SQL and Database Testing

Query Your Way to Quality

THE MODERN QA ENGINEER SKILLSET

SQL and Database Testing: Query Your Way to Quality

From First SELECT to Schema Migrations, NoSQL, and GDPR Compliance

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

SQL and Database Testing: Query Your Way to Quality

From First SELECT to Schema Migrations, NoSQL, and GDPR Compliance

Book 15 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

About This Book

The UI shows what the user sees. The API shows what the application exchanges. The database shows what actually happened. This book teaches you to verify truth at the source – confirming that data was written correctly, constraints were enforced, and referential integrity was maintained, regardless of what the UI displayed.

SQL is the verification layer that sits beneath every API and UI test. When an API call creates a record, a database query confirms it was persisted correctly. When a bug report says "data is missing," SQL finds the truth. Database testing skills make every other form of testing more reliable.

This textbook covers the complete spectrum of database testing: from writing your first SELECT query to testing schema migrations, from SQL JOINs to NoSQL document stores, from constraint verification to GDPR-compliant data masking. Each chapter builds on the last, and every concept is reinforced with hands-on exercises.

Who This Book Is For

- **QA Engineers** who want to verify data beyond the UI and API layers
- **Software Developers** who need to write reliable database tests
- **SDET (Software Development Engineers in Test)** building automated test frameworks
- **Data Analysts** who want to understand testing practices

- **Students and Career Changers** entering quality assurance or software engineering
- **Technical Leads** who need to establish database testing standards on their teams

Prerequisites

- Basic understanding of what a database is (tables, rows, columns)
- Familiarity with at least one programming language (Python examples are used throughout)
- Access to a PostgreSQL database (local installation or Docker container)
- A text editor and terminal/command-line interface
- Willingness to break things in a test environment (that is how you learn)

How to Use This Book

Each chapter contains:

1. **Conceptual explanations** with real-world analogies
2. **SQL code examples** you can run immediately
3. **Python test code** demonstrating automation patterns
4. **Practice schemas** you can create in your own database
5. **Exercises** at three difficulty levels: Beginner, Intermediate, and Advanced
6. **Pro Tips** highlighting best practices from industry experience
7. **Common Mistakes** callouts warning about frequent pitfalls

Work through the chapters in order for your first reading. After that, use individual chapters as reference material.

Conventions Used

```
Code blocks like this contain SQL or Python code you can execute.
```

> **Pro Tip**: Callouts like this highlight best practices and efficiency techniques.

> **Common Mistake**: Callouts like this warn about frequent errors that cost teams hours of debugging.

Results and expected outputs appear in comments within code blocks:

```sql
SELECT COUNT(*) FROM users;
-- Expected: 42
```

Table of Contents

Part I: SQL Essentials

Chapter 1: Your First SELECT – Reading the Source of Truth

1.1 Why Query the Database Directly?

Every application has layers. A user clicks a button in the UI, which sends an HTTP request to the API, which executes business logic, which writes data to the database. Each layer can introduce bugs. The database is the final layer – the source of truth.

Consider this scenario: a user creates an account through your web application. The UI shows "Account created successfully." The API returned HTTP 201. But did the data actually persist? Without querying the database, you are trusting every layer in between.

```
User Action          UI Response         API Response        Database Reality
-----------          -----------         ------------        ----------------
Click "Register"     "Success!"          201 Created         ???

The database is the only place to verify what ACTUALLY happened.
```

1.2 The Basic SELECT Statement

The SELECT statement retrieves data from one or more tables. It is the most fundamental SQL command and the one you will use most often as a QA engineer.

```
-- The simplest SELECT: get everything from a table
SELECT * FROM users;

-- Select specific columns (preferred for verification)
SELECT id, name, email, role, created_at
FROM users;

-- Select with a WHERE clause to find specific records
SELECT id, name, email, role, created_at
FROM users
WHERE email = 'newuser@test.com';
```

> **Pro Tip**: Avoid `SELECT *` in test automation code. Explicitly list the columns you need. This makes your tests resilient to schema changes – adding a new column will not break your query.

1.3 Verification Queries – The QA Engineer's Bread and Butter

After performing an action through the UI or API, verify the result at the database level. The database is the source of truth – if the UI says "order created" but the database has no record, the data was not actually persisted.

```sql
-- Verify user was stored correctly after API creation
SELECT id, name, email, role, created_at
FROM users
WHERE email = 'newuser@test.com';

-- Verify soft-delete worked (deleted_at should be set, not NULL)
SELECT id, email, deleted_at
FROM users
WHERE email = 'removed@test.com'
  AND deleted_at IS NOT NULL;

-- Verify order total matches line items (returns rows only if inconsistent)
SELECT o.id, o.total, SUM(li.quantity * li.unit_price) AS calculated_total
FROM orders o
JOIN line_items li ON li.order_id = o.id
WHERE o.id = 42
GROUP BY o.id, o.total
HAVING o.total != SUM(li.quantity * li.unit_price);
```

The third query is a pattern worth memorizing: it returns rows only when there is a discrepancy. No rows returned means the data is consistent.

1.4 Why DB Verification Matters

Scenario	API Response	Database Reality	Without DB Check
Caching bug	Returns 201 Created	No row in users table	Bug goes undetected
Race condition	Returns success	Duplicate rows created	Data corruption undetected
Default value bug	Returns user with role=null	Role column is NULL	Missing default undetected
Soft delete bug	Returns 204 Deleted	deleted_at is still NULL	Item appears deleted but is not

1.5 Verify Record Existence

The simplest verification: does this record exist?

```
-- Does this user exist?
SELECT COUNT(*) FROM users WHERE email = 'test@example.com';
-- Expected: 1

-- Does this order have line items?
SELECT COUNT(*) FROM line_items WHERE order_id = 'order-123';
-- Expected: > 0
```

1.6 Verify Default Values

When your application creates a record without specifying every field, the database should fill in sensible defaults.

```
-- Check that defaults are applied on creation
SELECT role, active, email_verified, created_at, updated_at
FROM users
WHERE email = 'newuser@test.com';
-- Expected: role='viewer', active=true, email_verified=false,
-- created_at is recent, updated_at equals created_at
```

1.7 Verify Timestamps

Timestamps are a frequent source of bugs: wrong timezone, not updating on modification, impossible dates.

```
-- Check that updated_at changes on modification
SELECT updated_at > created_at AS was_modified
FROM users
WHERE id = 123;
-- Expected: true (if user was updated)

-- Check that timestamps are reasonable (not in the future, not from 1970)
SELECT *
FROM orders
WHERE created_at > NOW()
   OR created_at < '2020-01-01';
-- Expected: no rows (no invalid timestamps)
```

1.8 Find Data Anomalies

These queries hunt for data integrity issues that indicate bugs in your application.

```
-- Find orphaned records (line items without orders)
SELECT li.id, li.order_id
FROM line_items li
LEFT JOIN orders o ON o.id = li.order_id
WHERE o.id IS NULL;

-- Find duplicate emails (data integrity issue)
SELECT email, COUNT(*)
FROM users
GROUP BY email
HAVING COUNT(*) > 1;

-- Find NULL values in required fields
SELECT id, name, email
FROM users
WHERE name IS NULL OR email IS NULL;
```

1.9 Connecting from Python – Your First Automated DB Test

Connecting to the database from your test code lets you verify persistence directly:

```
import psycopg2
import pytest

@pytest.fixture(scope="session")
def db():
    conn = psycopg2.connect(
        host="localhost",
        dbname="testdb",
        user="testuser",
        password="testpass"
    )
    yield conn
    conn.close()

def test_user_creation_persists(api_client, db):
    """Verify that creating a user via API persists to database."""
    api_client.post("/users", json={
        "name": "Alice",
        "email": "alice@test.com"
```

```
    })

    cursor = db.cursor()
    cursor.execute(
        "SELECT name, email, role FROM users WHERE email = %s",
        ("alice@test.com",)
    )
    row = cursor.fetchone()
    assert row is not None, "User was not persisted to database"
    assert row[0] == "Alice"
    assert row[2] == "viewer"  # Verify default role applied
```

1.10 Using DictCursor for Readable Assertions

Positional access (`row[0]`, `row[1]`) is fragile and hard to read. Use DictCursor instead:

```
from psycopg2.extras import DictCursor

def test_user_fields(db):
    cursor = db.cursor(cursor_factory=DictCursor)
    cursor.execute(
        "SELECT * FROM users WHERE email = %s",
        ("alice@test.com",)
    )
    user = cursor.fetchone()

    assert user["name"] == "Alice"
    assert user["role"] == "viewer"
    assert user["created_at"] is not None

def test_all_active_users_have_email(db):
    cursor = db.cursor(cursor_factory=DictCursor)
    cursor.execute("SELECT id, email FROM users WHERE active = true")
    users = cursor.fetchall()

    for user in users:
        assert user["email"] is not None, f"User {user['id']} has no email"
        assert "@" in user["email"], f"User {user['id']} has invalid email"
```

1.11 Parameterized Queries (Prevent SQL Injection)

Always use parameterized queries in test code:

```
# BAD: string interpolation -- SQL injection vulnerability
cursor.execute(f"SELECT * FROM users WHERE email = '{email}'")

# GOOD: parameterized query -- safe
cursor.execute("SELECT * FROM users WHERE email = %s", (email,))

# GOOD: named parameters
cursor.execute(
    "SELECT * FROM users WHERE email = %(email)s AND role = %(role)s",
    {"email": email, "role": "admin"}
)
```

> **Common Mistake**: Even in test code, never use string interpolation for SQL queries. Test data can contain special characters like single quotes (`O'Brien`) that will break your query. Parameterized queries handle this automatically. It is also a habit that prevents catastrophic mistakes when someone copies your pattern into production code.

Practice Schema 1: User Management

```
CREATE TABLE users (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(100) NOT NULL,
    email       VARCHAR(255) NOT NULL UNIQUE,
    role        VARCHAR(20) DEFAULT 'viewer',
    active      BOOLEAN DEFAULT true,
    email_verified BOOLEAN DEFAULT false,
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW(),
    deleted_at  TIMESTAMP
);

CREATE TABLE user_profiles (
    user_id     UUID PRIMARY KEY REFERENCES users(id),
    bio         TEXT,
    avatar_url  VARCHAR(500),
    phone       VARCHAR(20),
    timezone    VARCHAR(50) DEFAULT 'UTC'
);
```

Exercises

Beginner:

1. Write a SELECT query to find all users with the role 'admin'.
2. Write a query to count how many users have `email_verified = false`.
3. Write a query to find the most recently created user.

Intermediate:

4. Write a query that finds users whose `updated_at` equals `created_at` (never modified).
5. Write a Python test that creates a user via API and verifies the default role in the database.
6. Write a query to find all soft-deleted users (deleted_at IS NOT NULL) created in the last 30 days.

Advanced:

7. Write a verification query that returns rows ONLY if there is an inconsistency between the users table and user_profiles table (every user should have a profile).
8. Write a comprehensive Python test fixture that provides both API client and database cursor, with automatic cleanup.
9. Write a query that detects timestamps in the future or before the year 2020, across all timestamp columns in the users table.

Career Translation

Resume phrasing

- Authored SQL-based verification queries to validate data persistence across UI, API, and database layers, reducing production data bugs by 30%.
- Built automated database test fixtures in Python (psycopg2/pytest) to confirm record creation, default values, and timestamp integrity after every deployment.
- Developed parameterized query patterns for test automation, eliminating SQL injection risks and improving test reliability across 50+ verification scenarios.

Cover letter framing

My approach to quality starts at the source of truth: the database. I have built automated verification suites that go beyond UI and API checks to confirm that data is actually persisted correctly, defaults are applied, and timestamps are sane. This catches an entire class of bugs – caching inconsistencies, race conditions, silent failures – that surface-level testing misses entirely.

Interview framing

"I approach data verification by never trusting the UI or API response alone. After any state-changing operation, I write a SELECT query against the database to confirm persistence, check that defaults were applied, and validate timestamps. I optimize for specificity – explicit column lists rather than SELECT *, parameterized queries rather than string interpolation. The trade-off is that database tests are tightly coupled to the schema, so I keep them resilient by listing only the columns I need and using DictCursor for readable assertions."

What not to say

- "I just check the API response to verify data was saved." – This shows you trust intermediary layers blindly and miss entire categories of bugs.
- "I use SELECT * in all my test queries." – This signals fragile tests that break on any schema change.
- "I build my SQL queries with string concatenation." – This reveals ignorance of SQL injection risks and poor engineering habits.
- "I only test the happy path at the database level." – This misses NULL handling, default values, and data anomaly detection.

Interview Depth Check

Question 1

Prompt: You run an API test that creates a new user and the response is HTTP 201 with a valid JSON body. A week later, a customer reports their account does not exist. Walk me through how you would investigate this at the database level, and how you would prevent it in the future.

What a strong answer should cover:

- Querying the database directly to check if the row exists, was soft-deleted, or was never persisted
- Checking for caching layers that might return a stale 201 while the write actually failed
- Verifying default values (active flag, email_verified) and timestamps (created_at not null)
- Adding a database-level verification step to the automated test suite

Example answer:

- First, I would run `SELECT * FROM users WHERE email = '<customer_email>'` to determine if the row exists at all. If it does not, I would check if there is a deleted_at timestamp or if the record was never written.
- I would then investigate whether the API has a caching layer that might have returned a cached 201 response without actually persisting. I would look at the request logs and error logs tables for that time window.
- To prevent recurrence, I would add a database verification step to every user creation test: after the API returns 201, query the database to confirm the row exists, the default role is applied, and created_at is within an acceptable time window. This becomes a standard pattern across all creation endpoints.

Question 2

Prompt: Your team's test suite uses `SELECT *` throughout all database verification queries. A new migration adds a JSONB column to the users table, and suddenly 40% of your tests fail even though no business logic changed. What happened, and how would you redesign the verification approach?

What a strong answer should cover:

- Understanding that SELECT * returns all columns, so adding a new column changes the result shape
- Positional access (row[0], row[1]) breaks when column order changes
- The fix: use explicit column lists and DictCursor for named access
- Test resilience as a design principle

Example answer:

- The tests break because they use positional indexing (row[0], row[1]) against the results of SELECT *. When the new column is added, the positions shift for any columns that come after it in table order.
- I would refactor every verification query to list columns explicitly: `SELECT id, name, email, role FROM users` instead of `SELECT *`. Then I would switch from positional access to DictCursor so assertions read `user["role"] == "viewer"` instead of `row[3] == "viewer"`.
- Going forward, I would establish a team convention: no SELECT * in test automation, mandatory DictCursor for readability, and a review checklist item for schema resilience.

Question 3

Prompt: You need to verify that a soft-delete feature works correctly. The API returns 204 on DELETE, and the user no longer appears in the GET /users list. How would you verify this at the database level, and what edge cases would you check?

What a strong answer should cover:

- Confirming deleted_at is set to a recent timestamp (not NULL)
- Confirming the row still exists (it is a soft delete, not a hard delete)
- Checking that other fields (name, email) remain intact
- Edge cases: double-delete, re-activation, cascade effects on related records

Example answer:

- I would query `SELECT id, email, deleted_at FROM users WHERE email = '<test_email>'` and assert the row still exists with deleted_at set to a timestamp within the last few seconds.
- I would verify the row is excluded from the application's "active users" query by checking `WHERE deleted_at IS NULL` returns no match, while `WHERE deleted_at IS NOT NULL` does.
- Edge cases I would test: calling DELETE twice on the same user (idempotency), attempting to re-activate a soft-deleted user, and verifying that the user's orders and profiles are still intact (soft delete should not cascade to related records unless the business rules specify otherwise).

Chapter 2: Filtering and Sorting – Finding the Needle in the Haystack

2.1 The WHERE Clause in Depth

The WHERE clause filters rows based on conditions. Think of it as the "search criteria" for your query.

```
-- Equality
SELECT * FROM users WHERE role = 'admin';

-- Inequality
SELECT * FROM users WHERE role != 'admin';
SELECT * FROM users WHERE role <> 'admin';  -- Same thing

-- Comparison operators
SELECT * FROM orders WHERE total > 100.00;
SELECT * FROM orders WHERE total >= 100.00;
SELECT * FROM orders WHERE total < 50.00;
SELECT * FROM orders WHERE total BETWEEN 50.00 AND 100.00;
```

2.2 Combining Conditions with AND, OR, NOT

```sql
-- AND: both conditions must be true
SELECT * FROM users
WHERE role = 'admin' AND active = true;

-- OR: at least one condition must be true
SELECT * FROM users
WHERE role = 'admin' OR role = 'editor';

-- NOT: negation
SELECT * FROM users
WHERE NOT active;

-- Combining with parentheses (order of operations matters!)
SELECT * FROM users
WHERE (role = 'admin' OR role = 'editor')
  AND active = true;
```

> **Common Mistake**: Forgetting parentheses when combining AND and OR. `WHERE role = 'admin' OR role = 'editor' AND active = true` evaluates as `WHERE role = 'admin' OR (role = 'editor' AND active = true)` because AND has higher precedence. Always use parentheses to make your intent explicit.

2.3 Pattern Matching with LIKE and ILIKE

```sql
-- LIKE: case-sensitive pattern matching
SELECT * FROM users WHERE email LIKE '%@gmail.com';
SELECT * FROM users WHERE name LIKE 'John%';
SELECT * FROM users WHERE name LIKE '%son%';

-- ILIKE: case-insensitive (PostgreSQL)
SELECT * FROM users WHERE email ILIKE '%@gmail.com';

-- Wildcard characters:
--   % = any number of characters (including zero)
--   _ = exactly one character
SELECT * FROM users WHERE name LIKE 'J_hn';  -- John, Jahn, J9hn
```

2.4 Working with NULL

NULL is special in SQL. It is not a value – it is the absence of a value. You cannot compare to NULL with =.

```
-- WRONG: this will never return rows, even if deleted_at is NULL
SELECT * FROM users WHERE deleted_at = NULL;

-- CORRECT: use IS NULL / IS NOT NULL
SELECT * FROM users WHERE deleted_at IS NULL;       -- Active users
SELECT * FROM users WHERE deleted_at IS NOT NULL;   -- Soft-deleted users

-- COALESCE: replace NULL with a default
SELECT name, COALESCE(phone, 'No phone') AS phone
FROM users;
```

> **Common Mistake**: Using `= NULL` instead of `IS NULL`. This is one of the most common SQL mistakes. NULL is not equal to anything, not even itself. `NULL = NULL` evaluates to `NULL` (not `true`), so the condition is never satisfied.

2.5 The IN Operator

```
-- Instead of multiple OR conditions
SELECT * FROM users WHERE role IN ('admin', 'editor', 'moderator');

-- NOT IN
SELECT * FROM users WHERE role NOT IN ('viewer', 'guest');

-- IN with subquery (more on this in Chapter 5)
SELECT * FROM users WHERE id IN (
    SELECT user_id FROM orders WHERE total > 1000
);
```

2.6 Sorting with ORDER BY

```
-- Ascending (default)
SELECT * FROM users ORDER BY created_at ASC;

-- Descending
SELECT * FROM users ORDER BY created_at DESC;

-- Multiple sort criteria
SELECT * FROM users ORDER BY role ASC, created_at DESC;

-- Sorting with NULLs
SELECT * FROM users ORDER BY deleted_at NULLS FIRST;
SELECT * FROM users ORDER BY deleted_at NULLS LAST;
```

2.7 Limiting Results with LIMIT and OFFSET

```
-- First 10 results
SELECT * FROM users ORDER BY created_at DESC LIMIT 10;

-- Pagination: page 2, 10 results per page
SELECT * FROM users ORDER BY created_at DESC LIMIT 10 OFFSET 10;

-- Top N pattern
SELECT * FROM orders ORDER BY total DESC LIMIT 5;  -- Top 5 orders by value
```

> **Pro Tip**: When testing pagination in your application, verify it with SQL. Fetch page 1 and page 2 via the API, then run a single SQL query with the same ordering. The combined API results should match the SQL results exactly. This catches off-by-one errors in pagination logic.

Practice Schema 2: E-Commerce Orders

```
CREATE TABLE products (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(200) NOT NULL,
    sku         VARCHAR(50) UNIQUE NOT NULL,
    price       DECIMAL(10,2) NOT NULL CHECK (price >= 0),
    category    VARCHAR(50),
    in_stock    BOOLEAN DEFAULT true,
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE orders (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
    user_id     UUID NOT NULL REFERENCES users(id),
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','confirmed','shipped','delivered','cancelled')),
    total       DECIMAL(10,2) NOT NULL CHECK (total >= 0),
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW(),
    shipped_at  TIMESTAMP,
    delivered_at TIMESTAMP
);

CREATE TABLE line_items (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id    UUID NOT NULL REFERENCES orders(id) ON DELETE CASCADE,
    product_id  UUID NOT NULL REFERENCES products(id),
    quantity    INTEGER NOT NULL CHECK (quantity > 0),
    unit_price  DECIMAL(10,2) NOT NULL CHECK (unit_price >= 0)
);
```

Exercises

Beginner:

1. Write a query to find all products priced between $10 and $50.
2. Write a query to find all orders with status ‘pending’ or ‘confirmed’.
3. Write a query to find the 5 most expensive products.

Intermediate:

4. Write a query to find all orders that have been created but never shipped (shipped_at IS NULL and status is not ‘cancelled’).
5. Write a query to find products whose name contains “wireless” (case-insensitive).
6. Write a query to verify pagination: get page 3 of users (20 per page) sorted by name.

Advanced:

7. Write a query that finds orders where delivered_at is before shipped_at (a data integrity issue).
8. Write a query to find users whose email domain is not in a list of known email providers (gmail.com, outlook.com, yahoo.com).
9. Write a Python test that verifies the API’s sort order matches the database’s sort order for a product listing.

Career Translation

Resume phrasing

- Designed SQL-based data investigation queries using WHERE, LIKE, IN, and ORDER BY to isolate production anomalies, reducing average bug triage time by 40%.
- Built pagination verification tests comparing API responses against direct database queries, catching off-by-one errors before release.
- Identified data integrity issues (impossible timestamps, NULL required fields, invalid status values) through targeted filtering queries across 15+ tables.

Cover letter framing

I use SQL filtering as a diagnostic tool, not just a retrieval mechanism. When a bug report comes in, I write targeted WHERE clauses to scope the problem – how many users are affected, when did it start, what patterns exist. This investigative mindset turns vague bug reports into precise, actionable findings that developers can fix in hours instead of days.

Interview framing

"I approach data investigation by layering filters methodically. I start broad – how many records exist – then narrow with WHERE, BETWEEN, LIKE, and IS NULL to isolate the anomaly. I optimize for precision: ILIKE for case-insensitive searches, COALESCE for NULL handling, explicit NULLS FIRST/LAST in ORDER BY. The trade-off with aggressive filtering is that you might inadvertently exclude relevant data, so I always cross-check my filter criteria against the known bug symptoms."

What not to say

- "I just use equals comparisons in my WHERE clauses." – This misses NULL handling, pattern matching, and range queries entirely.
- "NULL and empty string are the same thing." – This is factually wrong in SQL and exposes a fundamental misunderstanding.
- "I do not test pagination because the framework handles it." – Pagination bugs are among the most common data display issues.

- "I sort results in the application layer, not in SQL." – This signals you do not understand how to verify database-level ordering or compare it against API behavior.

Interview Depth Check

Question 1

Prompt: Your application's product listing page shows products sorted by price ascending, but a customer reports that page 2 of results contains items cheaper than page 1. How would you use SQL to diagnose this, and what are the likely root causes?

What a strong answer should cover:

- Querying the database with the same ORDER BY and LIMIT/OFFSET to compare against the API results
- Understanding that non-deterministic sort order (multiple products with the same price) causes inconsistent pagination
- The fix: adding a secondary sort column (e.g., ORDER BY price ASC, id ASC) for deterministic pagination
- Checking if the API's pagination uses cursor-based or offset-based pagination

Example answer:

- I would run `SELECT id, name, price FROM products ORDER BY price ASC LIMIT 20 OFFSET 0` for page 1 and `LIMIT 20 OFFSET 20` for page 2. If the SQL results are correct but the API results differ, the bug is in the application layer.
- The most likely root cause is non-deterministic ordering: multiple products share the same price, so the database returns them in arbitrary order between requests. Between the page 1 and page 2 API calls, the order may shift.
- The fix is to add a tiebreaker column: `ORDER BY price ASC, id ASC`. I would also recommend the team consider cursor-based pagination for large datasets, as offset-based pagination has known performance and consistency issues.

Question 2

Prompt: You are investigating a bug where some users report their accounts are "deleted" but they can still log in. Describe how you would use NULL filtering and soft-delete queries to diagnose this.

What a strong answer should cover:

- The difference between IS NULL and = NULL in SQL
- Checking whether deleted_at is set or the application's delete logic has a bug
- Examining the login query to see if it filters on deleted_at IS NULL
- Looking for inconsistencies between the application's concept of "deleted" and the database state

Example answer:

- I would run `SELECT id, email, deleted_at, active FROM users WHERE email = '<user_email>'` to see the database state. If deleted_at is NULL, the delete operation never persisted. If deleted_at is set, the login flow is not filtering it out.
- Next, I would check the login query. A common bug is using `WHERE deleted_at = NULL` instead of `WHERE deleted_at IS NULL` – the former never matches because NULL comparisons always return NULL, not true. This means the login query effectively ignores the soft-delete flag.
- I would write a regression test that soft-deletes a user, then verifies that querying with `WHERE deleted_at IS NULL` excludes them. I would also add a data anomaly check: `SELECT COUNT(*) FROM users WHERE deleted_at IS NOT NULL AND active = true` to find users in an inconsistent state.

Question 3

Prompt: A data analyst asks you to help verify that the reporting dashboard's "orders by status" breakdown is accurate. The dashboard shows 500 pending, 1200 confirmed, and 300 shipped orders, but the analyst suspects the numbers are wrong. How would you approach this?

What a strong answer should cover:

- Writing a GROUP BY query with WHERE/IN to count orders by status
- Checking for NULL statuses, unexpected status values, or case sensitivity issues
- Considering whether the dashboard applies additional filters (date range, user type) that the analyst is unaware of
- Comparing the total count across both sources

Example answer:

- I would run `SELECT status, COUNT(*) FROM orders GROUP BY status ORDER BY COUNT(*) DESC` to get the ground truth. If these numbers differ from the dashboard, I would check the dashboard's query or API for additional filters.
- Common causes of mismatch: the dashboard excludes soft-deleted orders, applies a date range filter, or uses a cached query. I would also check `SELECT COUNT(*) FROM orders WHERE status IS NULL` to find orders with no status.
- I would also verify case sensitivity: `SELECT DISTINCT status FROM orders` to see if there are variations like "Pending" vs "pending". Then I would document the exact query the dashboard should use and create a recurring validation test.

Question 4

Prompt: You need to find all orders where the delivered_at timestamp is earlier than the shipped_at timestamp – a data integrity issue. Write the query and explain what additional anomalies you would check for in timestamp data.

What a strong answer should cover:

- The query: `WHERE delivered_at < shipped_at`
- Additional checks: timestamps in the future, timestamps before the company existed, NULL shipped_at with non-NULL delivered_at
- Systematic approach: checking all timestamp pairs for logical ordering

- Using BETWEEN and interval arithmetic for reasonable ranges

Example answer:

- The query is `SELECT id, shipped_at, delivered_at FROM orders WHERE delivered_at IS NOT NULL AND shipped_at IS NOT NULL AND delivered_at < shipped_at`. I include the NULL checks because comparing NULL values yields NULL, not false.
- Beyond that, I would check for: orders with `created_at > NOW()` (future timestamps), orders with `shipped_at < created_at` (shipped before created), and orders with `delivered_at IS NOT NULL AND shipped_at IS NULL` (delivered but never shipped).
- I would build a comprehensive timestamp validation query using CASE WHEN to flag each anomaly type, then run it as a nightly data quality check.

Chapter 3: JOINs – Connecting the Dots Across Tables

3.1 Why JOINs Matter

Real-world data lives in multiple tables. Users are in one table, orders in another, payments in a third. JOINs let you combine data from multiple tables in a single query. For a QA engineer, JOINs are essential for cross-table verification and investigation.

When you find a bug, SQL helps you investigate scope, patterns, and root cause. JOINs combine data across tables. Together with GROUP BY and HAVING, they are your most powerful investigation tools.

