for an `order_processing_queue` table.

Advanced
5. Design a distributed funds transfer (debit one account, credit another) that is safe under concurrent access. Include: lock ordering to prevent deadlocks, constraint to prevent negative balances, and proper transaction structure.
6. Explain why a long-running VACUUM can be blocked and what infrastructure implications this has. How do you monitor for this?

Solutions

1.

```
BEGIN;
-- This will fail (product_id=999 doesn't exist)
INSERT INTO orders (user_id, status, total_cents) VALUES (1, 'pending', 1
INSERT INTO payments (order_id, amount_cents, status, provider)
VALUES (999, 1000, 'pending', 'stripe');
-- Error: foreign key violation
ROLLBACK;
-- Both operations rolled back atomically
```

2. REPEATABLE READ. The first query takes a snapshot, and all subsequent queries in the same transaction see that same snapshot. Without it, data could change between queries giving an inconsistent report (e.g., counting orders in one query and summing their values in another could mismatch if orders are added/updated between queries).

4.

```
CREATE TABLE order_processing_queue (
    id         BIGSERIAL PRIMARY KEY,
    order_id   BIGINT NOT NULL REFERENCES orders(id),
    status     TEXT NOT NULL DEFAULT 'pending',
    created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Worker process:
BEGIN;
SELECT id, order_id
FROM order_processing_queue
WHERE status = 'pending'
ORDER BY created_at
LIMIT 1
FOR UPDATE SKIP LOCKED;

-- Process the order...

UPDATE order_processing_queue SET status = 'completed' WHERE id = $1;
COMMIT;
```

5.

```
-- Always lock in ascending account ID order to prevent deadlocks:
BEGIN;

-- Lock both rows in consistent order:
SELECT id, balance FROM accounts
WHERE id IN (1, 2)
ORDER BY id
FOR UPDATE;

-- Check sufficient funds:
-- (application verifies balance >= transfer_amount)

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

-- CHECK constraint (balance >= 0) will reject if insufficient funds

COMMIT;
```

Part 7 – Performance Engineering

Chapter 19 – The Query Planner and Optimizer

19.1 How the Planner Works

The query planner's job is to find the cheapest way to execute a query. "Cheapest" means minimum estimated cost (in arbitrary units representing disk I/O + CPU work).

```
Input SQL → Parse → Analyze → Rewrite → Plan → Execute
                                          ↑ THIS is the planner

Planning steps:
1. Generate all possible scan methods for each table
   (SeqScan, IndexScan, IndexOnlyScan, BitmapScan)

2. Generate all possible join orders and methods
   (NestedLoop, HashJoin, MergeJoin)
```

```
3. For each plan, estimate cost using:
   - Table statistics (pg_statistics)
   - Table size (pg_class.relpages, reltuples)
   - Index statistics

4. Choose minimum cost plan
```

Statistics are the planner's eyes:

```sql
-- Run ANALYZE to update statistics:
ANALYZE users;
ANALYZE orders;

-- Or let autovacuum do it (default):
-- autovacuum_analyze_threshold = 50 rows
-- autovacuum_analyze_scale_factor = 0.2 (20% of table changed)

-- See statistics:
SELECT attname, n_distinct, correlation
FROM pg_stats
WHERE tablename = 'orders' AND attname = 'status';
```

19.2 Scan Types

```
SEQUENTIAL SCAN (SeqScan)
  Read every page in the table, filter as you go.
  Cost: O(pages in table)
  Best when: fetching a large fraction of rows (>5-10%)

INDEX SCAN
  Traverse B-Tree → get ctid → fetch heap page → return tuple
  Cost: O(log n) per row + random I/O
  Best when: fetching few rows (pointer chasing is expensive at scale)

INDEX ONLY SCAN
  Traverse B-Tree → get data directly from index (if all columns indexed)
  Cost: O(log n) per row, minimal heap access
  Best when: query only needs columns in the index

BITMAP INDEX SCAN
  Phase 1: Scan index → build bitmap of matching pages
  Phase 2: Fetch heap pages in page order (sequential-ish)
  Best when: fetching medium fraction of rows (5-40%)
  Avoids random I/O by batching page fetches

Index: [page1, page3, page1, page5, page3] → sort → [page1, page1, page3,
Heap reads: page1, page3, page5 (each page read once)
```

Chapter 20 – EXPLAIN and EXPLAIN ANALYZE

20.1 Reading an Execution Plan

```
EXPLAIN SELECT * FROM orders WHERE user_id = 1;
```

```
                        QUERY PLAN
─────────────────────────────────────────────────────────────────
 Index Scan using orders_user_id_idx on orders  (cost=0.28..8.30 rows=2 w
   Index Cond: (user_id = 1)
```

Anatomy of a plan node:

```
 Index Scan using orders_user_id_idx on orders
 ├── Node type: Index Scan
 ├── Index used: orders_user_id_idx
 └── Table: orders

 (cost=0.28..8.30 rows=2 width=48)
        │          │       │     │
        │          │       │     └── Estimated avg row width (bytes)
        │          │       └──────── Estimated rows returned
        │          └──────────────── Estimated total cost
        └─────────────────────────── Estimated startup cost
                                      (cost to return first row)
```

Cost units: 1 unit ≈ read one 8KB page sequentially. Random I/O costs more (`random_page_cost` = 4.0 by default). CPU operations are tiny fractions.

