ADDALY · THE AI CLASSROOM

Data, SQL and Getting to the Answer

Most AI problems turn out to be data problems. This is where you learn to ask a database a question and defend the answer.

6 modules · 66 lessons · 29 figures · 6 case studies · 48,377 words · about 9 hours

Dr Mustafa Mun
Author and editor

ABOUT THIS BOOK

Data, SQL and Getting to the Answer, published by Addaly, 2026. Author and editor: **Dr Mustafa Mun**.

The manuscript was drafted with the assistance of a large language model and was edited and published under his name. That is stated plainly here rather than left to be discovered, because a reader is entitled to know how a book they are trusting was made.

This book is the printed form of the course of the same name at addaly.com/learn/data-and-sql. The website is the living version and is corrected first; where the two differ, the website is right.

Free to read, free to download, free to print, and free to teach from. There is no paid edition and there never will be. If you paid for this, somebody has sold you something that was given away.

Every diagram in it is drawn from data by code, not generated as a picture, so the numbers on the page are the numbers in the argument. Answers to every exercise, test and examination question are at the back, with a note on why each wrong option attracts people — those notes are the teaching, not the marking.

Contents

1. Tables, types and the question you are actually asking

Set up a working database on your own machine, read an unfamiliar schema well enough to say what one row of each table means and which columns join to which, and state the grain of any result you are about to compute before you compute it

2. Putting tables together

Combine any set of related tables without inflating a total: diagnose a join key that silently fails to match, recognise and defuse the fan trap between two child tables, choose between a join, a subquery, an EXISTS and a set operation, and generate the rows for periods in which nothing happened

1. Joins: reason about them, do not memorise them
2. The four joins, and what each one is for
3. When the join key looks right and matches nothing
4. Many-to-many, and the table in the middle
5. Two child tables, one very wrong total
6. Joining a table to itself
7. UNION, INTERSECT and EXCEPT
8. Subqueries: a query inside a query
9. CTEs: naming the steps
10. Generating the rows that are not there
11. Case study: Twelve lakh rupees that were never spent
12. Module test

3. Summaries, and the piles behind them

Compute any count, sum, average, percentile or rate at a stated grain with NULLs and time boundaries handled deliberately, produce subtotals and a wide report from long data in one query, and reconcile every total against a figure you already trust before sending it

1. GROUP BY: making piles and asking about each one
2. NULL, and why your counts are wrong
3. Averages: the four ways AVG misleads
4. Percentiles: describing a distribution instead of its mean
5. Grouping by day, week and month without lying about the boundaries
6. Rates: a numerator, a denominator, and the same grain for both
7. Listing the members of a pile: string_agg and array_agg
8. The row behind the MAX: which order was the biggest
9. Subtotals and grand totals in one query: ROLLUP and GROUPING
10. Wide and long: pivoting rows to columns and back
11. Reconciling: making a total tie out before you send it
12. Case study: The waiting time everybody was happy with
13. Module test

4. Windows, sequences and time

Answer any "compared with what came before" or "which one within each group" question — rank in group, top N per group, running total, change from the previous row, moving average, streaks, sessions and cohort retention — with a window function whose frame you can state, and join a fact to the value that was true at its own moment in time

1. A window: GROUP BY that keeps the rows
2. ROW_NUMBER, RANK and DENSE_RANK: three answers to "what position"
3. Top N per group in two lines
4. Running totals, and the frame you did not know you set
5. LAG and LEAD: the previous row, and the change since it
6. Moving averages: seven rows is not seven days
7. Streaks and sessions: gaps and islands
8. Cohorts and retention: how a month of new customers behaves afterwards
9. Dates, intervals and the two kinds of timestamp
10. As-of joins: the value that was true at the time
11. Case study: The retention curve that was falling because it was young
12. Module test

5. Data that arrives dirty

Load an exported CSV with mixed encodings, ambiguous dates and duplicate rows into a typed, deduplicated table by re-runnable query with every dropped row counted, decide and document what each missing value means, and produce a feature table for a model in which no column knows anything that happened after its row's timestamp

1. Cleaning real data without destroying it
2. Getting a file in without losing a row
3. Stage as text, then cast in a query you can re-run
4. Parsing dates that were typed by people
5. Text that looks identical and is not
6. Duplicates: finding them, and deciding which row survives
7. Assertions: queries that return no rows when everything is right
8. Missing values: NULL, empty, N/A, zero and minus one
9. Categories: canonical values, mapping tables and encoding for a model
10. A feature table: one row per entity, as of a moment
11. Leakage: when a feature knows the future
12. Why a query is slow, and what an index does
13. Case study: The eleven hundred rows that were not random
14. Module test

6. Changing data safely

Design and create a small schema whose keys and constraints refuse bad data, insert, update and delete inside a transaction you have rehearsed and can roll back, explain what happens when two writers touch the same row, alter a table that is in use without an outage, and give an analyst or an AI a read-only path into the database

1. CREATE TABLE: a constraint is a test that runs on every write
2. INSERT, and the INSERT you can run twice
3. UPDATE and DELETE: rehearse it as a SELECT
4. Transactions: making a mistake reversible
5. Two people, one row: lost updates and how the database prevents them
6. Changing a table that is in use
7. Views: a saved question, not a saved answer
8. JSON in a column: when it helps and what it costs
9. Keeping history: soft deletes, audit rows and who changed what
10. Who may do what: roles, GRANT and a read-only door
11. From a paragraph to a schema: a tuition centre
12. Case study: Four thousand marks, and the update that could not be rehearsed
13. Module test

Data, SQL and Getting to the Answer — final examination

The paper that opens the next course. Answers to everything follow it.

MODULE 1

Tables, types and the question you are actually asking

Before any clever query there is a table, and a table is a set of promises: one kind of thing per row, a declared type per column, a key that survives a name change. This block gets a real database onto the machine you already own, teaches the vocabulary a schema is written in, and ends with the single sentence that prevents most wrong answers in the rest of the course.

> BY THE END OF THIS MODULE YOU CAN
>
> Set up a working database on your own machine, read an unfamiliar schema well enough to say what one row of each table means and which columns join to which, and state the grain of any result you are about to compute before you compute it

1 · 7 min

When a spreadsheet stops working

> THE ONE THING TO KEEP
>
> A database stores each fact once and enforces rules; a spreadsheet stores copies and enforces nothing.

The sheet that worked for two years

Rukmini runs a tuition centre in Hyderabad. She keeps one spreadsheet: one row per payment, with the student's name, phone number, class, month,

amount and payment mode. For two years and about 3,000 rows it is fine. Then four things happen in the same month.

A student's family changes their phone number. That number appears in 34 rows. Rukmini fixes 31 of them.

Her assistant types "Class 9" in one row where everyone else has typed "Grade 9". The class-wise total is now wrong by eleven students, and nothing looks broken.

The file takes forty seconds to open. Two people edit it at the same time and one of them loses a morning's work.

A payment gets entered for a student who left last year. Nothing stops it. Nothing even notices.

Only one of these is a size problem. The other three are the same problem wearing different clothes: **the same fact is stored in many places, and nothing forces the copies to agree.**

One kind of thing per table

A database splits that sheet into tables. A table holds one kind of thing, one row per thing, with named columns that have declared types.

The same four problems, in a sheet and in two tables

One sheet, one row per payment

- The phone number appears in 34 rows. Thirty-one of them get corrected
- One row says Class 9 where the rest say Grade 9, and the class-wise total is quietly wrong
- A payment is entered for a student who left last year, and nothing notices
- Two people edit at once and one of them loses a morning

students and payments, two tables

- The phone number lives in one row. Change it once and every payment follows
- class_level is an int, so Grade 9 is refused at the moment it is typed
- REFERENCES students(id) rejects a payment for a student who does not exist
- Forty people write at once and nobody overwrites anybody

Only the fourth of these is a size problem. The other three are one problem — the same fact stored in several places, with nothing forcing the copies to agree — and the right-hand column removes it by storing each fact once and refusing the rows that break a rule.

```
CREATE TABLE students (
  id          bigserial     PRIMARY KEY,
  full_name   text          NOT NULL,
  phone       text,
  class_level int           NOT NULL,
  joined_on   date          NOT NULL
);

CREATE TABLE payments (
  id          bigserial     PRIMARY KEY,
  student_id  bigint        NOT NULL REFERENCES students(id),
  paid_on     date          NOT NULL,
  amount      numeric(10,2) NOT NULL,
  method      text          NOT NULL
);
```

Read what each line actually buys.

`PRIMARY KEY` means every student has one permanent id. Names change, spellings vary, two families are both called Reddy. The id is the thing the rest of the system points at.

The phone number now lives in exactly one row. Change it once and every payment ever made is instantly attached to the new number, because payments never stored a phone number at all.

`NOT NULL` means the database refuses a student with no name. Not a warning. A refusal.

`REFERENCES students(id)` means a payment for student 9,999 who does not exist is rejected. This is the rule the spreadsheet could never enforce, and it is the reason databases stay trustworthy while spreadsheets rot.

`numeric(10,2)` rather than a floating-point type, because money in binary floating point drifts. Rupees, naira and pesos all deserve exact arithmetic.

`int` on `class_level` means "Grade 9" typed into that column is an error, immediately, at the moment someone types it. Not a silent wrong total three months later.

What you get that a sheet cannot give you

Types and constraints. Bad data is rejected at the door instead of being cleaned up later. Later never comes.

Transactions. A refund that moves money out of one row and into another either fully happens or fully does not. There is no state where half of it landed.

Concurrency. Forty people can write at once. Nobody overwrites anybody.

Scale. Ten million rows, answered in milliseconds, with an index. You will see how in a later lesson.

A query language. This is the big one. A spreadsheet answers the questions you designed it to answer. A database answers questions you had not thought of when you built it. "Which students who joined before June have paid every month since?" is one query against the tables above and impossible against the sheet.

The honest part

A spreadsheet is not a failure mode. Under a few thousand rows, with one person writing and a throwaway question to answer, a spreadsheet is faster than a database and you should use it. Anyone who tells you otherwise is selling something.

The moment to move is specific: when more than one person writes, or when the same fact appears in two rows, or when you need the data to still be trustworthy in a year. Rukmini crossed all three lines in one month and did not notice any of them, because crossing them does not produce an error message. It produces a number that is quietly wrong.

BEFORE YOU MOVE ON

A small shop keeps one sheet with a row per sale, including the customer's name, phone and full address on every row. A regular customer moves house. What is the real problem this exposes?

A. Her address now sits in many rows and nothing forces those rows to agree with each other
B. The sheet will eventually grow too large to open, which is the point at which a database becomes necessary
C. Addresses are being stored as free text when they should be structured into separate fields
D. Each sale has no unique identifier, so there is no way to tell two sales apart

2 · 8 min

Getting a database onto the machine you have

THE ONE THING TO KEEP

You do not need a server to practise SQL: SQLite is one file, DuckDB queries a CSV where it lies, and Postgres is what to install when a job advert names it.

You do not need a server, a cloud account or a credit card

Most SQL courses assume you already have a database. Most people do not, and the setup step is where a lot of learners quietly stop. So here are four honest options, in order of how little they ask of you. All are free, all run offline, and none of them expire.

SQLite: a database that is one file

SQLite is the most widely deployed database in the world. It is inside your phone, your browser and your car. A whole database is a single file you can email to somebody.

If you have Python, you already have it:

```
python3 -c "import sqlite3;
sqlite3.connect('school.db').close()"
sqlite3 school.db
```

If the `sqlite3` command is missing, install it (`apt install sqlite3`, `brew install sqlite`), or use **DB Browser for SQLite** — a free graphical tool with a table view, an import-CSV button and a query tab. Job ads say Microsoft Access or Excel; DB Browser does the same teaching job for nothing, on Windows, macOS and Linux.

```
CREATE TABLE students (id integer primary key, full_name text,
class_level int);
INSERT INTO students (full_name, class_level) VALUES ('Rukmini
R', 9), ('Ade O', 10);
SELECT * FROM students;
```

The honest limitation, and it matters for this course: **SQLite does not really enforce types.** Its columns have type *affinity*, not type constraints, so `INSERT INTO students VALUES (1, 'Ade', 'Grade Nine')` succeeds. Everything else you learn transfers; that one protection does not exist until you turn on strict tables (`CREATE TABLE ... STRICT`) in a recent version. Do turn it on.

DuckDB: SQLite's analytical cousin

DuckDB is a single binary that queries CSV and Parquet files directly, without loading them first.

```
duckdb -c "SELECT city, count(*) FROM read_csv_auto('pay-
ments.csv') GROUP BY city"
```

That one line is the fastest route from a downloaded export to a real answer, and it is what most of this course's cleaning work is comfortable in. It is built

for reading large tables fast, not for many people writing at once — so it is a superb analysis tool and the wrong choice for an application's live data.