3.2 JOIN Types Reference

JOIN Type	Returns	Use Case
`INNER JOIN`	Only matching rows from both tables	Orders with their items
`LEFT JOIN`	All rows from left, matching from right (NULL if none)	Users who may or may not have orders
`RIGHT JOIN`	All rows from right, matching from left (NULL if none)	Less common; same as LEFT JOIN with tables swapped

continued on next page

JOIN Type	Returns	Use Case
`FULL OUTER JOIN`	All rows from both tables	Orphaned records on both sides
`CROSS JOIN`	Every combination of rows	Generating test data combinations

3.3 ASCII Art: Visualizing JOINs

To understand JOINs, picture two tables as circles in a Venn diagram:

```
Table A: users              Table B: orders
+--------+                  +--------+
| Alice  |                  | Ord-1  | (Alice's)
| Bob    |                  | Ord-2  | (Alice's)
| Carol  |                  | Ord-3  | (Dave's - but Dave is not in users!)
+--------+                  +--------+
```

INNER JOIN – Only rows that match in BOTH tables:

```
  users            orders
+--------+       +--------+
|        |       |        |
| Alice ========  Ord-1  |
|       \\========  Ord-2 |
|  Bob   |       |        |
| Carol  |       | Ord-3  |  <-- Dave's order: excluded (no matching user)
+--------+       +--------+

Result: Alice + Ord-1, Alice + Ord-2
Bob and Carol excluded (no orders). Ord-3 excluded (no matching user).
```

LEFT JOIN – All rows from LEFT table, matching from right (NULL if no match):

```
  users            orders
+--------+       +--------+
|        |       |        |
| Alice =========  Ord-1  |
|       \\========  Ord-2 |
| Bob ---+->NULL |        |
| Carol -+->NULL | Ord-3  |  <-- still excluded (no matching user)
+--------+       +--------+

Result: Alice + Ord-1, Alice + Ord-2, Bob + NULL, Carol + NULL
Every user appears. Ord-3 still excluded.
```

RIGHT JOIN – All rows from RIGHT table, matching from left (NULL if no match):

```
  users            orders
+--------+       +--------+
|        |       |        |
| Alice =========  Ord-1  |
|       \\========  Ord-2 |
|  Bob   |  NULL-+-- Ord-3|  <-- Dave's order: included with NULL user
| Carol  |       |        |
+--------+       +--------+

Result: Alice + Ord-1, Alice + Ord-2, NULL + Ord-3
Bob and Carol excluded. Ord-3 included with NULL user info.
```

FULL OUTER JOIN – ALL rows from BOTH tables:

```
  users            orders
+--------+       +--------+
|        |       |        |
| Alice =========  Ord-1  |
|       \\========  Ord-2 |
| Bob ---+->NULL |        |
| Carol -+->NULL | NULL-+-> Ord-3
+--------+       +--------+

Result: Alice+Ord-1, Alice+Ord-2, Bob+NULL, Carol+NULL, NULL+Ord-3
Everyone and everything appears.
```

CROSS JOIN – Every combination (Cartesian product):

```
  users     X     sizes
+--------+     +--------+
| Alice  |     | Small  |
| Bob    |     | Medium |
+--------+     | Large  |
               +--------+

Result: Alice+Small, Alice+Medium, Alice+Large,
        Bob+Small, Bob+Medium, Bob+Large
Total rows = 2 users x 3 sizes = 6 rows
```

3.4 Finding Orphaned Records

Orphaned records are a common data integrity issue. A LEFT JOIN followed by a NULL check reveals them:

```sql
-- Users who registered but never placed an order
SELECT u.id, u.email, u.created_at
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
WHERE o.id IS NULL;

-- Line items without a parent order (data integrity issue)
SELECT li.id, li.order_id, li.product_name
FROM line_items li
LEFT JOIN orders o ON o.id = li.order_id
WHERE o.id IS NULL;

-- Orders without line items (should never happen)
SELECT o.id, o.total, o.created_at
FROM orders o
LEFT JOIN line_items li ON li.order_id = o.id
WHERE li.id IS NULL;
```

> **Pro Tip**: The pattern `LEFT JOIN ... WHERE right_table.id IS NULL` is the standard way to find orphaned records. Memorize it. You will use it constantly.

3.5 Cross-Table Verification

Use JOINs to verify that data is consistent across tables:

```
-- Verify user's order count matches what the profile API shows
SELECT u.id, u.email, COUNT(o.id) AS actual_order_count
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
WHERE u.email = 'alice@test.com'
GROUP BY u.id, u.email;

-- Compare API response "total_spent" with database reality
SELECT u.id, u.email, COALESCE(SUM(o.total), 0) AS actual_total_spent
FROM users u
LEFT JOIN orders o ON o.user_id = u.id AND o.status = 'completed'
WHERE u.email = 'alice@test.com'
GROUP BY u.id, u.email;
```

3.6 Multi-Table Investigation

When investigating a bug, you often need to see data from three, four, or more tables at once:

```
-- Full order details: user, order, items, payment
SELECT
    u.email AS customer,
    o.id AS order_id,
    o.status AS order_status,
    li.product_name,
    li.quantity,
    li.unit_price,
    p.method AS payment_method,
    p.status AS payment_status
FROM orders o
JOIN users u ON u.id = o.user_id
JOIN line_items li ON li.order_id = o.id
LEFT JOIN payments p ON p.order_id = o.id
WHERE o.id = 'order-42';
```

Note the use of `LEFT JOIN` for payments – an order might not have a payment record yet (if it is still pending).

3.7 JOINs in Test Automation

```
def test_order_total_matches_line_items(db, api_client):
    """After creating an order, verify total equals sum of line items."""
    order = api_client.post("/orders", json={
        "items": [
            {"product_id": 1, "quantity": 2},
            {"product_id": 2, "quantity": 1}
        ]
    }).json()

    cursor = db.cursor()
    cursor.execute("""
        SELECT o.total, SUM(li.quantity * li.unit_price) AS calculated
        FROM orders o
        JOIN line_items li ON li.order_id = o.id
        WHERE o.id = %s
        GROUP BY o.total
    """, (order["id"],))
    row = cursor.fetchone()
    assert row is not None
    assert row[0] == row[1], f"Order total {row[0]} != calculated {row[1]}"

def test_no_orphaned_line_items(db):
    """No line items should exist without a parent order."""
    cursor = db.cursor()
    cursor.execute("""
        SELECT COUNT(*)
        FROM line_items li
        LEFT JOIN orders o ON o.id = li.order_id
        WHERE o.id IS NULL
    """)
    orphan_count = cursor.fetchone()[0]
    assert orphan_count == 0, f"Found {orphan_count} orphaned line items"
```

Practice Schema 3: Payments and Refunds

```
CREATE TABLE payments (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id    UUID NOT NULL REFERENCES orders(id),
    method      VARCHAR(20) NOT NULL
                CHECK (method IN ('credit_card','debit_card','paypal','bank_transfer')),
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','completed','failed','refunded')),
    amount      DECIMAL(10,2) NOT NULL CHECK (amount > 0),
    gateway_ref VARCHAR(100),
    created_at  TIMESTAMP DEFAULT NOW(),
    completed_at TIMESTAMP
);

CREATE TABLE refunds (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    payment_id  UUID NOT NULL REFERENCES payments(id),
    amount      DECIMAL(10,2) NOT NULL CHECK (amount > 0),
    reason      TEXT NOT NULL,
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','approved','rejected','completed')),
    created_at  TIMESTAMP DEFAULT NOW(),
    processed_at TIMESTAMP
);
```

Exercises

Beginner:

1. Write an INNER JOIN query that shows each order with its user's email address.
2. Write a LEFT JOIN query that lists all users and their order count (including users with zero orders).
3. Write a query that finds all orders that do NOT have a payment record.

Intermediate:

4. Write a query that finds line items whose product no longer exists in the products table (orphaned product reference).
5. Write a multi-table query that shows: user email, order ID, order total, payment status, and refund amount (if any).
6. Write a Python test that verifies no orphaned records exist across three related tables.

Advanced:

7. Write a query using FULL OUTER JOIN that finds mismatches between the orders table and the payments table (orders without payments AND payments without orders).
8. Write a CROSS JOIN query that generates all possible combinations of products and discount tiers for testing a pricing matrix.
9. Write a query that finds users who have orders but whose total_spent in the user_profiles table does not match the actual sum of their completed order totals.

Career Translation

Resume phrasing

- Wrote cross-table JOIN queries to detect orphaned records and referential integrity violations, identifying 200+ data inconsistencies before they reached production.
- Automated multi-table verification tests using INNER, LEFT, and FULL OUTER JOINs to validate data consistency across orders, payments, and user tables after every release.
- Designed orphaned-record detection queries adopted as standard CI pipeline checks, preventing data corruption in a system handling 50K+ daily transactions.

Cover letter framing

Cross-table data verification is where most testing strategies fall short. I write JOIN-based queries that catch orphaned records, mismatched totals, and broken references – the subtle data integrity issues that single-table tests miss entirely. This approach has repeatedly caught bugs that would have caused financial discrepancies or broken downstream reporting if they had reached production.

Interview framing

"I approach cross-table verification by thinking about relationships first: which tables reference each other, and what invariants should hold between them. I use LEFT JOINs with NULL checks as my standard orphan

detection pattern, INNER JOINs for consistency verification, and FULL OUTER JOINs when I need to find mismatches on both sides. The trade-off is query complexity – multi-table JOINs can be hard to debug, so I build them incrementally and verify each JOIN produces the expected row count before adding the next."

What not to say

- "I only test one table at a time." – This misses every cross-table data integrity issue.
- "I always use INNER JOIN because it is the simplest." – This shows you do not understand when LEFT, RIGHT, or FULL OUTER JOINs are necessary.
- "Orphaned records are the database team's problem." – This signals a siloed mindset; data integrity is everyone's responsibility.
- "I have never used a FULL OUTER JOIN." – While less common, FULL OUTER JOIN is the right tool for bidirectional mismatch detection.
- "I do not worry about JOIN performance in test queries." – Even test queries need to run efficiently, especially in CI pipelines.

Interview Depth Check

Question 1

Prompt: After a production deployment, you discover that some orders have line items referencing products that no longer exist in the products table. Explain how you would find these orphaned records, determine the scope of the problem, and design a test to prevent recurrence.

What a strong answer should cover:

- Using LEFT JOIN from line_items to products with a NULL check on products.id
- Scoping the problem: how many orders are affected, when did the orphans appear, which products were deleted
- Root cause analysis: missing foreign key constraint, or CASCADE delete that should have been RESTRICT
- Prevention: adding a constraint test and an orphan detection query to the CI pipeline

Example answer:

- I would run `SELECT li.id, li.order_id, li.product_id FROM line_items li LEFT JOIN products p ON p.id = li.product_id WHERE p.id IS NULL` to find all orphaned line items. Then I would JOIN with orders to see which customers are affected and when the orders were placed.
- To scope the timeline, I would add `ORDER BY li.created_at` and look for a cluster – if all orphans appeared after a specific date, it correlates with a deployment.
- The root cause is likely either a missing foreign key constraint or a hard delete of products that should have been a soft delete. I would add two tests: one that verifies the foreign key constraint exists (query pg_constraint), and one that runs the orphan detection query and asserts zero results. Both would run in CI.

Question 2

Prompt: Your e-commerce platform shows "Total Spent" on each user's profile page. A customer complains their total is wrong. How would you use JOINs to investigate the discrepancy between the displayed value and the actual database state?

What a strong answer should cover:

- Joining users to orders and aggregating with SUM to compute the actual total
- Filtering by appropriate order statuses (completed only, excluding cancelled/refunded)
- Comparing the computed total against the stored/cached value
- Checking for edge cases: refunds, partial payments, currency issues

Example answer:

- I would run `SELECT u.email, COALESCE(SUM(o.total), 0) AS actual_total FROM users u LEFT JOIN orders o ON o.user_id = u.id AND o.status = 'completed' WHERE u.email = '<customer_email>' GROUP BY u.email` to compute the real total from the orders table.

- Then I would compare this against whatever the profile page is showing – whether it comes from a cached field, a materialized view, or a real-time query. If the profile uses a stored value in user_profiles.total_spent, I would query that directly and compare.
- I would check for edge cases: are refunded orders still counted? Are there orders with status 'completed' but with a refund payment? I would also verify there are no duplicate orders (same user, same items, within seconds) inflating the total.

Question 3

Prompt: You are asked to write a "data health check" query that runs nightly and reports all referential integrity violations across the orders, payments, and line_items tables. Walk me through your design approach.

What a strong answer should cover:

- Using LEFT JOINs from child tables to parent tables to detect orphans in each direction
- Combining multiple checks into a single query or CTE chain
- Reporting the type of violation, affected record IDs, and count
- Making the output actionable (which team owns the fix, severity level)

Example answer:

- I would write a UNION query that combines three LEFT JOIN orphan checks: line_items without orders, payments without orders, and orders without users. Each SELECT would include a label column like 'orphaned_line_item' to identify the violation type.
- For completeness, I would also check the reverse: orders without any line items (which might be valid but is worth flagging), and orders without payments (which could indicate a checkout flow bug).
- The output would include violation_type, record_id, related_id, and created_at so the on-call team can prioritize. I would add a COUNT summary at the top and set up an alert if the total exceeds zero.

Question 4

Prompt: Explain the difference between using `WHERE o.id IS NULL` after a LEFT JOIN versus using `NOT EXISTS` with a correlated subquery. When would you prefer one over the other?

What a strong answer should cover:

- Both approaches find records in table A that have no match in table B
- LEFT JOIN + IS NULL is intuitive for simple cases and easy to extend with additional columns
- NOT EXISTS can be more efficient for large datasets because the optimizer can short-circuit
- NOT EXISTS handles NULL values in the join column correctly, while NOT IN does not

Example answer:

- `LEFT JOIN ... WHERE right.id IS NULL` is my default pattern because it is readable and lets me include columns from both tables in the result. It works perfectly for orphan detection.
- `NOT EXISTS` is preferable when I only need to know whether a match exists, not what the matching data looks like. The database can short-circuit the subquery as soon as it finds one match, which can be faster for large tables.
- I never use `NOT IN` with a subquery that might contain NULLs, because a single NULL in the subquery result causes the entire NOT IN condition to return no rows. NOT EXISTS does not have this problem.

Chapter 4: GROUP BY and Aggregation – Seeing the Big Picture

4.1 Understanding GROUP BY

GROUP BY collapses rows into groups. Instead of seeing every individual row, you see one row per group. This is essential for understanding patterns, distributions, and anomalies in your data.

```sql
-- Without GROUP BY: every individual order
SELECT user_id, total FROM orders;
-- Returns: 1000 rows (one per order)

-- With GROUP BY: summarized per user
SELECT user_id, COUNT(*) AS order_count, SUM(total) AS total_spent
FROM orders
GROUP BY user_id;
-- Returns: ~200 rows (one per user)
```

4.2 Aggregate Functions

Function	Purpose	Example
`COUNT(*)`	Count all rows	Total orders
`COUNT(column)`	Count non-NULL values	Orders with shipping addresses
`COUNT(DISTINCT column)`	Count unique values	Unique customers who ordered
`SUM(column)`	Sum values	Total revenue
`AVG(column)`	Average value	Average order value
`MIN(column)`	Minimum value	Smallest order
`MAX(column)`	Maximum value	Largest order

```sql
-- Summary statistics for orders
SELECT
    COUNT(*) AS total_orders,
    COUNT(DISTINCT user_id) AS unique_customers,
    ROUND(AVG(total), 2) AS avg_order_value,
    MIN(total) AS smallest_order,
    MAX(total) AS largest_order,
    SUM(total) AS total_revenue
FROM orders
WHERE status = 'completed';
```

4.3 HAVING – Filtering Groups

WHERE filters individual rows before grouping. HAVING filters groups after aggregation.

```
-- WHERE: filter rows before grouping
SELECT user_id, COUNT(*) AS order_count
FROM orders
WHERE status = 'completed'    -- Only count completed orders
GROUP BY user_id;

-- HAVING: filter groups after aggregation
SELECT user_id, COUNT(*) AS order_count
FROM orders
WHERE status = 'completed'
GROUP BY user_id
HAVING COUNT(*) > 5;          -- Only users with more than 5 completed orders
```

> **Common Mistake**: Confusing WHERE and HAVING. Use WHERE to filter rows before they are grouped. Use HAVING to filter groups after aggregation. You cannot use aggregate functions (COUNT, SUM, etc.) in a WHERE clause.

4.4 Detecting Duplicates

Duplicate detection is one of the most valuable GROUP BY patterns for QA:

```
-- Duplicate emails (data integrity issue)
SELECT email, COUNT(*) AS count
FROM users
GROUP BY email
HAVING COUNT(*) > 1;

-- Duplicate orders for the same user within 1 minute (possible double-click bug)
SELECT user_id, COUNT(*) AS order_count, MIN(created_at), MAX(created_at)
FROM orders
WHERE created_at > NOW() - INTERVAL '1 hour'
GROUP BY user_id
HAVING COUNT(*) > 1
   AND MAX(created_at) - MIN(created_at) < INTERVAL '1 minute';
```

The second query is a powerful bug detection tool. If a user has multiple orders within one minute, it likely indicates a double-submission bug in the UI.

4.5 Error Analysis Queries

These queries are invaluable during incident investigation:

```
-- Error distribution in last 24 hours
SELECT error_code, COUNT(*) AS occurrences,
       MIN(created_at) AS first_seen,
       MAX(created_at) AS last_seen
FROM error_logs
WHERE created_at > NOW() - INTERVAL '24 hours'
GROUP BY error_code
ORDER BY occurrences DESC;

-- Errors by hour (find peak error times)
SELECT DATE_TRUNC('hour', created_at) AS hour,
       COUNT(*) AS error_count
FROM error_logs
WHERE created_at > NOW() - INTERVAL '7 days'
GROUP BY DATE_TRUNC('hour', created_at)
ORDER BY hour;

-- Error rate by endpoint
SELECT endpoint, COUNT(*) AS total_requests,
       SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END) AS errors,
       ROUND(100.0 * SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END)
             / COUNT(*), 2) AS error_rate
FROM request_logs
WHERE created_at > NOW() - INTERVAL '24 hours'
GROUP BY endpoint
HAVING SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END) > 0
ORDER BY error_rate DESC;
```

> **Pro Tip**: The CASE WHEN pattern inside SUM is extremely versatile. It lets you count specific conditions within a group. Think of it as a "conditional count" – count only the rows that match your criteria.

4.6 Business Metrics Verification

Use GROUP BY to verify that business metrics displayed in the UI match the database reality:

```
-- Orders per status (verify expected distribution)
SELECT status, COUNT(*) AS count
FROM orders
GROUP BY status
ORDER BY count DESC;

-- Average order value by month
SELECT DATE_TRUNC('month', created_at) AS month,
       COUNT(*) AS order_count,
       ROUND(AVG(total), 2) AS avg_order_value,
       ROUND(SUM(total), 2) AS total_revenue
FROM orders
WHERE status = 'completed'
GROUP BY DATE_TRUNC('month', created_at)
ORDER BY month;

-- Top 10 customers by order count
SELECT u.email, COUNT(o.id) AS order_count, SUM(o.total) AS total_spent
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.status = 'completed'
GROUP BY u.email
ORDER BY order_count DESC
LIMIT 10;
```

4.7 DATE_TRUNC for Time-Based Analysis

DATE_TRUNC is a PostgreSQL function that truncates a timestamp to a specified precision:

```
-- Group by day
SELECT DATE_TRUNC('day', created_at) AS day, COUNT(*) AS signups
FROM users
GROUP BY DATE_TRUNC('day', created_at)
ORDER BY day DESC
LIMIT 30;

-- Group by week
SELECT DATE_TRUNC('week', created_at) AS week, COUNT(*) AS orders
FROM orders
GROUP BY DATE_TRUNC('week', created_at)
ORDER BY week DESC;

-- Group by month
SELECT DATE_TRUNC('month', created_at) AS month, COUNT(*) AS revenue
FROM orders
WHERE status = 'completed'
GROUP BY DATE_TRUNC('month', created_at)
ORDER BY month;
```

Practice Schema 4: Error Logging

```
CREATE TABLE error_logs (
    id          BIGSERIAL PRIMARY KEY,
    error_code  VARCHAR(10) NOT NULL,
    message     TEXT,
    stack_trace TEXT,
    endpoint    VARCHAR(200),
    user_id     UUID REFERENCES users(id),
    severity    VARCHAR(10) CHECK (severity IN ('low','medium','high','critical')),
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE request_logs (
    id          BIGSERIAL PRIMARY KEY,
    method      VARCHAR(10) NOT NULL,
    endpoint    VARCHAR(200) NOT NULL,
    status_code INTEGER NOT NULL,
    response_ms INTEGER,
    user_id     UUID REFERENCES users(id),
    created_at  TIMESTAMP DEFAULT NOW()
);
```

Exercises

Beginner:

1. Write a query to count the number of users in each role.
2. Write a query to find the total revenue from completed orders.
3. Write a query to find the average response time per endpoint from request_logs.

Intermediate:

4. Write a query to find duplicate emails in the users table, showing the count and all IDs involved.
5. Write a query that shows error counts grouped by hour for the last 7 days, identifying the peak error hour.
6. Write a query that computes the error rate (percentage of 5xx responses) for each API endpoint.

Advanced:

7. Write a query that detects possible double-submission bugs: users with multiple orders created within 60 seconds of each other.
8. Write a query that shows month-over-month growth in order count and revenue.
9. Write a Python test that verifies the dashboard API's "orders by status" endpoint matches the GROUP BY query result.

Career Translation

Resume phrasing

- Built aggregation-based data quality queries using GROUP BY, HAVING, and DATE_TRUNC to detect duplicate records and anomalous patterns across 10M+ row datasets.
- Automated business metrics verification by comparing dashboard API responses against direct SQL aggregations, catching 12 reporting discrepancies in a single quarter.
- Designed error-rate analysis queries grouping by endpoint, time window, and severity that became the standard incident investigation toolkit for the QA and SRE teams.

Cover letter framing

Aggregation queries are how I turn raw data into actionable intelligence. Whether it is detecting duplicate records that indicate a double-submission bug, computing error rates by endpoint to prioritize testing effort, or verifying that the revenue dashboard matches the database reality, GROUP BY is my primary tool for understanding the big picture. I have built aggregation-based data quality checks that run in CI and have caught reporting bugs worth six figures in potential revenue impact.

Interview framing

"I approach data analysis by starting with the question: what does the distribution look like? GROUP BY with COUNT, SUM, and AVG gives me the shape of the data. HAVING lets me filter for anomalies – duplicates, outliers, groups that should not exist. DATE_TRUNC gives me time-based patterns. I optimize for clarity: I always alias aggregate columns, order results meaningfully, and use CASE WHEN inside SUM for conditional counting. The trade-off is that aggregation hides individual records, so when I find an anomaly in a group, I always drill back down to the row level."

What not to say

- "I use WHERE to filter aggregated results." – This confuses WHERE and HAVING, which is a fundamental SQL misunderstanding.
- "I do not know the difference between COUNT*() and COUNT(column)." – COUNT()* counts all rows; COUNT(column) counts non-NULL values. Interviewers will test this.
- "I just export data to Excel for analysis." – This signals you cannot do analysis at the SQL level, which is slower and less repeatable.
- "I have never used DATE_TRUNC or time-based grouping." – Time-based analysis is essential for error investigation and trend detection.

Interview Depth Check

Question 1

Prompt: Your e-commerce platform processed 10,000 orders yesterday, but the finance team says the revenue report shows a number that is $15,000 higher than expected. How would you use GROUP BY and aggregation to investigate the discrepancy?

What a strong answer should cover:

- Grouping orders by status to see if cancelled or refunded orders are being incorrectly included
- Using SUM with CASE WHEN to compute revenue for different status categories
- Checking for duplicate orders (same user, same items, within seconds)
- Comparing the query's total against the report's total to isolate where the $15K difference originates

Example answer:

- I would start with `SELECT status, COUNT(*), SUM(total) FROM orders WHERE created_at::date = CURRENT_DATE - 1 GROUP BY status` to see the revenue breakdown by status. If cancelled or refunded orders are being summed, that explains the inflation.
- Next, I would check for duplicates: `SELECT user_id, COUNT(*), SUM(total) FROM orders WHERE created_at::date = CURRENT_DATE`

`- 1 GROUP BY user_id HAVING COUNT(*) > 1 AND MAX(created_at) - MIN(created_at) < INTERVAL '1 minute'`. Duplicate orders from double-clicks would inflate the revenue.

- I would compute the exact expected revenue as `SELECT SUM(total) FROM orders WHERE status IN ('completed', 'shipped', 'delivered') AND created_at::date = CURRENT_DATE - 1` and compare against the reported number. The difference should point directly to the included/excluded statuses.

Question 2

Prompt: You suspect there is a double-submission bug in your checkout flow. Users might be accidentally creating duplicate orders when they click the "Place Order" button multiple times. Design a SQL query to detect this pattern and explain how you would quantify the impact.

What a strong answer should cover:

- Grouping by user_id and filtering with HAVING COUNT(*) > 1
- Adding a time window constraint (orders within 60 seconds of each other)
- Using MIN and MAX timestamps to show the gap between duplicates
- Quantifying impact: number of affected users, total duplicate revenue, affected time period

Example answer:

- Detection query: `SELECT user_id, COUNT(*) AS order_count, MIN(created_at) AS first, MAX(created_at) AS last, SUM(total) AS duplicate_total FROM orders WHERE created_at > NOW() - INTERVAL '30 days' GROUP BY user_id HAVING COUNT(*) > 1 AND MAX(created_at) - MIN(created_at) < INTERVAL '1 minute'`.
- To quantify impact, I would wrap this in a CTE and aggregate: total affected users, total duplicate orders, total duplicate revenue. I would also group by DATE_TRUNC('day', first) to see if the problem started after a specific deployment.
- I would present this to the team with both the detection query and a recommended fix: either idempotency keys on the API or a UI de-

bounce on the button. The query becomes a regression test that runs nightly.

Question 3

Prompt: Your monitoring shows that the error rate for the /api/checkout endpoint spiked from 0.5% to 8% between 2:00 AM and 3:00 AM last night. Write the SQL queries you would use to investigate this incident, and explain your investigation methodology.

What a strong answer should cover:

- Using DATE_TRUNC to group by hour and identify the exact spike window
- Filtering by endpoint and status_code to isolate the specific error types
- Using CASE WHEN inside SUM to compute error rates
- Correlating with other tables (deployments, config changes) to identify root cause

Example answer:

- First, I would confirm the spike: `SELECT DATE_TRUNC('hour', created_at) AS hour, COUNT(*) AS total, SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END) AS errors, ROUND(100.0 * SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END) / COUNT(*), 2) AS error_rate FROM request_logs WHERE endpoint = '/api/checkout' AND created_at > NOW() - INTERVAL '24 hours' GROUP BY hour ORDER BY hour`.
- Then I would drill into the error types during the spike: `SELECT error_code, message, COUNT(*) FROM error_logs WHERE endpoint = '/api/checkout' AND created_at BETWEEN '2025-01-15 02:00' AND '2025-01-15 03:00' GROUP BY error_code, message ORDER BY COUNT(*) DESC`.
- Finally, I would check if the errors affected specific users or were global: `SELECT COUNT(DISTINCT user_id) AS affected_users FROM error_logs WHERE ...`. This tells me whether it was a systemic issue or limited to a subset of users.

Chapter 5: Subqueries and CTEs – Queries Within Queries

5.1 What Are Subqueries?

A subquery is a query nested inside another query. It answers "compared to what?" questions:

```
-- Users who have placed orders above the average order value
SELECT u.email, o.total
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.total > (SELECT AVG(total) FROM orders WHERE status = 'completed');
```

The inner query `(SELECT AVG(total) FROM orders WHERE status = 'completed')` runs first, producing a single number. The outer query then uses that number to filter results.

5.2 Types of Subqueries

Scalar subquery – returns a single value:

```
-- Orders above the average
SELECT * FROM orders
WHERE total > (SELECT AVG(total) FROM orders);
```

List subquery – returns a list of values (used with IN, NOT IN):

```
-- Products that have never been ordered
SELECT p.name
FROM products p
WHERE p.id NOT IN (
    SELECT DISTINCT product_id FROM line_items
);
```

Correlated subquery – references the outer query (runs once per row):

```
-- Most recent order for each user
SELECT u.email, o.id, o.total, o.created_at
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.created_at = (
    SELECT MAX(o2.created_at)
    FROM orders o2
    WHERE o2.user_id = u.id
);
```

> **Common Mistake**: Using NOT IN with a subquery that might return NULL values. If the subquery returns even one NULL, the entire NOT IN condition evaluates to NULL (not true), and you get zero results. Use NOT EXISTS instead, or add `WHERE column IS NOT NULL` to the subquery.

5.3 EXISTS and NOT EXISTS

EXISTS is often more efficient than IN for large datasets, and it handles NULL correctly:

```
-- Users who have at least one order (EXISTS)
SELECT u.email
FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.user_id = u.id
);

-- Users who have never ordered (NOT EXISTS)
SELECT u.email
FROM users u
WHERE NOT EXISTS (
    SELECT 1 FROM orders o WHERE o.user_id = u.id
);

-- Products with at least one review
SELECT p.name
FROM products p
WHERE EXISTS (
    SELECT 1 FROM reviews r WHERE r.product_id = p.id
);
```

5.4 Common Table Expressions (CTEs)

CTEs use the WITH keyword to create named temporary result sets. They make complex queries readable:

```
-- Without CTE (hard to read)
SELECT u.email, order_summary.total_spent
FROM users u
JOIN (
    SELECT user_id, SUM(total) AS total_spent
    FROM orders
    WHERE status = 'completed'
    GROUP BY user_id
) order_summary ON order_summary.user_id = u.id
WHERE order_summary.total_spent > 1000;

-- With CTE (much clearer)
WITH order_summary AS (
    SELECT user_id, SUM(total) AS total_spent
    FROM orders
    WHERE status = 'completed'
    GROUP BY user_id
)
SELECT u.email, os.total_spent
FROM users u
JOIN order_summary os ON os.user_id = u.id
WHERE os.total_spent > 1000;
```

5.5 Multiple CTEs

You can chain multiple CTEs, each building on the previous:

```
WITH active_users AS (
    SELECT id, email
    FROM users
    WHERE active = true AND deleted_at IS NULL
),
user_orders AS (
    SELECT au.email, COUNT(o.id) AS order_count, SUM(o.total) AS total_spent
    FROM active_users au
    JOIN orders o ON o.user_id = au.id
    WHERE o.status = 'completed'
    GROUP BY au.email
),
high_value_users AS (
    SELECT email, order_count, total_spent
    FROM user_orders
    WHERE total_spent > 500
)
SELECT * FROM high_value_users
ORDER BY total_spent DESC;
```

> **Pro Tip**: CTEs are your best friend for complex investigation queries. Break the problem into steps: first find the relevant users, then their orders, then aggregate, then filter. Each CTE is a self-contained, testable step.