20.2 EXPLAIN ANALYZE — Actual vs Estimated

```
EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT)
SELECT
    u.username,
    COUNT(o.id) as order_count
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
GROUP BY u.id, u.username;
```

```
                              QUERY PLAN
─────────────────────────────────────────────────────────────────────────
 HashAggregate  (cost=24.50..26.50 rows=200 width=40)
                (actual time=0.125..0.131 rows=5 loops=1)
   Group Key: u.id, u.username
   Batches: 1  Memory Usage: 24kB
   ->  Hash Left Join  (cost=12.75..22.00 rows=500 width=32)
                       (actual time=0.067..0.102 rows=6 loops=1)
         Hash Cond: (o.user_id = u.id)
         ->  Seq Scan on orders  (cost=0.00..1.05 rows=5 width=12)
                                 (actual time=0.014..0.020 rows=5 loops=1
               Buffers: shared hit=1
         ->  Hash  (cost=11.00..11.00 rows=100 width=28)
                   (actual time=0.036..0.036 rows=5 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on users  (cost=0.00..11.00 rows=100 width=28
                                      (actual time=0.015..0.023 rows=5 lo
                     Buffers: shared hit=1
 Planning Time: 0.312 ms
 Execution Time: 0.213 ms
```

Reading the plan (bottom-up):

```
Step 1: Seq Scan on users  → reads all users (5 rows, 1 buffer hit)
Step 2: Hash               → builds hash table from users
Step 3: Seq Scan on orders → reads all orders (5 rows, 1 buffer hit)
Step 4: Hash Left Join     → probes hash table for each order row
Step 5: HashAggregate      → groups by user and counts

"Buffers: shared hit=1" means the page was already in shared_buffers (no
"shared read=X" would mean disk reads.
```

20.3 Spotting Performance Problems in EXPLAIN

```
Red flags to look for:

1. HUGE rows estimate vs actual:
   (cost=... rows=50000) (actual... rows=3)
   → Stale statistics. Run ANALYZE.

2. Sequential scan on large table:
   Seq Scan on orders (actual rows=8000000)
   → Missing index? Or is the query returning >10% of rows?

3. Nested Loop with many loops:
   Nested Loop (loops=500000)
   → Inner side isn't using an index. O(n²) problem.

4. Hash Join spilling to disk:
   Batches: 8  (> 1 means work_mem was exceeded, spilled to disk)
   → Increase work_mem for this query.

5. Sort with large memory:
   Sort Method: external merge  Disk: 128MB
   → work_mem too low for this sort.

6. Filter removing many rows:
   Filter: (status = 'delivered')
   Rows Removed by Filter: 9850000
   → Index on status would help.
```

20.4 Debugging a Slow Query — Worked Example

Problem: The order history page is slow. The query:

```sql
SELECT o.id, o.status, o.total_cents, o.created_at,
       u.username
FROM orders o
JOIN users u ON u.id = o.user_id
WHERE o.user_id = 1
  AND o.created_at > now() - INTERVAL '30 days'
ORDER BY o.created_at DESC
LIMIT 20;
```

Step 1: Run EXPLAIN ANALYZE:

```
Limit  (actual time=1823.5..1823.6 rows=20 loops=1)
  ->  Sort  (actual time=1823.4..1823.5 rows=20 loops=1)
        Sort Key: o.created_at DESC
        Sort Method: external merge  Disk: 512MB   ← RED FLAG
        ->  Hash Join  (actual time=...)
              Hash Cond: (o.user_id = u.id)
              ->  Seq Scan on orders                  ← RED FLAG
                    Filter: ((user_id=1) AND (created_at > ...))
                    Rows Removed by Filter: 9,800,000  ← RED FLAG
              ->  Hash  ...
```

Diagnosis:

1. Sequential scan on orders — no index on user_id or created_at
2. 9.8M rows filtered down to ~20 — extremely wasteful
3. Sort spilling to disk — 512MB for a LIMIT 20 query is absurd

Fix:

```
CREATE INDEX ON orders (user_id, created_at DESC);
```

After the index:

```
Limit  (actual time=0.072..0.092 rows=20 loops=1)
  ->  Index Scan Backward using orders_user_id_created_at_idx on orders
        Index Cond: ((user_id=1) AND (created_at > '2024-01-15 ...'))
        (actual time=0.052..0.082 rows=20 loops=1)
        Buffers: shared hit=3
```

1823ms → 0.09ms. **20,000× faster.**

The composite index on `(user_id, created_at DESC)` lets PostgreSQL:

1. Jump directly to user_id=1 rows
2. Scan them in descending date order
3. Stop after 20 rows (no sort needed!)

Chapter 19–20 Exercises

Beginner

1. Run `EXPLAIN SELECT * FROM users WHERE email = 'alice@example.com'`. What scan type does it use? Why?
2. Add `ANALYZE` to your EXPLAIN and identify: what is the startup cost, total cost, estimated rows, and actual rows?

Intermediate
3. Run EXPLAIN ANALYZE on the multi-table JOIN query from Chapter 7. Identify the join strategy. Is it optimal?
4. Write a query that would benefit from an Index Only Scan and create the appropriate covering index.

Advanced
5. You have a query: `SELECT * FROM orders WHERE status = 'pending' ORDER BY created_at LIMIT 100` on a table with 5M orders where 2% are pending. Walk through: what indexes would help, why, and predict what EXPLAIN would show.
6. Explain the "N+1 query problem" in terms of PostgreSQL execution plans. Write a before/after query pair showing how to fix it.