PostgreSQL: the one the jobs ask for

Postgres is the reference implementation for almost everything in this course. Install it locally:

```
# Debian / Ubuntu
sudo apt install postgresql
sudo -u postgres createdb school
psql school

# macOS
brew install postgresql@17 && brew services start postgresql@17
createdb school && psql school
```

Or run it in Docker without installing anything permanent:

```
docker run --name pg -e POSTGRES_PASSWORD=dev -p 5432:5432 -d
postgres:17
psql "postgres://postgres:dev@localhost:5432/postgres"
```

If your laptop cannot run either, several providers give a small Postgres database on a free tier that never asks for a card. That is enough for every query in this course.

If all you have is a phone

This is a real constraint for a lot of readers, so it gets a real answer rather than an apology. On Android, Termux installs `sqlite3` and `python` from a package manager and gives you a proper shell. On any phone, a browser-based SQL sandbox will run SQLite compiled to WebAssembly entirely on the device. Neither is comfortable for eight-hour days. Both are entirely sufficient for learning to think in tables, which is the part that transfers.

Get some data in

Empty databases teach nothing. Two moves:

```
-- Postgres, from a file on the server's disk
COPY payments FROM '/tmp/payments.csv' WITH (FORMAT csv, HEADER
true);

-- Postgres, from a file on your own machine, via psql
\copy payments FROM 'payments.csv' WITH (FORMAT csv, HEADER
true)
```

That backslash matters. `COPY` runs as the database server and looks on the server's disk. `\copy` runs in your client and reads your file. Almost every "permission denied" on a first import is this distinction.

In SQLite: `.mode csv` then `.import payments.csv payments`. In DuckDB you can skip importing entirely.

Good public data to practise on: a national statistics office's open data portal, your city's transport authority, or the World Bank's open datasets. Prefer data you have some feel for — you will spot a wrong answer in data about your own city and you will not spot it in a synthetic sales table.

What to actually do now

Create one database. Load one CSV of at least a few thousand rows. Run `SELECT * FROM yourtable LIMIT 20` and read it. That is the whole exercise, and doing it once removes the friction that otherwise sits between you and every remaining lesson in this course.

BEFORE YOU MOVE ON

A colleague sends you a 3-million-row CSV export and asks for the number of rows per branch by tomorrow morning. You have an ordinary laptop and no database installed. Which route gets a correct answer with the least ceremony?

A. Set up Postgres, create a table with the right column types, import the CSV with COPY, then run the count against the indexed table.
B. Query the CSV directly with DuckDB, since it reads the file in place and is built for scanning large tables.
C. Open the file in a spreadsheet and use a pivot table.
D. Load it into SQLite, because SQLite is the most widely deployed database and works everywhere.

3 · 8 min

What a column type actually buys you

THE ONE THING TO KEEP

A column type is a refusal the database makes on your behalf; choose numeric for money, because a binary float cannot hold 0.10 exactly and the errors add up.

A type is a refusal

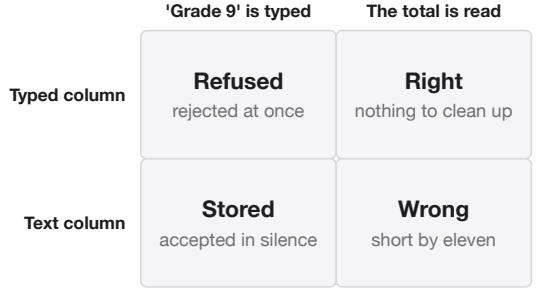
Every column in a table has a declared type, and the type is not documentation. It is a rule the database enforces at the moment of writing. `class_level int` does not mean "we intend to put numbers here". It means an attempt to store `Grade Nine` fails, loudly, in front of the person who typed it, instead of quietly three months later in a total.

That is the whole value proposition. A type moves an error from the moment of *reading* — where it is expensive, subtle and yours — to the moment of *writing*, where it is cheap and belongs to whoever caused it.

The types you will actually use

Whole numbers. `int` holds roughly ±2.1 billion. `bigint` holds ±9.2 quintillion. Use `int` for a class level or a quantity, `bigint` for anything that counts events or acts as an id. Tables have overflowed a 32-bit id column in production more than once; the fix is a long outage.

A type moves the error from read time to write time



The typed column produces an error in front of the person who caused it, at the moment they caused it. The text column produces no error at all: it produces a class-wise total that is short by eleven students three months later, and looks perfectly ordinary.

Money and exact decimals. `numeric(10,2)` in Postgres, `DECIMAL(10,2)` elsewhere. Ten digits total, two after the point. It is stored exactly.

Approximate numbers. `real` and `double precision` are binary floating point. They are fast, they are right for sensor readings and physics, and they are wrong for money:

```
SELECT 0.1::float + 0.2::float = 0.3::float;   -- false
SELECT 0.1::numeric + 0.2::numeric = 0.3::numeric;  -- true
```

That first line is not a bug in the database. One tenth has no exact representation in binary, exactly as one third has none in decimal. Sum a hundred thousand float prices and the total will be a few paise or cents away from the truth, in a direction nobody can predict. Finance teams notice.

Text. In Postgres, `text` and `varchar(n)` are the same machinery; `varchar(n)` merely adds a length check. Use `text` unless a length limit is a genuine business rule. A `varchar(20)` on a name column is not a rule, it is a guess about the world's names, and it will be wrong.

Dates and times. `date` for a calendar day. `timestamptz` for a moment in time. `timestamp` (without time zone) is the one that silently causes trouble, and the next module gives it a lesson of its own.

Boolean. `true`, `false`, and — this is the part people forget — `NULL`. A three-state column pretending to be two-state is a recurring source of wrong counts.

Identifiers. `bigserial` or `generated always as identity` for an auto-incrementing id; `uuid` when ids must be generated in several places without collisions.

The constraints that go next to the type

A type says what shape the value has. Constraints say what values are allowed.

```
CREATE TABLE payments (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  student_id  bigint       NOT NULL REFERENCES students(id),
  paid_on     date         NOT NULL DEFAULT current_date,
  amount      numeric(10,2) NOT NULL CHECK (amount > 0),
  method      text         NOT NULL CHECK (method IN
('cash','upi','card','transfer')),
  reference   text         UNIQUE
);
```

Read what each buys. `NOT NULL` refuses a payment with no amount. `CHECK (amount > 0)` refuses a negative charge dressed as a payment — refunds get their own sign convention or their own table, decided once, on purpose. `CHECK (method IN (...))` is a small controlled vocabulary that stops `UPI`, `Upi` and `upi ` becoming three payment methods. `UNIQUE` on the provider's reference means re-running an import cannot create a duplicate charge; the second insert fails rather than doubling your revenue.

Six lines of schema removing six classes of bug permanently, for free, at write time. This is the cheapest quality work available anywhere in data.

Where types stop helping

Be honest about the limit. A type checks *shape*, never *meaning*. `amount numeric(10,2)` accepts 4,500 rupees typed into a column that holds dollars. `paid_on date` accepts 2019 in a table that only covers this year. `class_level int` accepts 97.

For meaning you need CHECK constraints where the rule is stable (`class_level BETWEEN 1 AND 12`), and you need the profiling habits from the cleaning lessons where it is not. A well-typed table is not a correct table. It is a table with one entire category of nonsense already excluded, so your attention is free for the categories that are left.

The rule of thumb

Choose the narrowest type that can hold every legitimate value, and no wider. Narrow types reject bad data, use less memory, and — because the database knows more about them — let the query planner make better decisions later. `text` for everything is the choice that costs you at every subsequent stage.

BEFORE YOU MOVE ON

A finance report sums 40,000 order totals stored in a `double precision` column and lands a few cents away from the payment provider's figure, with the gap varying in size and direction from run to run. What is the most likely cause?

A. Rounding is being applied at display time but not at storage time, so the report and the provider round each total differently before summing.

B. The column is too narrow and some large totals are overflowing, losing precision at the top end.

C. Some rows were inserted with NULL totals, and SUM is skipping them.

D. Binary floating point cannot represent most decimal fractions exactly, so each stored total is minutely off and the errors accumulate.

4 · 8 min

Keys: how a database knows one thing from another

THE ONE THING TO KEEP

A primary key answers "is this the same thing as before?", so it must be meaningless and permanent; a business rule such as a unique email sits beside it and is allowed to change.

The question a key answers

Two rows say `Ade Okafor`. Are they one person who ordered twice, or two people who share a common name? Nothing about the text can tell you. A key is the column, or set of columns, that settles it — the thing the database and the rest of your system agree means *this one, and no other*.

Natural keys and their disappointments

A natural key is a value that already exists in the world and looks unique: an email address, a phone number, a national ID, a vehicle registration.

Each one fails in a way you will meet:

- **Email.** People share them (family accounts), change them, and reuse them — an abandoned address gets reassigned by the provider a year later.
- **Phone number.** Recycled aggressively by telecoms. A number that was one customer's is another's within months.
- **National ID.** Not everybody has one, transcription errors are common, and storing it makes your database a target and a legal responsibility.
- **A product's SKU.** Reissued by suppliers. Changed at rebrand.

The deeper problem is not uniqueness, it is *stability*. If the key changes, every row that referenced it must change too, and any exported report becomes unjoinable to the present.

Surrogate keys

So most tables use a surrogate key: a meaningless number the database generates and never changes.

```
CREATE TABLE customers (
  id     bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  email  text NOT NULL UNIQUE,
  phone  text
);
```

Both properties are present and they are different. `PRIMARY KEY` on `id` is the internal identity — permanent, meaningless, what other tables point at. `UNIQUE` on `email` is a business rule — two customers may not share an address today, and if that rule changes you drop the constraint without touching a single foreign key.

That separation is the whole design. Identity is internal and stable; business rules sit beside it and are allowed to change.

Sequential id or UUID? A `bigint` identity is small, sorts naturally, and produces compact indexes because new rows land at the end of the B-tree. A `uuid` can be generated on a phone that is offline, or by three services that never talk to each other, without collision — at the cost of 16 bytes instead of 8 and, for random UUIDs, writes scattered across the whole index, which measurably slows heavy insert workloads. Newer time-ordered UUID versions recover most of that locality. Pick a bigint unless you genuinely need to mint ids in several places.

Composite keys

Some tables have no single identifying column, and inventing one is worse than admitting it:

```
CREATE TABLE enrolments (
  student_id bigint REFERENCES students(id),
  course_id  bigint REFERENCES courses(id),
  enrolled_on date NOT NULL,
  PRIMARY KEY (student_id, course_id)
);
```

That primary key is a statement of fact: a student is enrolled in a course once. If they can re-enrol next term, the key is wrong and the real key includes the term. Get this decision right and a whole family of duplicate-row bugs never happens.

Foreign keys and what happens on delete

`REFERENCES students(id)` means the database checks, on every insert and update, that the student exists. It also decides what happens when the student is deleted, and this is a choice you must make rather than accept:

```
student_id bigint REFERENCES students(id) ON DELETE RESTRICT
-- refuse the delete
student_id bigint REFERENCES students(id) ON DELETE CASCADE
-- delete the payments too
student_id bigint REFERENCES students(id) ON DELETE SET NULL
-- orphan them deliberately
```

`RESTRICT` is the safe default for anything financial: you should not be able to make a year of payments vanish by tidying up a student record. `CASCADE` is right for rows that have no meaning alone — the items of a deleted draft order. `SET NULL` is right when the fact survives its owner, and it requires the column to be nullable, which every later query must then handle.

The single most common data-integrity failure in real systems is a foreign key that was never declared, "for performance". You save a few microseconds per insert and buy an unknown number of rows pointing at customers who no longer exist, discovered years later by a join that quietly returns fewer rows than it should.

Where a key does not exist yet

Some data arrives with no reliable key at all — a spreadsheet of donations with names and dates and nothing else. You cannot conjure identity out of it. What you can do is be explicit: create a surrogate id for each row on import, keep the raw text, and treat "is this the same person as that?" as a separate, evidence-based step with its own lesson later in this course. The mistake is quietly assuming the name is a key, joining on it, and reporting the result.

BEFORE YOU MOVE ON

A `customers` table uses `email` as its primary key, and `orders`, `tickets` and `invoices` all reference it. A customer asks to change her email address. What is the damaging consequence of this design?

A. Email addresses are case-sensitive in some systems, so the change may create a duplicate customer row that the unique constraint fails to catch, splitting her history in two.

B. The unique constraint will reject the new address if anybody else has ever used it.

C. The identifying value must be rewritten in every referencing table, and every earlier report or export can no longer be joined to current data.

D. Storing an email address as a key breaches data protection rules in most countries.

5 · 9 min

Where each fact should live

THE ONE THING TO KEEP

Store each fact once, where it is true; a value copied onto a row is only a duplicate if it is still supposed to change when the original does.