5.6 CTEs for Data Validation

CTEs are excellent for building multi-step validation queries:

```
-- Find data inconsistencies across multiple tables
WITH order_totals AS (
    SELECT order_id, SUM(quantity * unit_price) AS calculated_total
    FROM line_items
    GROUP BY order_id
),
mismatched_orders AS (
    SELECT o.id, o.total AS stored_total, ot.calculated_total,
           o.total - ot.calculated_total AS difference
    FROM orders o
    JOIN order_totals ot ON ot.order_id = o.id
    WHERE o.total != ot.calculated_total
)
SELECT mo.*, u.email AS customer_email
FROM mismatched_orders mo
JOIN users u ON u.id = (SELECT user_id FROM orders WHERE id = mo.id)
ORDER BY ABS(mo.difference) DESC;
```

5.7 Recursive CTEs (Advanced)

Recursive CTEs can traverse hierarchical data:

```
-- Find all subcategories of a parent category
WITH RECURSIVE category_tree AS (
    -- Base case: start with the parent
    SELECT id, name, parent_id, 0 AS depth
    FROM categories
    WHERE id = 'electronics'

    UNION ALL

    -- Recursive case: find children
    SELECT c.id, c.name, c.parent_id, ct.depth + 1
    FROM categories c
    JOIN category_tree ct ON c.parent_id = ct.id
)
SELECT * FROM category_tree ORDER BY depth, name;
```

Practice Schema 5: Reviews and Ratings

```
CREATE TABLE reviews (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    product_id  UUID NOT NULL REFERENCES products(id),
    user_id     UUID NOT NULL REFERENCES users(id),
    rating      INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
    title       VARCHAR(200),
    body        TEXT,
    verified    BOOLEAN DEFAULT false,
    created_at  TIMESTAMP DEFAULT NOW(),
    UNIQUE(product_id, user_id)  -- One review per user per product
);

CREATE TABLE categories (
    id          VARCHAR(50) PRIMARY KEY,
    name        VARCHAR(100) NOT NULL,
    parent_id   VARCHAR(50) REFERENCES categories(id),
    sort_order  INTEGER DEFAULT 0
);
```

Exercises

Beginner:

1. Write a subquery to find all users who have placed at least one order.
2. Write a subquery to find products that are more expensive than the average product price.
3. Rewrite a subquery as a CTE.

Intermediate:

4. Write a CTE that calculates each user's total spending, then finds users who spend more than twice the average.
5. Write a NOT EXISTS query to find products that have never been reviewed.
6. Write a multi-CTE query that identifies users with orders but no payments.

Advanced:

7. Write a recursive CTE that traverses a category tree and counts products at each level.
8. Write a CTE-based data validation query that checks for order total mismatches, orphaned line items, and duplicate reviews in a single query.
9. Write a correlated subquery that finds the second-highest order total for each user.

Career Translation

Resume phrasing

- Architected multi-CTE data validation queries that verify order total consistency, referential integrity, and duplicate detection in a single pass across 8 related tables.
- Replaced ad-hoc investigation scripts with reusable CTE-based validation pipelines, reducing data quality investigation time from hours to minutes.
- Built correlated subquery patterns to detect edge cases (second-highest values, missing relationships, threshold violations) that surface-level tests consistently missed.

Cover letter framing

Complex data validation requires breaking problems into manageable steps. I use CTEs to build layered validation queries where each step is self-contained and testable – first identify the relevant records, then aggregate, then compare against expected values. This systematic approach has let me build single-query data health checks that would otherwise require multiple scripts and manual correlation.

Interview framing

"I approach complex queries by decomposing them into CTEs – named, self-contained steps that I can test individually before combining. For validation, I typically chain three CTEs: one to compute the expected state, one to fetch the actual state, and one to compare them. I use EXISTS over

IN when NULL safety matters, and correlated subqueries when I need per-row comparisons. The trade-off is that CTEs can be slower than a single optimized query in some databases, but the readability and maintainability are worth it for test code."

What not to say

- "I write one giant query with multiple nested subqueries." – This signals unreadable, unmaintainable code.
- "I do not know the difference between a CTE and a subquery." – CTEs are named and reusable; subqueries are inline and anonymous.
- "I avoid subqueries because they are slow." – Blanket performance assumptions without understanding the query planner.
- "I have never used EXISTS or NOT EXISTS." – These are essential for NULL-safe existence checks.
- "I copy-paste the same subquery multiple times in a query." – This is exactly what CTEs solve.

Interview Depth Check

Question 1

Prompt: You need to build a single query that identifies all data integrity issues across an order management system: orders where the total does not match the sum of line items, line items referencing non-existent products, and users with duplicate email addresses. How would you structure this using CTEs?

What a strong answer should cover:

- Using separate CTEs for each validation check
- A final UNION ALL to combine all issues into a single result set with a "violation_type" label
- Understanding that CTEs make each check independently testable
- Handling NULL values and edge cases (orders with no line items)

Example answer:

- I would write three CTEs: `mismatched_totals` joins orders to line_items and compares SUM(quantity * unit_price) against orders.total using HAVING to surface mismatches. `orphaned_items` does a LEFT JOIN from line_items to products with WHERE products.id IS NULL. `duplicate_emails` groups users by email with HAVING COUNT(*) > 1.
- The final SELECT uses UNION ALL: `SELECT 'total_mismatch' AS type, order_id AS id FROM mismatched_totals UNION ALL SELECT 'orphaned_item', line_item_id FROM orphaned_items UNION ALL SELECT 'duplicate_email', email FROM duplicate_emails`.
- Each CTE can be run independently during debugging. The UNION ALL result gives a unified view of all violations. I would schedule this as a nightly data health check.

Question 2

Prompt: A product manager asks: "Which customers are in the top 10% by spending but have never left a review?" Explain how you would build this query step by step, and what each CTE does.

What a strong answer should cover:

- First CTE: calculate total spending per user from completed orders
- Second CTE: determine the 90th percentile spending threshold
- Third CTE or final query: filter for users above the threshold who do not exist in the reviews table
- Using NOT EXISTS rather than NOT IN for NULL safety

Example answer:

- CTE 1 (`user_spending`): `SELECT user_id, SUM(total) AS total_spent FROM orders WHERE status = 'completed' GROUP BY user_id`.
- CTE 2 (`spending_threshold`): `SELECT PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY total_spent) AS p90 FROM user_spending`.
- Final query: `SELECT us.user_id, u.email, us.total_spent FROM user_spending us JOIN users u ON u.id = us.user_id WHERE`

```
us.total_spent >= (SELECT p90 FROM spending_threshold)
AND NOT EXISTS (SELECT 1 FROM reviews r WHERE r.user_id =
us.user_id) ORDER BY us.total_spent DESC.
```

- I use NOT EXISTS instead of NOT IN because if any user_id in reviews is NULL, NOT IN would return zero results. Each CTE is a self-contained, testable step.

Question 3

Prompt: You discover that a correlated subquery in your test suite takes 45 seconds to run on a 500K-row table. The query finds the most recent order for each user. How would you diagnose the performance issue and rewrite the query?

What a strong answer should cover:

- Understanding that correlated subqueries execute once per outer row, leading to O(n ^ 2) behavior
- Rewriting using a CTE with window functions (ROW_NUMBER) as an alternative
- Using DISTINCT ON (PostgreSQL-specific) as another option
- Checking for missing indexes on the join/filter columns

Example answer:

- The correlated subquery `WHERE o.created_at = (SELECT MAX(created_at) FROM orders WHERE user_id = u.id)` runs the inner SELECT for every user row – 500K users means 500K subquery executions.
- Rewrite with a window function: `WITH ranked AS (SELECT *, ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY created_at DESC) AS rn FROM orders) SELECT * FROM ranked WHERE rn = 1`. This scans the orders table once.
- Alternatively, in PostgreSQL: `SELECT DISTINCT ON (user_id) * FROM orders ORDER BY user_id, created_at DESC`. This is the most concise option.
- I would also check for an index on `orders(user_id, created_at DESC)` which would dramatically speed up any of these approaches.

Chapter 6: INSERT, UPDATE, DELETE – Manipulating Test Data

6.1 Why Direct Data Manipulation?

Setting up test data directly in the database is faster than going through the UI or API. INSERT, UPDATE, and DELETE let you create precise test states, manipulate data for edge-case testing, and clean up after test runs. But with power comes responsibility – unsafe data manipulation in test environments can corrupt shared resources.

Method	Time to Create User + 3 Orders	Dependencies
UI clicks	Minutes	Browser, page loads, forms
API calls	Seconds	Auth flow, business logic
Direct SQL	Milliseconds	Database connection only

For complex test scenarios (user with 100 orders, specific date distributions, edge-case data), SQL is orders of magnitude faster.

6.2 INSERT: Creating Test Data

```
-- Create a test user with specific attributes
INSERT INTO users (id, name, email, role, created_at)
VALUES ('test-uuid-001', 'Test User', 'testuser@automation.com', 'admin',
        NOW() - INTERVAL '90 days');

-- Create multiple orders for the test user
INSERT INTO orders (id, user_id, status, total, created_at) VALUES
    ('order-001', 'test-uuid-001', 'completed', 99.99,
     NOW() - INTERVAL '30 days'),
    ('order-002', 'test-uuid-001', 'pending',  149.50,
     NOW() - INTERVAL '1 day'),
    ('order-003', 'test-uuid-001', 'cancelled', 75.00,
     NOW() - INTERVAL '15 days');

-- Create line items for an order
INSERT INTO line_items (order_id, product_id, quantity, unit_price) VALUES
    ('order-001', 'prod-a', 2, 25.00),
    ('order-001', 'prod-b', 1, 49.99);
```

6.3 INSERT with RETURNING

PostgreSQL's RETURNING clause lets you see what was inserted (especially useful when the database generates IDs or default values):

```
-- Get the generated UUID back
INSERT INTO users (name, email)
VALUES ('New User', 'new@test.com')
RETURNING id, role, created_at;

-- Result: id='<generated-uuid>', role='viewer', created_at='2025-01-15 10:30:00'
```

6.4 UPDATE: Modifying Test State

```
-- Set a user's password to expired (test expired password flow)
UPDATE users
SET password_expires_at = NOW() - INTERVAL '1 day'
WHERE email = 'testuser@automation.com';

-- Set an order to a specific status (test status-dependent logic)
UPDATE orders
SET status = 'shipped', shipped_at = NOW() - INTERVAL '2 hours'
WHERE id = 'order-001';

-- Simulate a user who has been inactive for 6 months
UPDATE users
SET last_login_at = NOW() - INTERVAL '6 months'
WHERE email = 'testuser@automation.com';
```

> **Common Mistake**: Running an UPDATE without a WHERE clause. `UPDATE users SET role = 'admin'` will set EVERY user's role to admin. Always double-check your WHERE clause before executing. In a shared test environment, this can break other people's tests.

6.5 DELETE: Cleaning Up

```
-- Cleanup in reverse dependency order (child records first)
DELETE FROM line_items WHERE order_id IN (
    SELECT id FROM orders WHERE user_id = 'test-uuid-001'
);
DELETE FROM orders WHERE user_id = 'test-uuid-001';
DELETE FROM users WHERE id = 'test-uuid-001';
```

6.6 Why Dependency Order Matters

Foreign key constraints prevent deleting a parent record while child records exist. If you try to delete a user who has orders, the database will reject the deletion. Always delete in reverse dependency order: line_items first, then orders, then users.

```
Dependency Chain:
users  <--  orders  <--  line_items  <--  (nothing)

Deletion Order (reverse):
line_items  -->  orders  -->  users
```

> **Pro Tip**: If your foreign keys are defined with `ON DELETE CASCADE`, deleting the parent automatically deletes all children. But never assume this – verify the cascade behavior is set up correctly (see Chapter 8: Constraints).

6.7 Bulk Data Generation

For performance testing or large data set scenarios:

```
-- Generate 1000 test users
INSERT INTO users (id, name, email, role, created_at)
SELECT
    'perf-test-' || generate_series AS id,
    'Perf User ' || generate_series AS name,
    'perftest' || generate_series || '@test.com' AS email,
    CASE WHEN generate_series % 3 = 0 THEN 'admin'
         WHEN generate_series % 3 = 1 THEN 'editor'
         ELSE 'viewer' END AS role,
    NOW() - (generate_series || ' days')::INTERVAL AS created_at
FROM generate_series(1, 1000);
```

6.8 Identifiable Test Data

Always use prefixes that make test data easy to find and clean up:

```
-- Prefix all test data for easy identification and cleanup
INSERT INTO users (id, name, email) VALUES
    ('test-auto-001', 'AUTO Test User 1', 'auto-test-1@automation.test'),
    ('test-auto-002', 'AUTO Test User 2', 'auto-test-2@automation.test');

-- Emergency cleanup: find and remove all automation test data
DELETE FROM orders WHERE user_id LIKE 'test-auto-%';
DELETE FROM users WHERE id LIKE 'test-auto-%';
-- OR
DELETE FROM users WHERE email LIKE '%@automation.test';
```

6.9 Safe Practices Summary

Practice	Why
Use transactions with rollback	`BEGIN; ... ROLLBACK;` – test data never persists
Use identifiable prefixes	`test-*`, `automation-*` – easy to find and clean up
Never run destructive SQL against production	Use read-only credentials for prod verification
Clean up in reverse dependency order	Avoid foreign key constraint violations
Use unique identifiers	Prevent collisions between parallel test runs

Practice Schema 6: Inventory Management

```
CREATE TABLE warehouses (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(100) NOT NULL,
    location    VARCHAR(200) NOT NULL,
    capacity    INTEGER NOT NULL CHECK (capacity > 0)
);

CREATE TABLE inventory (
    id           UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    warehouse_id UUID NOT NULL REFERENCES warehouses(id),
    product_id   UUID NOT NULL REFERENCES products(id),
    quantity     INTEGER NOT NULL DEFAULT 0 CHECK (quantity >= 0),
    reorder_level INTEGER DEFAULT 10,
    last_restocked TIMESTAMP,
    UNIQUE(warehouse_id, product_id)
);

CREATE TABLE inventory_transactions (
    id           UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    inventory_id UUID NOT NULL REFERENCES inventory(id),
    change_type  VARCHAR(20) NOT NULL
                 CHECK (change_type IN ('restock','sale','adjustment','return')),
    quantity_change INTEGER NOT NULL,  -- positive for additions, negative for removals
    reference_id VARCHAR(100),         -- order_id, PO number, etc.
    created_at   TIMESTAMP DEFAULT NOW()
);
```

Exercises

Beginner:

1. Write INSERT statements to create a user with 3 orders, each having 2 line items.
2. Write an UPDATE statement that simulates a user having an expired trial account.
3. Write DELETE statements that clean up all test data in the correct dependency order.

Intermediate:

4. Write a bulk data generation script that creates 500 users with random roles distributed as 10% admin, 30% editor, 60% viewer.
5. Write INSERT statements that create an inventory scenario where one product is below its reorder level and another is fully stocked.
6. Write a Python function that generates test data using the Faker library and inserts it into all tables.

Advanced:

7. Write a data generation script that creates a realistic 90-day order history with seasonal patterns (more orders on weekdays, fewer on weekends).
8. Write a comprehensive cleanup function that discovers all tables with test data (using the `test-` prefix) and deletes in the correct dependency order automatically.
9. Write an INSERT…SELECT statement that copies production schema data patterns into a test database with masked values.

Career Translation

Resume phrasing

- Designed test data management strategies using direct SQL INSERT/UPDATE/DELETE operations, reducing test setup time from minutes (API-based) to milliseconds (SQL-based) across 200+ test cases.
- Built reusable bulk data generation scripts using generate_series and identifiable prefixes, enabling performance testing with realistic 100K+ record datasets.
- Established test data cleanup protocols with dependency-aware deletion order, eliminating foreign key constraint failures and reducing test flakiness by 60%.

Cover letter framing

Efficient test data management is the foundation of a reliable test suite. I build SQL-based data setup and teardown that is orders of magnitude faster than API-based approaches, uses identifiable prefixes for safe cleanup, and respects foreign key dependencies. This lets the team run comprehensive tests in seconds rather than minutes, which directly improves development velocity and confidence in deployments.

Interview framing

"I approach test data by choosing the right tool for the job. For unit and integration tests, I set up data directly via SQL because it is faster and bypasses application logic I am not testing. I always use identifiable prefixes like 'test-auto-' so I can find and clean up test data with a single DELETE. I delete in reverse dependency order – children before parents – to avoid constraint violations. For complex scenarios, I use generate_series for bulk generation and RETURNING to capture generated IDs. The trade-off is that direct SQL bypasses application validation, so I use it for setup only, not for testing creation logic."

What not to say

- "I create test data through the UI because it is the most realistic." – This is extremely slow and not scalable for automated testing.
- "I just run DELETE FROM table_name without a WHERE clause to clean up." – This nukes all data, including other people's test data in shared environments.
- "I do not worry about cleanup because we reset the database nightly." – This means tests are not isolated and can interfere with each other during the day.
- "I hardcode IDs and hope they do not collide." – This causes random failures in parallel test execution.

Interview Depth Check

Question 1

Prompt: You are setting up test data for a scenario that requires a user with 50 orders spanning the last 90 days, each with 2-5 line items, and specific status distributions (60% completed, 20% shipped, 20% pending). How would you design this data generation in SQL?

What a strong answer should cover:

- Using generate_series for bulk insertion
- CASE WHEN with modulo arithmetic for status distribution
- Random interval offsets for realistic date distribution
- INSERT with RETURNING to capture generated IDs for child records

- Identifiable prefixes for cleanup

Example answer:

- I would start with the user: `INSERT INTO users (id, name, email) VALUES ('test-bulk-user', 'Bulk Test User', 'bulk@test.com')`.
- For orders: `INSERT INTO orders (id, user_id, status, total, created_at) SELECT 'test-order-' || i, 'test-bulk-user', CASE WHEN i % 5 < 3 THEN 'completed' WHEN i % 5 < 4 THEN 'shipped' ELSE 'pending' END, ROUND((RANDOM() * 200 + 10)::numeric, 2), NOW() - (i * INTERVAL '1.8 days') FROM generate_series(1, 50) AS i`.
- For line items, I would use a cross join with generate_series to create 2-5 items per order, referencing existing product IDs. All IDs are prefixed with 'test-' for cleanup: `DELETE FROM line_items WHERE order_id LIKE 'test-order-%'; DELETE FROM orders WHERE id LIKE 'test-order-%'; DELETE FROM users WHERE id = 'test-bulk-user'`.

Question 2

Prompt: Your test suite runs 500 tests in parallel against a shared test database. Tests are stepping on each other's data – one test deletes a user that another test just created. How would you design a data isolation strategy that allows parallel execution?

What a strong answer should cover:

- Unique prefixes per test run (timestamp or UUID-based)
- Each test operates only on its own prefixed data
- Transaction-based isolation where possible (BEGIN/ROLLBACK)
- Separate schemas or databases for true isolation in critical scenarios

Example answer:

- I would assign each test run a unique prefix using a UUID or timestamp: `test-run-a1b2c3-`. Every INSERT in that test uses this prefix for IDs, emails, and names. Every query includes `WHERE id LIKE 'test-run-a1b2c3-%'` so tests never read or modify each other's data.

- For tests that can use it, I would wrap each test in a transaction with ROLLBACK (the most elegant solution). For tests that go through the API and cannot share a transaction, the prefix strategy is the fallback.
- Cleanup runs `DELETE FROM ... WHERE id LIKE 'test-run-a1b2c3-%'` in reverse dependency order at the end of each test, regardless of pass/fail (using a pytest fixture with yield).

Question 3

Prompt: You accidentally run an UPDATE statement without a WHERE clause in a shared test environment: `UPDATE users SET role = 'admin'`. Every user in the database is now an admin. What are the immediate steps you would take, and how would you prevent this from happening again?

What a strong answer should cover:

- Immediate response: check if the operation was in a transaction (ROLLBACK if so)
- If committed: restore from backup or use point-in-time recovery
- Prevention: use transactions for all manual SQL, require WHERE clauses in code reviews
- Tooling: read-only connections by default, LIMIT on UPDATE/DELETE statements

Example answer:

- If the connection is still open and I used BEGIN, I would immediately ROLLBACK. If it was auto-committed, I would check if there is a recent backup or point-in-time recovery available. In PostgreSQL, I would look at the WAL for the original values.
- For prevention, I would implement several safeguards: all test SQL runs inside transactions by default (BEGIN at fixture setup, ROLLBACK at teardown). All UPDATE and DELETE statements in test code must include a WHERE clause that references the test-specific prefix. I would configure the test database user with `SET statement_timeout = '10s'` so runaway queries are killed automatically.
- I would also advocate for a pre-commit hook that scans SQL strings for UPDATE or DELETE without WHERE clauses.

Chapter 7: Transactions – The Safety Net

7.1 What Are Transactions?

Transactions ensure that a series of database operations are atomic – they either all succeed or all fail. There is no partial state. This is critical for test data management.

```
-- Everything succeeds or nothing does
BEGIN;

INSERT INTO users (id, name, email, role)
VALUES ('test-uuid-002', 'Transaction User', 'txn@test.com', 'viewer');

INSERT INTO orders (id, user_id, status, total)
VALUES ('order-txn', 'test-uuid-002', 'pending', 50.00);

-- If either INSERT fails, ROLLBACK undoes everything
COMMIT;
```

7.2 ACID Properties

Property	Meaning	Testing Implication
Atomicity	All or nothing	If one INSERT fails, none persist
Consistency	Data remains valid	Constraints are enforced within the transaction
Isolation	Transactions do not interfere	Parallel tests do not see each other's uncommitted data
Durability	Committed data persists	After COMMIT, data survives a crash

7.3 Transaction-Based Test Isolation

The most powerful pattern for test data management: wrap each test in a transaction and roll it back afterward. The database returns to its original state after every test.

```
@pytest.fixture
def db_transaction(db):
    """Wrap each test in a transaction that rolls back."""
    db.autocommit = False
    yield db
    db.rollback()  # All changes are undone after each test

def test_user_creation(db_transaction, api_client):
    api_client.post("/users", json={
        "name": "Alice", "email": "alice@test.com"
    })

    cursor = db_transaction.cursor()
    cursor.execute("SELECT * FROM users WHERE email = 'alice@test.com'")
    assert cursor.fetchone() is not None
    # After the test, db.rollback() removes the user automatically

def test_another_test(db_transaction):
    # This test starts with a clean database -- no Alice from the previous test
    cursor = db_transaction.cursor()
    cursor.execute("SELECT * FROM users WHERE email = 'alice@test.com'")
    assert cursor.fetchone() is None  # Alice does not exist
```

7.4 Transaction Limitations

Scenario	Transaction Rollback Works?
Test creates data via direct SQL	Yes
Test creates data via API call to same DB	Yes (if API uses same connection)
Test creates data via API call to separate service	No (different connection/ transaction)
Test modifies external systems (S3, email)	No (cannot rollback external side effects)

7.5 Cleanup Fixtures for API Tests

For API tests where rollback is not possible, use cleanup fixtures instead:

```
@pytest.fixture
def create_user(api_client):
    created_ids = []

    def _create(**kwargs):
        r = api_client.post("/users", json=kwargs)
        user = r.json()
        created_ids.append(user["id"])
        return user

    yield _create

    for uid in created_ids:
        api_client.delete(f"/users/{uid}")
```

7.6 Savepoints – Nested Transactions

Savepoints allow partial rollback within a transaction:

```
BEGIN;

INSERT INTO users (id, name, email) VALUES ('sp-user', 'SP User', 'sp@test.com');

SAVEPOINT before_order;

INSERT INTO orders (id, user_id, total)
VALUES ('sp-order', 'sp-user', -50.00);  -- This might fail (negative total)

-- If order fails, rollback to savepoint (user still exists)
ROLLBACK TO SAVEPOINT before_order;

-- User is still inserted, order is not
COMMIT;
```

7.7 Read-Only Production Access

```
@pytest.fixture(scope="session")
def prod_db():
    """Read-only connection to production for verification queries only."""
    conn = psycopg2.connect(
        host=os.environ["PROD_DB_HOST"],
        dbname="production",
        user="readonly_user",
        password=os.environ["PROD_DB_READONLY_PASS"],
        options="-c default_transaction_read_only=on"  # Extra safety
    )
    yield conn
    conn.close()
```

> **Common Mistake**: Using the same database credentials for test setup and production verification. Always use separate credentials. Production connections should use a read-only database user AND the `default_transaction_read_only=on` option for double protection.

Practice Schema 7: Financial Accounts

```
CREATE TABLE accounts (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id     UUID NOT NULL REFERENCES users(id),
    name        VARCHAR(100) NOT NULL,
    balance     DECIMAL(12,2) NOT NULL DEFAULT 0.00,
    currency    VARCHAR(3) NOT NULL DEFAULT 'USD',
    status      VARCHAR(20) DEFAULT 'active'
                CHECK (status IN ('active','frozen','closed')),
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE transfers (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    from_account_id UUID NOT NULL REFERENCES accounts(id),
    to_account_id   UUID NOT NULL REFERENCES accounts(id),
    amount          DECIMAL(12,2) NOT NULL CHECK (amount > 0),
    status          VARCHAR(20) DEFAULT 'pending'
                    CHECK (status IN ('pending','completed','failed','reversed')),
    created_at      TIMESTAMP DEFAULT NOW(),
    completed_at    TIMESTAMP,
    CHECK (from_account_id != to_account_id)
);
```

Exercises

Beginner:

1. Write a BEGIN/COMMIT block that inserts a user and their first order atomically.
2. Write a BEGIN/ROLLBACK block and verify that the data does not persist.
3. Create a pytest fixture that wraps each test in a transaction with automatic rollback.

Intermediate:

4. Write a Python test that uses savepoints to test multiple scenarios within a single transaction.
5. Write a cleanup fixture for API-created data that tracks all created resources and deletes them in reverse order.
6. Write a test that verifies transaction isolation: start two connections, insert data in one, verify it is not visible in the other until committed.

Advanced:

7. Write a fund transfer test that verifies atomicity: if the debit succeeds but the credit fails, neither operation should persist.
8. Write a test that simulates concurrent fund transfers to the same account and verifies the final balance is correct (test for race conditions).
9. Design a hybrid test data strategy that uses transactions for direct SQL tests and API cleanup for integration tests, with a shared pytest fixture hierarchy.

Career Translation

Resume phrasing

- Implemented transaction-based test isolation (BEGIN/ROLLBACK) that eliminated test data pollution and reduced test suite flakiness by 75% across 400+ database tests.

- Designed a hybrid test data strategy using transactions for SQL-level tests and API cleanup fixtures for integration tests, ensuring zero data leakage between test runs.
- Established production database access protocols using read-only credentials and default_transaction_read_only, preventing accidental writes during live verification.

Cover letter framing

Test isolation is the difference between a test suite you trust and one you restart when it fails. I use transaction-based isolation – wrapping each test in BEGIN/ROLLBACK – so every test starts with a clean database without expensive setup or teardown. For integration tests that cannot use transactions, I design cleanup fixtures that track and remove all created resources. This approach has eliminated an entire class of test failures caused by data pollution.

Interview framing

"I approach test isolation by defaulting to transaction rollback: wrap each test in BEGIN, run the test, ROLLBACK. The database returns to its original state in milliseconds. For API integration tests where the test and the application use different connections, I fall back to cleanup fixtures that track created resources and delete them in reverse dependency order. I always use read-only credentials for production verification with the extra safety of default_transaction_read_only. The trade-off is that transaction rollback only works when the test and the database share a connection – cross-service scenarios require explicit cleanup."

What not to say

- "I just truncate all tables between tests." – This is slow and destroys data that other parallel tests might need.
- "I do not worry about test isolation because tests run sequentially." – Sequential execution does not prevent data pollution between tests.
- "I use the same database credentials for test setup and production verification." – This is a serious security risk.

- "I have never used savepoints." – Savepoints enable partial rollback, which is essential for testing error scenarios within a transaction.
- "Transactions are just for the application, not for tests." – This misses one of the most powerful test data management patterns.

Interview Depth Check

Question 1

Prompt: Your test suite has 500 tests that each create data via the API. The suite takes 45 minutes to run because each test cleans up its data via DELETE statements. How would you redesign the data management strategy to run the suite in under 10 minutes?