Solutions

4.

```
-- Index Only Scan covers all needed columns in the index:
CREATE INDEX ON orders (user_id, status, total_cents, created_at);

-- This query uses Index Only Scan (no heap fetch needed):
SELECT user_id, status, total_cents, created_at
FROM orders
WHERE user_id = 1;
```

5.

```
-- 2% of 5M = 100K pending orders. Index will help significantly.
-- Composite index:
CREATE INDEX ON orders (status, created_at)
WHERE status = 'pending';  -- partial index!

-- EXPLAIN would show:
-- Index Scan using orders_status_created_at_idx on orders
--   Index Cond: (status = 'pending')
--   (actual rows=100, loops=1)
-- No sort needed because index provides created_at order
-- Limit stops after 100 rows
-- vs without index: SeqScan 5M rows, sort all pending, take 100
```

Part 8 – PostgreSQL in Production

Chapter 21 – Connection Pooling

21.1 The Connection Problem

Each PostgreSQL connection is an OS process consuming ~5-10MB RAM. At 500 connections: 2.5-5GB just for process overhead.

```
Without pooling:
  App Servers (500 threads each with a DB connection)
  ────────────────────────────────────────────────
  App1: conn1, conn2, conn3, ... conn100
  App2: conn101, conn102, ... conn200
  ...

  PostgreSQL: 500 backend processes
  Memory: 500 × 8MB = 4GB just for backends

  Problem: Most connections idle at any moment,
  wasting memory and adding scheduling overhead.

With PgBouncer (transaction pooling):
  App Servers                PgBouncer              PostgreSQL
  ──────────────────         ─────────────          ──────────
  500 "connections"   →      20 real connections →  20 backends
  (to PgBouncer)             (to PostgreSQL)        50MB RAM

  PgBouncer assigns a real connection to an app
  connection only for the duration of a transaction.
  After COMMIT, the real connection returns to pool.
```

21.2 PgBouncer Configuration

```
# /etc/pgbouncer/pgbouncer.ini

[databases]
mydb = host=localhost port=5432 dbname=mydb

[pgbouncer]
listen_addr = 0.0.0.0
listen_port = 6432

# POOLING MODE:
# session     → one real connection per client session (minimal benefit)
# transaction → real connection per transaction (BEST for most apps)
# statement   → per statement (breaks transactions! avoid)
pool_mode = transaction

# How many real PostgreSQL connections per database/user pair:
default_pool_size = 25

# Max waiting clients (queue size):
max_client_conn = 1000

# Connections kept open even when idle:
min_pool_size = 5

# Return connection to pool after N seconds idle:
server_idle_timeout = 600

# Reject client if can't get a connection within N seconds:
query_wait_timeout = 30
```

21.3 What Breaks With Transaction Pooling

```
NOT compatible with transaction pooling:

1. SET statements that persist across transactions:
   SET search_path = myschema;  ← resets when connection returned to pool
   FIX: Use schema-qualified names, or SET LOCAL

2. Advisory locks:
   pg_try_advisory_lock()  ← connection-scoped, lost on pool return
   FIX: Use application-level locking or row locks
```

```
3. LISTEN/NOTIFY:
   LISTEN channel;  ← session-scoped
   FIX: Use a dedicated long-lived connection for pub/sub

4. Prepared statements with PREPARE:
   PREPARE myplan AS SELECT...;
   FIX: Use protocol-level prepared statements (driver handles this)
   or disable statement-level prepared statements in PgBouncer
```

Chapter 22 – Backups and Restores

22.1 Backup Types

Method	Description
pg_dump	Logical dump. SQL or binary. Per-database. Can restore to different PostgreSQL versions. Consistent snapshot (uses MVCC). Does NOT capture WAL position.
pg_dumpall	Like pg_dump but includes roles and tablespaces.
pg_basebackup	Physical backup. Copies data directory. Requires streaming replication setup. Used as base for PITR and standbys.
PITR (WAL archiving)	Point-in-Time Recovery. Base backup + WAL segments → restore to any point. Production gold standard for RPO near zero.

22.2 pg_dump

```
# Dump to custom format (recommended — supports parallel restore):
pg_dump -Fc -d mydb -f mydb_backup.dump

# Dump to SQL (human readable, slower restore):
pg_dump -d mydb -f mydb_backup.sql
```

```
# Dump specific table:
pg_dump -Fc -d mydb -t orders -f orders_backup.dump

# Parallel dump (4 workers):
pg_dump -Fd -j 4 -d mydb -f mydb_backup_dir/

# Restore:
pg_restore -Fc -d mydb_restored mydb_backup.dump

# Parallel restore:
pg_restore -Fd -j 4 -d mydb_restored mydb_backup_dir/
```

22.3 PITR (Point-in-Time Recovery)

```
Setup:
1. Configure WAL archiving in postgresql.conf:
   wal_level = replica
   archive_mode = on
   archive_command = 'cp %p /backup/wal/%f'

2. Take base backup:
   pg_basebackup -D /backup/base -Ft -Xs -P -R

3. WAL segments are continuously archived as changes happen.

Recovery scenario:
  "Someone dropped the users table at 14:32. Recover to 14:31."

Step 1: Stop PostgreSQL
Step 2: Restore base backup to data directory
Step 3: Create recovery.conf (or recovery signal file):
        restore_command = 'cp /backup/wal/%f %p'
        recovery_target_time = '2024-01-15 14:31:00'
Step 4: Start PostgreSQL
        → Replays WAL from backup up to 14:31
        → Stops before the DROP TABLE

Recovery Time Objective (RTO): minutes to hours depending on WAL volume
Recovery Point Objective (RPO): seconds to last archived WAL segment
```

Chapter 23 – Replication and HA

23.1 Streaming Replication

```
Primary                         Standby
─────────────────────           ──────────────────────

WAL generated by writes
  ↓
WAL written to pg_wal/
  ↓
WAL sender process ──────────→ WAL receiver process
                    (TCP/IP)          ↓
                              WAL applied to data files

Result: Standby is a near-real-time copy of primary.
Lag: typically milliseconds on LAN.
```

Synchronous vs Asynchronous:

```
ASYNCHRONOUS (default):
  COMMIT returns to app after WAL flushed on PRIMARY only.
  Standby may lag by milliseconds-seconds.
  Risk: If primary fails before standby receives WAL, data loss.
  Best for: most applications, high throughput.

SYNCHRONOUS:
  COMMIT returns after WAL flushed on PRIMARY AND at least one STANDBY.
  No data loss on failover (RPO=0).
  Cost: ~2× latency per write (one extra network round-trip).
  Best for: financial systems requiring zero data loss.

# postgresql.conf on primary:
synchronous_standby_names = 'standby1'  # enables sync replication
```