The rule, before the jargon

Normalisation has an intimidating vocabulary and one idea underneath it: **every fact is stored once, in the table about the thing it is a fact about.** Everything else is a consequence.

Take a badly shaped table:

```
order_id | customer_name | customer_phone | customer_city |
product | qty | price
```

Three of those columns are facts about the customer, not about the order. So they are repeated on every order the customer places. Change the phone number and you must change it in 34 places, and last week's export still holds the old one. This is exactly the tuition-centre spreadsheet from the first lesson, and it is what normalisation removes.

The three steps, in plain language

Step one: no repeating groups. A column named `phone1, phone2, phone3` is a list crammed into a row. So is a `tags` column holding `"urgent,billing,vip"`. Both are fine until you have to ask "how many customers have a billing tag", at which point you are writing string searches instead of counting rows. The fix is a second table with one row per phone or per tag.

Step two: no partial dependency. If your key is `(order_id, product_id)` and a column depends only on `order_id` — the order date, say — it belongs in the orders table, not repeated on every line item. Otherwise two line items can disagree about when their own order happened.

Step three: no transitive dependency. A column that depends on another non-key column belongs elsewhere. `customer_city` depends on the customer, and the customer depends on the order. Move it to `customers`.

Applied to the table above:

```
customers   (id, full_name, phone, city)
orders      (id, customer_id, placed_at, status)
order_items (order_id, product_id, qty, unit_price)
products    (id, name, current_price)
```

Now the phone number exists in one row. The city exists in one row. A product's name exists in one row.

The exception that is not an exception

`order_items.unit_price` looks like a violation — the price is a fact about the product, surely? It is not. `products.current_price` is what the product costs *now*. `order_items.unit_price` is what this customer *was charged*, which is a fact about that line of that order and must never change again. If you normalise it away and join to `products` for a price, every historical invoice silently rewrites itself the next time somebody runs a sale.

The same reasoning applies to a delivery address copied onto a shipped order, and a tax rate recorded on an invoice. **Anything that was true at a moment and must stay true is a fact about the event, not about the entity.** Copying it in is correct design, not duplication.

That distinction — a current value versus a recorded value — is the one people most often get wrong, in both directions.

When to denormalise on purpose

Normalisation is a default, not a religion. Deliberate denormalisation is a real technique with a real cost:

- A reporting table with one wide row per order, built nightly, because forty analysts joining six tables is slower and more error-prone than one scan.
- A cached `order_count` on the customer row, because the count is read on every page load and computed rarely.

Both are fine if two conditions hold: the derived value has exactly one place that writes it, and there is a way to rebuild it from the source of truth. A cached count updated in three different code paths will drift, and once it has drifted nobody can tell which number is real. Write it in one place, and keep a job that recomputes it and complains when it disagreed.

The rule is: **normalise the system of record, denormalise the copies you make from it.**

How to tell if a table is wrong

Three questions, faster than any theory:

1. **Can I change one true fact about the world by updating one row?** If a change of city needs an UPDATE with a WHERE clause that touches many rows, the city is in the wrong table.
2. **Can two rows disagree with each other about the same fact?** If `orders` has both `customer_city` and a `customer_id` pointing to a customer with a different city, your data already contains a contradiction and no query can tell you which side is right.
3. **Is there a column whose value is a list?** Commas inside a cell mean a missing table.

The honest cost

Normalised data needs joins, and joins are the part of SQL people find hard — which is most of the next module. Very wide, very denormalised tables are

genuinely easier to query and are exactly what analytical warehouses build on purpose.

The trade is: normalised data is harder to read and almost impossible to make inconsistent; denormalised data is easy to read and drifts out of agreement with itself unless something rebuilds it. Systems that take writes should be normalised. Tables built for reading can be as flat as you like, as long as everyone knows they are derived.

BEFORE YOU MOVE ON

An `invoices` table stores `tax_rate` on every row, even though each customer's region and each region's current rate already live in `customers` and `tax_rates`. A reviewer says the column duplicates a fact and breaks normalisation. Is the reviewer right?

A. Yes. The rate depends on the region, which depends on the customer, so it is a transitive dependency and belongs only in the tax table, reached by a join.

B. Yes, but an acceptable one, since joining to the tax table on every invoice read would be too slow.

C. No. The rate charged on that invoice was true at that moment and must stay fixed, so it is a fact about the invoice rather than about the region.

D. No, because tax rates are set by governments and are therefore reference data, which is always duplicated.

6 · 7 min

SELECT and WHERE: asking for rows

> THE ONE THING TO KEEP
>
> WHERE tests one row at a time, in isolation; it never sees the other rows that share a customer.

Every query has the same skeleton

```
SELECT   full_name, class_level
FROM     students
WHERE    class_level = 9
ORDER BY joined_on DESC
LIMIT    20;
```

`FROM` names where the rows come from. `WHERE` decides which rows survive. `SELECT` decides which columns of the survivors you see. `ORDER BY` sorts them. `LIMIT` cuts the list short.

Here is the thing that clears up half of all beginner confusion: **you write SELECT first, but the database runs FROM first, then WHERE, then SELECT.** The rows are chosen before the columns are picked. That is why this works perfectly well:

```
SELECT full_name FROM students WHERE phone IS NOT NULL;
```

You filtered on a column you never asked to see. Nothing strange happened. WHERE was looking at whole rows; SELECT trimmed them afterwards.

WHERE tests one row at a time

This is the mental model to hold. The database walks the table, takes one row, evaluates your condition against the values *in that row only*, and keeps it if

the answer is true. It cannot see the previous row. It cannot see the other rows belonging to the same customer. One row, in isolation, every time.

So given a `payments` table where Ade has two rows — 3,000 naira in cash and 8,000 naira by transfer:

```
SELECT student_id FROM payments
WHERE method = 'cash' AND amount > 5000;
```

Ade does not appear. No single row of his is both cash and over 5,000. If you wanted "students who have paid cash *and* who have some payment over 5,000", that is a different question and needs the grouping you will meet in a later lesson.

Writing conditions

Text goes in single quotes. Double quotes mean something else entirely (a column name), and using them for text is one of the first errors you will hit.

```
WHERE method = 'upi'                    -- exact match, case
sensitive
WHERE method IN ('upi', 'card', 'cash') -- any of these
WHERE amount >= 5000 AND amount < 20000 -- a range
WHERE full_name LIKE 'A%'               -- starts with A
WHERE full_name ILIKE '%kumar%'         -- contains, case-in-
sensitive (Postgres)
```

For an apostrophe inside text, double it: `WHERE city = 'Coeur d''Alene'`.

The AND/OR trap. AND binds tighter than OR, exactly like multiplication binds tighter than addition. This:

```
WHERE country = 'NG' OR country = 'KE' AND amount > 5000
```

means `country = 'NG'` **or** `(country = 'KE' and amount > 5000)`. Every Nigerian payment comes back, whatever the amount. The database is not

wrong; you did not say what you meant. Put brackets in whenever you mix them, even when you are sure:

```
WHERE (country = 'NG' OR country = 'KE') AND amount > 5000
```

Dates. Prefer half-open ranges over BETWEEN when the column stores a time as well as a date:

```
WHERE paid_on >= '2026-03-01' AND paid_on < '2026-04-01'
```

`BETWEEN '2026-03-01' AND '2026-03-31'` looks equivalent and quietly drops everything that happened on 31 March after midnight, because `2026-03-31 14:02` is greater than `2026-03-31 00:00`. That bug has cost more than one company a month-end report.

Look before you count

The habit that will save you the most time is boring. Before you aggregate anything, run the query with `LIMIT 20` and read the rows with your eyes. Do the amounts look like amounts. Do the dates land in the range you expected. Is that column full of the string `'N/A'`.

A count is a single number and a single number cannot look wrong. Twenty rows can look wrong immediately.

BEFORE YOU MOVE ON

A payments table has one row per payment. Ade has two rows: 3,000 in cash and 8,000 by transfer. Someone runs `SELECT student_id FROM payments WHERE method = 'cash' AND amount > 5000;` Does Ade appear in the result?

A. Yes, because he has a cash payment and he has a payment over 5,000, so both conditions hold for him

B. No, because no single row of his satisfies both conditions at once

C. Yes, because the database combines a student's rows before applying the filter

D. No, because AND requires the two conditions to refer to the same column

7 · 7 min

Sorting, limits, and the rows that move between runs

THE ONE THING TO KEEP

Without ORDER BY a result has no order, and with ORDER BY on a column that ties it has no order among the ties; add a unique column as the final sort key before you paginate.

A query without ORDER BY has no order

This surprises people, so state it plainly: unless you write `ORDER BY`, the database is free to return rows in any order it likes, and it will change its mind. The same query can come back in a different order after an index is added, after rows are updated, or when the planner decides to use several CPU cores.

It usually looks stable, which is what makes it dangerous. A report that has been fine for eight months starts showing a different "top 10" because someone ran a maintenance job.

```
SELECT full_name FROM students LIMIT 10;           -- ten ar-
bitrary students
SELECT full_name FROM students ORDER BY id LIMIT 10; -- the
same ten, always
```

`LIMIT` without `ORDER BY` does not mean "the first ten". It means "any ten, stop early".

Ties do the same thing at a smaller scale

Now the subtler version. You sort by a column with repeated values:

```
SELECT id, full_name, class_level
FROM students
ORDER BY class_level DESC
LIMIT 20;
```

If forty students are in class 10, the twenty you get are an arbitrary twenty of the forty, and re-running may hand you a different twenty. The rule that fixes it is short: **always end ORDER BY with something unique.**

```
ORDER BY class_level DESC, id
```

Now the ordering is total. Nothing can move. Paginating a list without a unique tiebreaker is the cause of the classic bug where a row appears on both page 1 and page 2, and another row appears on neither.

Where NULLs sort

Ascending in Postgres, NULLs come last. Descending, they come first. Other databases disagree — MySQL puts NULLs first when ascending. So a "worst performers" report ordered `ORDER BY score DESC` may open with every student who has no score at all.

Say what you want:

```
ORDER BY score DESC NULLS LAST
```

SQLite and MySQL lack that clause; the portable trick is to sort on a flag first:

```
ORDER BY (score IS NULL), score DESC
```

`false` sorts before `true`, so rows with a score come first.

Sorting text is not alphabetical

`ORDER BY full_name` sorts by the database's *collation*, and collations disagree about real questions. Whether uppercase sorts before lowercase. Whether `Ångström` files under A or after Z. Whether `ä` equals `a`. Whether accents are ignored. A Postgres cluster initialised with the C locale sorts by raw byte value, which puts every capital letter before every lowercase one, so `Zara` precedes `ade`.

That is not broken, it is a different definition of order. If a person will read the list, ask for a human collation explicitly:

```
SELECT full_name FROM students ORDER BY full_name COLLATE
"en_US.utf8";
```

And be aware of the operational side: a collation change during an operating system upgrade can silently invalidate text indexes, because the index was built in one order and the database now believes in another. This has caused real corruption incidents. It is one of the few database problems that a completely correct query cannot protect you from.

Sorting by something you computed

You can order by an alias or by position, and one of those is a trap:

```
SELECT city, count(*) AS orders
FROM orders GROUP BY city
ORDER BY orders DESC;     -- fine, clear

ORDER BY 2 DESC;          -- works, but breaks the day someone
adds a column
```

Ordinals are convenient in a throwaway query and a liability in anything saved.

DISTINCT is a reshaping, not a filter

`SELECT DISTINCT city FROM customers` returns each city once. Two things worth knowing.

First, DISTINCT applies to the *whole* selected row, not the first column. `SELECT DISTINCT city, id FROM customers` returns every row, because the id makes each one unique. People write this expecting one row per city and get the full table.

Second, DISTINCT is often a sign that a join is multiplying rows. The instinct to add it to make a count look right is the instinct to hide the fault rather than find it. If a `SELECT DISTINCT` fixes your row count, go back and ask which join matched twice — the DISTINCT will also have quietly merged genuinely different rows that happened to look alike.

The habit

Two rules that cost nothing and save whole afternoons:

- Anything with `LIMIT` gets an `ORDER BY`.
- Any `ORDER BY` ends with a unique column.

Then a query returns the same rows today, tomorrow and after the database is upgraded, which is the minimum standard for a number anybody is going to act on.

BEFORE YOU MOVE ON

A results page runs `ORDER BY created_at LIMIT 20 OFFSET 20`, and hundreds of rows share the same `created_at` because they were bulk-imported. A user reports that some items appear on page 1 and again on page 2, while others never appear at all. What is happening?

A. Rows created between the two page requests shift everything down by one position, so page 2 repeats the last row from page 1 and hides another.

B. Nothing, as long as `created_at` has an index, because the index fixes the order of equal values.

C. Rows sharing a timestamp may come back in a different order per request, so a tied row can land on both pages or on neither.