What a strong answer should cover:

- Switching from DELETE-based cleanup to transaction-based rollback for tests that can use it
- Identifying which tests can share a database connection (direct SQL tests) vs. which need API cleanup
- Using parallel execution with data isolation (unique prefixes per worker)
- Database-level optimizations: in-memory test database, unlogged tables

Example answer:

- I would categorize tests into two groups: those that only verify via SQL (can use transactions) and those that test through the API (need explicit cleanup). For the first group, I would wrap each test in a transaction with ROLLBACK – cleanup becomes instantaneous.
- For API tests, I would switch from individual DELETE statements to batch cleanup at the end of each test class rather than each test. I would use identifiable prefixes and delete all test data in a single pass: `DELETE FROM orders WHERE user_id LIKE 'test-%'`.
- For parallel execution, each worker gets a unique prefix. I would also consider running the test database with `fsync = off` and using `UNLOGGED` tables for test schemas, since durability does not matter in test environments. Combined, this typically achieves a 5-10x speedup.

Question 2

Prompt: You are testing a fund transfer feature. The transfer involves debiting one account, crediting another, and creating a transfer record – all in a single transaction. How would you test that this transaction is truly atomic: that if any step fails, none of the changes persist?

What a strong answer should cover:

- Setting up two accounts with known balances
- Triggering a failure in the middle of the transaction (e.g., insufficient funds, constraint violation)
- Verifying that both account balances remain unchanged after the failure
- Verifying that no transfer record was created
- Testing the success case to confirm all three changes persist together

Example answer:

- I would set up Account A with $100 and Account B with $50, then attempt to transfer $200 from A to B (exceeding A's balance). The stored procedure or application code should raise an error.
- After the error, I would verify: Account A still has $100 (`SELECT balance FROM accounts WHERE id = 'acct-a'`), Account B still has $50, and no transfer record exists for this transaction.
- For the success case, I would transfer $30 from A to B and verify: A has $70, B has $80, and a transfer record exists with status 'completed'. The key is verifying all three changes persisted, not just one.
- I would also test with savepoints: transfer from A to B succeeds, but a subsequent transfer from B to C fails – the first transfer should persist if they are in separate savepoints, or both should roll back if they are in the same transaction.

Question 3

Prompt: Your team wants to run verification queries against the production database after each deployment. What safeguards would you implement to ensure these queries cannot accidentally modify production data?

What a strong answer should cover:

- Read-only database user with only SELECT privileges
- Connection-level read_only setting as a second layer of protection
- Query timeout to prevent long-running queries from impacting production performance
- Network-level restrictions (VPN, IP allowlist)
- Monitoring and audit logging for production database access

Example answer:

- First, I would create a dedicated read-only database user: `CREATE USER readonly_qa WITH PASSWORD '...' LOGIN; GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly_qa`. No INSERT, UPDATE, or DELETE privileges.
- Second, I would set the connection-level safety: `options="-c default_transaction_read_only=on"` in the connection string. Even if the user had write privileges by mistake, this connection refuses writes.
- Third, I would set `statement_timeout = '30s'` to prevent queries from running long enough to impact production. I would also run all production queries through a read replica rather than the primary.
- Fourth, I would log all production database access in our monitoring system and require two-person approval for any new production verification query.

Part II: Database Testing

Chapter 8: Constraints – The Database's Built-In Test Suite

8.1 Why Constraints Are Your Safety Net

Database constraints are the database's built-in test suite. They enforce data integrity rules at the lowest level – even if the application code has a bug, constraints prevent invalid data from being stored. Testing constraints verifies that the safety net is in place.

"If the database has the constraint, why test it?" Because:

1. **Constraints can be accidentally removed** during migrations
2. **Constraints may not exist** even when you assume they do
3. **Application code may bypass constraints** (e.g., ORM bug, raw SQL)
4. **Constraint behavior varies** (CASCADE vs RESTRICT, case sensitivity)
5. **Documentation** – constraint tests document the data integrity rules

8.2 Types of Constraints

Constraint	Purpose	What to Test
PRIMARY KEY	Unique row identifier	Insert duplicate PK should fail

continued on next page

Constraint	Purpose	What to Test
FOREIGN KEY	References must point to existing rows	Insert with non-existent FK should fail
UNIQUE	No duplicate values	Duplicate email should return constraint violation
NOT NULL	Column must have a value	NULL for required field should fail
CHECK	Custom validation	`CHECK (price >= 0)` – negative price should fail
DEFAULT	Provides value when none specified	Verify default is applied when column is omitted

8.3 Testing Primary Key Constraints

```
def test_primary_key_prevents_duplicates(db):
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
        ("user-001", "First", "first@test.com")
    )
    with pytest.raises(psycopg2.errors.UniqueViolation):
        cursor.execute(
            "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
            ("user-001", "Duplicate", "duplicate@test.com")
        )
```

8.4 Testing Foreign Key Constraints

Foreign keys have different behaviors when the referenced row is deleted. You must test each:

```
def test_foreign_key_prevents_orphans(db):
    """Cannot create an order for a non-existent user."""
    cursor = db.cursor()
    with pytest.raises(psycopg2.errors.ForeignKeyViolation):
        cursor.execute(
            "INSERT INTO orders (user_id, total) VALUES (%s, %s)",
            ("nonexistent-user-id", 99.99)
        )
```

```
def test_foreign_key_cascade_delete(db):
    """When a user is deleted, their orders should also be deleted (if CASCADE)."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
        ("cascade-test", "Cascade User", "cascade@test.com"))
    cursor.execute(
        "INSERT INTO orders (id, user_id, total) VALUES (%s, %s, %s)",
        ("cascade-order", "cascade-test", 50.00))

    cursor.execute("DELETE FROM users WHERE id = %s", ("cascade-test",))

    cursor.execute(
        "SELECT COUNT(*) FROM orders WHERE id = %s", ("cascade-order",))
    assert cursor.fetchone()[0] == 0, "Order should be cascade-deleted with user"

def test_foreign_key_restrict_delete(db):
    """If RESTRICT: deleting a user with orders should fail."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
        ("restrict-test", "Restrict User", "restrict@test.com"))
    cursor.execute(
        "INSERT INTO orders (id, user_id, total) VALUES (%s, %s, %s)",
        ("restrict-order", "restrict-test", 50.00))

    with pytest.raises(psycopg2.errors.ForeignKeyViolation):
        cursor.execute(
            "DELETE FROM users WHERE id = %s", ("restrict-test",))
```

```
Foreign Key Behaviors Visualized:

CASCADE:     Delete parent  -->  Children automatically deleted
             users(Alice)   -->  orders(Alice's orders) GONE

RESTRICT:    Delete parent  -->  ERROR! Children exist
             users(Alice)   -->  BLOCKED (Alice has orders)

SET NULL:    Delete parent  -->  Children's FK set to NULL
             users(Alice)   -->  orders.user_id = NULL

SET DEFAULT: Delete parent  -->  Children's FK set to default value
             users(Alice)   -->  orders.user_id = <default>
```

8.5 Testing Unique Constraints

```
def test_unique_email_constraint(db):
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (name, email) VALUES (%s, %s)",
        ("User A", "dup@test.com")
    )
    with pytest.raises(psycopg2.errors.UniqueViolation):
        cursor.execute(
            "INSERT INTO users (name, email) VALUES (%s, %s)",
            ("User B", "dup@test.com")
        )

def test_unique_constraint_case_sensitivity(db):
    """Test whether the unique constraint is case-sensitive."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (name, email) VALUES (%s, %s)",
        ("User A", "test@example.com"))
    try:
        cursor.execute(
            "INSERT INTO users (name, email) VALUES (%s, %s)",
            ("User B", "TEST@EXAMPLE.COM"))
        # If this succeeds, the constraint is case-sensitive
        # (this may or may not be desired behavior)
    except psycopg2.errors.UniqueViolation:
        # Constraint is case-insensitive -- usually desired for email
        pass
```

> **Pro Tip**: For emails, you almost always want case-insensitive uniqueness. If your unique constraint is case-sensitive, users could create accounts with "John@Test.com" and "john@test.com" – two accounts for the same person. Test this explicitly.

8.6 Testing NOT NULL Constraints

```
def test_not_null_email(db):
    cursor = db.cursor()
    with pytest.raises(psycopg2.errors.NotNullViolation):
        cursor.execute(
            "INSERT INTO users (name, email) VALUES (%s, %s)",
            ("No Email User", None)
        )

def test_not_null_name(db):
    cursor = db.cursor()
    with pytest.raises(psycopg2.errors.NotNullViolation):
        cursor.execute(
            "INSERT INTO users (name, email) VALUES (%s, %s)",
            (None, "valid@test.com")
        )
```

8.7 Testing CHECK Constraints

```
def test_check_price_non_negative(db):
    """Price must be >= 0."""
    cursor = db.cursor()
    with pytest.raises(psycopg2.errors.CheckViolation):
        cursor.execute(
            "INSERT INTO products (name, price) VALUES (%s, %s)",
            ("Bad Product", -10.00)
        )

def test_check_allows_zero_price(db):
    """Zero price should be allowed (free products)."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO products (name, price) VALUES (%s, %s)",
        ("Free Product", 0.00)
    )
    # Should succeed without error

def test_check_status_enum(db):
    """Status must be one of the allowed values."""
    cursor = db.cursor()
    with pytest.raises(psycopg2.errors.CheckViolation):
        cursor.execute(
            "INSERT INTO orders (user_id, status, total) VALUES (%s, %s, %s)",
            ("user-001", "invalid_status", 50.00)
        )
```

8.8 Testing Default Values

```
def test_default_role_applied(db):
    """When role is not specified, default should be 'viewer'."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (name, email) VALUES (%s, %s) RETURNING role",
        ("Default Role User", "default@test.com")
    )
    role = cursor.fetchone()[0]
    assert role == "viewer"

def test_default_timestamp_applied(db):
    """created_at should default to current time."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO users (name, email) VALUES (%s, %s) RETURNING created_at",
        ("Timestamp User", "timestamp@test.com")
    )
    created_at = cursor.fetchone()[0]
    assert created_at is not None
    from datetime import datetime, timedelta
    assert datetime.now() - created_at < timedelta(seconds=5)
```

8.9 Discovering Existing Constraints

Before writing tests, discover what constraints already exist:

```
-- List all constraints on a table (PostgreSQL)
SELECT con.conname, con.contype, pg_get_constraintdef(con.oid)
FROM pg_constraint con
JOIN pg_class rel ON rel.oid = con.conrelid
WHERE rel.relname = 'users';

-- contype key:
--   p = primary key
--   f = foreign key
--   u = unique
--   c = check
```

Practice Schema 8: Content Management

```
CREATE TABLE authors (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(100) NOT NULL,
    email       VARCHAR(255) NOT NULL UNIQUE,
```

```
    bio         TEXT,
    active      BOOLEAN DEFAULT true,
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE articles (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    author_id   UUID NOT NULL REFERENCES authors(id) ON DELETE RESTRICT,
    title       VARCHAR(300) NOT NULL,
    slug        VARCHAR(300) NOT NULL UNIQUE,
    body        TEXT NOT NULL,
    status      VARCHAR(20) DEFAULT 'draft'
                CHECK (status IN ('draft','review','published','archived')),
    published_at TIMESTAMP,
    word_count  INTEGER GENERATED ALWAYS AS (
                    array_length(string_to_array(body, ' '), 1)
                ) STORED,
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW(),
    CHECK (published_at IS NULL OR status = 'published')
);

CREATE TABLE tags (
    id   UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE article_tags (
    article_id UUID NOT NULL REFERENCES articles(id) ON DELETE CASCADE,
    tag_id     UUID NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
    PRIMARY KEY (article_id, tag_id)
);
```

Exercises

Beginner:

1. Write tests for PRIMARY KEY and UNIQUE constraints on the authors table.
2. Write a test that verifies the default status of a new article is 'draft'.
3. Write a test that verifies you cannot insert an article with an invalid status.

Intermediate:

4. Write tests for both CASCADE and RESTRICT foreign key behaviors in the article/tag system.
5. Write a test that verifies the CHECK constraint: published_at can only be set when status is 'published'.

6. Write a test for the generated column word_count – insert an article and verify the count matches.

Advanced:

7. Write a comprehensive constraint test suite that discovers all constraints on a table dynamically (by querying pg_constraint) and generates test cases for each.
8. Write a test that verifies cascade behavior across three levels: deleting an article cascades to article_tags but NOT to tags themselves.
9. Write a migration test that verifies constraints survive a schema migration (apply migration, check constraints still exist).

Career Translation

Resume phrasing

- Developed comprehensive constraint test suites covering PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK, and DEFAULT constraints across 20+ tables, catching 8 missing constraints before production.
- Automated constraint discovery by querying pg_constraint and generating test cases dynamically, ensuring new constraints are tested without manual test creation.
- Tested CASCADE vs. RESTRICT foreign key behaviors systematically, preventing 3 data loss incidents caused by incorrect cascade configurations.
- Validated case-sensitivity behavior of UNIQUE constraints on email fields, identifying a duplicate account vulnerability that affected 200+ users.

Cover letter framing

I treat database constraints as the application's last line of defense – and I test them as rigorously as application code. My constraint test suites verify that every PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, and CHECK constraint works exactly as specified, including edge cases like case sensitivity and cascade behavior. This approach has caught missing

constraints that would have allowed invalid data into production, and it documents the data integrity rules in executable form.

Interview framing

"I approach constraint testing by first discovering what constraints exist – querying pg_constraint gives me the ground truth. Then I write both positive tests (valid data should be accepted) and negative tests (invalid data should be rejected with the correct error). I pay special attention to foreign key behaviors: CASCADE, RESTRICT, SET NULL – each has different implications and must be tested explicitly. I also test defaults and case sensitivity, because those are the constraints that fail silently. The trade-off is that constraint tests are tightly coupled to the schema, so I use dynamic discovery to generate tests that adapt automatically when constraints change."

What not to say

- "If the database has constraints, I do not need to test them." – Constraints can be accidentally removed by migrations; testing verifies they still exist.
- "I only test that valid data is accepted." – Negative testing (invalid data should be rejected) is equally important.
- "CASCADE and RESTRICT are the same thing." – They are opposite behaviors. Confusing them risks data loss.
- "I do not test default values because they are trivial." – Missing or incorrect defaults are a frequent source of production bugs.

Interview Depth Check

Question 1

Prompt: You join a new team and discover that the orders table has no foreign key constraint on the user_id column – it was supposed to reference users(id) but the constraint was never added. How would you discover this, assess the damage, and safely add the constraint?

What a strong answer should cover:

- Querying pg_constraint or information_schema to verify the constraint is missing
- Running a query to find existing orphaned records (orders with user_ids not in users)
- Cleaning up orphaned data before adding the constraint
- Adding the constraint with NOT VALID first (avoids locking), then VALIDATE separately

Example answer:

- Discovery: `SELECT con.conname FROM pg_constraint con JOIN pg_class rel ON rel.oid = con.conrelid WHERE rel.relname = 'orders' AND con.contype = 'f'`. If no foreign key references users, the constraint is missing.
- Damage assessment: `SELECT COUNT(*) FROM orders o LEFT JOIN users u ON u.id = o.user_id WHERE u.id IS NULL`. This tells me how many orphaned orders exist.
- Cleanup: Depending on business rules, I would either soft-delete the orphaned orders, assign them to a placeholder user, or archive them. I would never silently delete order records.
- Safe addition: `ALTER TABLE orders ADD CONSTRAINT fk_orders_user_id FOREIGN KEY (user_id) REFERENCES users(id) NOT VALID` first (no lock, no validation of existing data), then `ALTER TABLE orders VALIDATE CONSTRAINT fk_orders_user_id` separately. This avoids locking the table during validation.

Question 2

Prompt: Your application allows users to register with email addresses, and the emails should be unique. However, users report they can create accounts with "John@Example.com" and "john@example.com" – two different accounts for the same email. What is happening at the database level, and how would you fix and test it?

What a strong answer should cover:

- The UNIQUE constraint on email is case-sensitive by default in PostgreSQL
- Options: citext extension, a functional unique index on LOWER(email), or application-level normalization
- Testing both the current behavior and the fixed behavior
- Data cleanup for existing duplicates before applying the fix

Example answer:

- The UNIQUE constraint on VARCHAR is case-sensitive. 'John@Example.com' and 'john@example.com' are different values to the database, so both are allowed.
- Fix option 1: change the column type to `citext` (case-insensitive text). Fix option 2: create a unique index on `LOWER(email)`: `CREATE UNIQUE INDEX idx_users_email_lower ON users (LOWER(email))`.
- Before applying the fix, I would find existing duplicates: `SELECT LOWER(email), COUNT(*) FROM users GROUP BY LOWER(email) HAVING COUNT(*) > 1`. These must be merged or deactivated first.
- Test: insert 'test@test.com', then attempt to insert 'TEST@TEST.COM'. The second insert should raise UniqueViolation. I would also test mixed case in the middle: 'Test@Test.COM'.

Question 3

Prompt: A developer adds a CHECK constraint `CHECK (status IN ('pending','active','suspended'))` to the users table. But the migration fails in staging because existing users have status values of 'inactive' and 'banned'. How should this migration be designed to handle existing data, and how would you test it?

What a strong answer should cover:

- The migration needs a data transformation step before adding the constraint
- Mapping old values to new values (e.g., 'inactive' -> 'suspended', 'banned' -> 'suspended')

- Adding the constraint with NOT VALID, then transforming data, then VALIDATE
- Testing that existing data is transformed correctly and the constraint rejects new invalid values

Example answer:

- The migration should have three steps: (1) UPDATE users SET status = 'suspended' WHERE status IN ('inactive', 'banned') to map old values to the new enum, (2) ALTER TABLE users ADD CONSTRAINT check_user_status CHECK (status IN ('pending','active','suspended')) NOT VALID, (3) ALTER TABLE users VALIDATE CONSTRAINT check_user_status.
- Tests: verify the UPDATE changed the correct number of rows (`SELECT COUNT(*) FROM users WHERE status IN ('inactive','banned')` should be 0 after step 1). Verify the constraint exists after step 2. Attempt to INSERT a user with status 'banned' and verify it raises CheckViolation. Verify existing users still have their original data except for the status field.
- I would also test rollback: the rollback script should drop the constraint and reverse the UPDATE (this requires storing the original values, which is why I would use a separate mapping table for production migrations).

Question 4

Prompt: Your team uses ON DELETE CASCADE on the line_items foreign key to orders, but ON DELETE RESTRICT on the orders foreign key to users. A junior developer is confused about why they can delete an order (and its line items disappear) but cannot delete a user who has orders. Explain the difference and how you would test both behaviors.

What a strong answer should cover:

- CASCADE automatically deletes child records when the parent is deleted
- RESTRICT prevents deletion of the parent if any child records exist
- Testing CASCADE: delete parent, verify children are gone

- Testing RESTRICT: attempt to delete parent with children, verify it raises ForeignKeyViolation
- The business rationale for each choice

Example answer:

- CASCADE on line_items means: when an order is deleted, all its line items are automatically removed. The database handles the cleanup. This makes sense because line items have no meaning without their parent order.
- RESTRICT on orders means: you cannot delete a user who has orders. The database blocks the deletion to preserve the order history. This makes sense because deleting a user should not erase their transaction history.
- Test for CASCADE: insert an order with 3 line items, delete the order, verify `SELECT COUNT(*) FROM line_items WHERE order_id = '<order_id>'` returns 0.
- Test for RESTRICT: insert a user, create an order for that user, attempt to delete the user, and assert the deletion raises `psycopg2.errors.ForeignKeyViolation`. Then delete the order first, then delete the user successfully.

Chapter 9: Schema Migrations – Testing the Riskiest Deployment

9.1 What Makes Migrations Risky

Schema migrations are one of the riskiest deployment operations. A bad migration can cause data loss, corruption, or extended downtime. Testing migrations before they reach production is essential – and often overlooked.

Risk	Example	Impact
Data loss	Dropping a column that still has data	Irreversible data destruction

continued on next page

Risk	Example	Impact
Data corruption	Type conversion truncates values	Silent data modification
Downtime	Adding an index on a large table without `CONCURRENTLY`	Table locked for minutes
Failed rollback	Rollback script does not restore original schema	Cannot undo a bad migration
Default value bug	New NOT NULL column without default – existing rows fail	Application errors
Foreign key break	Adding FK to column with orphaned data	Migration fails

9.2 The Migration Testing Checklist

Test	How
Migration applies cleanly	Run on a copy of production schema
Rollback works	Apply, roll back – schema returns to original state
Data is preserved	Insert known data before migration, verify after
Default values	New NOT NULL column must have a default – verify existing rows
Performance	Adding index on large table must not lock for minutes
Application compatibility	New schema works with both old and new application code

9.3 Testing Migrations with Shell Scripts

```
#!/bin/bash
set -euo pipefail

echo "=== Creating test database from production schema ==="
pg_dump --schema-only production_db > schema.sql
createdb migration_test && psql migration_test < schema.sql

echo "=== Loading seed data ==="
psql migration_test < test_seed_data.sql

echo "=== Recording pre-migration state ==="
psql migration_test -c "SELECT COUNT(*) FROM users" > pre_counts.txt
psql migration_test -c "\d users" > pre_schema.txt

echo "=== Applying migration ==="
psql migration_test < migrations/0042_add_phone_column.sql

echo "=== Verifying migration ==="
psql migration_test -c "\d users"
psql migration_test -c "SELECT COUNT(*) FROM users"
psql migration_test -c "SELECT phone FROM users LIMIT 5"

echo "=== Testing rollback ==="
psql migration_test < migrations/0042_rollback.sql

echo "=== Verifying rollback ==="
psql migration_test -c "\d users"
psql migration_test -c "SELECT COUNT(*) FROM users"

echo "=== Cleanup ==="
dropdb migration_test

echo "=== Migration test PASSED ==="
```

9.4 Testing Migrations in Python

```
import subprocess
import psycopg2
import pytest
import time

@pytest.fixture
def migration_db():
    """Create a disposable database for migration testing."""
```

```
    db_name = "migration_test_" + str(int(time.time()))

    subprocess.run(["createdb", db_name], check=True)
    subprocess.run(
        f"pg_dump --schema-only production_db | psql {db_name}",
        shell=True, check=True
    )

    conn = psycopg2.connect(dbname=db_name)
    yield conn, db_name

    conn.close()
    subprocess.run(["dropdb", db_name], check=True)

def test_migration_applies_cleanly(migration_db):
    conn, db_name = migration_db
    subprocess.run(
        f"psql {db_name} < migrations/0042_add_phone_column.sql",
        shell=True, check=True
    )
    cursor = conn.cursor()
    cursor.execute("""
        SELECT column_name, data_type, is_nullable
        FROM information_schema.columns
        WHERE table_name = 'users' AND column_name = 'phone'
    """)
    row = cursor.fetchone()
    assert row is not None, "Phone column was not created"
    assert row[1] == "character varying"

def test_migration_preserves_data(migration_db):
    conn, db_name = migration_db
    cursor = conn.cursor()

    cursor.execute("""
        INSERT INTO users (name, email) VALUES
        ('Alice', 'alice@test.com'),
        ('Bob', 'bob@test.com')
    """)
    conn.commit()

    subprocess.run(
        f"psql {db_name} < migrations/0042_add_phone_column.sql",
        shell=True, check=True
    )

    conn = psycopg2.connect(dbname=db_name)
    cursor = conn.cursor()
    cursor.execute("SELECT name, email FROM users ORDER BY name")
    rows = cursor.fetchall()
    assert len(rows) == 2
    assert rows[0][0] == "Alice"
    assert rows[1][0] == "Bob"

def test_migration_rollback(migration_db):
    conn, db_name = migration_db

    subprocess.run(
        f"psql {db_name} < migrations/0042_add_phone_column.sql",
        shell=True, check=True
    )
    subprocess.run(
        f"psql {db_name} < migrations/0042_rollback.sql",
        shell=True, check=True
    )

    conn = psycopg2.connect(dbname=db_name)
```

```
    cursor = conn.cursor()
    cursor.execute("""
        SELECT column_name FROM information_schema.columns
        WHERE table_name = 'users' AND column_name = 'phone'
    """)
    assert cursor.fetchone() is None, "Phone column should not exist after rollback"

def test_not_null_column_has_default(migration_db):
    conn, db_name = migration_db
    cursor = conn.cursor()

    cursor.execute(
        "INSERT INTO users (name, email) VALUES ('Test', 'test@test.com')")
    conn.commit()

    subprocess.run(
        f"psql {db_name} < migrations/0043_add_status_column.sql",
        shell=True, check=True
    )

    conn = psycopg2.connect(dbname=db_name)
    cursor = conn.cursor()
    cursor.execute(
        "SELECT status FROM users WHERE email = 'test@test.com'")
    status = cursor.fetchone()[0]
    assert status is not None, "Existing rows should have default status value"
    assert status == "active", "Default status should be 'active'"
```

9.5 Performance Considerations

Adding an index to a table with millions of rows can lock the table for minutes:

```
-- BAD: locks the table during index creation
CREATE INDEX idx_users_email ON users (email);

-- GOOD: creates the index without locking (PostgreSQL)
CREATE INDEX CONCURRENTLY idx_users_email ON users (email);
```

```
def test_migration_completes_in_time(migration_db):
    """Migration should complete within acceptable time."""
    conn, db_name = migration_db

    cursor = conn.cursor()
    cursor.execute("""
        INSERT INTO users (name, email)
        SELECT 'User ' || i, 'user' || i || '@test.com'
        FROM generate_series(1, 100000) AS i
    """)
    conn.commit()

    start = time.time()
    subprocess.run(
        f"psql {db_name} < migrations/0042_add_phone_column.sql",
        shell=True, check=True
    )
    duration = time.time() - start

    assert duration < 30, f"Migration took {duration:.1f}s (limit: 30s)"
```

> **Common Mistake**: Not testing migrations with realistic data volumes. A migration that takes milliseconds on a table with 10 rows can take minutes on a table with 10 million rows. Always test with a data volume that approximates production.

9.6 Migration Testing in CI

```
# GitHub Actions migration testing
migration-test:
  runs-on: ubuntu-latest
  services:
    postgres:
      image: postgres:15
      env:
        POSTGRES_PASSWORD: testpass
      ports:
        - 5432:5432
  steps:
    - uses: actions/checkout@v4
    - name: Load production schema
      run: psql -h localhost -U postgres -f schema_dump.sql
    - name: Run migration
      run: psql -h localhost -U postgres -f migrations/latest.sql
    - name: Verify migration
      run: pytest tests/migrations/ -v
    - name: Test rollback
      run: psql -h localhost -U postgres -f migrations/latest_rollback.sql
```

Exercises

Beginner:

1. Write a migration that adds a new nullable column and test that it applies cleanly.
2. Write a rollback script for the migration above and verify the column is removed.
3. Verify that data inserted before the migration still exists after.

Intermediate:

4. Write a migration that adds a NOT NULL column with a default value. Test with existing data.
5. Write a performance test that measures migration time with 100,000 rows.
6. Add migration testing to a CI pipeline configuration.

Advanced:

7. Write a migration that renames a column while preserving all data, indexes, and constraints. Test each.

8. Write a migration that changes a column type (e.g., VARCHAR to TEXT) and verify no data truncation occurs.
9. Design a comprehensive migration test framework that automatically discovers migrations and tests apply/rollback/data-preservation for each.

Career Translation

Resume phrasing

- Established a schema migration testing pipeline in CI that validates apply, rollback, and data preservation for every migration before it reaches staging, eliminating 100% of migration-related production incidents.
- Designed migration performance tests with production-scale data volumes (100K+ rows), catching a 12-minute table lock issue that was reduced to 2 seconds using CREATE INDEX CONCURRENTLY.
- Built automated migration test framework that clones production schema, seeds data, applies migration, verifies data integrity, tests rollback, and tears down – all in under 60 seconds.

Cover letter framing

Schema migrations are the highest-risk deployment operation in most systems. I treat every migration as a potential data loss event and test accordingly: does it apply cleanly? Does rollback restore the original state? Is existing data preserved? Does it complete within acceptable time on production-scale data? My migration testing pipeline has caught column-type truncation bugs, missing defaults on NOT NULL columns, and table-locking index creations – all before they reached production.

Interview framing

"I approach migration testing with a checklist: apply cleanly, rollback cleanly, preserve existing data, complete in acceptable time, and maintain constraint integrity. I test against a copy of the production schema with realistic data volumes, because a migration that works on 10 rows can lock a table with 10 million rows. I use CREATE INDEX CONCURRENTLY instead of CREATE INDEX, add NOT NULL columns with defaults,

and always write a rollback script. The trade-off is that migration tests add 1-2 minutes to the CI pipeline, but one bad migration can cause hours of downtime."

What not to say

- "I test migrations on an empty database." – This misses data preservation issues and performance problems.
- "I do not write rollback scripts because we can always fix forward." – Fix-forward works until it does not. Rollback is your safety net.
- "Migrations are the DBA's responsibility, not QA's." – Migration testing is a collaboration between QA, development, and database teams.
- "I add NOT NULL columns without defaults because the table is small." – The table might be small today but grow tomorrow. Always add defaults.
- "I have never timed a migration." – Performance is a critical migration concern.

Interview Depth Check

Question 1

Prompt: You need to add a NOT NULL column called `phone_verified` (boolean) to a users table with 5 million rows. The team wants to deploy during business hours with zero downtime. Walk me through how you would design, test, and deploy this migration safely.