23.2 Failover and HA Concepts

```
Production HA setup:

┌──────────────────────────────────────────────────────────────┐
│                       LOAD BALANCER                          │
│                    (HAProxy / AWS ALB)                              │
└──────────────────────────────┬──────────────────────────────────────┘
                               │
               ┌───────────────┴──────────────┐
               ▼                              ▼
┌─────────────────┐            ┌────────────────────┐
│    PRIMARY      │            │   READ REPLICA     │
│  (reads+writes) │──WAL────→  │  (reads only)      │
│                 │            │                    │
└─────────────────┘            └────────────────────┘
         │
    Patroni / Repmgr
    (HA manager)
         │
         ├── Monitors primary health
         ├── Promotes standby on failure
         └── Updates DNS / load balancer

Automated failover timeline:
T=0    Primary fails (hardware failure, crash)
T=5s   HA manager detects failure (health check interval)
T=10s  Standby promoted to primary
T=15s  DNS/VIP updated to point to new primary
T=20s  Application reconnects (connection retries)
Total downtime: ~20 seconds (RTO)
```

23.3 Safe Schema Migrations

```
The ALTER TABLE problem:
  ALTER TABLE orders ADD COLUMN is_fraud BOOLEAN DEFAULT false;

  On a table with 10M rows, this acquires ACCESS EXCLUSIVE lock.
  This blocks ALL reads and writes until complete.
  Duration: could be minutes.

  During migration: every query waits → connection pile-up → service outa
```

Safe migration strategies:

```
-- SAFE: Add nullable column (instant):
ALTER TABLE orders ADD COLUMN is_fraud BOOLEAN;
-- No lock needed! NULLable columns with no default are free in PostgreSQ
-- SAFE: Add column with DEFAULT (PostgreSQL 11+, instant):
ALTER TABLE orders ADD COLUMN processed_at TIMESTAMPTZ DEFAULT now();
-- PostgreSQL 11+ stores the default in catalog, doesn't rewrite table.

-- SAFE: Create index without blocking:
CREATE INDEX CONCURRENTLY orders_user_id_idx ON orders (user_id);
-- Takes ShareUpdateExclusiveLock (doesn't block reads/writes)
-- Takes longer than normal CREATE INDEX but doesn't block.

-- SAFE: Add NOT NULL constraint (PostgreSQL 12+):
-- Step 1: Add nullable column
ALTER TABLE orders ADD COLUMN new_status TEXT;
-- Step 2: Backfill data
UPDATE orders SET new_status = status WHERE new_status IS NULL;
-- Step 3: Add CHECK constraint (validates but doesn't scan table in PG12
ALTER TABLE orders ADD CONSTRAINT orders_new_status_nn
    CHECK (new_status IS NOT NULL) NOT VALID;
-- Step 4: Validate constraint (concurrent, non-blocking):
ALTER TABLE orders VALIDATE CONSTRAINT orders_new_status_nn;
-- Step 5: Now set NOT NULL (fast, uses validated constraint):
ALTER TABLE orders ALTER COLUMN new_status SET NOT NULL;
-- Step 6: Drop constraint:
ALTER TABLE orders DROP CONSTRAINT orders_new_status_nn;
```

Chapter 21–23 Exercises

Beginner

1. What are the three PgBouncer pool modes? Which is recommended for most web applications?
2. Write a pg_dump command to back up only the orders table to a custom-format file.

Intermediate

3. You need to add a `last_login` TIMESTAMPTZ column to a users table with 50M rows in production. Write the safest migration plan.
4. Your primary database fails at 3:47 AM. Walk through the failover steps and what information you need to assess data loss.

Advanced

5. Design a complete HA architecture for an e-commerce application requiring: RPO < 1 minute, RTO < 30 seconds, read scale-out for analytics queries. Draw the component diagram.

Solutions

3.

```sql
-- All steps are safe for production (no extended locks):

-- 1. Add nullable column (instant in PostgreSQL 11+):
ALTER TABLE users ADD COLUMN last_login TIMESTAMPTZ;

-- 2. Backfill from sessions table (run during low traffic, in batches):
UPDATE users u
SET last_login = (
    SELECT MAX(created_at) FROM sessions s WHERE s.user_id = u.id
)
WHERE last_login IS NULL;

-- 3. Create index concurrently if needed:
CREATE INDEX CONCURRENTLY ON users (last_login)
WHERE last_login IS NOT NULL;

-- No table rewrite, no extended locking, no downtime.
```

Part 9 – Security & Reliability

Chapter 24 – Roles and Permissions

24.1 PostgreSQL Role System

```
In PostgreSQL, "users" and "groups" are both ROLES.
A role with LOGIN privilege = a user.
A role without LOGIN = a group/permission bundle.

Role hierarchy example:

role: readonly
  └── Can SELECT on all tables in schema public

role: readwrite
```

```
    └── inherits: readonly
    └── Can INSERT, UPDATE, DELETE on all tables

role: app_user (LOGIN)
    └── member of: readwrite
    └── This is what your application connects as

role: report_user (LOGIN)
    └── member of: readonly
    └── Used by analytics/reporting tools


-- Create roles:
CREATE ROLE readonly;
CREATE ROLE readwrite;
CREATE ROLE app_user LOGIN PASSWORD 'secure_password';
CREATE ROLE report_user LOGIN PASSWORD 'another_password';

-- Grant permissions to roles:
GRANT USAGE ON SCHEMA public TO readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT ON TABLES TO readonly;

GRANT readonly TO readwrite;  -- readwrite inherits readonly
GRANT INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO readwrite;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT INSERT, UPDATE, DELETE ON TABLES TO readwrite;

-- Assign roles to users:
GRANT readwrite TO app_user;
GRANT readonly TO report_user;

-- Restrict app_user to specific tables:
REVOKE ALL ON TABLE payments FROM readwrite;
GRANT SELECT, INSERT ON payments TO app_user;
-- (app can insert payments but not delete them)
```

24.2 Row-Level Security (RLS)

RLS lets you add row-level predicates to table access — every query automatically gets a WHERE clause added based on the current user.