D. `OFFSET` skips rows before sorting, so the twenty skipped rows are not the twenty shown on page 1.

8 · 7 min

CASE, COALESCE and computing a column that does not exist

THE ONE THING TO KEEP

CASE tests its branches strictly top to bottom and returns the first match, so the order you write the conditions in is part of the logic, not a matter of style.

SELECT can compute, not just retrieve

Every column in a SELECT list can be an expression. This is where a query stops fetching data and starts producing an answer.

```
SELECT
  full_name,
  amount,
  amount * 0.18                          AS tax,
  round(amount / 12.0, 2)                AS monthly,
  upper(left(full_name, 1))              AS initial,
  paid_on + interval '30 days'           AS due
FROM payments JOIN students ON ...
```

`AS` names the result. Without a name you get something like `?column?`, which is unusable downstream.

CASE: an if-statement inside a query

```
SELECT
  full_name,
  amount,
  CASE
    WHEN amount >= 20000 THEN 'large'
    WHEN amount >= 5000  THEN 'medium'
    ELSE 'small'
  END AS size_band
FROM payments;
```

Two mechanics decide everything CASE does.

It stops at the first true branch. There is no fall-through. So the order of the WHENs is the logic. Reverse the two above and every payment over 20,000 is labelled `medium`, because `amount >= 5000` was tested first and matched. This is the single most common CASE bug, and it produces a perfectly plausible-looking result.

A missing ELSE means NULL. Leave out the ELSE and any row matching nothing gets NULL rather than an error. Then a later `WHERE size_band <> 'small'` drops those rows entirely, for reasons explained in the NULL lesson. Write the ELSE, even when you are sure it cannot be reached — especially then.

Note also that every branch must return the same type. `THEN 0 ... ELSE 'none'` will be rejected, or worse, silently coerced in a database that is trying to be helpful.

The short forms

`COALESCE(a, b, c)` returns the first argument that is not NULL. It is the standard way to supply a default:

```
COALESCE(nickname, full_name, 'Unknown')
```

`NULLIF(a, b)` returns NULL when the two are equal. Its main use is defusing division by zero, which is an error rather than a NULL in most databases:

```
SELECT successes::numeric / NULLIF(attempts, 0) AS rate FROM
funnels;
```

Zero attempts now yields NULL — "we cannot say" — instead of aborting the whole query. That is usually what you want, and it is honest: there genuinely is no rate.

`GREATEST(a, b)` and `LEAST(a, b)` compare values across a row, unlike MAX and MIN which compare down a column. Worth knowing because the confusion between the two is common.

CASE inside an aggregate, which is the powerful trick

This deserves its own mention because it unlocks a lot:

```
SELECT
  city,
  count(*)                                          AS al-
l_orders,
  count(*) FILTER (WHERE status = 'refunded')       AS refund-
ed,
  sum(CASE WHEN status = 'refunded' THEN amount ELSE 0 END) AS
refunded_value
FROM orders
GROUP BY city;
```

One pass over the table producing several differently-filtered counts side by side. The `FILTER` clause is standard SQL and supported by Postgres and SQLite; `sum(CASE ...)` is the portable form that works everywhere including MySQL and older engines. The aggregation module goes much further with this.

Where an expression costs you

Computing in SELECT is free-ish; computing in WHERE can be expensive:

```
WHERE upper(city) = 'LAGOS'          -- cannot use an index on
city
WHERE city ILIKE 'lagos'             -- also cannot, usually
WHERE city IN ('Lagos','lagos','LAGOS')  -- can
```

Wrapping a column in a function hides its stored value from the index, as the index lesson explains in detail. The clean answer is usually to fix the data once — store the city normalised — rather than to normalise it on every read forever.

Readability, briefly

A CASE with nine branches nested inside another CASE is where queries go to die. Two ways out.

Give the logic a name with a CTE:

```
WITH banded AS (
  SELECT id, amount,
         CASE WHEN amount >= 20000 THEN 'large' ... END AS band
  FROM payments
)
SELECT band, count(*) FROM banded GROUP BY band;
```

Or, when the bands are business rules that change, put them in a small table and join to it. A `band_thresholds` table with three rows lets someone change the definition of "large" without editing SQL — and, more importantly, makes the definition visible to people who cannot read SQL at all. That last point is worth more than it sounds.

BEFORE YOU MOVE ON

A query labels orders with `CASE WHEN amount > 1000 THEN 'high' WHEN amount > 5000 THEN 'vip' END`. What does it produce for an order of 7,500, and for an order of exactly 1,000?

A. Orders over 5000 get both labels, so they appear twice in a grouped count.

B. Orders over 5000 are labelled 'vip', because the more specific condition wins, and orders of exactly 1000 are labelled 'high' since the boundary is inclusive.

C. Orders of exactly 1000 are labelled 'high' and everything below is NULL.

D. Orders over 5000 are labelled 'high', because that branch is tested first and matches; orders of 1000 or less are NULL.

9 · 9 min

Reading a database somebody else built

THE ONE THING TO KEEP

In a schema you did not build, let the foreign keys draw the map, state each table's grain, and hunt for the status and soft-delete columns that silently filter everything.

The situation

You are given access to a database with 180 tables, no documentation, and a question to answer by Thursday. The person who designed it left. This is the normal case, not the unlucky one, and there is a routine for it that takes about forty minutes.

Find out what exists, and how big it is

```
-- Postgres: tables with a live row estimate, largest first
SELECT relname, n_live_tup
FROM pg_stat_user_tables
ORDER BY n_live_tup DESC
LIMIT 40;
```

Size is the fastest guide to importance. The three biggest tables are almost always the event tables — the things that happen — and the small ones are the entities and the lookup lists. A table with 12 rows is a controlled vocabulary; a table with 400 million is a log.

Portable equivalents: `\dt` and `\d tablename` in psql, `.tables` and `.schema` in SQLite, `SHOW TABLES` and `DESCRIBE t` in MySQL, and `information_schema.columns` everywhere.

```
SELECT column_name, data_type, is_nullable, column_default
FROM information_schema.columns
WHERE table_name = 'orders'
ORDER BY ordinal_position;
```

Let the foreign keys draw the map

If the designer declared foreign keys, the structure is already written down and you can read it:

```
SELECT
  tc.table_name      AS child,
  kcu.column_name    AS child_column,
  ccu.table_name     AS parent
FROM information_schema.table_constraints tc
JOIN information_schema.key_column_usage kcu
  ON kcu.constraint_name = tc.constraint_name
JOIN information_schema.constraint_column_usage ccu
  ON ccu.constraint_name = tc.constraint_name
WHERE tc.constraint_type = 'FOREIGN KEY'
ORDER BY child;
```

That single result is the entity relationship diagram, generated from the truth rather than from a document somebody stopped updating in 2021.

When it returns nothing — common in databases built by application frameworks that manage relationships in code — fall back to naming conventions. A column called `customer_id` in `orders` almost certainly points at `customers.id`. Verify rather than assume:

```
SELECT count(*) FROM orders o
LEFT JOIN customers c ON c.id = o.customer_id
WHERE c.id IS NULL;      -- how many orders point at a customer
that does not exist
```

A non-zero answer tells you two things at once: the guess is probably right, and the data has orphans nobody is watching.

Establish the grain of each table you will use

For every table you plan to query, find the sentence "one row of this table is one _____". Check it rather than believing the name:

```
SELECT count(*), count(DISTINCT order_id) FROM order_items;
```

Equal numbers mean one row per order, and the table is misnamed. Different numbers mean one row per line item, as expected. Two minutes here prevents the row-multiplication bugs that the joins module is entirely about.

Find the columns that silently filter everything

Almost every mature database has at least one of these, and missing it is how you produce a confidently wrong number:

- **Soft deletes**: `deleted_at`, `is_deleted`, `archived`. Rows are still there. Every honest query needs `WHERE deleted_at IS NULL`, and nobody will tell you.
- **Test data**: `is_test`, `environment`, an internal customer whose name is `QA Test`. Payment systems in particular are full of it.
- **Status columns**: an `orders` table where `status` runs `cart`, `pending`, `paid`, `cancelled`. Counting all rows counts abandoned baskets as orders.
- **Multi-tenancy**: `tenant_id` or `org_id`. Forgetting it sums every customer of the business together.

Find them by listing the distinct values of every low-cardinality column:

```
SELECT status, count(*) FROM orders GROUP BY status ORDER BY 2
DESC;
```

Do this for each suspicious column. It takes minutes and it is the single highest-yield habit in this lesson.

Read the application, if you can

Column meanings often live in code, not in the database. A grep of the application repository for the table name will usually find the model definition, and the comment above it, and the migration that added the mystery column with a message explaining why. When there is a repository you can read, read it — it is documentation that had to be true.

Ask one person one question

The last step is social. Find whoever queries this database most and ask a single, specific question: *"If I want revenue for last month, which table and which filters do you use?"* Not "can you explain the schema", which invites an hour of vagueness. A concrete question gets you their working query, and their working query contains all the filters nobody wrote down.

Then write what you learned into a file in your own repository. Three paragraphs: the grain of each table you touched, the filters that are always required, and the one thing that surprised you. You are going to be the person somebody asks next.

BEFORE YOU MOVE ON

You are asked for last quarter's revenue from a database you have never seen. You have found `orders` with `amount` and `placed_at`, and the sum looks plausible. Which single check most changes whether the figure is right?

A. Confirming that `amount` is stored as `numeric` rather than a floating-point type, since binary rounding across a quarter of orders could compound into a visibly wrong total.

B. Confirming there is an index on `placed_at` so the query runs in reasonable time.

C. Checking the collation used to sort the customer names in the output.

D. Listing the distinct values of `status` and any soft-delete or test flags, to see how many rows are abandoned baskets, cancellations or internal tests.

10 · 8 min

Grain: the sentence to say before you write the query

THE ONE THING TO KEEP

Say "one row of my result is one ____" before writing the query; every aggregate reads the joined grain, not the original table's, so a join that changes the grain is the whole hazard.

One sentence, and most wrong answers do not happen

Before writing any query, finish this sentence out loud:

> ***One row of my result is one _____.***

One customer. One order. One customer-month. One payment attempt. If you cannot finish it, you are not ready to write the query, and the number you produce will be wrong in a way that does not look wrong.

That sentence names the **grain** of your result. Grain is the most useful idea in this course and almost nobody is taught it explicitly.

Every table has a grain too

Before combining tables you need to know the grain of each. It is a fact you can measure, not a matter of opinion:

```
SELECT count(*) AS rows,
       count(DISTINCT order_id) AS orders
FROM order_items;
-- rows 84,201 | orders 31,940  → one row per line item
```

Names lie. A table called `daily_metrics` may hold one row per day per country per product, which is not daily anything. A table called `users` may hold

one row per user per sign-up source, so a user who arrived twice appears twice. Measure it once; write the answer down.

Joining changes the grain, and that is the entire hazard

Here is the mechanism stated as a rule you can apply without thinking:

Joining a table at grain A to a table at grain B gives you grain B, whenever many B rows belong to one A row.

So joining `customers` (one row per customer) to `orders` (one row per order) gives you one row per *order*, with customer details repeated. Every customer-level number in that result — credit limit, signup date, lifetime value — is now duplicated once per order and cannot be summed. The joins lesson shows exactly what that does to a SUM.

You have three moves, and picking the right one is the skill:

1. **Aggregate the many side first**, then join. One row per customer stays one row per customer.
2. **Join, then group back up.** Works, but any customer-level column you SUM after joining is wrong; only COUNT and the aggregates over the many side are safe.
3. **Use a scalar subquery or a lateral join** to attach one summarised value per row without changing grain at all.

```
-- Option 1: aggregate first, grain stays one row per customer
WITH per_customer AS (
  SELECT customer_id, sum(amount) AS spend, count(*) AS orders
  FROM orders GROUP BY customer_id
)
SELECT c.id, c.full_name, c.credit_limit,
       coalesce(p.spend, 0) AS spend
FROM customers c
LEFT JOIN per_customer p ON p.customer_id = c.id;
```

Read the grain at each step. The CTE is one row per customer. `customers` is one row per customer. Joining one-to-one preserves it. `sum(credit_limit)` over that result is now correct, and it was not correct before.

The grain of a report is a business decision

"Revenue by month" sounds unambiguous until you ask what a month is a property of. Order date? Payment settlement date? The month the subscription covers? Three different grains, three different tables, three different numbers, all defensible.

Similarly "one row per customer per month" needs a decision about customers with no activity that month. Do they appear with zero, or not appear at all? A grid that only contains rows where something happened will show a gap where a real zero should be, and the chart drawn from it will interpolate straight across the gap.