What a strong answer should cover:

- Adding the column as nullable first, then backfilling, then setting NOT NULL
- Or adding with a DEFAULT value (PostgreSQL 11+ adds NOT NULL with DEFAULT without rewriting the table)
- Testing with production-scale data to verify performance
- Testing that existing rows get the default value
- Rollback plan

Example answer:

- In PostgreSQL 11+, I would use `ALTER TABLE users ADD COLUMN phone_verified BOOLEAN NOT NULL DEFAULT false`. This is a metadata-only operation that does not rewrite the table, so it completes instantly even on 5 million rows.
- I would test this on a clone of production: seed 5 million rows, run the migration, verify it completes in under 1 second, verify all existing rows have `phone_verified = false`, and verify new inserts work with and without specifying the column.
- Rollback: `ALTER TABLE users DROP COLUMN phone_verified`. I would test that the rollback also completes quickly and that existing data is preserved.
- If we were on an older PostgreSQL version, I would split the migration: (1) add nullable column, (2) backfill in batches of 10K rows, (3) add NOT NULL constraint with NOT VALID, (4) validate in a separate step.

Question 2

Prompt: A migration renames the column `email` to `email_address` on the users table. After deploying the migration to staging, the application throws errors because the ORM still references the old column name. How should this migration have been designed, and what tests would have caught this?

What a strong answer should cover:

- The migration should be split into phases: add new column, backfill, update application, drop old column
- Or use a database view/alias for backward compatibility
- Application compatibility testing: old code against new schema and new code against old schema
- Testing that all queries referencing the old column still work during the transition

Example answer:

- This migration should follow the "expand-contract" pattern: Phase 1 (expand) adds `email_address` and a trigger that keeps both columns in sync. Phase 2 updates all application code to use `email_address`. Phase 3 (contract) drops the old `email` column and the sync trigger.
- Tests for Phase 1: verify both columns exist, verify the sync trigger works (updating one column updates the other), verify existing data is copied.
- The compatibility test that would have caught this: deploy the migration, then run the existing test suite without any code changes. If tests fail, the migration is not backward-compatible.
- I would also query `information_schema.columns` before and after to verify the schema change, and run `SELECT email FROM users LIMIT 1` to verify the old column still exists during the transition period.

Question 3

Prompt: Your CI pipeline runs migration tests against an empty database, and they always pass. But in staging (which has 2 years of production-like data), the migration fails because an ALTER TABLE ADD CONSTRAINT CHECK finds existing rows that violate the constraint. What went wrong and how would you fix the testing approach?

What a strong answer should cover:

- Empty database tests miss data-dependent failures entirely
- Migration tests must seed realistic data including edge cases and legacy values
- Using NOT VALID to add constraints without checking existing data, then cleaning up, then VALIDATE
- Testing with a production schema dump (pg_dump –schema-only) plus representative data

Example answer:

- The fundamental problem is that empty database tests verify syntax, not data compatibility. The CHECK constraint passed because there were no rows to violate it. In staging, legacy data had values outside the constraint's allowed range.
- Fix: migration tests should use `pg_dump --schema-only` from production to get the real schema, then seed representative data including known edge cases. For a CHECK constraint on status, I would seed rows with every status value that exists in production, including any legacy values.
- For the migration itself: use `ADD CONSTRAINT ... NOT VALID` to add the constraint without checking existing data, then run an UPDATE to fix legacy values, then `VALIDATE CONSTRAINT` to verify all rows now comply. Each step should be tested independently.
- Going forward, I would add a CI step that dumps production data distributions (`SELECT DISTINCT status, COUNT(*) FROM users GROUP BY status`) and uses them to generate representative test data.

Chapter 10: Stored Procedures and Triggers – Testing Hidden Logic

10.1 Why Business Logic Lives in the Database

Business logic that lives in the database – stored procedures, functions, and triggers – must be tested like any other code. These are particularly common in financial systems, inventory management, and anywhere that data integrity is critical enough to enforce at the database level.

Advantage	Example
Atomicity	Calculate order total and update inventory in a single transaction
Consistency	Ensure constraints are enforced regardless of which application writes data

continued on next page

Advantage	Example
Performance	Aggregate millions of rows without transferring data to the application
Security	Application only calls procedures, never writes raw SQL

The trade-off: database logic is harder to version control, harder to test, and harder to debug than application code. This is exactly why testing it is important.

10.2 Testing Stored Procedures

```
def test_calculate_order_total(db):
    cursor = db.cursor()

    cursor.execute(
        "INSERT INTO orders (id, status) VALUES ('test-order', 'pending')")
    cursor.execute("""
        INSERT INTO line_items (order_id, quantity, unit_price) VALUES
        ('test-order', 2, 10.00),
        ('test-order', 1, 25.50)
    """)

    cursor.execute("SELECT calculate_order_total('test-order')")
    total = cursor.fetchone()[0]

    expected = (2 * 10.00) + (1 * 25.50)  # 45.50
    assert total == expected, f"Expected {expected}, got {total}"
```

10.3 Testing Edge Cases

```
def test_calculate_order_total_empty_order(db):
    """Order with no line items should return 0."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO orders (id, status) VALUES ('empty-order', 'pending')")
    cursor.execute("SELECT calculate_order_total('empty-order')")
    assert cursor.fetchone()[0] == 0

def test_calculate_order_total_nonexistent(db):
    """Non-existent order should raise an error or return NULL."""
    cursor = db.cursor()
    cursor.execute("SELECT calculate_order_total('nonexistent')")
    result = cursor.fetchone()[0]
    assert result is None or result == 0
```

```
def test_calculate_order_total_large_quantities(db):
    """Test with large quantities to check for overflow."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO orders (id, status) VALUES ('big-order', 'pending')")
    cursor.execute("""
        INSERT INTO line_items (order_id, quantity, unit_price) VALUES
        ('big-order', 999999, 9999.99)
    """)
    cursor.execute("SELECT calculate_order_total('big-order')")
    total = cursor.fetchone()[0]
    expected = 999999 * 9999.99
    assert abs(total - expected) < 0.01

def test_calculate_order_total_precision(db):
    """Test decimal precision (common bug with money calculations)."""
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO orders (id, status) VALUES ('precise-order', 'pending')")
    cursor.execute("""
        INSERT INTO line_items (order_id, quantity, unit_price) VALUES
        ('precise-order', 3, 0.10)
    """)
    cursor.execute("SELECT calculate_order_total('precise-order')")
    total = cursor.fetchone()[0]
    # 3 * 0.10 should be exactly 0.30, not 0.30000000000000004
    assert total == 0.30, f"Precision error: expected 0.30, got {total}"
```

> **Pro Tip**: Financial calculations are the number one source of precision bugs. Always use DECIMAL/NUMERIC types for money, never FLOAT or DOUBLE. Test with values that expose floating-point issues: 0.10, 0.01, and values that do not have exact binary representations.

10.4 Testing Triggers

Triggers run automatically when data is inserted, updated, or deleted. They are invisible to the application – which makes them especially important to test.

updated_at Trigger

```
def test_updated_at_trigger(db):
    """Verify that updated_at changes when a row is modified."""
    cursor = db.cursor()

    cursor.execute("""
        INSERT INTO users (id, name, email)
        VALUES ('trigger-test', 'Original Name', 'trigger@test.com')
        RETURNING created_at, updated_at
    """)
    created_at, updated_at = cursor.fetchone()
    assert created_at == updated_at

    import time
    time.sleep(1)
    cursor.execute("""
        UPDATE users SET name = 'Updated Name'
        WHERE id = 'trigger-test'
    """)

    cursor.execute(
        "SELECT updated_at FROM users WHERE id = 'trigger-test'")
    new_updated_at = cursor.fetchone()[0]
    assert new_updated_at > updated_at, \
        "updated_at should be newer after UPDATE"
```

Audit Trail Trigger

```
def test_audit_trail_on_update(db):
    """Verify that changes to users are logged in audit table."""
    cursor = db.cursor()

    cursor.execute("""
        INSERT INTO users (id, name, email, role)
        VALUES ('audit-test', 'Audit User', 'audit@test.com', 'viewer')
    """)

    cursor.execute("""
        UPDATE users SET role = 'admin' WHERE id = 'audit-test'
    """)

    cursor.execute("""
        SELECT old_value, new_value, changed_field, changed_at
        FROM audit_log
        WHERE table_name = 'users' AND record_id = 'audit-test'
```

```
        ORDER BY changed_at DESC LIMIT 1
    """)
    row = cursor.fetchone()
    assert row is not None, "Audit log entry should exist"
    assert row[0] == "viewer"   # old_value
    assert row[1] == "admin"    # new_value
    assert row[2] == "role"     # changed_field

def test_audit_trail_on_delete(db):
    """Verify that deletions are logged."""
    cursor = db.cursor()
    cursor.execute("""
        INSERT INTO users (id, name, email)
        VALUES ('delete-audit', 'Delete Me', 'delete@test.com')
    """)
    cursor.execute("DELETE FROM users WHERE id = 'delete-audit'")

    cursor.execute("""
        SELECT action FROM audit_log
        WHERE table_name = 'users' AND record_id = 'delete-audit'
    """)
    row = cursor.fetchone()
    assert row is not None
    assert row[0] == "DELETE"
```

10.5 Testing Computed and Derived Values

```
def test_user_order_count_computed(db):
    """Verify computed order_count matches actual orders."""
    cursor = db.cursor()

    cursor.execute(
        "INSERT INTO users (id, name, email) "
        "VALUES ('count-test', 'Count User', 'count@test.com')")
    cursor.execute("""
        INSERT INTO orders (user_id, status, total) VALUES
        ('count-test', 'completed', 50.00),
        ('count-test', 'completed', 75.00),
        ('count-test', 'cancelled', 25.00)
    """)

    cursor.execute(
        "SELECT order_count FROM user_stats WHERE user_id = 'count-test'")
    computed_count = cursor.fetchone()[0]

    cursor.execute(
        "SELECT COUNT(*) FROM orders WHERE user_id = 'count-test'")
    actual_count = cursor.fetchone()[0]

    assert computed_count == actual_count
```

10.6 Testing Procedure Error Handling

```
def test_transfer_funds_insufficient_balance(db):
    """Transfer should fail if source account has insufficient funds."""
    cursor = db.cursor()

    cursor.execute("""
        INSERT INTO accounts (id, balance) VALUES
        ('acct-a', 100.00),
        ('acct-b', 50.00)
    """)

    with pytest.raises(Exception):
        cursor.execute(
            "SELECT transfer_funds('acct-a', 'acct-b', 200.00)")

    cursor.execute(
        "SELECT balance FROM accounts WHERE id = 'acct-a'")
    assert cursor.fetchone()[0] == 100.00  # Unchanged
    cursor.execute(
        "SELECT balance FROM accounts WHERE id = 'acct-b'")
    assert cursor.fetchone()[0] == 50.00   # Unchanged
```

Practice Schema 9: Audit and Compliance

```
CREATE TABLE audit_log (
    id           BIGSERIAL PRIMARY KEY,
    table_name   VARCHAR(100) NOT NULL,
    record_id    VARCHAR(100) NOT NULL,
    action       VARCHAR(10) NOT NULL CHECK (action IN ('INSERT','UPDATE','DELETE')),
    changed_field VARCHAR(100),
    old_value    TEXT,
    new_value    TEXT,
    changed_by   UUID REFERENCES users(id),
    changed_at   TIMESTAMP DEFAULT NOW()
);

CREATE TABLE user_stats (
    user_id      UUID PRIMARY KEY REFERENCES users(id),
    order_count  INTEGER DEFAULT 0,
    total_spent  DECIMAL(12,2) DEFAULT 0.00,
    last_order_at TIMESTAMP,
    updated_at   TIMESTAMP DEFAULT NOW()
);

-- Example trigger function
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER set_updated_at
    BEFORE UPDATE ON users
    FOR EACH ROW
    EXECUTE FUNCTION update_timestamp();
```

Exercises

Beginner:

1. Write a test for a `calculate_order_total` function with normal input.
2. Write a test that verifies the `updated_at` trigger fires on UPDATE.
3. Write a test for a stored function with an empty input.

Intermediate:

4. Write tests for an audit trail trigger covering INSERT, UPDATE, and DELETE actions.
5. Write tests for a fund transfer procedure: success case and insufficient balance case.

6. Write a precision test for financial calculations using values like 0.10 and 0.01.

Advanced:

7. Write a comprehensive test suite for a stored procedure that handles inventory changes: restock, sale, adjustment, and return.
8. Write a test that verifies trigger behavior under concurrent modifications (two simultaneous updates to the same row).
9. Write a test for a trigger that prevents deletion of admin users and logs the attempted deletion.

Career Translation

Resume phrasing

- Built test suites for 15+ stored procedures and database triggers, covering edge cases (empty inputs, overflow, precision errors) that application-layer tests missed entirely.
- Validated audit trail triggers across INSERT, UPDATE, and DELETE operations, ensuring compliance with SOX requirements for financial data traceability.
- Identified decimal precision bugs in financial stored procedures by testing with boundary values (0.10, 0.01), preventing a $50K+ calculation error in production billing.

Cover letter framing

Database logic – stored procedures, triggers, computed columns – is some of the hardest code to test because it is invisible to the application layer. I have built test suites that treat database logic as first-class code: testing procedures with normal inputs, edge cases, and error conditions, and verifying that triggers fire correctly on every data modification event. This approach has caught precision bugs in financial calculations and missing audit trail entries that would have been compliance violations.

Interview framing

"I approach database logic testing the same way I approach application code testing: positive cases, negative cases, edge cases, and error handling. For stored procedures, I test with normal input, empty input, NULL input, boundary values, and values that might cause overflow or precision errors. For triggers, I verify they fire on INSERT, UPDATE, and DELETE, and I check that the trigger's side effects are correct (updated_at changes, audit log entry exists). I always use DECIMAL for financial calculations, never FLOAT. The trade-off is that database tests are slower than unit tests, but database logic handles critical business rules that justify the investment."

What not to say

- "I trust stored procedures because the DBA wrote them." – All code needs testing, regardless of who wrote it.
- "I only test the happy path for database functions." – Edge cases in database logic cause the most expensive bugs.
- "I use FLOAT for money calculations in tests." – FLOAT has precision issues that DECIMAL avoids.
- "I do not test triggers because they are automatic." – Automatic means invisible, which makes testing even more important.
- "Audit trails are a compliance team concern, not a testing concern." – QA verifies that the audit trail works correctly.

Interview Depth Check

Question 1

Prompt: Your application uses a trigger that automatically updates the `updated_at` timestamp whenever a row in the users table is modified. A developer reports that the `updated_at` field is not changing when they update a user's profile through the API. How would you diagnose whether the problem is in the trigger, the application, or the ORM?

What a strong answer should cover:

- Testing the trigger directly via SQL UPDATE to isolate the database layer
- Checking if the ORM is bypassing the trigger (some ORMs set updated_at themselves)
- Verifying the trigger exists and is enabled
- Checking if the trigger only fires on specific columns

Example answer:

- First, I would test the trigger directly: `UPDATE users SET name = 'Changed' WHERE id = '<test_id>'`, then `SELECT updated_at FROM users WHERE id = '<test_id>'`. If updated_at changed, the trigger works and the problem is in the application layer.
- Next, I would check if the ORM is interfering. Some ORMs explicitly set updated_at in their UPDATE statements, which means the trigger fires but the ORM's value overwrites it. I would enable query logging to see the exact SQL the ORM generates.
- I would also verify the trigger exists: `SELECT tgname, tgenabled FROM pg_trigger WHERE tgrelid = 'users'::regclass`. A trigger with tgenabled = 'D' is disabled.
- If the API updates the user_profiles table (not the users table), the trigger would not fire because it is attached to users, not user_profiles. This is a common oversight.

Question 2

Prompt: You are testing a `transfer_funds` stored procedure that debits one account and credits another. The procedure should reject transfers when the source account has insufficient funds. How would you design a comprehensive test suite for this procedure?

What a strong answer should cover:

- Success case: valid transfer updates both balances and creates a transfer record
- Insufficient funds: balances remain unchanged, transfer record shows 'failed'

- Edge cases: transfer to same account, zero amount, negative amount, exact balance transfer
- Precision testing: amounts like 0.10, 0.01 that expose floating-point issues
- Concurrency: two simultaneous transfers from the same account

Example answer:

- Success case: Account A has $100, Account B has $50. Transfer $30. Verify A has $70, B has $80, transfer record has status 'completed' and correct amount.
- Insufficient funds: Account A has $100. Transfer $200. Verify A still has $100, B still has $50, and the procedure raises an exception. Verify no transfer record with status 'completed' exists.
- Boundary case: Transfer exactly $100 from an account with $100. This should succeed, leaving balance at $0. Then attempt another $1 transfer – should fail.
- Precision test: Transfer $0.10 three times. Final balance should decrease by exactly $0.30, not $0.30000000000000004.
- Same-account transfer: The CHECK constraint `from_account_id != to_account_id` should prevent this, but I would test it explicitly.

Question 3

Prompt: Your team has implemented an audit trail trigger that logs all changes to the users table into an audit_log table. After a compliance audit, you discover that DELETE operations are not being logged. How would you investigate this and design tests to prevent recurrence?

What a strong answer should cover:

- Checking the trigger definition: is it set to fire on DELETE or only INSERT/UPDATE?
- Verifying the trigger function handles the DELETE case (TG_OP = 'DELETE')
- Writing explicit tests for INSERT, UPDATE, and DELETE audit trail entries
- Checking that the audit log captures the correct old and new values

Example answer:

- First, I would check the trigger definition: `SELECT tgname, tgtype FROM pg_trigger WHERE tgrelid = 'users'::regclass`. The tgtype bitmask indicates which operations fire the trigger. If DELETE is not included, the trigger was defined with `AFTER INSERT OR UPDATE` but not `OR DELETE`.
- Next, I would check the trigger function for a `WHEN TG_OP = 'DELETE'` branch. If this branch is missing, DELETE operations fire the trigger but the function does not handle them (or handles them incorrectly).
- Tests I would add: (1) INSERT a user, verify audit_log has an entry with action='INSERT'. (2) UPDATE the user's role, verify audit_log has action='UPDATE' with correct old_value and new_value. (3) DELETE the user, verify audit_log has action='DELETE' with the deleted data.
- I would also test that the audit_log entry captures the correct changed_by user, and that the trigger does not prevent the actual operation from succeeding (a trigger error should not block legitimate operations – or it should, depending on requirements).

Question 4

Prompt: A financial system uses a stored procedure that calculates order totals. During testing, you find that `3 * 0.10` returns `0.30000000000000004` instead of `0.30`. Explain why this happens and how you would fix it at the database level.

What a strong answer should cover:

- FLOAT/DOUBLE types use binary floating-point representation, which cannot exactly represent 0.10
- DECIMAL/NUMERIC types use base-10 representation, which handles currency correctly
- The fix: use DECIMAL(10,2) for all monetary values
- Testing with known precision-sensitive values: 0.10, 0.01, 1/3

Example answer:

- This happens because the column or calculation uses FLOAT or DOUBLE PRECISION, which stores numbers in binary. The value 0.10 cannot be represented exactly in binary (like 1/3 cannot be represented exactly in decimal), so it becomes 0.10000000000000005551… internally.
- The fix is to use DECIMAL(10,2) or NUMERIC(10,2) for all monetary values and calculations. DECIMAL uses base-10 arithmetic, so `3 * 0.10` is exactly `0.30`.
- In the stored procedure, I would verify that all variables and parameters use DECIMAL, not FLOAT. I would also check the line_items.unit_price and orders.total column types.
- My test would explicitly check: insert line items with quantity=3 and unit_price=0.10, call calculate_order_total, and assert the result equals exactly 0.30 (not approximately 0.30).

Part III: Beyond SQL

Chapter 11: NoSQL Fundamentals – Beyond Relational Tables

11.1 The NoSQL Landscape

Not all data lives in relational databases. MongoDB stores documents, Redis caches key-value pairs, and DynamoDB provides serverless key-value/document storage. As a QA engineer, you need to know how to test these systems – the patterns differ significantly from SQL.

Database	Type	Primary Use	QA Testing Focus
MongoDB	Document	Flexible schemas, content management	Schema validation, aggregation pipelines
Redis	Key-Value / Cache	Caching, sessions, rate limiting	TTL expiration, cache invalidation
DynamoDB	Key-Value / Document	Serverless, high-scale	Capacity limits, GSI consistency
Elasticsearch	Search engine	Full-text search, log aggregation	Index mapping, query accuracy
Cassandra	Wide-column	Time-series, high-write	Consistency levels, partition sizing

11.2 MongoDB Testing

MongoDB stores data as JSON-like documents (BSON). Documents in the same collection can have different structures – which is both a feature and a testing challenge.

Document Structure Validation

```
from pymongo import MongoClient
import pytest

@pytest.fixture(scope="session")
def mongo_db():
    client = MongoClient("mongodb://localhost:27017")
    db = client["testdb"]
    yield db
    client.close()

def test_user_document_structure(mongo_db):
    """Verify user documents have expected fields."""
    mongo_db.users.insert_one({
        "name": "Alice",
        "email": "alice@test.com",
        "tags": ["admin"],
        "profile": {"bio": "Test user", "avatar_url": None}
    })

    user = mongo_db.users.find_one({"email": "alice@test.com"})
    assert user is not None
    assert isinstance(user["tags"], list)
    assert "admin" in user["tags"]
    assert isinstance(user["profile"], dict)
    assert "bio" in user["profile"]
```

Schema Validation

```
def test_schema_validation_enforced(mongo_db):
    """If schema validation is configured, invalid documents
    should be rejected."""
    mongo_db.command({
        "collMod": "strict_users",
        "validator": {
            "$jsonSchema": {
                "bsonType": "object",
                "required": ["name", "email"],
```

```
            "properties": {
                "name": {"bsonType": "string"},
                "email": {
                    "bsonType": "string",
                    "pattern": "^.+@.+$"
                }
            }
        }
    }
})

# Valid document should succeed
mongo_db.strict_users.insert_one({
    "name": "Valid", "email": "valid@test.com"
})

# Invalid document should fail
from pymongo.errors import WriteError
with pytest.raises(WriteError):
    mongo_db.strict_users.insert_one(
        {"name": "Invalid"})  # Missing email
```

Aggregation Pipeline Testing

```
def test_aggregation_pipeline(mongo_db):
    """Test an aggregation pipeline that calculates user statistics."""
    mongo_db.orders.insert_many([
        {"user_id": "user-1", "total": 50.00, "status": "completed"},
        {"user_id": "user-1", "total": 75.00, "status": "completed"},
        {"user_id": "user-1", "total": 25.00, "status": "cancelled"},
        {"user_id": "user-2", "total": 100.00, "status": "completed"},
    ])

    pipeline = [
        {"$match": {"status": "completed"}},
        {"$group": {
            "_id": "$user_id",
            "total_spent": {"$sum": "$total"},
            "order_count": {"$sum": 1}
        }},
        {"$sort": {"total_spent": -1}}
    ]
    results = list(mongo_db.orders.aggregate(pipeline))

    assert len(results) == 2
    assert results[0]["_id"] == "user-1"
    assert results[0]["total_spent"] == 125.00
    assert results[0]["order_count"] == 2
```

11.3 Redis Testing

Redis is primarily used as a cache, session store, or message broker. Testing Redis focuses on TTL behavior, cache invalidation, and data structure correctness.

TTL Expiration

```
import redis
import time

@pytest.fixture
def redis_client():
    client = redis.Redis(host="localhost", port=6379, db=1)
    yield client
    client.flushdb()

def test_cache_ttl_expiration(redis_client):
    """Cached data should expire after TTL."""
    redis_client.setex("session:abc", 2, "user-data")  # 2 second TTL

    assert redis_client.get("session:abc") is not None

    time.sleep(3)
    assert redis_client.get("session:abc") is None

def test_cache_ttl_value(redis_client):
    """Verify the TTL is set correctly."""
    redis_client.setex("session:def", 300, "data")
    ttl = redis_client.ttl("session:def")
    assert 298 <= ttl <= 300
```

Cache Invalidation

```
def test_cache_invalidation_on_update(redis_client, api_client):
    """When a user is updated via API, their cache should be invalidated."""
    api_client.get("/users/123")
    assert redis_client.get("user:123") is not None

    api_client.put("/users/123", json={"name": "Updated Name"})

    assert redis_client.get("user:123") is None

    response = api_client.get("/users/123")
    assert response.json()["name"] == "Updated Name"
    assert redis_client.get("user:123") is not None
```

Redis Data Structures

```
def test_redis_sorted_set_leaderboard(redis_client):
    """Test a leaderboard using Redis sorted set."""
    redis_client.zadd("leaderboard", {
        "alice": 100, "bob": 85, "charlie": 92
    })

    top = redis_client.zrevrange("leaderboard", 0, 1, withscores=True)
    assert top[0] == (b"alice", 100.0)
    assert top[1] == (b"charlie", 92.0)

    rank = redis_client.zrevrank("leaderboard", "bob")
    assert rank == 2  # 0-indexed

def test_redis_list_as_queue(redis_client):
    """Test a job queue using Redis list."""
    redis_client.rpush("job_queue", "job-1", "job-2", "job-3")
    assert redis_client.llen("job_queue") == 3

    job = redis_client.lpop("job_queue")
    assert job == b"job-1"
    assert redis_client.llen("job_queue") == 2
```

11.4 DynamoDB Testing

```
import boto3
from moto import mock_dynamodb

@mock_dynamodb
def test_dynamodb_crud():
    """Test CRUD operations on DynamoDB."""
    dynamodb = boto3.resource("dynamodb", region_name="us-east-1")

    table = dynamodb.create_table(
        TableName="users",
        KeySchema=[{"AttributeName": "id", "KeyType": "HASH"}],
        AttributeDefinitions=[
            {"AttributeName": "id", "AttributeType": "S"}
        ],
        BillingMode="PAY_PER_REQUEST"
    )

    table.put_item(Item={
        "id": "user-1", "name": "Alice", "email": "alice@test.com"
    })

    response = table.get_item(Key={"id": "user-1"})
    assert response["Item"]["name"] == "Alice"

    response = table.get_item(Key={"id": "nonexistent"})
    assert "Item" not in response
```

11.5 SQL vs NoSQL Testing Comparison

```
+--------------------+--------------------------+--------------------------+
| Aspect             | SQL Testing              | NoSQL Testing            |
+--------------------+--------------------------+--------------------------+
| Schema             | Enforced by database     | Enforced by application  |
|                    | (test constraints)       | (test document structure)|
+--------------------+--------------------------+--------------------------+
| Relationships      | JOINs and foreign keys   | Embedded documents or    |
|                    | (test referential        | manual references (test  |
|                    |  integrity)              |  consistency)            |
+--------------------+--------------------------+--------------------------+
| Transactions       | Full ACID                | Varies by database       |
|                    | (test atomicity)         | (test eventual           |
|                    |                          |  consistency)            |
+--------------------+--------------------------+--------------------------+
| Query Language     | Standard SQL             | Database-specific        |
|                    | (portable skills)        | (learn each one)         |
+--------------------+--------------------------+--------------------------+
| Caching/TTL        | Not built-in             | Core feature (Redis)     |
|                    |                          | (test expiration)        |
+--------------------+--------------------------+--------------------------+
```

Exercises

Beginner:

1. Write a MongoDB test: insert a document, query it back, verify all fields.
2. Write a Redis test: set a key with TTL, verify it exists, wait for expiration, verify it is gone.
3. Write a DynamoDB test using moto: create a table, insert an item, read it back.

Intermediate:

4. Write a MongoDB aggregation pipeline test that groups orders by status and calculates totals.
5. Write a Redis test for rate limiting: increment a counter key, verify it resets after the window.
6. Write a test that verifies cache invalidation: update data via API, verify Redis cache was cleared.

Advanced:

7. Write a comprehensive MongoDB test that validates document structure across an entire collection (no documents missing required fields).
8. Write a Redis pub/sub test that verifies a message published to a channel is received by a subscriber.
9. Compare constraint enforcement between PostgreSQL and MongoDB: write equivalent tests for both showing the differences.

Career Translation

Resume phrasing

- Extended database testing capabilities beyond SQL to cover MongoDB document validation, Redis cache TTL verification, and DynamoDB CRUD operations using pymongo, redis-py, and moto.
- Built cache invalidation test suites verifying that stale data is purged from Redis within expected TTL windows after API updates, reducing stale-data customer complaints by 90%.
- Designed MongoDB aggregation pipeline tests validating complex data transformations, ensuring reporting accuracy across document store and relational database hybrid architectures.

Cover letter framing

Modern applications rarely use a single database technology. I bring testing expertise across the NoSQL spectrum – MongoDB document validation, Redis cache behavior, DynamoDB operations – in addition to deep SQL skills. I understand the testing differences: SQL databases enforce schema at the database level, while NoSQL databases push schema enforcement to the application, which means my tests must cover document structure, TTL expiration, and eventual consistency patterns that do not exist in the relational world.

Interview framing

"I approach NoSQL testing by focusing on what the database does not enforce automatically. In PostgreSQL, constraints prevent invalid data. In MongoDB, schema validation is optional, so I test document structure explicitly. In Redis, TTL expiration and cache invalidation are the critical behaviors to verify. In DynamoDB, I use moto for local testing and focus on partition key design and GSI consistency. The trade-off is that NoSQL tests are more application-specific – there is no universal constraint system to lean on, so every validation must be written explicitly in test code."