Use case: Multi-tenant SaaS — users can only see their own data.

```
-- Enable RLS on orders:
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;

-- Create policy: users can only see their own orders:
CREATE POLICY orders_user_isolation ON orders
    USING (user_id = current_setting('app.current_user_id')::INTEGER);

-- In your application, before each query:
SET LOCAL app.current_user_id = '42';
SELECT * FROM orders;  -- automatically filtered to user_id=42

-- Admins bypass RLS:
ALTER TABLE orders FORCE ROW LEVEL SECURITY;
GRANT BYPASSRLS TO admin_role;

-- Separate read and write policies:
CREATE POLICY orders_read ON orders
    FOR SELECT
    USING (user_id = current_setting('app.current_user_id')::INTEGER);

CREATE POLICY orders_insert ON orders
    FOR INSERT
    WITH CHECK (user_id = current_setting('app.current_user_id')::INTEGEF
```

24.3 Auditing

```
-- Audit table:
CREATE TABLE audit_log (
    id          BIGSERIAL PRIMARY KEY,
    table_name  TEXT NOT NULL,
    operation   TEXT NOT NULL,  -- INSERT, UPDATE, DELETE
    old_data    JSONB,
    new_data    JSONB,
    changed_by  TEXT NOT NULL DEFAULT current_user,
    changed_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Audit trigger:
CREATE OR REPLACE FUNCTION audit_trigger_function()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
```

```
        INSERT INTO audit_log (table_name, operation, new_data)
        VALUES (TG_TABLE_NAME, 'INSERT', to_jsonb(NEW));
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO audit_log (table_name, operation, old_data, new_data)
        VALUES (TG_TABLE_NAME, 'UPDATE', to_jsonb(OLD), to_jsonb(NEW));
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO audit_log (table_name, operation, old_data)
        VALUES (TG_TABLE_NAME, 'DELETE', to_jsonb(OLD));
    END IF;
    RETURN COALESCE(NEW, OLD);
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

-- Apply to sensitive tables:
CREATE TRIGGER orders_audit
    AFTER INSERT OR UPDATE OR DELETE ON orders
    FOR EACH ROW EXECUTE FUNCTION audit_trigger_function();

CREATE TRIGGER payments_audit
    AFTER INSERT OR UPDATE OR DELETE ON payments
    FOR EACH ROW EXECUTE FUNCTION audit_trigger_function();
```

Chapter 24 Exercises

Beginner

1. Create a role `analytics_user` that can only SELECT from the orders and payments tables.
2. List all roles in a PostgreSQL cluster and show which roles they are members of.

Intermediate
3. Implement RLS on the logs table so that each service can only read its own logs (based on a `app.current_service` setting).
4. Modify the audit trigger to also record the session_user and the application_name from pg_stat_activity.

Advanced
5. Design a complete RBAC system for a multi-tenant SaaS with three user types: superadmin (sees all tenants), tenant_admin (manages their tenant), tenant_user (read-only their tenant). Include RLS policies and role hierarchy.

Solutions

1.

```
CREATE ROLE analytics_user LOGIN PASSWORD 'password';
GRANT USAGE ON SCHEMA public TO analytics_user;
GRANT SELECT ON orders, payments TO analytics_user;
```

2.

```
SELECT r.rolname, r.rolsuper, r.rolinherit, r.rolcanlogin,
       array_agg(m.roleid::regrole::TEXT) AS member_of
FROM pg_roles r
LEFT JOIN pg_auth_members m ON m.member = r.oid
GROUP BY r.rolname, r.rolsuper, r.rolinherit, r.rolcanlogin
ORDER BY r.rolname;
```

3.

```
ALTER TABLE logs ENABLE ROW LEVEL SECURITY;

CREATE POLICY logs_service_isolation ON logs
    FOR ALL
    USING (service = current_setting('app.current_service', true));

-- 'true' in current_setting means: return NULL if not set (don't error)
-- When app.current_service is NULL, no rows are visible (safe default)
```

Part 10 – Engineer Playbook

Chapter 25 – Schema Design Patterns

25.1 Soft Deletes

```
-- Pattern: mark rows deleted instead of removing them
ALTER TABLE users ADD COLUMN deleted_at TIMESTAMPTZ;

-- "Delete":
UPDATE users SET deleted_at = now() WHERE id = 42;
```

```sql
-- All queries must filter:
SELECT * FROM users WHERE deleted_at IS NULL;

-- Better: create a view:
CREATE VIEW active_users AS
    SELECT * FROM users WHERE deleted_at IS NULL;

-- Index for performance:
CREATE INDEX ON users (deleted_at) WHERE deleted_at IS NULL;
-- Partial index: only NULL rows indexed → tiny, fast
```

25.2 Status State Machine

```sql
-- Enforce valid status transitions via trigger:
CREATE OR REPLACE FUNCTION validate_order_status_transition()
RETURNS TRIGGER AS $$
DECLARE
    valid_transitions JSONB := '{
        "pending":    ["processing", "cancelled"],
        "processing": ["shipped", "cancelled"],
        "shipped":    ["delivered"],
        "delivered":  [],
        "cancelled":  []
    }';
BEGIN
    IF OLD.status = NEW.status THEN
        RETURN NEW;  -- no change
    END IF;

    IF NOT (valid_transitions->OLD.status) @> to_jsonb(NEW.status) THEN
        RAISE EXCEPTION 'Invalid status transition: % → %',
            OLD.status, NEW.status;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER enforce_order_status_transition
    BEFORE UPDATE OF status ON orders
    FOR EACH ROW EXECUTE FUNCTION validate_order_status_transition();
```