State the grain in words when you send the number:

The grain changes at the join, and every aggregate reads the new one

customers	one row per customer
orders	one row per order
customers JOIN orders	one row per order; the customer's columns repeat
sum(credit_limit) over that result	each limit counted once per order — wrong
Aggregate orders first, then join one to one	one row per customer, and the sum is right

Read the grain at each step. A customer-level column summed after a one-to-many join is inflated once per order, and nothing in the output shows it. Aggregating the many side first keeps the grain, and only then is sum(credit_limit) the number it looks like.

> *One row is one customer-month, for every customer who existed at the start of the month, including months with no orders (shown as zero), by order placement date in Asia/Kolkata.*

Nobody enjoys writing that sentence. It has ended more disagreements than any query I have seen.

A check you can run

After writing any query, before believing it, compare the row count against the grain you claimed:

```
SELECT count(*), count(DISTINCT customer_id) FROM my_result;
```

Claimed one row per customer? Those two numbers must match. If they do not, a join multiplied and every per-customer aggregate is inflated by an amount you cannot see in the output.

This takes eight seconds and it catches the majority of the errors that make it into meetings. It is the cheapest quality check in the whole of data work, and the reason it is not universal is simply that most people were never taught to name the grain in the first place.

BEFORE YOU MOVE ON

A query joins `customers` to `orders` on `customer_id` and, in one pass, computes `sum(orders.amount)` and `sum(customers.credit_limit)`. Which of the two totals can be trusted?

A. Both totals are correct, because each SUM only reads its own table's column.

B. The order total is right and the credit-limit total is inflated, because the join repeated each customer's limit on every one of their orders.

C. Both totals are inflated, because a join multiplies rows on both sides of the relationship, and every SUM computed after it will therefore double-count in proportion.

D. Neither total can be trusted until DISTINCT is added to both aggregates.

11 · 9 min

A right number and a useful number are not the same

THE ONE THING TO KEEP

A correct query answers the question you typed, which is rarely the question you were asked.

The query was correct. The answer was wrong.

This is the lesson that separates people who can write SQL from people you can trust with a decision. Your query can be flawless — correct joins, correct filters, no NULL traps — and still produce a number that misleads everyone in the room.

Here is the shape of it, four ways.

The definition nobody agreed on

"How many active users do we have?" You write the query. You return 41,200.

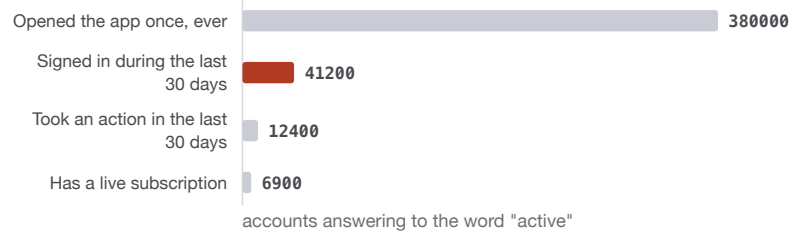
But nobody said what active means. Signed in during the last 30 days? Took any action at all? Has a live subscription? Opened the app once, ever, and never deleted the account? Those four numbers might be 41,200 and 12,400 and 6,900 and 380,000. All four are correct. Three of them will get someone fired.

The fix is not technical. Before the query, write the sentence: *"one row of my result is one account that performed at least one action between 1 and 31 March, in the account's own time zone."* Send that sentence to whoever asked, before you send them a number.

The denominator that moved

A team reports that conversion rate rose from 2.1% to 3.4% after a redesign. The numerator query is correct. The denominator query is correct. What happened is that an analytics script broke on mobile, so a third of the visits stopped being recorded. Fewer visits, same purchases, higher rate.

Four correct answers to "how many active users do we have?"



Every one of these is produced by a query with no bug in it. The number that gets reported is decided by a definition nobody wrote down, and the gap between the largest and the smallest is a factor of fifty-five. Send the sentence before you send the number.

Whenever a ratio moves, look at the top and the bottom separately. A ratio hides which half changed, and it is very often the half you were not thinking about.

Survivorship

A subscription business computes churn as: of the customers currently in the customers table, what share have status `'cancelled'`. It returns 4%.

The problem is that customers who cancelled and were then purged, or archived, or moved to another table, are not in the denominator at all. You are measuring churn among the people who did not leave. Adding a date filter does not help; the missing people are missing before any filter runs.

The general form: whenever your population is defined by having survived, you cannot use it to measure survival.

The average that lands where nobody is

Mean order value on an Indian marketplace: ₹4,800. Ninety percent of orders are between ₹300 and ₹900. A handful of bulk business orders are ₹200,000

each. The mean is arithmetically perfect and describes no customer who has ever existed.

Report the median. Report the 90th percentile next to it. If the median is ₹620 and the mean is ₹4,800, that gap is itself the finding, and it is more interesting than either number alone.

Time zones and late data, briefly

"Yesterday's revenue" computed in UTC for a business in Asia/Kolkata moves the boundary by five and a half hours, which reliably shifts the evening rush into the wrong day. And today's number is always low, because payments settle, webhooks retry and refunds land days later. A dashboard whose last bar always dips has taught its whole company that things are getting worse.

What to do about all of this

Say the sentence first. "One row of my result is _____, counted over _____, between _____ and _____ in _____ time zone, excluding _____." If any blank is unfillable, go and ask.

Check the number against something you already know. Last month's figure. A total from the finance system. Ten rows counted by hand. A number with nothing to compare it against is a number nobody can challenge, including you.

Try to break your own filter. Change a condition so the result *should* move. If it does not move, your filter is not doing anything, and you have found that out for free instead of in a meeting.

Ship the query with the number. Always. A figure in a message with no query behind it cannot be argued with, and things that cannot be argued with are how organisations end up confidently wrong for a year.

BEFORE YOU MOVE ON

A subscription business calculates churn as: of the customers currently in the customers table, what share have status 'cancelled'. The SQL is correct and returns 4%. Why might 4% be the wrong answer to "what is our churn rate"?

A. The result has no date range, so it mixes every year together; adding `WHERE cancelled_at >= ...` fixes it
B. Some customers marked 'cancelled' later returned, so the numerator overstates the loss
C. Customers who cancelled and were later removed or archived are not in the denominator, so this measures churn only among people still on file
D. 4% is a monthly figure and needs to be annualised before it can be reported as a churn rate

12 · 8 min

Asking an AI for SQL without being lied to

THE ONE THING TO KEEP

Paste your schema, ask for assumptions first, and check the number against something you already know.

What an AI is actually good at here

Be fair to the tool before you are suspicious of it. A language model will write a window function you would have spent twenty minutes looking up. It will translate a query between Postgres and BigQuery dialects. It will explain a query plan you cannot read. It will produce a decent first draft five times faster than you can type one. This is real and you should use it.

What it cannot do is know things it was not told, and its failure mode is not silence. It is confidence.

Give it the schema, every time

The most common wrong query from an AI is not a logic error. It is a guess about your column names, dressed as knowledge. It will write `orders.total_amount` when your column is `amount_cents`. It will invent a `users.last_login_at` because most schemas have one.

So paste the actual definitions. `\d orders` in psql, or the CREATE TABLE statements, plus one sentence per table saying what a row means:

> *Here are three tables. One row of `payments` is one attempted charge, including failures. One row of `orders` is one basket someone checked out. An order can have several payment attempts.*

That last sentence — the one about several payment attempts — prevents more wrong queries than anything else you can write, because it is exactly the fact that produces row multiplication in a join.

Ask for the assumptions before the SQL

A useful prompt shape:

> *Before writing the query, list what you are assuming about the schema and about the definitions of the terms I used. Then write the query.*

This converts invisible guesses into a list you can check in ten seconds. It will say things like "I assumed 'active' means a login in the last 30 days" and "I assumed refunds are stored as negative amounts in the same table." One of those is usually wrong, and it is much cheaper to catch it there than in the output.

Verify like you would verify a stranger's work

Because that is what it is.

Run it with a LIMIT first and read the rows. Not the count. The rows.

Check the row count against what you expect. If the orders table has 40,000 rows and the joined result has 61,000, a join is multiplying, and the

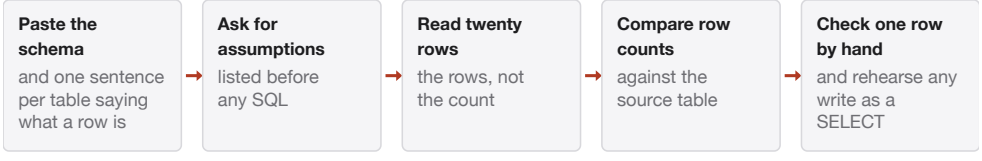
AI's total revenue is now wrong in a way that looks entirely plausible. This is the single most dangerous failure in AI-written SQL: the result is not absurd, it is just too big by 40%.

Check one row by hand. Pick a customer, look them up in the source, verify their number.

Compare a total to something you already know. Last month's figure, the finance export, anything.

Break it on purpose. Change a filter so the number should move. If it does not move, the filter is not doing anything.

Getting SQL out of a model without being confidently misled



The dangerous output is not the query that errors — that looks after itself. It is the clean, reasonable-looking number returned on the first try. Every step here is cheap, and each one converts an invisible guess into something you can check.

The rule for anything that writes

Never run a generated `UPDATE` or `DELETE` directly. Run the WHERE clause as a SELECT first, and look at what would be affected:

```
-- Before: see exactly what the generated statement would touch
SELECT count(*), min(id), max(id) FROM orders WHERE status =
'pendng';

-- If you must run it, run it where you can take it back
BEGIN;
UPDATE orders SET status = 'pending' WHERE status = 'pendng';
-- inspect, then:
ROLLBACK;   -- or COMMIT, once you are sure
```

In that example the typo is in the data, not the statement, and the SELECT would have returned 3 rows instead of the 40,000 you feared, or zero when you expected thousands. Either way you learn it before anything changes.

The asymmetry worth remembering

SQL that fails is free. You see an error, you fix it, nobody is harmed. SQL that succeeds and is wrong costs you a decision, and it costs it silently, sometimes months later.

So spend your scepticism where the risk actually is. Not on the queries that error — those look after themselves — but on the ones that return a clean, reasonable-looking number on the first try. That is the output you should stare at hardest, whether a model wrote it or you did.

BEFORE YOU MOVE ON

An AI writes a query joining orders to payments and reports total revenue of 42.1 lakh rupees. It runs without error and the figure is in a believable range. What is the strongest reason to still not trust it?

A. Models are trained on older material, so it may have used syntax that is deprecated in your database version

B. An order with more than one payment attempt is duplicated by the join, inflating the total in a way that still looks believable

C. The query may have used a poor plan and computed the total over a partial scan of the table

D. Money should be aggregated in application code rather than in SQL, where rounding is harder to control

CASE STUDY · MODULE 1

The fee sheet that could not be trusted twice

Vidyankur, a coaching centre in Nagpur with 620 students, four staff and one spreadsheet, three weeks before the annual fee drive

Sunita Deshpande has run Vidyankur for nine years. The centre's entire record is one spreadsheet named `fees_master_FINAL_v4.xlsx`: one row per payment, with the student's name, parent's phone number, class, batch, month, amount and mode. It has 41,000 rows. It takes fifty seconds to open, and for nine years it has been good enough.

In January three things happen in the same fortnight.

A parent calls to say she has been paying for her son since June and the centre has just sent her a defaulter notice. She is right. Her phone number changed in August; the office updated it in the rows they could find, and missed eleven. The defaulter list is built by matching on phone number, so those eleven payments now belong to nobody.

The second is quieter. Sunita asks her office manager, Faiz, how many students are in Class 10 Maths. He gives her 214. Her senior teacher gives her 227 from the same file an hour later. Neither number is wrong: Faiz counted rows where `batch` is `Class 10 Maths`, and the teacher counted rows where `batch` starts with `Class 10`, which also catches `Class 10 Maths (Evening)` — a batch someone created in July by typing a new value into the column. Nothing in the file says which of the two is the real batch list, because there is no batch list. There are only whatever strings people have typed.

The third is that Faiz, tidying up, sorts the sheet by amount to find the largest payments and does not extend the selection to the whole row. Two thousand rows now have somebody else's amount against their name. He notices within a minute and undoes it. Nobody can prove the undo was complete.

Sunita's nephew, who is finishing a diploma, offers to rebuild the whole thing as a small Postgres database: a `students` table with a generated id and the phone as a plain column, a `batches` table with one row per batch, an `enrolments` table in the middle because a student can be in several batches, and a

`payments` table whose `student_id` and `batch_id` point at rows that must exist. He estimates eleven working days, most of it spent deciding what the existing rows mean rather than typing SQL.

Eleven working days lands squarely on the fee drive, which is the fortnight that pays for the year. During the rebuild there is no defaulter list, because the old sheet is frozen and the new tables are not loaded. Faiz cannot write SQL. Sunita cannot write SQL. If the nephew goes back to college in March, the centre owns a database it cannot query, having given up a spreadsheet it could at least look at.