What not to say

- "I only work with SQL databases." – Most modern applications use multiple database technologies.
- "MongoDB does not have schemas, so there is nothing to test." – MongoDB has optional schema validation, and document structure should always be tested.
- "I do not test cache expiration because it is infrastructure." – Cache TTL bugs cause stale data bugs that directly affect users.
- "I test DynamoDB against the real AWS service." – This is slow and expensive; moto provides a local mock for testing.
- "NoSQL databases are just like SQL without the schema." – They have fundamentally different consistency models, query patterns, and testing requirements.

Interview Depth Check

Question 1

Prompt: Your application uses Redis as a session cache with a 30-minute TTL. Users report being randomly logged out after 5 minutes. How would you diagnose this, and what Redis-specific tests would you write to prevent recurrence?

What a strong answer should cover:

- Checking the actual TTL set on session keys using redis-cli TTL command
- Verifying whether SETEX is being called with the correct TTL value

- Testing for TTL reset on activity (should the TTL extend on each request?)
- Checking for cache eviction due to memory pressure (maxmemory-policy)
- Writing a test that sets a key, checks TTL, waits, and verifies expiration timing

Example answer:

- First, I would check the actual TTL on a session key: `redis-cli TTL session:<user_id>`. If it shows 300 (5 minutes) instead of 1800 (30 minutes), the application is setting the wrong TTL.
- I would check the application code for SETEX calls and verify the expiration parameter. Common bugs: using seconds when the config is in minutes, or a hardcoded value overriding the config.
- I would also check Redis memory: `redis-cli INFO memory`. If maxmemory is hit, Redis may be evicting keys based on the maxmemory-policy (e.g., volatile-lru evicts keys with TTLs first).
- Tests: (1) Set a session key with SETEX, immediately check TTL is within expected range. (2) Set a key, sleep for 2 seconds, verify TTL decreased by approximately 2. (3) Set a key with 2-second TTL, sleep for 3 seconds, verify the key is gone. (4) Verify that the application refreshes the TTL on activity by making an authenticated request and checking the TTL increased.

Question 2

Prompt: Your team is migrating a feature from PostgreSQL to MongoDB. In PostgreSQL, referential integrity between users and orders is enforced by a foreign key constraint. How would you ensure the same data integrity in MongoDB, and what tests would you write?

What a strong answer should cover:

- MongoDB does not have foreign key constraints; referential integrity is the application's responsibility
- Options: embed orders within user documents, or maintain separate collections with application-level enforcement

- Tests: verify no orders reference non-existent users, verify document structure, verify application rejects orphan creation
- The trade-off between embedding (consistency) and referencing (flexibility)

Example answer:

- MongoDB has no foreign key constraints, so if orders are in a separate collection with a user_id field, nothing prevents inserting an order with a non-existent user_id. The application must enforce this.
- I would write a collection-wide integrity check: `db.orders.aggregate([{$lookup: {from: "users", localField: "user_id", foreignField: "_id", as: "user"}}, {$match: {"user": {$size: 0}}}])` – this finds all orders with no matching user.
- I would also test the application's enforcement: attempt to create an order via the API with a non-existent user_id and verify it is rejected. Then test directly against MongoDB to verify that an insert with an invalid user_id succeeds (proving the database does not enforce it) – this documents the gap.
- If the team chooses document embedding (orders inside user documents), I would test that the embedded array structure is correct and that querying by order fields still works efficiently.

Question 3

Prompt: You are testing a feature that uses MongoDB aggregation pipelines to generate sales reports. The pipeline groups orders by status, calculates totals, and sorts by revenue. How would you design tests for this pipeline, and what pitfalls would you watch for?

What a strong answer should cover:

- Inserting known test data with predictable aggregation results
- Testing with edge cases: empty collections, single document, documents with missing fields
- Verifying sort order and grouping accuracy

- Pitfalls: field type mismatches (string vs number), missing fields returning null vs 0, $match stage filtering too aggressively

Example answer:

- I would insert a controlled dataset: 5 orders with known statuses and totals, then run the pipeline and verify the results against manually calculated expected values. For example: 2 completed orders at $50 and $75 should produce a group with count=2 and total=$125.
- Edge cases: run the pipeline on an empty collection (should return empty results, not an error). Insert an order with a missing `total` field and verify the pipeline handles it (does $sum treat null as 0 or skip it?). Insert an order where `total` is a string instead of a number and verify the pipeline either rejects it or handles the type mismatch.
- I would also test the sort: if two status groups have the same revenue, is the sort stable? Does the pipeline correctly handle the `cancelled` status (should it be excluded from revenue calculations?).
- Pitfall I always check: MongoDB aggregation treats `$sum` of null as 0, but `$avg` of null is null. This difference causes reporting bugs when some documents have missing fields.

Chapter 12: Data Masking and Anonymization – Compliance by Design

12.1 Why Data Masking Matters

Test environments need realistic data without exposing real customer information. Data masking transforms production data into safe test data while preserving format, relationships, and statistical properties. Compliance regulations (GDPR, HIPAA, PCI DSS) make this not just a best practice but a legal requirement.

Without Masking	With Masking
Test environments contain real customer data	Test data looks realistic but is fake
Data breach in staging exposes real customers	Breach in staging exposes nothing sensitive

continued on next page

Without Masking	With Masking
Compliance violations (GDPR, HIPAA)	Compliance by design
Cannot share test environments with contractors	Safe to share with anyone
Production data copy requires security approvals	Masked copy is safe to distribute

12.2 Masking Techniques

Data Type	Original	Masked	Technique
Email	john@company.com	user_8a3f@masked.test	Hash-based replacement
Phone	+1-555-867-5309	+1-555-000-0042	Format-preserving randomization
SSN	123-45-6789	XXX-XX-6789	Partial redaction
Credit card	4111111111111111	4111XXXXXXXX1111	First/last four preserved
Name	John Smith	David Wilson	Lookup-based substitution
Address	123 Main St, NY	456 Oak Ave, CA	Randomized replacement
Date of birth	1985-03-15	1985-07-22	Day/month shuffled, year preserved
Salary	$85,000	$82,000	Range-preserving perturbation

```
Data Masking Flow:

+------------------+       +------------------+       +------------------+
| Production DB    | ----> | Masking Engine   | ----> | Test DB          |
|                  |       |                  |       |                  |
| john@company.com |       | Apply rules:     |       | user_8a3f@test   |
| 123-45-6789      |       | - Hash emails    |       | XXX-XX-6789      |
| John Smith       |       | - Redact SSN     |       | David Wilson     |
| 4111...1111      |       | - Substitute     |       | 4111XXXX1111     |
+------------------+       +------------------+       +------------------+

                           Preserves:
                           - Format (emails look like emails)
                           - Relationships (same user = same masked name)
                           - Referential integrity (FKs still work)
                           - Statistical properties (counts, averages)
```

12.3 Testing Format Preservation

Masked data must look like the original data type. Emails must look like emails. Phone numbers must be valid phone number format.

```
def test_masked_emails_have_valid_format(masked_db):
    cursor = masked_db.cursor()
    cursor.execute("SELECT email FROM users LIMIT 100")
    for row in cursor.fetchall():
        email = row[0]
        assert "@" in email, f"Invalid email format: {email}"
        assert "." in email.split("@")[1], f"Invalid domain: {email}"
```

12.4 Testing Relationship Preservation

The same customer should have the same masked name across all tables:

```
def test_masked_relationships_consistent(masked_db):
    cursor = masked_db.cursor()
    cursor.execute("""
        SELECT u.email AS user_email, o.customer_email AS order_email
        FROM users u
        JOIN orders o ON o.user_id = u.id
        LIMIT 50
    """)
    for row in cursor.fetchall():
        assert row[0] == row[1], \
            f"Email mismatch: users.email={row[0]}, " \
            f"orders.customer_email={row[1]}"
```

12.5 Testing Referential Integrity

Foreign keys must still resolve. Masked data should not break joins:

```
def test_masked_data_referential_integrity(masked_db):
    cursor = masked_db.cursor()

    cursor.execute("""
        SELECT COUNT(*)
        FROM orders o
        LEFT JOIN users u ON u.id = o.user_id
        WHERE u.id IS NULL
    """)
    orphans = cursor.fetchone()[0]
    assert orphans == 0, \
        f"Found {orphans} orders referencing non-existent users"
```

12.6 Testing Irreversibility

Masking must be one-way. It should not be possible to reverse the masking to recover original data:

```
def test_masking_is_irreversible(original_db, masked_db):
    """No original values should appear in masked data."""
    orig_cursor = original_db.cursor()
    orig_cursor.execute("SELECT email FROM users LIMIT 100")
    original_emails = {row[0] for row in orig_cursor.fetchall()}

    masked_cursor = masked_db.cursor()
    masked_cursor.execute("SELECT email FROM users")
    masked_emails = {row[0] for row in masked_cursor.fetchall()}

    overlap = original_emails & masked_emails
    assert len(overlap) == 0, \
        f"Original emails found in masked data: {overlap}"
```

12.7 Statistical Preservation

For analytics and reporting tests, masked data should preserve statistical properties:

```
def test_masked_data_preserves_statistics(original_db, masked_db):
    """Distribution of key metrics should be similar after masking."""
    for db, label in [(original_db, "original"), (masked_db, "masked")]:
        cursor = db.cursor()
        cursor.execute("SELECT COUNT(*) FROM orders")
        count = cursor.fetchone()[0]
        cursor.execute("SELECT AVG(total) FROM orders")
        avg_total = cursor.fetchone()[0]
        print(f"{label}: {count} orders, avg total ${avg_total:.2f}")
```

12.8 Compliance Requirements

Regulation	Impact on Testing
GDPR	Test environments must use masked data; verify deletion removes all references
HIPAA	No real patient data in test environments, ever
PCI DSS	Test with valid-format but fake card numbers (4111…)
SOC 2	Access to production data must be audited; masked data reduces audit scope
CCPA	Similar to GDPR – California residents' data must be protected

GDPR-Specific Tests

```
def test_gdpr_deletion_removes_all_references(api, db):
    """GDPR right to erasure: deleting a user removes ALL their data."""
    user = create_user_with_full_history(api)

    api.delete(f"/users/{user['id']}/gdpr-delete")

    cursor = db.cursor()

    cursor.execute(
        "SELECT * FROM users WHERE id = %s", (user["id"],))
    assert cursor.fetchone() is None, "User record still exists"

    cursor.execute(
        "SELECT * FROM orders WHERE user_id = %s", (user["id"],))
    assert cursor.fetchone() is None, "User orders still exist"

    cursor.execute(
        "SELECT * FROM audit_log WHERE user_id = %s", (user["id"],))
    assert cursor.fetchone() is None, "User audit trail still exists"

    cursor.execute(
        "SELECT * FROM session_logs WHERE user_id = %s", (user["id"],))
    assert cursor.fetchone() is None, "User session logs still exist"
```

12.9 Masking Tools

Tool	Type	Best For
pg_anonymize	PostgreSQL extension	Simple rules-based masking
Delphix	Enterprise platform	Large-scale, automated masking
DataMasker	Commercial tool	SQL Server and Oracle
Faker (Python/JS)	Library	Generating fake data for test environments
Custom scripts	DIY	Full control over masking logic

Simple Masking with Faker

```
from faker import Faker

fake = Faker()

def mask_users_table(source_conn, target_conn):
    source = source_conn.cursor()
    target = target_conn.cursor()

    source.execute("SELECT id, name, email, phone FROM users")
    for row in source.fetchall():
        target.execute(
            "INSERT INTO users (id, name, email, phone) "
            "VALUES (%s, %s, %s, %s)",
            (row[0], fake.name(), fake.email(), fake.phone_number())
        )
    target_conn.commit()
```

Practice Schema 10: Healthcare Records (Masking Exercise)

```
CREATE TABLE patients (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    first_name      VARCHAR(100) NOT NULL,
    last_name       VARCHAR(100) NOT NULL,
    date_of_birth   DATE NOT NULL,
    ssn             VARCHAR(11) UNIQUE,
    email           VARCHAR(255),
    phone           VARCHAR(20),
    insurance_id    VARCHAR(50),
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE medical_records (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    patient_id      UUID NOT NULL REFERENCES patients(id),
    provider_id     UUID NOT NULL,
    diagnosis_code  VARCHAR(10) NOT NULL,     -- ICD-10 code
    diagnosis_text  TEXT NOT NULL,
    visit_date      DATE NOT NULL,
    notes           TEXT,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE prescriptions (
```

```
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    record_id       UUID NOT NULL REFERENCES medical_records(id),
    medication      VARCHAR(200) NOT NULL,
    dosage          VARCHAR(100) NOT NULL,
    frequency       VARCHAR(100) NOT NULL,
    start_date      DATE NOT NULL,
    end_date        DATE,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE session_logs (
    id              BIGSERIAL PRIMARY KEY,
    user_id         UUID,
    ip_address      INET,
    action          VARCHAR(100),
    resource        VARCHAR(200),
    created_at      TIMESTAMP DEFAULT NOW()
);
```

Exercises

Beginner:

1. Write a masking function that replaces email addresses while preserving the @ symbol and domain format.
2. Write a test that verifies masked SSNs show only the last four digits (XXX-XX-6789 format).
3. Write a test that verifies all patients in the masked database have non-NULL names and valid date_of_birth.

Intermediate:

4. Write a complete masking script that transforms the patients table: names, SSNs, emails, and phone numbers.
5. Write tests that verify relationship consistency: the same patient_id in medical_records maps to the same masked patient.
6. Write a GDPR deletion test across patients, medical_records, prescriptions, and session_logs.

Advanced:

7. Write a statistical preservation test: verify that the distribution of diagnosis codes is similar between original and masked datasets.
8. Write a masking pipeline that handles the full healthcare schema (patients, records, prescriptions) while preserving all foreign key relationships.
9. Design a masking verification test suite that can be run against any masked database to validate format, consistency, referential integrity, and irreversibility.

Career Translation

Resume phrasing

- Designed and validated data masking pipelines for GDPR and HIPAA compliance, verifying format preservation, referential integrity, and irreversibility across 12 tables containing PII.
- Built automated masking verification test suites that confirmed zero original PII appeared in test environments, eliminating compliance risk for a team of 30 developers and contractors.
- Implemented GDPR right-to-erasure tests verifying complete data removal across users, orders, audit logs, and session logs – all foreign key relationships and cascade behaviors confirmed.

Cover letter framing

Compliance is not a checkbox – it is a testable property. I build automated verification suites that confirm masked data preserves format (emails still look like emails), maintains referential integrity (foreign keys still resolve), and is irreversible (no original PII can be recovered). This approach gives legal and compliance teams concrete evidence that test environments are safe, and it gives the development team confidence to work with realistic data without regulatory risk.

Interview framing

"I approach data masking as four testable properties: format preservation, relationship consistency, referential integrity, and irreversibility. Format preservation means masked emails still pass email validation, masked phone numbers still have the right digit count. Relationship consistency means the same customer has the same masked name everywhere they appear. Referential integrity means foreign keys still resolve after masking. Irreversibility means no original value appears anywhere in the masked dataset. I optimize for automation – these checks should run every time the masking pipeline produces a new dataset. The trade-off is that deterministic masking (same input always produces same output) is easier to test for consistency but slightly less secure than random masking."

What not to say

- "We just copy production data into test environments." – This is a compliance violation under GDPR, HIPAA, and PCI DSS.
- "Masking is not my responsibility as a QA engineer." – If you test with data, you are responsible for ensuring it is safe.
- "We mask names and emails, that is enough." – Addresses, phone numbers, SSNs, IP addresses, and session logs all contain PII.
- "I test masking by spot-checking a few records manually." – Manual spot-checks do not scale. Automated verification is required.
- "Masked data does not need to be realistic." – Unrealistic masked data produces unrealistic test results.

Interview Depth Check

Question 1

Prompt: Your company receives a GDPR "right to erasure" request from a customer. How would you design a test that verifies the deletion is complete across all database tables, including audit logs and session records?

What a strong answer should cover:

- Identifying all tables that contain user data (direct and indirect references)
- Testing that the DELETE cascades or explicitly removes data from every table
- Verifying audit logs and session logs are also purged (not just the primary user record)
- Edge cases: the user's data in backups, cached aggregations, and third-party systems

Example answer:

- I would create a user with a full data footprint: profile, orders, payments, reviews, audit log entries, and session logs. Then I would call the GDPR deletion endpoint.
- Verification queries for each table: `SELECT COUNT(*) FROM users WHERE id = '<user_id>'` (should be 0), `SELECT COUNT(*) FROM orders WHERE user_id = '<user_id>'` (should be 0 or orders should be anonymized), `SELECT COUNT(*) FROM audit_log WHERE changed_by = '<user_id>'` (should be 0), `SELECT COUNT(*) FROM session_logs WHERE user_id = '<user_id>'` (should be 0).
- I would also search for the user's email across all text columns: if any table stores email as a denormalized field, it must also be purged. I would use `information_schema.columns` to find all VARCHAR columns and search each one.
- Edge case: verify that the deletion itself is logged (for compliance proof), but the log entry does not contain the deleted user's PII.

Question 2

Prompt: Your masking pipeline replaces real emails with hash-based masked emails. After running the pipeline, a developer reports that some masked emails appear in multiple user records – two different customers have the same masked email. What went wrong, and how would you test for this?

What a strong answer should cover:

- Hash collisions or insufficient entropy in the masking function
- The masking function might be hashing only part of the email (e.g., domain only)
- Testing: verify uniqueness of masked values matches uniqueness of original values
- Relationship preservation: the same original email should always produce the same masked email

Example answer:

- If two different original emails produce the same masked email, the hash function has a collision or is truncating output. For example, if the masking function only hashes the local part and two emails share the same local part with different domains (john@gmail.com, john@yahoo.com), they might collide.
- Test 1: `SELECT email, COUNT(*) FROM masked_users GROUP BY email HAVING COUNT(*) > 1`. If this returns results, we have duplicate masked emails.
- Test 2: Compare uniqueness counts: `SELECT COUNT(DISTINCT email) FROM original_users` should equal `SELECT COUNT(DISTINCT email) FROM masked_users`. If the masked count is lower, collisions occurred.
- Test 3: Verify determinism: the same original email in different tables should produce the same masked email. `SELECT u.email AS user_email, o.customer_email AS order_email FROM masked_users u JOIN masked_orders o ON o.user_id = u.id` – these should match.
- Fix: use a longer hash output, include the full email (not just the local part) in the hash input, and add a salt that is consistent across tables.

Question 3

Prompt: You need to create a test environment with masked production data for a team of external contractors. The data must be realistic enough

for feature development but must not contain any real PII. How would you validate that the masked dataset meets both requirements?

What a strong answer should cover:

- Format preservation: emails, phones, SSNs all pass format validation
- Statistical preservation: record counts, distributions, and relationships match production patterns
- Irreversibility: no overlap between original and masked PII values
- Referential integrity: all JOINs that work in production also work in the masked dataset
- Documenting what was masked and the masking technique used

Example answer:

- Format preservation tests: `SELECT email FROM masked_users WHERE email NOT LIKE '%@%.%'` should return zero rows. `SELECT phone FROM masked_users WHERE LENGTH(REPLACE(phone, '-', '')) != 10` should return zero rows (assuming US phone format).
- Statistical preservation: `SELECT status, COUNT(*) FROM masked_orders GROUP BY status` should produce similar distributions to production. Total order count should be identical.
- Irreversibility: `SELECT COUNT(*) FROM original_users ou JOIN masked_users mu ON ou.email = mu.email` should return zero – no original emails appear in the masked dataset. Repeat for names, phones, SSNs, and addresses.
- Referential integrity: `SELECT COUNT(*) FROM masked_orders o LEFT JOIN masked_users u ON u.id = o.user_id WHERE u.id IS NULL` should return zero.
- I would package these tests as a reusable validation suite that runs automatically every time the masking pipeline produces a new dataset, with a pass/fail report that the compliance team can review.

Question 4

Prompt: A colleague suggests using `UPDATE users SET email = CONCAT('user_', id, '@masked.test')` as the masking strategy. Evaluate this approach: what does it get right, and what are its weaknesses?

What a strong answer should cover:

- Strengths: deterministic, preserves email format, produces unique values, simple to implement
- Weaknesses: reversible (if you know the user's ID, you know the pattern), does not preserve statistical properties (all emails have the same domain), does not mask other PII
- The ID in the email leaks the mapping between masked and original data
- Better alternatives: hash-based masking, Faker-based substitution

Example answer:

- This approach gets format preservation right: every masked email has an @ symbol and a domain. It is deterministic, so the same user always gets the same masked email across tables. It is unique, so no collisions.
- The critical weakness is reversibility: the user ID is embedded in the email in plain text. Anyone with database access can map `user_abc123@masked.test` back to user `abc123` and look up their real data in a backup or the original database. Masking must be one-way.
- Another weakness: all masked emails share the `@masked.test` domain, which breaks any tests that depend on email domain distribution (e.g., "show me orders from Gmail users").
- A better approach: `CONCAT(MD5(email || 'salt'), '@masked.test')`. The MD5 hash is one-way, the salt prevents rainbow table attacks, and the format is preserved. For better statistical properties, use Faker to generate realistic-looking emails with varied domains.

Part IV: Putting It All Together

Chapter 13: Capstone Project – The Complete Database Test Suite

13.1 Project Overview

You are the QA engineer for "ShopFast," an e-commerce platform. Your task is to build a comprehensive database test suite that covers every topic in this book. The project uses the following unified schema that combines elements from all practice schemas.

13.2 The ShopFast Database Schema

```
-- ============================================================
-- ShopFast E-Commerce Database Schema
-- ============================================================

-- Users and Authentication
CREATE TABLE users (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name            VARCHAR(100) NOT NULL,
    email           VARCHAR(255) NOT NULL UNIQUE,
    password_hash   VARCHAR(255) NOT NULL,
    role            VARCHAR(20) DEFAULT 'customer'
                    CHECK (role IN ('customer','seller','admin','support')),
    active          BOOLEAN DEFAULT true,
    email_verified  BOOLEAN DEFAULT false,
    last_login_at   TIMESTAMP,
    created_at      TIMESTAMP DEFAULT NOW(),
    updated_at      TIMESTAMP DEFAULT NOW(),
    deleted_at      TIMESTAMP
);

-- Products and Catalog
CREATE TABLE categories (
    id          VARCHAR(50) PRIMARY KEY,
```

```
    name        VARCHAR(100) NOT NULL,
    parent_id   VARCHAR(50) REFERENCES categories(id),
    sort_order  INTEGER DEFAULT 0
);

CREATE TABLE products (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    seller_id   UUID NOT NULL REFERENCES users(id),
    name        VARCHAR(200) NOT NULL,
    sku         VARCHAR(50) UNIQUE NOT NULL,
    description TEXT,
    price       DECIMAL(10,2) NOT NULL CHECK (price >= 0),
    category_id VARCHAR(50) REFERENCES categories(id),
    active      BOOLEAN DEFAULT true,
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW()
);

-- Orders and Line Items
CREATE TABLE orders (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id     UUID NOT NULL REFERENCES users(id),
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN
                    ('pending','confirmed','shipped','delivered','cancelled','refunded')),
    subtotal    DECIMAL(10,2) NOT NULL CHECK (subtotal >= 0),
    tax         DECIMAL(10,2) NOT NULL DEFAULT 0 CHECK (tax >= 0),
    total       DECIMAL(10,2) NOT NULL CHECK (total >= 0),
    shipping_address TEXT,
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW(),
    shipped_at  TIMESTAMP,
    delivered_at TIMESTAMP
);

CREATE TABLE line_items (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id    UUID NOT NULL REFERENCES orders(id) ON DELETE CASCADE,
    product_id  UUID NOT NULL REFERENCES products(id),
    quantity    INTEGER NOT NULL CHECK (quantity > 0),
    unit_price  DECIMAL(10,2) NOT NULL CHECK (unit_price >= 0)
);

-- Payments
CREATE TABLE payments (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id    UUID NOT NULL REFERENCES orders(id),
    method      VARCHAR(20) NOT NULL
                CHECK (method IN ('credit_card','debit_card','paypal','bank_transfer')),
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','completed','failed','refunded')),
    amount      DECIMAL(10,2) NOT NULL CHECK (amount > 0),
    gateway_ref VARCHAR(100),
    created_at  TIMESTAMP DEFAULT NOW()
);

-- Reviews
CREATE TABLE reviews (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    product_id  UUID NOT NULL REFERENCES products(id),
    user_id     UUID NOT NULL REFERENCES users(id),
    rating      INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
    title       VARCHAR(200),
    body        TEXT,
    verified    BOOLEAN DEFAULT false,
    created_at  TIMESTAMP DEFAULT NOW(),
    UNIQUE(product_id, user_id)
```

```
);

-- Inventory
CREATE TABLE inventory (
    product_id  UUID PRIMARY KEY REFERENCES products(id),
    quantity    INTEGER NOT NULL DEFAULT 0 CHECK (quantity >= 0),
    reorder_level INTEGER DEFAULT 10,
    last_restocked TIMESTAMP
);

-- Audit Log
CREATE TABLE audit_log (
    id          BIGSERIAL PRIMARY KEY,
    table_name  VARCHAR(100) NOT NULL,
    record_id   VARCHAR(100) NOT NULL,
    action      VARCHAR(10) NOT NULL
                CHECK (action IN ('INSERT','UPDATE','DELETE')),
    changed_field VARCHAR(100),
    old_value   TEXT,
    new_value   TEXT,
    changed_by  UUID REFERENCES users(id),
    changed_at  TIMESTAMP DEFAULT NOW()
);

-- Triggers
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER users_updated_at BEFORE UPDATE ON users
    FOR EACH ROW EXECUTE FUNCTION update_timestamp();

CREATE TRIGGER products_updated_at BEFORE UPDATE ON products
    FOR EACH ROW EXECUTE FUNCTION update_timestamp();

CREATE TRIGGER orders_updated_at BEFORE UPDATE ON orders
    FOR EACH ROW EXECUTE FUNCTION update_timestamp();
```

13.3 Capstone Tasks

Complete the following tasks to build your test suite:

Task 1: Data Setup (Chapter 6)

Write a comprehensive test data setup script that creates:

- 5 users (1 admin, 2 sellers, 2 customers)
- 3 categories with a parent-child hierarchy
- 10 products across the categories
- 5 orders with varying statuses
- Line items, payments, reviews, and inventory records

Use identifiable prefixes (`capstone-`) for all test data.

Task 2: SELECT Verification (Chapters 1-2)

Write verification queries for:

- All products in a specific category (including subcategories)
- All users who have never placed an order
- Orders with invalid timestamps (delivered_at before shipped_at)
- Products that are active but have zero inventory

Task 3: JOIN Investigation (Chapter 3)

Write investigation queries for:

- Full order details (user, items, products, payment)
- Orphaned records across all tables
- Users with orders but no payments

Task 4: Aggregation Analysis (Chapter 4)

Write analysis queries for:

- Revenue by category (requires JOIN through line_items to products)
- Average rating per product with at least 3 reviews
- Duplicate detection across key unique fields

Task 5: CTE-Based Validation (Chapter 5)

Write a single multi-CTE validation query that checks:

- Order totals match sum of line items
- All foreign keys resolve
- No orphaned records exist
- Review ratings are within bounds

Task 6: Constraint Testing (Chapter 8)

Write Python tests for:

- Every constraint type on the orders table
- Foreign key behavior (CASCADE on line_items, RESTRICT implied on users)
- Check constraint on review ratings
- Unique constraint on product SKU

Task 7: Migration Testing (Chapter 9)

Design and test a migration that:

- Adds a `discount_code` column to orders
- Adds a `discount_amount` column with CHECK >= 0
- Modifies the total CHECK constraint to account for discounts
- Includes rollback

Task 8: Trigger Testing (Chapter 10)

Write tests for:

- The `updated_at` trigger on users, products, and orders
- Audit log trigger (if implemented)
- Computed inventory values after order completion

Task 9: Data Masking (Chapter 12)

Write a masking pipeline and tests for:

- User PII (name, email)
- Shipping addresses
- Payment gateway references
- Verify format preservation, relationship consistency, and irreversibility

13.4 Evaluation Criteria

Criteria	Points
Test data setup is complete and uses identifiable prefixes	10
Verification queries are correct and handle edge cases	15
JOIN queries find all orphaned records	15
Aggregation queries match expected results	10
CTE validation query is comprehensive	10
Constraint tests cover all types with positive and negative cases	15

continued on next page

Criteria	Points
Migration tests verify apply, rollback, and data preservation	10
Trigger tests verify automatic behavior	10
Data masking tests verify all four properties	5
Total	**100**

Career Translation

Resume phrasing

- Architected end-to-end database test suites covering data setup, SELECT verification, JOIN investigation, aggregation analysis, CTE validation, constraint testing, migration testing, trigger testing, and data masking across a unified e-commerce schema.
- Delivered a capstone-quality database testing portfolio demonstrating mastery of SQL fundamentals, data integrity verification, and compliance testing – all patterns applicable to production systems handling 100K+ daily transactions.
- Designed reusable test frameworks combining direct SQL verification, Python/pytest automation, and CI pipeline integration for comprehensive database quality assurance.