25.3 Versioned Records (Temporal Tables)

```
-- Keep full history of price changes:
CREATE TABLE product_price_history (
    id           BIGSERIAL PRIMARY KEY,
    product_id  INTEGER NOT NULL REFERENCES products(id),
    price_cents INTEGER NOT NULL,
    effective_from TIMESTAMPTZ NOT NULL,
    effective_to   TIMESTAMPTZ,  -- NULL = current price
    changed_by  INTEGER REFERENCES users(id)
);

-- Get price at a specific point in time:
SELECT price_cents
FROM product_price_history
WHERE product_id = 1
  AND effective_from <= '2024-01-15 12:00:00'
  AND (effective_to IS NULL OR effective_to > '2024-01-15 12:00:00');
```

Chapter 26 – Anti-Patterns

26.1 The NOT IN / NULL Trap

```
-- DANGEROUS: returns 0 rows if orders has ANY row with NULL user_id:
SELECT id FROM users
WHERE id NOT IN (SELECT user_id FROM orders);

-- SAFE: use NOT EXISTS:
SELECT id FROM users u
WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.user_id = u.id);

-- Proof:
SELECT 1 WHERE 5 NOT IN (1, 2, NULL);
-- Returns 0 rows! Because: 5 NOT IN (1, 2, NULL)
-- = 5 != 1 AND 5 != 2 AND 5 != NULL
-- = true AND true AND UNKNOWN = UNKNOWN → filtered out
```

26.2 SELECT * in Production Code

```
-- NEVER use SELECT * in application code:
SELECT * FROM orders JOIN order_items ON ...
-- Problem: adding a column to orders changes your result set
-- Problem: fetches columns you don't need (JSONB, TEXT fields = huge)
-- Problem: can't use Index Only Scan

-- ALWAYS name columns:
SELECT o.id, o.status, o.total_cents, oi.product_id, oi.quantity
FROM orders o JOIN order_items oi ON oi.order_id = o.id
```

26.3 Storing Money as FLOAT

```
-- WRONG: floating point arithmetic loses precision:
CREATE TABLE payments (amount FLOAT);
INSERT INTO payments VALUES (0.1), (0.2);
SELECT SUM(amount) FROM payments;
-- → 0.30000000000000004 (!)

-- RIGHT: store as integer cents:
CREATE TABLE payments (amount_cents INTEGER);
INSERT INTO payments VALUES (10), (20);
SELECT SUM(amount_cents) FROM payments;
-- → 30 (exact)
-- Display: 30 / 100.0 = 0.30

-- Alternative: NUMERIC (exact decimal):
CREATE TABLE payments (amount NUMERIC(12, 2));
```

26.4 Missing Indexes on Foreign Keys

```
-- After every REFERENCES, create an index:
CREATE TABLE orders (
    user_id INTEGER REFERENCES users(id)
    -- users.id has index (PK), but orders.user_id does NOT
);

-- Always add:
CREATE INDEX ON orders (user_id);

-- Why: DELETE FROM users WHERE id=X must verify no child rows exist.
-- Without the FK index, this is a full sequential scan of orders.
-- At 10M orders: 10M page reads per user deletion.
```

26.5 Unbounded Queries

```
-- DANGEROUS: could return millions of rows:
SELECT * FROM logs WHERE level = 'DEBUG';

-- ALWAYS add LIMIT in application queries:
SELECT * FROM logs WHERE level = 'DEBUG' LIMIT 1000;

-- For pagination, use keyset pagination (not OFFSET):
-- WRONG (offset pagination):
SELECT * FROM orders ORDER BY id LIMIT 20 OFFSET 10000;
-- PostgreSQL still reads 10020 rows to skip to offset 10000!

-- RIGHT (keyset pagination):
SELECT * FROM orders WHERE id > 10000 ORDER BY id LIMIT 20;
-- Uses index, reads exactly 20 rows regardless of "page"
```

Chapter 27 – Interview-Grade SQL Problems

Problem 1: Second Highest Salary

Find the user with the second highest total spending.

```
-- Method 1: OFFSET
SELECT username, total_spent
FROM (
    SELECT u.username, SUM(o.total_cents) AS total_spent
    FROM users u JOIN orders o ON o.user_id = u.id
    GROUP BY u.id, u.username
) t
ORDER BY total_spent DESC
LIMIT 1 OFFSET 1;

-- Method 2: Window function (handles ties correctly)
WITH ranked AS (
    SELECT
        u.username,
        SUM(o.total_cents) AS total_spent,
        DENSE_RANK() OVER (ORDER BY SUM(o.total_cents) DESC) AS rnk
    FROM users u JOIN orders o ON o.user_id = u.id
    GROUP BY u.id, u.username
)
SELECT username, total_spent
FROM ranked
WHERE rnk = 2;
```

Problem 2: Consecutive Order Dates

Find users who placed orders on 3 or more consecutive days.

```
WITH order_dates AS (
    SELECT DISTINCT
        user_id,
        DATE(created_at) AS order_date
    FROM orders
),
with_gaps AS (
    SELECT
        user_id,
        order_date,
        order_date - ROW_NUMBER() OVER (
            PARTITION BY user_id ORDER BY order_date
        ) * INTERVAL '1 day' AS grp
    FROM order_dates
)
SELECT u.username, MIN(order_date) AS streak_start,
       MAX(order_date) AS streak_end,
       COUNT(*) AS consecutive_days
FROM with_gaps wg
JOIN users u ON u.id = wg.user_id
GROUP BY wg.user_id, u.username, wg.grp
HAVING COUNT(*) >= 3
ORDER BY consecutive_days DESC;
```