Against that: the fee drive is the exact moment when the defaulter list has to be right. A notice sent to a parent who has paid costs a student. Last year the centre lost four that way, which is about a hundred and forty thousand rupees a year in fees, and Sunita only knows about the four who complained.

There is a third option nobody has costed. Keep the spreadsheet as the working record for this year, and spend two days building only the parts that stop the specific bleeding: a real list of batches, a real list of students with a permanent id written back into the sheet, and a nightly export into a database that answers the two questions — who owes what, and who is in which batch — while all writing continues in the sheet as before.

That option leaves the sheet's other faults untouched for a year. It also means the same fact lives in two places for a year, which is the problem the rebuild exists to remove.

> **STOP HERE**
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Sunita took the third option, and it half worked. Two days produced a `batches` table with 31 rows, a `students` table with 620 rows and a permanent id, and a script that loaded the sheet nightly into Postgres. Assigning a perma-

nent id forced the office to resolve identity by hand for the first time: 620 students in the database against 664 distinct name-and-phone combinations in the sheet, of which 31 were spelling variants, 9 were siblings sharing a parent's phone number, and 4 were genuinely two people with the same name in the same batch. The nine siblings were the ones that mattered — the defaulter list had been merging them for years, so a family with two children was marked paid when one of them had paid. The batch list found the evening batch and two more nobody had known existed. Faiz learned five queries and stopped asking anyone for counts. The rebuild never happened, and eighteen months later the nightly export is still the only trustworthy record, with the sheet still being edited by four people. Sunita's own summary: the two days bought the answers, and paid for them with a copy of every fact, which somebody will have to unpick.

Faiz and the senior teacher both counted Class 10 Maths and got different numbers. Which of the ideas in this module names the fault, and what would fix it permanently?

It is a missing controlled vocabulary and a missing table. `batch` is a free-text column, so its values are whatever people typed, and `Class 10 Maths (Evening)` is a batch that exists in the data and nowhere in anyone's head. The permanent fix is a `batches` table with one row per batch and a foreign key from enrolments to it, so a batch that does not exist cannot be typed. The immediate fix is `SELECT batch, count(*) FROM sheet GROUP BY batch ORDER BY 1` and reading the tail of the list, which is where the variants live.

The defaulter list matched on the parent's phone number. Explain, using the keys lesson, why that was going to fail no matter how careful the office was.

A phone number is a natural key, and natural keys fail on stability rather than on uniqueness. When the number changes, every row that referenced it must change too, and any row that is missed is silently orphaned — which is exactly what happened to the eleven payments. It also fails on uniqueness here, because siblings share a parent's number, so two students collapse into one. A surrogate key — a meaningless, permanent id the database generates — carries identity, and the phone number sits beside it as an ordinary column that is allowed to change.

Sunita's third option leaves the same facts in two places for a year. Under what conditions is that a defensible piece of denormalisation rather than a mess?

It is defensible when exactly one place writes the derived copy and there is a way to rebuild it from the source of truth. Here the sheet is the system of record and the database is rebuilt from it nightly, so the copy has one writer and can always be regenerated. It stops being defensible the moment anyone starts editing the database directly, because then two writers exist, the copies can disagree, and nobody can say which is real.

MODULE 1 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Ade has two rows in `payments`: 3,000 naira in cash, and 8,000 naira by transfer. `SELECT student_id FROM payments WHERE method = 'cash' AND amount > 5000` does not return him. Why not?

A. No single row of his is both cash and over 5,000
B. WHERE cannot combine a text test and a number test in one condition
C. SELECT runs before WHERE, so the method column had already been discarded
D. Both of Ade's rows are dropped because their two amounts disagree with each other

2 A report shows the twenty highest classes with `ORDER BY class_level DESC LIMIT 20`, and forty students share class 10. Different students appear on different runs. What fixes it?

A. Creating an index on class_level so the order becomes fixed
B. Ending the ORDER BY with a column that cannot tie, such as the id
C. Adding DISTINCT so that tied rows are not returned twice
D. Raising the LIMIT until every student in the tied class is included

3 A CASE is meant to label amounts of 20,000 or more as `large` and 5,000 or more as `medium`, but the 5,000 branch is written first. What does the query return?

A. An error, because the two conditions overlap
B. Both labels for large payments, with the last branch winning
C. Every payment over 20,000 labelled medium
D. NULL for payments over 20,000, because no branch claims them

4 `SELECT sum(c.credit_limit) FROM customers c JOIN orders o ON o.customer_id = c.id` returns a figure many times the sum of the column in `customers`. What happened?

A. The sum is right; a join only adds columns to each row

B. Each customer's limit is added once for every order they placed

C. The query should have errored, since credit_limit is not in a GROUP BY

D. Only customers with at least one order are counted, so the figure is slightly low

5 In Postgres, `SELECT 0.1::float + 0.2::float = 0.3::float` is false. What is the mechanism?

A. One tenth has no exact binary form, as one third has none in decimal

B. The float type keeps only two decimal places, so the third is lost

C. Comparing two floats with = always yields unknown, in the way NULL does

D. Postgres rounds every floating-point result to six significant digits before comparing

6 You are given access to an unfamiliar database and asked for last month's revenue from `orders`. Which habit most often prevents a confidently wrong number?

A. Reading the entity relationship diagram in the team's documentation folder

B. Adding DISTINCT, so that duplicated orders cannot inflate the total

C. Listing the distinct values of every low-cardinality column

D. Filtering to the last thirty days rather than to a calendar month

MODULE 2

Putting tables together

Joins are where most SQL goes wrong, and almost always for one of four reasons: a key that does not match, a relationship that matches more than once, two child tables counted at the same time, or a filter placed on the wrong side. This block works through each mechanism, then adds the other ways of combining rows — set operations, subqueries, CTEs and a generated spine — so you can pick the one that keeps your grain intact.

> BY THE END OF THIS MODULE YOU CAN
>
> Combine any set of related tables without inflating a total: diagnose a join key that silently fails to match, recognise and defuse the fan trap between two child tables, choose between a join, a subquery, an EXISTS and a set operation, and generate the rows for periods in which nothing happened

---

13 · 9 min

Joins: reason about them, do not memorise them

> THE ONE THING TO KEEP
>
> A join emits one row per match; zero matches and two matches are where every join bug lives.

Forget the Venn diagrams

The circles do not tell you what actually happens, which is this:

For each row on the left, the database looks for rows on the right where the ON condition is true, and emits one output row per match.

Everything follows from that one sentence. In particular, two questions decide the behaviour of any join you will ever write:

1. What happens to a left row with **zero** matches?
2. What happens to a left row with **two** matches?

An inner join drops the zero-match row. A left join keeps it and fills the right-hand columns with NULL. That is the entire difference between them.

And for two matches, both kinds of join do the same thing: they emit two rows. The left row is duplicated. This is not a bug and there is no flag to turn it off. It is what a join is.

Row multiplication, which is where the money goes wrong

```
SELECT c.name, c.credit_limit, o.amount
FROM customers c
JOIN orders o ON o.customer_id = c.id;
```

If Mariana has three orders, Mariana appears three times, and her credit limit appears three times with her. Now watch this go wrong:

```
SELECT SUM(c.credit_limit)
FROM customers c
JOIN orders o ON o.customer_id = c.id;
```

The total credit limit is now roughly the number of orders times the average limit. It is enormous. It is also completely plausible-looking, which is the dangerous part. Nothing errors. You get a number.

The defence is a sentence you say before you write the query: **"one row of my result is _____."**

- "One row per order, with the customer's city attached." Each order has exactly one customer, so at most one match. Joining is safe.
- "One row per customer, with their total order value." That is one row per customer, but the join produces one row per order. So the join has to be followed by a grouping, or the total belongs in a subquery.

If you cannot finish that sentence, you are not ready to write the join.

The trap that turns a LEFT JOIN into an inner join

This is the single most common join bug in working code.

```
SELECT o.id, r.reason
FROM orders o
LEFT JOIN refunds r ON r.order_id = o.id
WHERE r.reason <> 'damaged';
```

The intent is "all orders, with refund reason where there is one, excluding damage refunds." What you get is only orders that have a refund.

Why: the LEFT JOIN faithfully keeps the unmatched orders and sets `r.reason` to NULL for them. Then WHERE asks whether NULL is different from `'damaged'`, and the answer is not true — it is unknown. WHERE keeps only rows where the test is true. Every unmatched order is discarded a step after the join carefully preserved it.

Two fixes, and they mean different things:

```
-- Filter the right-hand side before matching: keeps all orders,
-- and simply never attaches a damage refund to them.
LEFT JOIN refunds r ON r.order_id = o.id AND r.reason <> 'damaged'

-- Filter after matching, but spare the unmatched rows explicitly.
WHERE (r.reason <> 'damaged' OR r.id IS NULL)
```

General rule: a condition on the right-hand table of a LEFT JOIN belongs in ON. A condition on the left-hand table belongs in WHERE.

The anti-join, which is worth knowing by name

"Orders that have no refund at all":

```
SELECT o.id
FROM orders o
LEFT JOIN refunds r ON r.order_id = o.id
WHERE r.id IS NULL;
```

Join everything, then keep only the rows where the right side came back empty. Read it as "look for a match, and keep the failures." It is the standard way to ask "which of these has none of those", and once you have seen it you will use it constantly: students with no payment this month, products never sold, users who never opened the app.

The reasoning habit

When a join surprises you, do not reach for a different join type and hope. Count. Run `SELECT COUNT(*)` on the left table alone, then on the join. If the join returned more rows than the left table has, something on the right matched more than once, and you have found your bug. If it returned fewer, an inner join or a WHERE clause is dropping rows you meant to keep.

BEFORE YOU MOVE ON

A query LEFT JOINs orders to refunds and adds `WHERE r.reason <> 'damaged'`. It returns 812 rows, but the orders table has 40,000 rows and the team expected roughly that many. What has happened?

A. The refunds table contains duplicate rows, so the join multiplied some orders and the count is unreliable

B. A LEFT JOIN guarantees every left row survives, so the count cannot have dropped and the 40,000 figure must be wrong

C. For orders with no refund, r.reason is NULL, the WHERE test is not true, and those rows are discarded after the join keeps them

D. The join direction is backwards and a RIGHT JOIN from refunds to orders would restore the missing rows

14 · 8 min

The four joins, and what each one is for

THE ONE THING TO KEEP

All four joins share one mechanism, one output row per match, and differ only in what they do with the leftovers; the row count after a join is the check that tells you whether a key matched more than once.

One mechanism, four decisions about the leftovers

Every join does the same thing: for each row on the left, find rows on the right where the ON condition holds, and emit one output row per match. The four join types differ in a single respect — what happens to rows that found nothing.

- **INNER JOIN**: unmatched rows on either side are dropped.
- **LEFT JOIN**: every left row survives; unmatched ones get NULLs on the right.

- **RIGHT JOIN**: the mirror image. Every right row survives.
- **FULL OUTER JOIN**: everything from both sides survives, with NULLs wherever there was no match.

That is the whole taxonomy. There is nothing else to memorise.

When each is the right choice

INNER is right when the relationship is guaranteed and you want it enforced. `orders` joined to `customers` on a NOT NULL foreign key should match every time; if the row count drops, you have found a data problem and you want to know.

One mechanism, and what the join types do with the leftovers

The join types differ in one column of this table, not two. Zero matches is the only thing INNER and LEFT disagree about; two matches produces two output rows in both, which is where a total silently doubles.

LEFT is right whenever the left table is your population. "All students, with their payments" is a left join, because a student who has never paid is still a student and must appear in the report. This is the join you will write most often, and the reason is worth internalising: **the left table defines who is in the answer**, and the right table only adds detail.

RIGHT is genuinely rare in practice. Anything a right join does, a left join with the tables swapped does more readably. Its main appearance in real code is accidental, when somebody reorders a query and does not adjust the join type.

FULL OUTER has one great use: reconciliation. When you have two systems that should agree and want to see everything that does not line up:

```
SELECT
  coalesce(a.reference, b.reference) AS reference,
  a.amount AS in_ledger,
  b.amount AS in_bank
FROM ledger a
FULL OUTER JOIN bank b ON b.reference = a.reference
WHERE a.reference IS NULL       -- in the bank, missing from
the ledger
   OR b.reference IS NULL       -- in the ledger, missing from
the bank
   OR a.amount <> b.amount;     -- present in both, disagreeing
```

Three failure modes in one result set. MySQL has no FULL OUTER JOIN; the standard workaround is a `LEFT JOIN` unioned with a `RIGHT JOIN`.

The join that has no ON clause

`CROSS JOIN` pairs every left row with every right row. Ten rows against twelve gives 120. It is deliberate and useful for generating grids — every product against every month — and it is also what you get by accident when you write an old-style comma join and forget the WHERE:

```
SELECT * FROM orders, customers;   -- 31,940 x 8,200 = 261 mil-
lion rows
```

Use explicit `JOIN ... ON` syntax always. It puts the join condition next to the join, so a missing condition is a syntax error instead of a query that runs for an hour.