Cover letter framing

I bring a complete database testing skill set to the table – from writing verification SELECTs to testing schema migrations, from constraint validation to GDPR-compliant data masking. My approach is systematic: I build layered test suites where each layer (data integrity, referential consistency, business logic, compliance) catches a different class of defect. This comprehensive approach has consistently caught bugs that surface-level testing misses, and the test suites I build become reusable assets that serve teams long after the initial project.

Interview framing

"I approach database testing as a multi-layered discipline. Layer one is data integrity: do records exist with correct values? Layer two is referential consistency: do relationships between tables hold? Layer three is business logic: do stored procedures and triggers produce correct results? Layer four is compliance: is PII properly masked, are deletions complete? I build each layer as a separate test module with its own fixtures and assertions, so the team can run targeted checks or the full suite. The trade-off is upfront investment in test infrastructure, but the payoff is a test suite that catches defects across every database-related concern."

What not to say

- "I test the database by testing the UI." – UI tests do not verify database-level correctness.
- "I only write tests for new features, not existing data integrity." – Data quality degrades over time without ongoing verification.
- "I do not need a test data strategy because we use production data." – This is both a compliance risk and a test reliability problem.
- "I build one big test that checks everything." – Monolithic tests are hard to debug, maintain, and parallelize.
- "Database testing is just running a few SELECT queries." – Comprehensive database testing covers constraints, migrations, triggers, procedures, masking, and more.

Interview Depth Check

Question 1

Prompt: You are hired as the sole QA engineer for a startup that has an e-commerce platform with 20 tables, zero database tests, and a history of data integrity issues. You have two weeks to establish a database testing foundation. How would you prioritize your work?

What a strong answer should cover:

- Week 1: discover existing constraints, identify the most critical tables, write orphan detection and duplicate detection queries
- Week 2: add constraint tests for the most critical tables, set up migration testing in CI, create a nightly data health check
- Prioritization: focus on tables that handle money (orders, payments) and user identity (users, authentication)
- Quick wins: orphan detection, duplicate detection, and timestamp anomaly queries

Example answer:

- Day 1-2: Discovery. I would query pg_constraint for every table to map existing constraints. I would also run orphan detection queries (LEFT JOIN + IS NULL) across all foreign key relationships and duplicate detection (GROUP BY + HAVING COUNT > 1) on all unique columns. This gives me the current state of data integrity.
- Day 3-5: Critical table tests. I would write constraint tests for orders, payments, and users – the tables where bugs cost money. I would test every constraint type: PK, FK, UNIQUE, NOT NULL, CHECK, and DEFAULT. I would also write a CTE-based validation query that checks order totals match line item sums.
- Week 2: Infrastructure. I would set up migration testing in CI (clone schema, seed data, apply migration, verify, rollback, verify). I would create a nightly data health check that runs all orphan and duplicate detection queries and alerts on non-zero results. Finally, I would document the test patterns so future engineers can extend the suite.

Question 2

Prompt: The ShopFast platform has a bug where some orders show a total of $0.00 even though they have line items with non-zero prices. Walk me through a complete investigation using the SQL skills from this book, starting from detection through root cause analysis.

What a strong answer should cover:

- Detection: find all zero-total orders that have non-zero line items
- Scope: how many orders are affected, when did they start appearing, which users are impacted
- Root cause candidates: trigger not firing, stored procedure bug, race condition between insert and total calculation
- Using JOINs, GROUP BY, CTEs, and timestamp analysis for the investigation

Example answer:

- Detection: `WITH zero_orders AS (SELECT o.id, o.total, o.created_at FROM orders o WHERE o.total = 0.00), line_item_totals AS (SELECT order_id, SUM(quantity * unit_price) AS calculated FROM line_items GROUP BY order_id) SELECT zo.id, zo.total, lt.calculated, zo.created_at FROM zero_orders zo JOIN line_item_totals lt ON lt.order_id = zo.id WHERE lt.calculated > 0 ORDER BY zo.created_at`. This finds the affected orders.
- Scope: `SELECT DATE_TRUNC('day', created_at) AS day, COUNT(*) FROM (above query) GROUP BY day ORDER BY day`. If the zero-total orders started on a specific date, it correlates with a deployment.
- Root cause: I would check if the calculate_order_total stored procedure is being called during order creation. I would also check if a trigger that updates order.total exists and is enabled. If the timing correlates with a migration, I would check if the migration dropped or modified the trigger.
- Fix verification: after the fix is deployed, I would run the detection query again and verify zero results. I would also add this query as a permanent CI check.

Question 3

Prompt: You need to present your database test suite to the engineering leadership team. They want to know: what does it catch that other testing layers (unit tests, API tests, UI tests) do not? Give concrete examples for each testing layer comparison.

What a strong answer should cover:

- vs. unit tests: database tests catch constraint violations, trigger side effects, and stored procedure logic that unit tests mock away
- vs. API tests: database tests catch caching bugs, silent write failures, and incorrect defaults that API responses mask
- vs. UI tests: database tests catch data persistence issues, orphaned records, and referential integrity violations invisible to the UI
- The complementary nature: database tests do not replace other layers, they add a layer that catches different defects

Example answer:

- vs. unit tests: Unit tests mock the database, so they cannot catch constraint violations (a duplicate email that the UNIQUE constraint should reject), trigger failures (updated_at not changing), or stored procedure bugs (calculate_order_total returning wrong values). Database tests execute real SQL against a real database.
- vs. API tests: An API test creates a user and checks the 201 response. But the API might have a caching layer that returns 201 while the database write failed. Or the API might not return default values (role, created_at) that the database should have set. Database tests verify the actual persisted state.
- vs. UI tests: The UI shows "Order Placed" but the database has no order record (silent failure). The UI shows 5 line items but the database has 6 (orphaned record). The UI shows the correct total but the database stores a different value (display vs. persistence mismatch). Database tests catch all of these.
- I would frame it as: "Other testing layers verify what the system says happened. Database tests verify what actually happened."

Question 4

Prompt: Your capstone test suite needs to handle both PostgreSQL (primary data store) and MongoDB (product catalog cache). How would you structure your test fixtures, assertions, and cleanup to handle both database technologies in a single test run?

What a strong answer should cover:

- Separate fixtures for each database connection
- Shared test data factories that insert into both databases
- Cross-database consistency tests (PostgreSQL product data matches MongoDB cache)
- Independent cleanup strategies: transactions for PostgreSQL, collection drops for MongoDB
- Configuration management for connection strings

Example answer:

- I would create separate pytest fixtures: `pg_db` for PostgreSQL (using transaction rollback) and `mongo_db` for MongoDB (using collection cleanup). Both are session-scoped for connection reuse.
- A shared `product_factory` fixture inserts a product into PostgreSQL and the MongoDB cache simultaneously, returning the product ID. This ensures both databases have consistent test data.
- Cross-database tests: after inserting a product via the API, verify it exists in PostgreSQL (`SELECT * FROM products WHERE sku = '...'`) and in MongoDB (`db.products.find_one({"sku": "..."})`). Compare field values to ensure the cache matches the source of truth.
- Cleanup: PostgreSQL uses transaction ROLLBACK for instantaneous cleanup. MongoDB uses `db.products.delete_many({"_test": True})` where all test documents are tagged with a `_test` field.
- I would structure test modules by concern: `test_pg_constraints.py`, `test_mongo_schema.py`, `test_cross_db_consistency.py`, each importing the appropriate fixtures.

Appendices

Appendix A: Practice Database Schemas

This appendix consolidates all ten practice schemas referenced throughout the book. Each schema is self-contained and can be created in any PostgreSQL database.

Schema 1: User Management (Chapter 1)

```
CREATE TABLE users (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name            VARCHAR(100) NOT NULL,
    email           VARCHAR(255) NOT NULL UNIQUE,
    role            VARCHAR(20) DEFAULT 'viewer',
    active          BOOLEAN DEFAULT true,
    email_verified  BOOLEAN DEFAULT false,
    created_at      TIMESTAMP DEFAULT NOW(),
    updated_at      TIMESTAMP DEFAULT NOW(),
    deleted_at      TIMESTAMP
);

CREATE TABLE user_profiles (
    user_id     UUID PRIMARY KEY REFERENCES users(id),
    bio         TEXT,
    avatar_url  VARCHAR(500),
    phone       VARCHAR(20),
    timezone    VARCHAR(50) DEFAULT 'UTC'
);
```

Schema 2: E-Commerce Orders (Chapter 2)

```
CREATE TABLE products (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(200) NOT NULL,
    sku         VARCHAR(50) UNIQUE NOT NULL,
    price       DECIMAL(10,2) NOT NULL CHECK (price >= 0),
    category    VARCHAR(50),
    in_stock    BOOLEAN DEFAULT true,
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE orders (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id     UUID NOT NULL REFERENCES users(id),
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','confirmed','shipped','delivered','cancelled')),
    total       DECIMAL(10,2) NOT NULL CHECK (total >= 0),
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW(),
    shipped_at  TIMESTAMP,
    delivered_at TIMESTAMP
);

CREATE TABLE line_items (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id    UUID NOT NULL REFERENCES orders(id) ON DELETE CASCADE,
    product_id  UUID NOT NULL REFERENCES products(id),
    quantity    INTEGER NOT NULL CHECK (quantity > 0),
    unit_price  DECIMAL(10,2) NOT NULL CHECK (unit_price >= 0)
);
```

Schema 3: Payments and Refunds (Chapter 3)

```
CREATE TABLE payments (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id    UUID NOT NULL REFERENCES orders(id),
    method      VARCHAR(20) NOT NULL
                CHECK (method IN ('credit_card','debit_card','paypal','bank_transfer')),
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','completed','failed','refunded')),
    amount      DECIMAL(10,2) NOT NULL CHECK (amount > 0),
    gateway_ref VARCHAR(100),
    created_at  TIMESTAMP DEFAULT NOW(),
    completed_at TIMESTAMP
);

CREATE TABLE refunds (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    payment_id  UUID NOT NULL REFERENCES payments(id),
    amount      DECIMAL(10,2) NOT NULL CHECK (amount > 0),
    reason      TEXT NOT NULL,
    status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending','approved','rejected','completed')),
    created_at  TIMESTAMP DEFAULT NOW(),
    processed_at TIMESTAMP
);
```

Schema 4: Error Logging (Chapter 4)

```
CREATE TABLE error_logs (
    id          BIGSERIAL PRIMARY KEY,
    error_code  VARCHAR(10) NOT NULL,
    message     TEXT,
    stack_trace TEXT,
    endpoint    VARCHAR(200),
    user_id     UUID REFERENCES users(id),
    severity    VARCHAR(10) CHECK (severity IN ('low','medium','high','critical')),
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE request_logs (
    id          BIGSERIAL PRIMARY KEY,
    method      VARCHAR(10) NOT NULL,
    endpoint    VARCHAR(200) NOT NULL,
    status_code INTEGER NOT NULL,
    response_ms INTEGER,
    user_id     UUID REFERENCES users(id),
    created_at  TIMESTAMP DEFAULT NOW()
);
```

Schema 5: Reviews and Categories (Chapter 5)

```
CREATE TABLE reviews (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    product_id  UUID NOT NULL REFERENCES products(id),
    user_id     UUID NOT NULL REFERENCES users(id),
    rating      INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
    title       VARCHAR(200),
    body        TEXT,
    verified    BOOLEAN DEFAULT false,
    created_at  TIMESTAMP DEFAULT NOW(),
    UNIQUE(product_id, user_id)
);

CREATE TABLE categories (
    id          VARCHAR(50) PRIMARY KEY,
    name        VARCHAR(100) NOT NULL,
    parent_id   VARCHAR(50) REFERENCES categories(id),
    sort_order  INTEGER DEFAULT 0
);
```

Schema 6: Inventory Management (Chapter 6)

```
CREATE TABLE warehouses (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(100) NOT NULL,
    location    VARCHAR(200) NOT NULL,
    capacity    INTEGER NOT NULL CHECK (capacity > 0)
);

CREATE TABLE inventory (
    id           UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    warehouse_id UUID NOT NULL REFERENCES warehouses(id),
    product_id   UUID NOT NULL REFERENCES products(id),
    quantity     INTEGER NOT NULL DEFAULT 0 CHECK (quantity >= 0),
    reorder_level INTEGER DEFAULT 10,
    last_restocked TIMESTAMP,
    UNIQUE(warehouse_id, product_id)
);

CREATE TABLE inventory_transactions (
    id           UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    inventory_id UUID NOT NULL REFERENCES inventory(id),
    change_type  VARCHAR(20) NOT NULL
                 CHECK (change_type IN ('restock','sale','adjustment','return')),
    quantity_change INTEGER NOT NULL,
    reference_id VARCHAR(100),
    created_at   TIMESTAMP DEFAULT NOW()
);
```

Schema 7: Financial Accounts (Chapter 7)

```
CREATE TABLE accounts (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id     UUID NOT NULL REFERENCES users(id),
    name        VARCHAR(100) NOT NULL,
    balance     DECIMAL(12,2) NOT NULL DEFAULT 0.00,
    currency    VARCHAR(3) NOT NULL DEFAULT 'USD',
    status      VARCHAR(20) DEFAULT 'active'
                CHECK (status IN ('active','frozen','closed')),
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE transfers (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    from_account_id UUID NOT NULL REFERENCES accounts(id),
    to_account_id   UUID NOT NULL REFERENCES accounts(id),
    amount          DECIMAL(12,2) NOT NULL CHECK (amount > 0),
    status          VARCHAR(20) DEFAULT 'pending'
                    CHECK (status IN ('pending','completed','failed','reversed')),
    created_at      TIMESTAMP DEFAULT NOW(),
    completed_at    TIMESTAMP,
    CHECK (from_account_id != to_account_id)
);
```

Schema 8: Content Management (Chapter 8)

```
CREATE TABLE authors (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        VARCHAR(100) NOT NULL,
    email       VARCHAR(255) NOT NULL UNIQUE,
    bio         TEXT,
    active      BOOLEAN DEFAULT true,
    created_at  TIMESTAMP DEFAULT NOW()
);

CREATE TABLE articles (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    author_id   UUID NOT NULL REFERENCES authors(id) ON DELETE RESTRICT,
    title       VARCHAR(300) NOT NULL,
    slug        VARCHAR(300) NOT NULL UNIQUE,
    body        TEXT NOT NULL,
    status      VARCHAR(20) DEFAULT 'draft'
                CHECK (status IN ('draft','review','published','archived')),
    published_at TIMESTAMP,
    created_at  TIMESTAMP DEFAULT NOW(),
    updated_at  TIMESTAMP DEFAULT NOW(),
    CHECK (published_at IS NULL OR status = 'published')
);

CREATE TABLE tags (
```

```
    id   UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE article_tags (
    article_id UUID NOT NULL REFERENCES articles(id) ON DELETE CASCADE,
    tag_id     UUID NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
    PRIMARY KEY (article_id, tag_id)
);
```

Schema 9: Audit and Compliance (Chapter 10)

```
CREATE TABLE audit_log (
    id           BIGSERIAL PRIMARY KEY,
    table_name   VARCHAR(100) NOT NULL,
    record_id    VARCHAR(100) NOT NULL,
    action       VARCHAR(10) NOT NULL
                 CHECK (action IN ('INSERT','UPDATE','DELETE')),
    changed_field VARCHAR(100),
    old_value    TEXT,
    new_value    TEXT,
    changed_by   UUID REFERENCES users(id),
    changed_at   TIMESTAMP DEFAULT NOW()
);

CREATE TABLE user_stats (
    user_id      UUID PRIMARY KEY REFERENCES users(id),
    order_count  INTEGER DEFAULT 0,
    total_spent  DECIMAL(12,2) DEFAULT 0.00,
    last_order_at TIMESTAMP,
    updated_at   TIMESTAMP DEFAULT NOW()
);
```

Schema 10: Healthcare Records (Chapter 12)

```
CREATE TABLE patients (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    first_name      VARCHAR(100) NOT NULL,
    last_name       VARCHAR(100) NOT NULL,
    date_of_birth   DATE NOT NULL,
    ssn             VARCHAR(11) UNIQUE,
    email           VARCHAR(255),
    phone           VARCHAR(20),
```

```
    insurance_id    VARCHAR(50),
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE medical_records (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    patient_id      UUID NOT NULL REFERENCES patients(id),
    provider_id     UUID NOT NULL,
    diagnosis_code  VARCHAR(10) NOT NULL,
    diagnosis_text  TEXT NOT NULL,
    visit_date      DATE NOT NULL,
    notes           TEXT,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE prescriptions (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    record_id       UUID NOT NULL REFERENCES medical_records(id),
    medication      VARCHAR(200) NOT NULL,
    dosage          VARCHAR(100) NOT NULL,
    frequency       VARCHAR(100) NOT NULL,
    start_date      DATE NOT NULL,
    end_date        DATE,
    created_at      TIMESTAMP DEFAULT NOW()
);

CREATE TABLE session_logs (
    id              BIGSERIAL PRIMARY KEY,
    user_id         UUID,
    ip_address      INET,
    action          VARCHAR(100),
    resource        VARCHAR(200),
    created_at      TIMESTAMP DEFAULT NOW()
);
```

Appendix B: 100 SQL Exercises

Difficulty Levels

- **(B)** = Beginner: Basic syntax, single table operations
- **(I)** = Intermediate: JOINs, aggregations, subqueries
- **(A)** = Advanced: CTEs, complex business logic, performance considerations

Use the practice schemas from Appendix A for all exercises.

SELECT and Filtering (Exercises 1-15)

1. (B) Select all active users sorted by creation date descending.

```
SELECT * FROM users WHERE active = true ORDER BY created_at DESC;
```

2. (B) Find all products priced over $100.

```
SELECT name, price FROM products WHERE price > 100.00;
```

3. (B) Count the number of users in each role.

```
SELECT role, COUNT(*) AS count FROM users GROUP BY role;
```

4. (B) Find all orders with status 'pending' created in the last 7 days.

```
SELECT * FROM orders
WHERE status = 'pending' AND created_at > NOW() - INTERVAL '7 days';
```

5. (B) Find users whose email contains 'gmail'.

```
SELECT name, email FROM users WHERE email ILIKE '%gmail%';
```

6. (I) Find all products that are out of stock (in_stock = false) but have a price greater than zero.

```
SELECT name, sku, price FROM products
WHERE in_stock = false AND price > 0;
```

7. (I) Find orders where the delivered_at timestamp is before shipped_at (data anomaly).

```
SELECT id, shipped_at, delivered_at FROM orders
WHERE delivered_at < shipped_at;
```

8. (I) Find users who have not logged in for more than 90 days.

```
SELECT name, email, last_login_at FROM users
WHERE last_login_at < NOW() - INTERVAL '90 days'
   OR last_login_at IS NULL;
```

9. (I) List the top 10 most expensive products with their category.

```
SELECT name, price, category FROM products
ORDER BY price DESC LIMIT 10;
```

10. (A) Find products whose price changed (updated_at > created_at) but are still below the average product price.

```
SELECT name, price FROM products
WHERE updated_at > created_at
  AND price < (SELECT AVG(price) FROM products);
```

11. (B) Select all soft-deleted users (deleted_at IS NOT NULL).

```
SELECT id, name, email, deleted_at FROM users
WHERE deleted_at IS NOT NULL;
```

12. (B) Find all orders with a total between $50 and $200.

```
SELECT id, total, status FROM orders
WHERE total BETWEEN 50.00 AND 200.00;
```

13. (I) Find products that match multiple categories using IN.

```
SELECT name, category FROM products
WHERE category IN ('electronics', 'clothing', 'home');
```

14. (I) Find the most recent order for each status.

```
SELECT DISTINCT ON (status) status, id, total, created_at
FROM orders ORDER BY status, created_at DESC;
```

15. (A) Find users whose name starts with the same letter as their email domain.

```
SELECT name, email FROM users
WHERE LOWER(LEFT(name, 1)) = LOWER(LEFT(SPLIT_PART(email, '@', 2), 1));
```

JOINs (Exercises 16-35)

16. (B) List all orders with the customer's name and email.

```
SELECT o.id, o.total, u.name, u.email
FROM orders o JOIN users u ON u.id = o.user_id;
```

17. (B) Find all users who have never placed an order.

```
SELECT u.name, u.email FROM users u
LEFT JOIN orders o ON o.user_id = u.id
WHERE o.id IS NULL;
```

18. (B) List each order with its line items and product names.

```
SELECT o.id AS order_id, p.name AS product, li.quantity, li.unit_price
FROM orders o
JOIN line_items li ON li.order_id = o.id
JOIN products p ON p.id = li.product_id;
```

19. (I) Find orphaned line items (line items whose order does not exist).

```
SELECT li.id, li.order_id FROM line_items li
LEFT JOIN orders o ON o.id = li.order_id
WHERE o.id IS NULL;
```

20. (I) Find orders that have no payment record.

```
SELECT o.id, o.total, o.status FROM orders o
LEFT JOIN payments p ON p.order_id = o.id
WHERE p.id IS NULL AND o.status != 'cancelled';
```

21. (I) Show each product with its average review rating.

```
SELECT p.name, ROUND(AVG(r.rating), 1) AS avg_rating, COUNT(r.id) AS review_count
FROM products p
LEFT JOIN reviews r ON r.product_id = p.id
GROUP BY p.name;
```

22. (I) Find payments without a corresponding order (orphaned payments).

```
SELECT p.id, p.order_id, p.amount FROM payments p
LEFT JOIN orders o ON o.id = p.order_id
WHERE o.id IS NULL;
```

23. (I) List all products with their inventory quantity and warehouse location.

```
SELECT p.name, i.quantity, w.name AS warehouse, w.location
FROM products p
JOIN inventory i ON i.product_id = p.id
JOIN warehouses w ON w.id = i.warehouse_id;
```

24. (A) Full order details: user, order, items, products, payment, review status.

```
SELECT u.email, o.id AS order_id, o.status,
       p.name AS product, li.quantity, li.unit_price,
       pay.method AS payment_method, pay.status AS payment_status
FROM orders o
JOIN users u ON u.id = o.user_id
JOIN line_items li ON li.order_id = o.id
JOIN products p ON p.id = li.product_id
LEFT JOIN payments pay ON pay.order_id = o.id;
```

25. (A) Find users whose profile timezone does not match the most common timezone in the system.

```
WITH common_tz AS (
    SELECT timezone FROM user_profiles
    GROUP BY timezone ORDER BY COUNT(*) DESC LIMIT 1
)
SELECT u.name, up.timezone
FROM users u
JOIN user_profiles up ON up.user_id = u.id
WHERE up.timezone != (SELECT timezone FROM common_tz);
```

26. (B) Count orders per user.

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

27. (I) Find refunds where the refund amount exceeds the original payment.

```
SELECT r.id, r.amount AS refund_amount, p.amount AS payment_amount
FROM refunds r
JOIN payments p ON p.id = r.payment_id
WHERE r.amount > p.amount;
```

28. (I) Find products that have been ordered but never reviewed.

```
SELECT DISTINCT p.name FROM products p
JOIN line_items li ON li.product_id = p.id
LEFT JOIN reviews r ON r.product_id = p.id
WHERE r.id IS NULL;
```

29. (A) Find sellers whose products have an average rating below 3.0.

```
SELECT u.name AS seller, ROUND(AVG(r.rating), 1) AS avg_rating
FROM users u
JOIN products p ON p.seller_id = u.id
JOIN reviews r ON r.product_id = p.id
GROUP BY u.name
HAVING AVG(r.rating) < 3.0;
```

30. (A) FULL OUTER JOIN to find mismatched records between orders and payments.

```
SELECT o.id AS order_id, o.total, p.id AS payment_id, p.amount
FROM orders o
FULL OUTER JOIN payments p ON p.order_id = o.id
WHERE o.id IS NULL OR p.id IS NULL;
```

31. (I) Find inventory items below their reorder level.

```
SELECT p.name, i.quantity, i.reorder_level
FROM inventory i JOIN products p ON p.id = i.product_id
WHERE i.quantity < i.reorder_level;
```

32. (I) List each category with the number of products it contains.

```
SELECT c.name, COUNT(p.id) AS product_count
FROM categories c
LEFT JOIN products p ON p.category_id = c.id
GROUP BY c.name ORDER BY product_count DESC;
```

33. (A) Self-join: find users who share the same email domain.

```
SELECT u1.email, u2.email
FROM users u1
JOIN users u2 ON SPLIT_PART(u1.email,'@',2) = SPLIT_PART(u2.email,'@',2)
WHERE u1.id < u2.id;
```

34. (A) Find all orders where every line item's product is out of stock.

```sql
SELECT o.id FROM orders o
WHERE NOT EXISTS (
    SELECT 1 FROM line_items li
    JOIN products p ON p.id = li.product_id
    WHERE li.order_id = o.id AND p.in_stock = true
);
```

35. **(A)** CROSS JOIN: generate test combinations of products and discount percentages.

```sql
SELECT p.name, d.pct AS discount_pct,
       ROUND(p.price * (1 - d.pct/100.0), 2) AS discounted_price
FROM products p
CROSS JOIN (VALUES (5),(10),(15),(20),(25)) AS d(pct);
```

GROUP BY and Aggregation (Exercises 36-55)

36. **(B)** Count orders by status.

```sql
SELECT status, COUNT(*) FROM orders GROUP BY status ORDER BY COUNT(*) DESC;
```

37. **(B)** Calculate total revenue from completed orders.

```sql
SELECT SUM(total) AS total_revenue FROM orders WHERE status = 'delivered';
```

38. **(B)** Find the average order total.

```sql
SELECT ROUND(AVG(total), 2) AS avg_order_total FROM orders;
```

39. **(I)** Find duplicate emails in the users table.

```sql
SELECT email, COUNT(*) FROM users GROUP BY email HAVING COUNT(*) > 1;
```

40. **(I)** Revenue by month for the last 12 months.

```sql
SELECT DATE_TRUNC('month', created_at) AS month,
       SUM(total) AS revenue, COUNT(*) AS order_count
FROM orders WHERE status IN ('delivered','shipped')
  AND created_at > NOW() - INTERVAL '12 months'
GROUP BY DATE_TRUNC('month', created_at) ORDER BY month;
```

41. **(I)** Average response time per endpoint with more than 100 requests.

```
SELECT endpoint, COUNT(*) AS requests, ROUND(AVG(response_ms)) AS avg_ms
FROM request_logs
GROUP BY endpoint HAVING COUNT(*) > 100
ORDER BY avg_ms DESC;
```

42. (I) Error rate by endpoint.

```
SELECT endpoint,
       COUNT(*) AS total,
       SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END) AS errors,
       ROUND(100.0 * SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END)
             / COUNT(*), 2) AS error_pct
FROM request_logs GROUP BY endpoint
HAVING SUM(CASE WHEN status_code >= 500 THEN 1 ELSE 0 END) > 0
ORDER BY error_pct DESC;
```

43. (I) Top 5 customers by total spending.

```
SELECT u.name, u.email, SUM(o.total) AS total_spent
FROM users u JOIN orders o ON o.user_id = u.id
WHERE o.status = 'delivered'
GROUP BY u.name, u.email
ORDER BY total_spent DESC LIMIT 5;
```

44. (A) Detect double-click orders: same user, multiple orders within 60 seconds.

```
SELECT user_id, COUNT(*) AS orders_in_window,
       MIN(created_at) AS first, MAX(created_at) AS last
FROM orders
WHERE created_at > NOW() - INTERVAL '24 hours'
GROUP BY user_id
HAVING COUNT(*) > 1
   AND MAX(created_at) - MIN(created_at) < INTERVAL '60 seconds';
```

45. (A) Errors by hour, showing hour with highest error count.

```
SELECT DATE_TRUNC('hour', created_at) AS hour, COUNT(*) AS errors
FROM error_logs
WHERE created_at > NOW() - INTERVAL '7 days'
GROUP BY DATE_TRUNC('hour', created_at)
ORDER BY errors DESC LIMIT 1;
```

46. (B) Count reviews per rating (1-5 distribution).

```sql
SELECT rating, COUNT(*) AS count FROM reviews
GROUP BY rating ORDER BY rating;
```

47. **(I)** Average order value by user role.

```sql
SELECT u.role, ROUND(AVG(o.total), 2) AS avg_order
FROM users u JOIN orders o ON o.user_id = u.id
GROUP BY u.role;
```

48. **(I)** Products with highest total sales volume.

```sql
SELECT p.name, SUM(li.quantity) AS total_units_sold,
       SUM(li.quantity * li.unit_price) AS total_revenue
FROM products p JOIN line_items li ON li.product_id = p.id
JOIN orders o ON o.id = li.order_id
WHERE o.status = 'delivered'
GROUP BY p.name ORDER BY total_revenue DESC LIMIT 10;
```

49. **(A)** Month-over-month order growth percentage.

```sql
WITH monthly AS (
    SELECT DATE_TRUNC('month', created_at) AS month, COUNT(*) AS orders
    FROM orders GROUP BY DATE_TRUNC('month', created_at)
)
SELECT month, orders,
       LAG(orders) OVER (ORDER BY month) AS prev_month,
       ROUND(100.0 * (orders - LAG(orders) OVER (ORDER BY month))
             / LAG(orders) OVER (ORDER BY month), 1) AS growth_pct
FROM monthly ORDER BY month;
```