How the gaps trick works:

```
user_id=1 orders on: Jan 1, Jan 2, Jan 3, Jan 7, Jan 8

ROW_NUMBER:  1,    2,    3,    4,    5
```

```
order_date - row * 1 day:
Jan 1 - 1d = Dec 31
Jan 2 - 2d = Dec 31
Jan 3 - 3d = Dec 31  ← same "group" for consecutive days!
Jan 7 - 4d = Jan 3
Jan 8 - 5d = Jan 3   ← new group for the second streak

GROUP BY this derived date → count of consecutive days per streak
```

Problem 3: Moving Average Revenue

Calculate 7-day moving average revenue, excluding cancelled orders.

```sql
WITH daily_revenue AS (
    SELECT
        DATE(created_at) AS day,
        SUM(total_cents) AS revenue
    FROM orders
    WHERE status != 'cancelled'
    GROUP BY 1
)
SELECT
    day,
    revenue,
    AVG(revenue) OVER (
        ORDER BY day
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS moving_avg_7d,
    COUNT(*) OVER (
        ORDER BY day
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS days_in_window  -- <7 at start
FROM daily_revenue
ORDER BY day;
```

Problem 4: Gaps and Islands

Find time periods where no orders were placed (gaps in order activity).

```sql
WITH daily_order_counts AS (
    SELECT
```

```
        generate_series(
            MIN(DATE(created_at)),
            MAX(DATE(created_at)),
            '1 day'::interval
        )::date AS day
    FROM orders
),
order_activity AS (
    SELECT
        d.day,
        COUNT(o.id) AS order_count,
        COUNT(o.id) = 0 AS is_gap
    FROM daily_order_counts d
    LEFT JOIN orders o ON DATE(o.created_at) = d.day
    GROUP BY d.day
),
islands AS (
    SELECT
        day,
        is_gap,
        day - ROW_NUMBER() OVER (
            PARTITION BY is_gap ORDER BY day
        ) * INTERVAL '1 day' AS grp
    FROM order_activity
)
SELECT
    MIN(day) AS gap_start,
    MAX(day) AS gap_end,
    COUNT(*) AS gap_days
FROM islands
WHERE is_gap = true
GROUP BY grp
ORDER BY gap_start;
```

Chapter 28 – Production War Stories

War Story 1: The Missing FK Index

Symptom: Database load spikes every night at midnight. Alerts firing. `pg_stat_activity` shows 50 sessions stuck in `SELECT` queries against the `payments` table.

Investigation:

```
-- Check running queries:
SELECT pid, now() - pg_stat_activity.query_start AS duration, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY duration DESC;

-- Found: DELETE FROM users WHERE created_at < now() - INTERVAL '2 years'
-- Duration: 8 minutes each, 50 concurrent

-- Check table/index sizes:
SELECT tablename, pg_size_pretty(pg_total_relation_size(tablename::text))
FROM pg_tables WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(tablename::text) DESC;

-- Check for sequential scans:
SELECT relname, seq_scan, idx_scan
FROM pg_stat_user_tables
ORDER BY seq_scan DESC;
-- payments: seq_scan = 15000, idx_scan = 100
```

Root cause: Nightly user cleanup (DELETE FROM users) caused PostgreSQL to check for related payments. The `payments.user_id` FK had no index → full sequential scan of payments table for *every user deleted*.

Fix:

```
CREATE INDEX CONCURRENTLY ON payments (user_id);
```

Midnight cleanup: 8 minutes → 0.3 seconds.

War Story 2: VACUUM Can't Keep Up

Symptom: Table bloat growing. Queries slowing down. `pg_stat_user_tables` shows `n_dead_tup` in the millions.

Diagnosis:

```
-- Check dead tuples:
SELECT relname, n_live_tup, n_dead_tup,
       round(100.0 * n_dead_tup / nullif(n_live_tup + n_dead_tup, 0), 2)
       last_autovacuum
```

```
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;

-- Check autovacuum settings:
SHOW autovacuum_vacuum_scale_factor;  -- 0.2 = triggers at 20% dead tuple

-- Check if VACUUM is being blocked:
SELECT pid, usename, application_name, state, wait_event_type, wait_event
FROM pg_stat_activity
WHERE wait_event_type = 'Lock';

-- Check oldest transaction (can block VACUUM from removing old versions)
SELECT pid, usename, state,
       age(backend_xid) AS xid_age,
       query
FROM pg_stat_activity
WHERE backend_xid IS NOT NULL
ORDER BY xid_age DESC;
```

Root cause: A long-running reporting query (3-hour analytics job) held an old transaction snapshot. VACUUM couldn't remove dead tuples because they might still be visible to this snapshot.

Fix:

1. Set `statement_timeout = '1h'` for report connections
2. Move heavy reports to a replica
3. Tune autovacuum to run more aggressively on high-churn tables:

```
ALTER TABLE orders SET (
    autovacuum_vacuum_scale_factor = 0.01,  -- 1% instead of 20%
    autovacuum_analyze_scale_factor = 0.005
);
```

War Story 3: The Serialization Anomaly

Symptom: Double-charging customers. Two payments for one order.

Root cause: Race condition in payment processing:

Worker A	Worker B
BEGIN;	BEGIN;

```
SELECT * FROM orders                SELECT * FROM orders
WHERE id=42 AND status='pending';   WHERE id=42 AND status='pending';
→ sees: {status: 'pending'}         → sees: {status: 'pending'}

-- Both workers think they're first!

INSERT INTO payments                INSERT INTO payments
(order_id, status) VALUES           (order_id, status) VALUES
(42, 'processing');                 (42, 'processing');

COMMIT;                             COMMIT;

-- Both committed! Two payments for one order.
```

Fix — use UPDATE as an atomic test-and-set:

```
-- Instead of SELECT then INSERT, use UPDATE with optimistic locking:
UPDATE orders
SET status = 'processing',
    processed_by = $worker_id
WHERE id = 42
  AND status = 'pending'
RETURNING id;

-- Only one worker gets rowcount=1 (the winner)
-- The other gets rowcount=0 and knows to stop
```

This is idiomatic PostgreSQL concurrency control. The UPDATE is atomic — two workers cannot both set the same row to 'processing'.