USING and NATURAL, and why to be careful

`USING (customer_id)` is shorthand for `ON a.customer_id = b.customer_id`, and it merges the two columns into one in the output. It is tidy and safe.

`NATURAL JOIN` joins on *every* column with a matching name in both tables. It reads beautifully and is a trap: the day somebody adds a `created_at` column to both tables, your natural join silently starts requiring those timestamps to

be equal, and your result becomes empty. Nothing errors. Do not use it in anything that has to keep working.

Reading a join out loud

The habit that makes joins stop being confusing is to narrate each one in a fixed shape:

> *For each **order**, find the **customer** whose id matches, of which there is exactly **one**, and it is **guaranteed to exist**.*

Three facts: the left thing, the right thing, and the multiplicity. If the multiplicity is "one, guaranteed", use INNER and expect the row count to stay the same. If it is "at most one", use LEFT and expect the row count to stay the same. If it is "possibly many", the row count will grow and you must decide what to do about that before you write an aggregate.

The count that checks your work

```
SELECT count(*) FROM orders;                    -- 31,940
SELECT count(*) FROM orders o
JOIN customers c ON c.id = o.customer_id;       -- 31,940 → good
```

Same number: every order matched exactly one customer. Fewer: an inner join dropped orders with a missing or non-matching customer, and you should find out how many and why. More: a customer id appears more than once in `customers`, which means either the table's grain is not what you thought or there are duplicates, and either way your revenue total is now too big.

Run this comparison every time. It is two queries, it takes ten seconds, and it turns join bugs from something you discover in a meeting into something you discover while writing.

BEFORE YOU MOVE ON

`orders` has 31,940 rows. `SELECT ... FROM orders o LEFT JOIN customers c ON c.id = o.customer_id` returns 34,100 rows. What do the extra 2,160 rows tell you?

A. Some orders have a NULL `customer_id`, and the LEFT JOIN is keeping them with NULL customer details.

B. Some `customer_id` values match more than one row in `customers`, so those orders were emitted more than once.

C. The join condition is missing, so the query has degenerated into a cross join.

D. A LEFT JOIN always returns at least as many rows as both tables combined, so 34,100 is expected and nothing needs checking.

15 · 8 min

When the join key looks right and matches nothing

THE ONE THING TO KEEP

Zero matched rows is usually a lie told by a type mismatch, an invisible character, a case or leading-zero difference, or a NULL; the diagnosis is to compare one pair of values by hand.

Zero rows is an answer, and it is usually a lie

You write a join you are confident about and get back nothing, or a fraction of what you expected. The condition looks correct. Both tables clearly contain the same customers. This lesson is the checklist, in the order the causes actually occur.

Cause one: the types differ

```
-- orders.customer_ref is text, customers.id is bigint
JOIN customers c ON c.id = o.customer_ref
```

Postgres will refuse this outright with a type error, which is the good outcome. MySQL will silently coerce the text to a number, so `'0012'` becomes 12 and matches, while `'abc'` becomes 0 and matches customer zero. SQLite compares a text value and an integer value as *never equal*, so you get zero rows and no complaint at all.

Three databases, three behaviours, one bug. Check the types before you check anything else:

```
SELECT column_name, data_type FROM information_schema.columns
WHERE table_name IN ('orders','customers') AND column_name LIKE
'%id%';
```

Cause two: invisible characters

A CSV export gives you `'CUST-4412 '` with a trailing space, or `'CUST-4412\r'` because the file has Windows line endings and the last column of each row swallowed the carriage return. Neither is visible when you print it.

```
SELECT '[' || customer_ref || ']', length(customer_ref)
FROM orders LIMIT 5;
-- [CUST-4412 ]  10
```

Bracketing the value and printing its length makes both problems obvious in one look. The fix on read is `btrim(customer_ref)`; the better fix is to clean it once on import so every future query does not have to remember.

Cause three: case and leading zeros

`'ade@example.com'` and `'Ade@Example.com'` are different strings to a case-sensitive database, and the domain part of an email address is officially case-

insensitive. Account codes suffer the same fate as `'00412'` versus `'412'` — a system that stored the code as text and another that stored it as a number will disagree forever.

Normalise on both sides, and prefer to do it once at write time. Normalising in the join means no index can help:

```
ON lower(btrim(a.email)) = lower(btrim(b.email))   -- correct,
and slow
```

Cause four: the key is not unique on the side you assumed

You believe `customers.email` is unique. Check rather than believe:

```
SELECT email, count(*) FROM customers GROUP BY email HAVING
count(*) > 1;
```

Any rows here mean your join will multiply. This is the check that catches the most expensive class of join bug, because unlike the previous three it produces a plausible number rather than an empty result.

Cause five: NULLs never match

`NULL = NULL` is unknown, not true, so a row with a NULL key never joins to anything — including another NULL. In an inner join it disappears. In a left join it survives with NULLs on the right. Both are correct and neither is announced.

If you need NULLs to match each other, say so explicitly with `IS NOT DISTINCT FROM`, which treats two NULLs as equal:

```
ON a.region IS NOT DISTINCT FROM b.region
```

Use it rarely and deliberately. Most of the time, a NULL join key means the data is incomplete and joining it away is the honest outcome.

The diagnosis routine

When a join returns the wrong number of rows, do not experiment with join types. Measure, in this order:

```
-- 1. How many keys on each side, and how many are distinct?
SELECT count(*), count(customer_ref), count(DISTINCT cus-
tomer_ref) FROM orders;
SELECT count(*), count(id), count(DISTINCT id) FROM customers;

-- 2. Take five real keys from the left and look for them on
the right.
SELECT DISTINCT customer_ref FROM orders LIMIT 5;
SELECT * FROM customers WHERE id::text IN ('CUST-4412', ...);

-- 3. How many left rows fail to match?
SELECT count(*) FROM orders o
LEFT JOIN customers c ON c.id::text = o.customer_ref
WHERE c.id IS NULL;
```

Step two is the one people skip and the one that solves it. Print the actual key from each side, side by side, and look at them. Ninety per cent of the time the answer is visible immediately: one has a prefix, one has padding, one is a different identifier entirely.

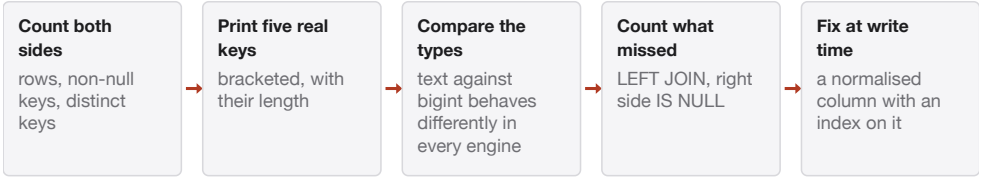
The structural fix

If you find yourself cleaning a key inside a join condition, that cleaning belongs earlier. Add a normalised column to the table, index it, and join on that:

```
ALTER TABLE customers ADD COLUMN email_key text
  GENERATED ALWAYS AS (lower(btrim(email))) STORED;
CREATE INDEX ON customers (email_key);
```

Now the normalisation happens once at write time, the index works, and no future query can forget the rule. A generated column is not available in every database, but a plain column populated on import achieves the same thing.

Diagnosing a join key, in the order the causes actually occur



Do not experiment with join types when a join returns the wrong number of rows. Measure. Step two is the one people skip and the one that solves it: print a real key from each side, bracketed, and look at them.

BEFORE YOU MOVE ON

The same join, `ON o.customer_ref = c.ref`, returns 18,000 matched rows in MySQL and zero rows in SQLite against a copy of the same data. What is the most likely explanation?

A. MySQL is using an index that SQLite lacks, so SQLite cannot find the matching rows.

B. The key columns have different types; MySQL coerces one to the other, while SQLite treats a text value and a number as never equal.

C. SQLite defaults to an inner join and MySQL defaults to a left join when the join type is omitted.

D. SQLite is case-sensitive on text comparisons and MySQL is not, so the keys differ only in capitalisation and every one of the 18,000 matches depends on it.

16 · 8 min

Many-to-many, and the table in the middle

THE ONE THING TO KEEP

A many-to-many relationship needs a table in the middle, and when you count across it a LEFT JOIN's unmatched row still counts as a row unless you count the child's column rather than `*`.

The relationship that needs its own table

A student takes several courses. A course has many students. There is no column you can add to either table that holds this, because a column holds one value and you need many.

The answer is a third table whose rows *are* the relationship:

```
CREATE TABLE enrolments (
  student_id  bigint NOT NULL REFERENCES students(id),
  course_id   bigint NOT NULL REFERENCES courses(id),
  enrolled_on date   NOT NULL DEFAULT current_date,
  status      text   NOT NULL DEFAULT 'active',
  PRIMARY KEY (student_id, course_id)
);
```

This is variously called a bridge, junction, join or associative table. The name does not matter. What matters is that it is a real table with a real grain — one row per student per course — and it can carry its own columns, which is the part people underuse. The date somebody enrolled, whether they dropped out, what they paid: all of these are facts about the *relationship*, not about the student or the course, and this is the only correct place for them.

Querying across it

Two joins, in order:

```
SELECT s.full_name, c.title, e.enrolled_on
FROM students s
JOIN enrolments e ON e.student_id = s.id
JOIN courses    c ON c.id = e.course_id
WHERE c.title = 'Statistics I';
```

Grain check: the result is one row per enrolment, which is one row per student per course. A student on three courses appears three times. That is correct and expected, and it is exactly the multiplication that ruins any per-student total you compute on top of it without grouping first.

Signs the bridge table is missing

You will meet data where somebody avoided the third table. Two symptoms:

A comma-separated column. `students.courses` holding `'stats,algebra,physics'`. Every question becomes a string search. `WHERE courses LIKE '%stats%'` also matches a course called `'biostats'`. Counting how many students take algebra requires counting commas. Adding a course means rewriting a string in every affected row, transactionally, without race conditions, which nobody does correctly.

Numbered columns. `course1, course2, course3`. The moment somebody takes a fourth, you are altering the schema, and every query has to be rewritten to check three columns instead of two. Ask "how many students take Statistics I" and you need `WHERE course1 = 'stats' OR course2 = 'stats' OR course3 = 'stats'`.

Both are the "no repeating groups" rule from the first module, wearing different clothes. The fix in both cases is to unpack them into a bridge table once and never think about it again.

Counting on both sides

Because the bridge table holds the relationship, both directions are the same shape of query:

```
-- students per course
SELECT c.title, count(*) AS students
FROM courses c JOIN enrolments e ON e.course_id = c.id
GROUP BY c.title;

-- courses per student
SELECT s.full_name, count(*) AS courses
FROM students s JOIN enrolments e ON e.student_id = s.id
GROUP BY s.full_name;
```

Note that both of these silently exclude students with no courses and courses with no students, because an inner join drops unmatched rows. If the report is about coverage — which courses nobody signed up for — you need a LEFT JOIN from the side you care about and `count(e.course_id)` rather than `count()`, since `count( )` would count the single NULL-filled row as one.

That distinction, `count(*)` versus `count(column)` after a left join, is the most common way an "empty" category is reported as having one member.

Two useful patterns

"Students taking both A and B" — a question that trips people up because no single row contains both:

```
SELECT student_id
FROM enrolments
WHERE course_id IN (11, 22)
GROUP BY student_id
HAVING count(DISTINCT course_id) = 2;
```

Group and count, rather than trying to join the table to itself. This generalises: change the 2 and the list and it answers "all of these", for any number.

"Courses with no students" — the anti-join:

```
SELECT c.* FROM courses c
LEFT JOIN enrolments e ON e.course_id = c.id
WHERE e.course_id IS NULL;
```

One design note

Whether the bridge table gets its own surrogate id, in addition to the composite primary key, is a real argument with no universal answer. A composite key states the rule that the pair is unique and is self-documenting. A surrogate id makes the table easier to reference from a fourth table and easier for some application frameworks to handle. Pick one, and if you add a surrogate id, keep a UNIQUE constraint on the pair — otherwise the rule you were relying on is gone and duplicate enrolments will appear.

BEFORE YOU MOVE ON

`SELECT c.title, count(*) FROM courses c LEFT JOIN enrolments e ON e.course_id = c.id GROUP BY c.title` shows 1 for a course you know has no students. Why?

A. The LEFT JOIN produced a separate row for the course itself, and `count(*)` includes that course row alongside its enrolment rows, adding one to every count.

B. The unmatched course still yields one output row with NULL enrolment columns, and `count(*)` counts rows whether or not those columns are NULL.