50. **(A)** Revenue cohort analysis: group users by registration month, track their spending over time.

```sql
WITH user_cohorts AS (
    SELECT id, DATE_TRUNC('month', created_at) AS cohort_month
    FROM users
)
SELECT uc.cohort_month,
       DATE_TRUNC('month', o.created_at) AS order_month,
       COUNT(DISTINCT uc.id) AS active_users,
       SUM(o.total) AS revenue
FROM user_cohorts uc
JOIN orders o ON o.user_id = uc.id
GROUP BY uc.cohort_month, DATE_TRUNC('month', o.created_at)
ORDER BY uc.cohort_month, order_month;
```

51. (B) Count products per category.

```
SELECT category, COUNT(*) FROM products GROUP BY category ORDER BY COUNT(*) DESC;
```

52. (I) Find categories with average product price above $50.

```
SELECT category, ROUND(AVG(price), 2) AS avg_price, COUNT(*) AS product_count
FROM products GROUP BY category HAVING AVG(price) > 50 ORDER BY avg_price DESC;
```

53. (I) Inventory value per warehouse.

```
SELECT w.name, SUM(i.quantity * p.price) AS inventory_value
FROM warehouses w
JOIN inventory i ON i.warehouse_id = w.id
JOIN products p ON p.id = i.product_id
GROUP BY w.name ORDER BY inventory_value DESC;
```

54. (A) Rolling 7-day average of daily order count.

```
WITH daily_orders AS (
    SELECT DATE_TRUNC('day', created_at) AS day, COUNT(*) AS orders
    FROM orders GROUP BY DATE_TRUNC('day', created_at)
)
SELECT day, orders,
       ROUND(AVG(orders) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND CURRENT ROW), 1)
       AS rolling_7day_avg
FROM daily_orders ORDER BY day DESC LIMIT 30;
```

55. (A) Percentile analysis of order totals.

```
SELECT
    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY total) AS p25,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY total) AS median,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY total) AS p75,
    PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY total) AS p95
FROM orders WHERE status = 'delivered';
```

Subqueries and CTEs (Exercises 56-70)

56. (B) Find users who have at least one completed order using EXISTS.

```
SELECT name, email FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.user_id = u.id AND o.status = 'delivered'
);
```

57. (I) Find products never ordered using NOT IN.

```
SELECT name FROM products
WHERE id NOT IN (SELECT DISTINCT product_id FROM line_items);
```

58. (I) Find users whose total spending exceeds the average using a CTE.

```
WITH user_spending AS (
    SELECT user_id, SUM(total) AS total_spent
    FROM orders WHERE status = 'delivered' GROUP BY user_id
)
SELECT u.name, us.total_spent FROM users u
JOIN user_spending us ON us.user_id = u.id
WHERE us.total_spent > (SELECT AVG(total_spent) FROM user_spending);
```

59. (I) Find the most recent order for each user.

```
SELECT u.email, o.id, o.total, o.created_at
FROM users u JOIN orders o ON o.user_id = u.id
WHERE o.created_at = (
    SELECT MAX(o2.created_at) FROM orders o2 WHERE o2.user_id = u.id
);
```

60. (I) Find products with above-average ratings.

```
WITH product_ratings AS (
    SELECT product_id, AVG(rating) AS avg_rating
    FROM reviews GROUP BY product_id
)
SELECT p.name, pr.avg_rating FROM products p
JOIN product_ratings pr ON pr.product_id = p.id
WHERE pr.avg_rating > (SELECT AVG(avg_rating) FROM product_ratings);
```

61. (A) Multi-CTE data validation query.

```
WITH orphaned_line_items AS (
    SELECT COUNT(*) AS count FROM line_items li
    LEFT JOIN orders o ON o.id = li.order_id WHERE o.id IS NULL
),
mismatched_totals AS (
    SELECT COUNT(*) AS count FROM orders o
    JOIN (SELECT order_id, SUM(quantity*unit_price) AS calc
          FROM line_items GROUP BY order_id) li ON li.order_id = o.id
    WHERE o.subtotal != li.calc
),
orphaned_payments AS (
    SELECT COUNT(*) AS count FROM payments p
    LEFT JOIN orders o ON o.id = p.order_id WHERE o.id IS NULL
)
SELECT 'orphaned_line_items' AS check, count FROM orphaned_line_items
UNION ALL
SELECT 'mismatched_totals', count FROM mismatched_totals
UNION ALL
SELECT 'orphaned_payments', count FROM orphaned_payments;
```

62. **(A)** Recursive CTE: traverse category hierarchy.

```
WITH RECURSIVE cat_tree AS (
    SELECT id, name, parent_id, 0 AS depth, name AS path
    FROM categories WHERE parent_id IS NULL
    UNION ALL
    SELECT c.id, c.name, c.parent_id, ct.depth+1, ct.path || ' > ' || c.name
    FROM categories c JOIN cat_tree ct ON c.parent_id = ct.id
)
SELECT * FROM cat_tree ORDER BY path;
```

63. **(I)** Find the second-highest order total per user.

```
WITH ranked AS (
    SELECT user_id, total,
           ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY total DESC) AS rn
    FROM orders
)
SELECT u.email, r.total AS second_highest
FROM ranked r JOIN users u ON u.id = r.user_id
WHERE r.rn = 2;
```

64. **(A)** Find users who ordered every product in a category.

```
WITH category_products AS (
    SELECT id FROM products WHERE category_id = 'electronics'
),
user_products AS (
    SELECT DISTINCT o.user_id, li.product_id
    FROM orders o JOIN line_items li ON li.order_id = o.id
    WHERE li.product_id IN (SELECT id FROM category_products)
)
SELECT u.name FROM users u
WHERE (SELECT COUNT(DISTINCT product_id) FROM user_products WHERE user_id = u.id)
    = (SELECT COUNT(*) FROM category_products);
```

65. **(I)** CTE: find products that need restocking.

```
WITH low_stock AS (
    SELECT product_id, quantity, reorder_level
    FROM inventory WHERE quantity < reorder_level
)
SELECT p.name, p.sku, ls.quantity, ls.reorder_level,
       ls.reorder_level - ls.quantity AS units_to_order
FROM low_stock ls JOIN products p ON p.id = ls.product_id
ORDER BY units_to_order DESC;
```

66. **(B)** Subquery in SELECT: show each order with the total number of orders for that user.

```
SELECT o.id, o.total,
       (SELECT COUNT(*) FROM orders o2 WHERE o2.user_id = o.user_id) AS user_total_orders
FROM orders o;
```

67. **(I)** Find orders whose total is more than 3 standard deviations above the mean.

```
WITH stats AS (
    SELECT AVG(total) AS mean, STDDEV(total) AS sd FROM orders
)
SELECT o.id, o.total FROM orders o, stats
WHERE o.total > stats.mean + 3 * stats.sd;
```

68. **(A)** CTE-based funnel analysis: registration to first order to repeat order.

```
WITH registered AS (
    SELECT id, created_at AS registered_at FROM users
),
first_order AS (
    SELECT user_id, MIN(created_at) AS first_order_at
    FROM orders GROUP BY user_id
),
repeat_order AS (
    SELECT user_id FROM orders GROUP BY user_id HAVING COUNT(*) > 1
)
SELECT
    COUNT(r.id) AS total_registered,
    COUNT(fo.user_id) AS placed_first_order,
    COUNT(ro.user_id) AS placed_repeat_order,
    ROUND(100.0 * COUNT(fo.user_id) / COUNT(r.id), 1) AS first_order_pct,
    ROUND(100.0 * COUNT(ro.user_id) / NULLIF(COUNT(fo.user_id), 0), 1) AS repeat_pct
FROM registered r
LEFT JOIN first_order fo ON fo.user_id = r.id
LEFT JOIN repeat_order ro ON ro.user_id = r.id;
```

69. (A) Window function: running total of revenue by day.

```
SELECT DATE_TRUNC('day', created_at) AS day,
       SUM(total) AS daily_revenue,
       SUM(SUM(total)) OVER (ORDER BY DATE_TRUNC('day', created_at)) AS running_total
FROM orders WHERE status = 'delivered'
GROUP BY DATE_TRUNC('day', created_at) ORDER BY day;
```

70. (A) Lateral join: top 3 orders per user.

```
SELECT u.email, top_orders.*
FROM users u
CROSS JOIN LATERAL (
    SELECT id, total, created_at FROM orders
    WHERE user_id = u.id ORDER BY total DESC LIMIT 3
) top_orders;
```

Data Manipulation (Exercises 71-80)

71. (B) Insert a new user with all required fields.

```
INSERT INTO users (name, email, password_hash)
VALUES ('Test User', 'test-ex71@test.com', 'hash_placeholder');
```

72. (B) Insert multiple products in a single statement.

```
INSERT INTO products (seller_id, name, sku, price, category_id) VALUES
    ((SELECT id FROM users WHERE role='seller' LIMIT 1), 'Widget A', 'WDG-A', 19.99, 'electronics'),
    ((SELECT id FROM users WHERE role='seller' LIMIT 1), 'Widget B', 'WDG-B', 29.99, 'electronics');
```

73. (I) Update all pending orders older than 30 days to cancelled.

```
UPDATE orders SET status = 'cancelled', updated_at = NOW()
WHERE status = 'pending' AND created_at < NOW() - INTERVAL '30 days';
```

74. **(I)** Delete test data in correct dependency order.

```
DELETE FROM line_items WHERE order_id IN (
    SELECT id FROM orders WHERE user_id IN (
        SELECT id FROM users WHERE email LIKE '%@automation.test'));
DELETE FROM orders WHERE user_id IN (
    SELECT id FROM users WHERE email LIKE '%@automation.test');
DELETE FROM users WHERE email LIKE '%@automation.test';
```

75. **(I)** Bulk generate 100 orders with random statuses.

```
INSERT INTO orders (user_id, status, subtotal, tax, total, created_at)
SELECT
    (SELECT id FROM users ORDER BY RANDOM() LIMIT 1),
    (ARRAY['pending','confirmed','shipped','delivered','cancelled'])[1 + (RANDOM()*4)::INT],
    ROUND((RANDOM() * 200 + 10)::NUMERIC, 2),
    0,
    ROUND((RANDOM() * 200 + 10)::NUMERIC, 2),
    NOW() - (RANDOM() * 90 || ' days')::INTERVAL
FROM generate_series(1, 100);
```

76. **(A)** Insert with conflict handling (upsert).

```
INSERT INTO inventory (product_id, quantity)
VALUES ('product-uuid', 50)
ON CONFLICT (product_id)
DO UPDATE SET quantity = inventory.quantity + 50,
              last_restocked = NOW();
```

77. **(B)** Update a user's role and verify the change.

```
UPDATE users SET role = 'admin' WHERE email = 'test@test.com';
SELECT role FROM users WHERE email = 'test@test.com';
-- Expected: admin
```

78. **(I)** Copy data from one table to another with transformation.

```
INSERT INTO user_stats (user_id, order_count, total_spent, last_order_at)
SELECT u.id, COUNT(o.id), COALESCE(SUM(o.total), 0), MAX(o.created_at)
FROM users u LEFT JOIN orders o ON o.user_id = u.id
GROUP BY u.id
ON CONFLICT (user_id) DO UPDATE SET
    order_count = EXCLUDED.order_count,
    total_spent = EXCLUDED.total_spent,
    last_order_at = EXCLUDED.last_order_at;
```

79. **(A)** Conditional update based on aggregate.

```
UPDATE products SET active = false
WHERE id IN (
    SELECT p.id FROM products p
    LEFT JOIN line_items li ON li.product_id = p.id
    LEFT JOIN orders o ON o.id = li.order_id
        AND o.created_at > NOW() - INTERVAL '1 year'
    GROUP BY p.id
    HAVING COUNT(o.id) = 0
);
```

80. **(A)** Generate realistic test data with correlated values.

```
WITH new_users AS (
    INSERT INTO users (id, name, email, password_hash, role, created_at)
    SELECT
        'gen-' || i,
        'User ' || i,
        'user' || i || '@generated.test',
        'hash_' || i,
        CASE WHEN i % 10 = 0 THEN 'admin'
             WHEN i % 5 = 0 THEN 'seller'
             ELSE 'customer' END,
        NOW() - (i || ' days')::INTERVAL
    FROM generate_series(1, 50) i
    RETURNING id, created_at
)
INSERT INTO orders (user_id, status, subtotal, tax, total, created_at)
SELECT nu.id, 'delivered',
       ROUND((RANDOM()*100+10)::NUMERIC,2) AS subtotal,
       0 AS tax,
       ROUND((RANDOM()*100+10)::NUMERIC,2) AS total,
       nu.created_at + (RANDOM()*30 || ' days')::INTERVAL
FROM new_users nu;
```

Constraints and Schema (Exercises 81-90)

81. **(B)** Test that inserting a duplicate email fails.

```
-- First insert succeeds:
INSERT INTO users (name, email, password_hash)
VALUES ('User A', 'unique-test@test.com', 'hash');
-- Second insert should raise UniqueViolation:
INSERT INTO users (name, email, password_hash)
VALUES ('User B', 'unique-test@test.com', 'hash');
```

82. **(B)** Test that inserting a NULL name fails.

```
INSERT INTO users (name, email, password_hash)
VALUES (NULL, 'null-name@test.com', 'hash');
-- Expected: NotNullViolation
```

83. **(I)** Test the CHECK constraint on order status.

```
INSERT INTO orders (user_id, status, subtotal, tax, total)
VALUES ('some-user-id', 'nonexistent_status', 0, 0, 0);
-- Expected: CheckViolation
```

84. **(I)** Test that negative prices are rejected.

```
INSERT INTO products (seller_id, name, sku, price)
VALUES ('some-seller-id', 'Bad Product', 'BAD-1', -5.00);
-- Expected: CheckViolation
```

85. **(I)** Test foreign key constraint: order referencing non-existent user.

```
INSERT INTO orders (user_id, status, subtotal, tax, total)
VALUES ('nonexistent-uuid', 'pending', 10, 0, 10);
-- Expected: ForeignKeyViolation
```

86. **(A)** Query all constraints on the orders table.

```
SELECT con.conname AS constraint_name,
       con.contype AS type,
       pg_get_constraintdef(con.oid) AS definition
FROM pg_constraint con
JOIN pg_class rel ON rel.oid = con.conrelid
WHERE rel.relname = 'orders';
```

87. **(A)** Test CASCADE delete: deleting an order should delete its line items.

```
INSERT INTO orders (id, user_id, status, subtotal, tax, total)
VALUES ('cascade-test-order', 'existing-user-id', 'pending', 10, 0, 10);
INSERT INTO line_items (order_id, product_id, quantity, unit_price)
VALUES ('cascade-test-order', 'existing-product-id', 1, 10.00);

DELETE FROM orders WHERE id = 'cascade-test-order';

SELECT COUNT(*) FROM line_items WHERE order_id = 'cascade-test-order';
-- Expected: 0
```

88. (A) Test that the transfers table CHECK prevents self-transfers.

```
INSERT INTO transfers (from_account_id, to_account_id, amount)
VALUES ('account-1', 'account-1', 50.00);
-- Expected: CheckViolation (from_account_id != to_account_id)
```

89. (A) Test the composite unique constraint on reviews (one review per user per product).

```
INSERT INTO reviews (product_id, user_id, rating, title)
VALUES ('product-1', 'user-1', 5, 'Great!');
INSERT INTO reviews (product_id, user_id, rating, title)
VALUES ('product-1', 'user-1', 4, 'Revised opinion');
-- Expected: UniqueViolation
```

90. (A) Verify default values for all columns that have them on the users table.

```
INSERT INTO users (name, email, password_hash)
VALUES ('Defaults Test', 'defaults@test.com', 'hash')
RETURNING role, active, email_verified, created_at, updated_at;
-- Expected: role='customer', active=true, email_verified=false,
-- created_at ~ NOW(), updated_at ~ NOW()
```

Advanced Topics (Exercises 91-100)

91. (I) Write a Python test that verifies the updated_at trigger.

```
def test_updated_at_trigger(db):
    cursor = db.cursor()
    cursor.execute("""
        INSERT INTO users (id, name, email, password_hash)
        VALUES ('trigger-91', 'Trigger Test', 'trigger91@test.com', 'hash')
        RETURNING created_at, updated_at
    """)
    c, u = cursor.fetchone()
    assert c == u
    import time; time.sleep(1)
    cursor.execute("UPDATE users SET name='Changed' WHERE id='trigger-91'")
    cursor.execute("SELECT updated_at FROM users WHERE id='trigger-91'")
    assert cursor.fetchone()[0] > u
```

92. (I) Write a migration test: add a column, verify data preserved.

```
def test_add_column_migration(migration_db):
    conn, db_name = migration_db
    cursor = conn.cursor()
    cursor.execute("INSERT INTO users (name,email,password_hash) "
                   "VALUES ('Mig','mig@t.com','h')")
    conn.commit()
    cursor.execute("ALTER TABLE users ADD COLUMN nickname VARCHAR(50)")
    cursor.execute("SELECT name FROM users WHERE email='mig@t.com'")
    assert cursor.fetchone()[0] == 'Mig'
```

93. **(A)** Write a Redis cache invalidation test.

```
def test_user_cache_invalidation(redis_client, api_client):
    api_client.get("/users/1")
    assert redis_client.get("user:1") is not None
    api_client.put("/users/1", json={"name": "New"})
    assert redis_client.get("user:1") is None
```

94. **(A)** Write a MongoDB aggregation test.

```
def test_order_stats_pipeline(mongo_db):
    mongo_db.orders.insert_many([
        {"user": "a", "total": 10, "status": "completed"},
        {"user": "a", "total": 20, "status": "completed"},
        {"user": "b", "total": 30, "status": "completed"},
    ])
    result = list(mongo_db.orders.aggregate([
        {"$match": {"status": "completed"}},
        {"$group": {"_id": "$user", "sum": {"$sum": "$total"}}}
    ]))
    sums = {r["_id"]: r["sum"] for r in result}
    assert sums["a"] == 30
    assert sums["b"] == 30
```

95. **(A)** Write a data masking verification test for emails.

```
def test_masked_emails(original_db, masked_db):
    orig = set(r[0] for r in original_db.cursor().execute(
        "SELECT email FROM users").fetchall())
    masked = set(r[0] for r in masked_db.cursor().execute(
        "SELECT email FROM users").fetchall())
    assert len(orig & masked) == 0, "Original emails found in masked data"
    for e in masked:
        assert "@" in e, f"Invalid masked email: {e}"
```

96. **(A)** Write a GDPR deletion test across four tables.

```
def test_gdpr_erasure(api, db):
    user = api.post("/users", json={"name":"GDPR","email":"gdpr@t.com"}).json()
    uid = user["id"]
    api.post(f"/users/{uid}/orders", json={"total": 10})
    api.delete(f"/users/{uid}/gdpr-delete")
    cursor = db.cursor()
    for table, col in [("users","id"),("orders","user_id"),
                       ("audit_log","user_id"),("session_logs","user_id")]:
        cursor.execute(f"SELECT COUNT(*) FROM {table} WHERE {col}=%s",(uid,))
        assert cursor.fetchone()[0] == 0, f"Data remains in {table}"
```

97. (A) Write a comprehensive data validation CTE query for the ShopFast schema.

```
WITH checks AS (
    SELECT 'orphan_line_items' AS name,
           (SELECT COUNT(*) FROM line_items li LEFT JOIN orders o ON o.id=li.order_id WHERE o.id IS NULL) AS failures
    UNION ALL SELECT 'orphan_payments',
           (SELECT COUNT(*) FROM payments p LEFT JOIN orders o ON o.id=p.order_id WHERE o.id IS NULL)
    UNION ALL SELECT 'mismatched_totals',
           (SELECT COUNT(*) FROM orders o JOIN
               (SELECT order_id, SUM(quantity*unit_price) AS c FROM line_items GROUP BY order_id) s
               ON s.order_id=o.id WHERE o.subtotal!=s.c)
    UNION ALL SELECT 'invalid_ratings',
           (SELECT COUNT(*) FROM reviews WHERE rating<1 OR rating>5)
    UNION ALL SELECT 'negative_inventory',
           (SELECT COUNT(*) FROM inventory WHERE quantity<0)
)
SELECT * FROM checks WHERE failures > 0;
-- Expected: no rows (all checks pass)
```

98. (A) Write a stored procedure performance test.

```
def test_calculate_total_performance(db):
    import time
    cursor = db.cursor()
    # Create order with 1000 line items
    cursor.execute("INSERT INTO orders (id,user_id,status,subtotal,tax,total) "
                   "VALUES ('perf-order','some-user','pending',0,0,0)")
    for i in range(1000):
        cursor.execute("INSERT INTO line_items (order_id,product_id,quantity,unit_price) "
                       "VALUES ('perf-order','some-product',1,%.2f)" % (i*0.01))
    start = time.time()
    cursor.execute("SELECT calculate_order_total('perf-order')")
    duration = time.time() - start
    assert duration < 1.0, f"Took {duration:.3f}s (limit: 1s)"
```

99. (A) Write a transaction isolation test.

```
def test_transaction_isolation(db_factory):
    conn1 = db_factory()
    conn2 = db_factory()
    conn1.autocommit = False
    conn2.autocommit = False

    cur1 = conn1.cursor()
    cur2 = conn2.cursor()

    cur1.execute("INSERT INTO users (id,name,email,password_hash) "
                 "VALUES ('iso-test','Iso','iso@t.com','h')")

    # Connection 2 should NOT see uncommitted data
    cur2.execute("SELECT COUNT(*) FROM users WHERE id='iso-test'")
    assert cur2.fetchone()[0] == 0

    conn1.commit()

    # Now connection 2 should see it (after refreshing its snapshot)
    conn2.commit()  # End current transaction to get fresh snapshot
    cur2.execute("SELECT COUNT(*) FROM users WHERE id='iso-test'")
    assert cur2.fetchone()[0] == 1
```

100. **(A)** Write a comprehensive end-to-end database test.

```
def test_full_order_lifecycle(db, api_client):
    """Test the complete order lifecycle at the database level."""
    # 1. Create user
    user = api_client.post("/users", json={
        "name": "E2E Test", "email": "e2e@test.com"
    }).json()

    # 2. Verify user in DB with defaults
    cur = db.cursor(cursor_factory=DictCursor)
    cur.execute("SELECT * FROM users WHERE id=%s", (user["id"],))
    db_user = cur.fetchone()
    assert db_user["role"] == "customer"
    assert db_user["active"] is True

    # 3. Create order
    order = api_client.post("/orders", json={
        "user_id": user["id"],
        "items": [{"product_id": "p1", "quantity": 2}]
    }).json()

    # 4. Verify order total matches line items
    cur.execute("""
        SELECT o.subtotal, SUM(li.quantity * li.unit_price) AS calc
        FROM orders o JOIN line_items li ON li.order_id = o.id
        WHERE o.id = %s GROUP BY o.subtotal
    """, (order["id"],))
```

```
    row = cur.fetchone()
    assert row["subtotal"] == row["calc"]

    # 5. Verify inventory decremented
    cur.execute("SELECT quantity FROM inventory WHERE product_id='p1'")
    inv = cur.fetchone()
    # quantity should have decreased by 2

    # 6. Process payment
    api_client.post(f"/orders/{order['id']}/pay", json={"method": "credit_card"})
    cur.execute("SELECT status FROM payments WHERE order_id=%s", (order["id"],))
    assert cur.fetchone()["status"] == "completed"

    # 7. Verify audit trail
    cur.execute("""
        SELECT COUNT(*) FROM audit_log
        WHERE table_name='orders' AND record_id=%s
    """, (order["id"],))
    assert cur.fetchone()[0] > 0
```

Appendix C: Glossary

ACID – Atomicity, Consistency, Isolation, Durability. The four properties that guarantee database transactions are processed reliably.

Aggregation – The process of combining multiple rows into a single summary row using functions like COUNT, SUM, AVG, MIN, MAX.

Audit Trail – A chronological record of changes made to database records, typically implemented with triggers that log old and new values.

BSON – Binary JSON. The document format used by MongoDB to store data internally.

CASCADE – A foreign key action that automatically deletes or updates child records when the parent record is deleted or updated.

CHECK Constraint – A rule that limits the values allowed in a column (e.g., `CHECK (price >= 0)`).

CTE (Common Table Expression) – A named temporary result set defined using the WITH keyword, used to simplify complex queries.

Correlated Subquery – A subquery that references columns from the outer query and is executed once for each row of the outer query.

CROSS JOIN – A JOIN that produces the Cartesian product of two tables (every combination of rows).

Cursor – A database object used to retrieve rows from a result set one at a time in application code.

Data Masking – The process of transforming sensitive data into realistic but fake data for use in non-production environments.

DDL (Data Definition Language) – SQL commands that define database structure: CREATE, ALTER, DROP.

DML (Data Manipulation Language) – SQL commands that manipulate data: SELECT, INSERT, UPDATE, DELETE.

DictCursor – A cursor type (in psycopg2) that returns rows as dictionaries instead of tuples, allowing column access by name.

Document Database – A NoSQL database that stores data as JSON-like documents (e.g., MongoDB).

Eventual Consistency – A consistency model where reads may return stale data, but all replicas will eventually converge to the same state.

Foreign Key – A constraint that enforces a link between data in two tables, ensuring referential integrity.

FULL OUTER JOIN – A JOIN that returns all rows from both tables, with NULL values where there is no match.

GDPR (General Data Protection Regulation) – European Union regulation governing the collection, storage, and processing of personal data.

GROUP BY – A SQL clause that groups rows sharing common values, typically used with aggregate functions.

GSI (Global Secondary Index) – In DynamoDB, an index with a partition key and optional sort key different from the base table.

HAVING – A SQL clause that filters groups created by GROUP BY, similar to WHERE but for aggregated data.

HIPAA – Health Insurance Portability and Accountability Act. US regulation governing the protection of health information.

Index – A database structure that improves query performance by providing fast lookup paths to data.

INNER JOIN – A JOIN that returns only rows where there is a match in both tables.

Key-Value Store – A NoSQL database that stores data as simple key-value pairs (e.g., Redis).

LEFT JOIN – A JOIN that returns all rows from the left table and matching rows from the right table (NULL if no match).

Migration – A versioned script that modifies the database schema (adding columns, creating tables, etc.).

NOT NULL Constraint – A rule that prevents a column from containing NULL values.

NoSQL – A category of databases that do not use the traditional relational table/row/column model. Includes document stores, key-value stores, wide-column stores, and graph databases.

Orphaned Record – A record that references a parent record that no longer exists, indicating a data integrity issue.

Parameterized Query – A query where user-supplied values are passed as parameters rather than concatenated into the SQL string, preventing SQL injection.

PCI DSS (Payment Card Industry Data Security Standard) – Security standard for organizations that handle credit card information.

PRIMARY KEY – A constraint that uniquely identifies each row in a table. Cannot be NULL and must be unique.

RESTRICT – A foreign key action that prevents deletion of a parent record while child records exist.

Rollback – The process of undoing all changes made within a transaction, returning the database to its previous state.

Savepoint – A named point within a transaction to which you can roll back without rolling back the entire transaction.

Scalar Subquery – A subquery that returns exactly one value (one row, one column).

Schema – The structure of a database: tables, columns, types, constraints, indexes, and relationships.

Soft Delete – A pattern where records are not physically deleted but marked as deleted (typically by setting a `deleted_at` timestamp).

SQL Injection – A security vulnerability where malicious SQL code is inserted into a query through unsanitized user input.

Stored Procedure – A pre-compiled SQL program stored in the database that can be called by name, often containing business logic.

Subquery – A query nested inside another query, used to provide values for the outer query's conditions.

Transaction – A sequence of database operations that are treated as a single unit of work (all succeed or all fail).

Trigger – A database object that automatically executes a function when a specified event (INSERT, UPDATE, DELETE) occurs on a table.

TTL (Time To Live) – The duration for which a cached value remains valid before it expires and is automatically deleted.

UNIQUE Constraint – A rule that ensures all values in a column (or combination of columns) are distinct.

Upsert – An operation that inserts a new record or updates an existing one if a conflict (e.g., duplicate key) is detected. In PostgreSQL: `INSERT ... ON CONFLICT ... DO UPDATE`.

Window Function – A function that performs a calculation across a set of rows related to the current row, without collapsing them into a single output row (e.g., ROW_NUMBER, LAG, SUM OVER).

Final Words

The database is where truth lives. No matter how polished the UI or how well-designed the API, the database reveals what actually happened. A QA engineer who can query the database directly transforms from someone who reports symptoms into someone who diagnoses root causes.

The skills in this book – from basic SELECT queries to migration testing, from constraint verification to data masking – form a complete toolkit for database-level quality assurance. Practice them consistently, and you will find bugs faster, write more reliable tests, and earn the trust of your development team.

> **Interview Talking Point**: "I use SQL as a verification layer in my test automation – after an API call creates a record, I query the database to confirm data was persisted with the right defaults and constraints. I am comfortable with JOINs, GROUP BY, subqueries, and transaction-based test data management. I also test database migrations separately: apply, verify data integrity, roll back, and verify again. For NoSQL stores like Redis and MongoDB, I test TTL behavior, document structure, and cache invalidation."

SQL and Database Testing: Query Your Way to Quality A comprehensive textbook for QA engineers, developers, and data professionals.

About This Sample

You have just read **Chapter 1 of SQL and Database Testing: Query Your Way to Quality** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-15-sql-database-testing/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.