Final Reference: Production Monitoring Queries

```
-- 1. Currently running queries (anything slow):
SELECT pid, now() - query_start AS duration,
       state, wait_event_type, wait_event,
       left(query, 100) AS query_snippet
FROM pg_stat_activity
WHERE state != 'idle'
  AND query_start < now() - INTERVAL '5 seconds'
ORDER BY duration DESC;
```

```
-- 2. Table bloat:
SELECT relname, pg_size_pretty(pg_relation_size(oid)) AS size,
       n_dead_tup, n_live_tup,
       round(100.0 * n_dead_tup / nullif(n_live_tup + n_dead_tup, 0), 2)
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 20;

-- 3. Index usage (unused indexes waste write performance):
SELECT schemaname, tablename, indexname,
       idx_scan, idx_tup_read,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
ORDER BY idx_scan ASC, pg_relation_size(indexrelid) DESC
LIMIT 20;

-- 4. Longest running transactions (can block VACUUM):
SELECT pid, usename,
       age(backend_xid) AS xid_age,
       now() - xact_start AS duration,
       state, query
FROM pg_stat_activity
WHERE xact_start IS NOT NULL
ORDER BY xact_start ASC
LIMIT 10;

-- 5. Lock waits:
SELECT
    blocked.pid AS blocked_pid,
    blocking.pid AS blocking_pid,
    blocked.query AS blocked_query,
    blocking.query AS blocking_query,
    now() - blocked.query_start AS wait_duration
FROM pg_stat_activity blocked
JOIN pg_stat_activity blocking
    ON blocking.pid = ANY(pg_blocking_pids(blocked.pid))
ORDER BY wait_duration DESC;

-- 6. Cache hit ratio (should be >99%):
SELECT
    sum(heap_blks_read) AS heap_read,
    sum(heap_blks_hit) AS heap_hit,
    round(100.0 * sum(heap_blks_hit) /
          nullif(sum(heap_blks_hit) + sum(heap_blks_read), 0), 2) AS cach
```

```
FROM pg_statio_user_tables;

-- 7. Replication lag:
SELECT client_addr, state,
       pg_size_pretty(pg_wal_lsn_diff(
           sent_lsn, replay_lsn)) AS replication_lag
FROM pg_stat_replication;

-- 8. Top queries by total time (requires pg_stat_statements):
SELECT left(query, 80) AS query,
       calls, total_exec_time::BIGINT AS total_ms,
       mean_exec_time::NUMERIC(10,2) AS avg_ms,
       rows
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 20;
```

Quick Reference: Index Decision Tree

```
What kind of query?
│
├── Equality / Range / Sort on one column?
│   └── B-Tree: CREATE INDEX ON table (column);
│
├── Multiple columns, equality + range?
│   └── Composite B-Tree: CREATE INDEX ON table (eq_col, range_col);
│       (equality columns first!)
│
├── Only query a small subset of rows?
│   └── Partial B-Tree: CREATE INDEX ON table (col) WHERE condition;
│
├── Query only needs indexed columns (no heap lookup)?
│   └── Covering index: CREATE INDEX ON table (col1) INCLUDE (col2, col3)
│
├── JSONB containment / key existence?
│   └── GIN: CREATE INDEX ON table USING GIN (jsonb_col);
│
├── Full-text search?
│   └── GIN on tsvector: CREATE INDEX ON table USING GIN (tsvector_col);
│
├── Array operations (@>, &&, etc.)?
```

```
│   └── GIN: CREATE INDEX ON table USING GIN (array_col);
│
├── Geometric / spatial data?
│   └── GiST: CREATE INDEX ON table USING GiST (point_col);
│
├── Very large table, append-ordered data, range queries?
│   └── BRIN: CREATE INDEX ON table USING BRIN (ts_col);
│
└── Can't lock the table during index build?
    └── Add CONCURRENTLY to any CREATE INDEX statement
```

Glossary

```
ACID              → Atomicity, Consistency, Isolation, Durability
Autovacuum        → Background process that reclaims dead tuples
BRIN              → Block Range Index — tiny index for sorted data
B-Tree            → Balanced Tree — default index type
CTE               → Common Table Expression (WITH clause)
ctid              → Physical tuple location (page, offset)
Dead tuple        → Row version marked deleted, not yet reclaimed
EXPLAIN           → Show query plan without executing
EXPLAIN ANALYZE→ Show query plan + actual execution statistics
FK                → Foreign Key constraint
GIN               → Generalized Inverted Index — for JSONB, arrays, FTS
GiST              → Generalized Search Tree — for geometric/range types
LSN               → Log Sequence Number — WAL position
MVCC              → Multi-Version Concurrency Control
OID               → Object Identifier — internal PostgreSQL ID
PITR              → Point-In-Time Recovery using WAL
RLS               → Row Level Security
RTO               → Recovery Time Objective
RPO               → Recovery Point Objective
Seq Scan          → Sequential table scan (reads every page)
Shared Buffers → PostgreSQL's shared page cache in RAM
SSI               → Serializable Snapshot Isolation
TOAST             → Storage for large field values (>2KB)
Tuple             → A row version in PostgreSQL storage
VACUUM            → Process to reclaim dead tuples
WAL               → Write-Ahead Log — durability mechanism
work_mem          → Per-operation memory for sorts and hash joins
xid               → Transaction ID
```

```
xmax            → Tuple field: transaction that deleted this version
xmin            → Tuple field: transaction that created this version
```

This book is intended as a living reference. Every concept builds on the previous. Re-read the internals sections (MVCC, WAL, planner) after operating a production database for six months — they will make much more sense.