C. `count(*)` cannot be used with GROUP BY on a left-joined table and should be `count(DISTINCT *)`.

D. The courses table contains a duplicate row for each course with no enrolments.

17 · 9 min

Two child tables, one very wrong total

THE ONE THING TO KEEP

Two one-to-many joins from the same parent multiply each other; pre-aggregate each child to one row per parent, then join the summaries.

The bug that survives code review

You have an order. It has line items. It also has payments. Both are perfectly ordinary one-to-many relationships. You write what looks like an obvious query:

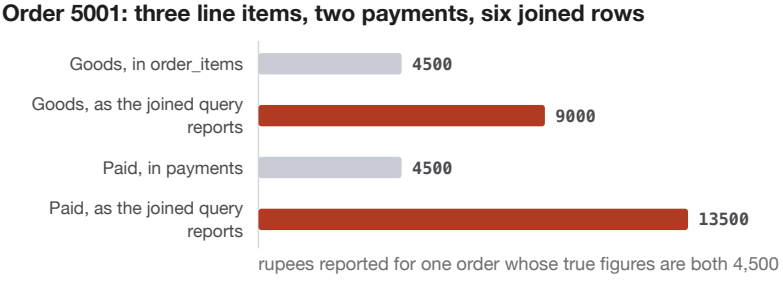
```
SELECT
  o.id,
  sum(i.qty * i.unit_price) AS goods,
  sum(p.amount)             AS paid
FROM orders o
JOIN order_items i ON i.order_id = o.id
JOIN payments    p ON p.order_id = o.id
GROUP BY o.id;
```

Order 5001 has 3 line items and 2 payments. The join produces 3 × 2 = 6 rows. Each line item appears twice, so `goods` is double the truth. Each payment appears three times, so `paid` is triple the truth. Neither number errors, neither looks absurd, and the ratio between them is even preserved badly enough that a sanity check on "did they pay the full amount" may still pass.

This is the **fan trap**, sometimes called the chasm trap in its sibling form. It is the single most expensive join bug in business reporting, and it is invisible in the query text. Nothing about those three lines looks wrong.

Why the multiplication happens

Go back to the mechanism. A join emits one row per match. When you join the parent to child table A, you get one row per A. When you then join that result to child table B, *each of those rows* looks for matches in B — so every A row is paired with every B row for the same parent. The output is the Cartesian product of the two children, per parent.



Each line item is emitted once per payment, so goods double. Each payment is emitted once per line item, so paid triples. Neither number errors and neither looks absurd. Pre-aggregate each child to one row per order, then join the summaries.

The rule: **two independent one-to-many joins from the same parent multiply each other.** Three children multiply three ways. It gets worse quickly: 5 items, 3 payments and 4 shipments give 60 rows for one order.

Spotting it

Two signs, both cheap.

First, the row count. If the joined result has far more rows than either child table, something is fanning. Second, an "impossible" total: revenue larger than the company's revenue, quantities that exceed inventory, a customer who apparently paid eleven times. Distrust any total you did not check against a known figure.

But the reliable detection is structural, not numerical: **count the one-to-many joins in the FROM clause.** More than one from the same parent, and you have a fan trap unless you did something about it.

The fixes

Pre-aggregate each child separately, then join the summaries. This is the correct default:

```
WITH item_totals AS (
  SELECT order_id, sum(qty * unit_price) AS goods
  FROM order_items GROUP BY order_id
),
payment_totals AS (
  SELECT order_id, sum(amount) AS paid
  FROM payments GROUP BY order_id
)
SELECT o.id,
       coalesce(it.goods, 0) AS goods,
       coalesce(pt.paid, 0)  AS paid
FROM orders o
LEFT JOIN item_totals    it ON it.order_id = o.id
LEFT JOIN payment_totals pt ON pt.order_id = o.id;
```

Each CTE has grain "one row per order". Joining one-to-one preserves it. Nothing multiplies, and the LEFT JOINs keep orders that have no payment yet — which is usually the interesting group.

Or use scalar subqueries, when there are only one or two values and readability matters more than speed:

```
SELECT o.id,
  (SELECT sum(qty * unit_price) FROM order_items WHERE order_id
= o.id) AS goods,
  (SELECT sum(amount)           FROM payments    WHERE order_id
= o.id) AS paid
FROM orders o;
```

Each subquery returns exactly one value per order, so the grain cannot change. On modern Postgres the planner often rewrites this into something close to the CTE version; on other engines it can run once per row, which is fine for a thousand orders and painful for a million.

Or `count(DISTINCT ...)`, but only for counts. `count(DISTINCT i.id)` survives the fan because it deduplicates. `sum(i.qty)` does not, and there is no `sum(DISTINCT)` that means what you want — `sum(DISTINCT amount)` would collapse two genuinely separate payments of ₹500 into one. This is why reaching for DISTINCT to "fix" a total is dangerous: it works for counting and quietly corrupts summing.

The chasm trap, briefly

The related failure is joining two children that have *no* rows in common, so an inner join returns nothing for orders that have items but no payments yet — and those are precisely the unpaid orders your report was about. Same cause, opposite symptom: the fan inflates, the chasm deletes. LEFT JOINs on pre-aggregated summaries avoid both.

The habit worth keeping

Before writing a query with more than one join, list the tables and write "one" or "many" next to each. If two "many" tables hang off the same parent, stop and pre-aggregate. It takes fifteen seconds and it is the difference between a report you can defend and one that is quietly wrong by a factor of six.

BEFORE YOU MOVE ON

A query on `orders` joins both `order_items` and `shipments` (about four shipments per order) and sums `qty * unit_price`, giving goods totals roughly four times the true value. Which fix is correct?

A. Add `DISTINCT` to the SELECT list so the duplicated rows are collapsed before aggregation, which restores one row per line item.

B. Change both joins to LEFT JOINs so no rows are dropped.

C. Group by `order_id, item_id, shipment_id` so each combination is counted once.

D. Aggregate `order_items` and `shipments` separately to one row per order first, then join those two summaries to `orders`.

18 · 8 min

Joining a table to itself

THE ONE THING TO KEEP

A self-join is the same table under two names; add `a.id < b.id` so that no row matches itself and no pair appears twice.

The same table, twice, with two names

A join does not care whether the two sides are different tables. When the relationship you want is between rows of the *same* table, you join it to itself and give each copy an alias:

```
SELECT e.full_name AS employee, m.full_name AS manager
FROM employees e
LEFT JOIN employees m ON m.id = e.manager_id;
```

`e` and `m` are the same table. The database does not find this strange. The LEFT JOIN matters: the chief executive has no manager, and an inner join would remove the one person the org chart is rooted on.

Three things self-joins are actually for

Hierarchies. Manager and report, category and parent category, comment and the comment it replies to, region within country. Any table with a column pointing at its own primary key.

Comparing a row to another row of the same kind. Two payments by the same student, this month's figure against last month's, a price against the price of the previous version.

Finding pairs. Which two products are bought together; which two students share a phone number.

For that last one, the essential detail:

```
SELECT a.student_id, b.student_id, a.phone
FROM students a
JOIN students b ON b.phone = a.phone AND b.id > a.id;
```

`b.id > a.id` does two jobs at once. It stops each student matching themselves, and it returns each pair once rather than twice in both orders. Without it you get every row paired with itself plus every genuine pair duplicated, which is the classic wrong answer to "how many duplicates do we have".

Consecutive rows, before window functions

A self-join can find the previous row per group:

```
SELECT p.id, p.paid_on, prev.paid_on AS previous_payment
FROM payments p
LEFT JOIN payments prev
  ON prev.student_id = p.student_id
 AND prev.paid_on = (
       SELECT max(paid_on) FROM payments x
       WHERE x.student_id = p.student_id AND x.paid_on < p.-
paid_on
     );
```

It works, and it is horrible. Read it twice and you can see why: the join condition contains a correlated subquery that runs per row. The window-function module replaces this whole construction with `lag(paid_on) OVER (PARTITION BY student_id ORDER BY paid_on)`, which is one line, faster, and obvious. Know the self-join version because you will meet it in old code and because it shows what the window function is doing underneath; do not write new code this way.

Walking a hierarchy of unknown depth

A plain self-join goes one level. Two joins go two levels. For a tree of unknown depth you need a recursive CTE:

```
WITH RECURSIVE chain AS (
  SELECT id, full_name, manager_id, 1 AS depth
  FROM employees WHERE id = 42            -- the anchor: where
to start
  UNION ALL
  SELECT e.id, e.full_name, e.manager_id, c.depth + 1
  FROM employees e
  JOIN chain c ON e.manager_id = c.id     -- the step: one lev-
el down
)
SELECT * FROM chain;
```

Two halves: a starting row, and a rule for getting from rows you have to rows you do not. The database applies the rule repeatedly until it produces nothing new.

The honest warning: if the data contains a cycle — someone is transitively their own manager, which happens in real HR systems after a reorganisation — this runs until it exhausts memory. Guard it with a depth limit (`WHERE c.depth < 20`) or by carrying the path travelled so far and refusing to revisit a node. Add the guard when you write it, not after the incident.

Performance, briefly

A self-join reads the table twice, and an index on the joined column is doing double duty. `employees(manager_id)` is the index that makes the org-chart query fast. Without it, a table of 50,000 employees means 50,000 lookups against an unindexed column — a quadratic amount of work, which is why these queries have a reputation for suddenly becoming slow when a company grows.

Naming

Alias every self-join with something meaningful. `e` and `m` for employee and manager. `a` and `b` for a symmetric pair. Never `t1` and `t2`, because six months later nobody, including you, will remember which side was which — and in a self-join the two sides are indistinguishable except by their names.

BEFORE YOU MOVE ON

To find students who share a phone number in an 800-row table, you write `students a JOIN students b ON a.phone = b.phone`, expecting about six pairs, and get 812 rows. What produced that count?

A. The join is matching NULL phone numbers to each other, adding a row for every pair of students with no number recorded on file.

B. Every student matches themselves, giving 800 rows, and each of the 6 real pairs is returned twice, once in each direction.

C. A self-join always produces the square of the table size unless a LIMIT is applied.

D. The phone column has trailing whitespace on some rows, creating spurious matches.

19 · 7 min

UNION, INTERSECT and EXCEPT

THE ONE THING TO KEEP

UNION removes rows that are identical in every column, so it silently deletes genuine repeats; UNION ALL keeps everything and is the one to reach for by default.

Stacking results instead of widening them

A join adds columns. A set operation adds rows. When two queries produce the same shape of result and you want them in one list, you stack them:

```
SELECT id, full_name, 'student' AS kind FROM students
UNION ALL
SELECT id, full_name, 'teacher' AS kind FROM teachers;
```

The rules are strict and they are what make this predictable. Both sides need the same number of columns, in the same order, with compatible types. Column names come from the first query; the second query's names are ignored entirely, which surprises people who carefully alias them.

UNION versus UNION ALL, which is not a style choice

`UNION ALL` concatenates. `UNION` concatenates **and removes duplicate rows**, which requires sorting or hashing the entire result.

Two consequences.

The performance one: on a million rows, `UNION` can be several times slower than `UNION ALL` for no benefit if the two sides cannot overlap. Students and teachers are different tables; there is nothing to deduplicate. Use `UNION ALL` unless you specifically want deduplication.

The correctness one, which matters more: `UNION` silently deletes genuine data. Two payments of ₹500 from the same student on the same day, identical in every selected column, become one payment. Your total is now short by ₹500 and nothing indicates it. If the rows carry an id, include it and the problem disappears; if they do not, `UNION` is a data-destroying operation dressed as tidiness.

INTERSECT and EXCEPT

`INTERSECT` returns rows appearing in both results. `EXCEPT` (called `MINUS` in Oracle) returns rows in the first that are not in the second. Both deduplicate by default, with `ALL` variants that do not.

Their best use is comparing two datasets that should be identical:

```
-- rows the ledger has and the bank does not
SELECT reference, amount FROM ledger
EXCEPT
SELECT reference, amount FROM bank;

-- and the other direction, which is a different question
SELECT reference, amount FROM bank
EXCEPT
SELECT reference, amount FROM ledger;
```

Running both directions is the point. `EXCEPT` is not symmetric, and "what did we miss" and "what did they add" are different findings. This pair of queries is the fastest reconciliation tool in SQL, and it also works beautifully for verifying a migration: run the same aggregate against the old and new systems, `EXCEPT` in both directions, and an empty result on both is a proof rather than an impression.

When to prefer a set operation to a join

`EXCEPT` and `NOT EXISTS` can answer the same question:

```
SELECT id FROM students
EXCEPT
SELECT student_id FROM payments;

SELECT s.id FROM students s
WHERE NOT EXISTS (SELECT 1 FROM payments p WHERE p.student_id =
s.id);
```

The `EXCEPT` version is shorter and reads well when you want a single column. The `NOT EXISTS` version lets you select other columns from `students`, which the `EXCEPT` version cannot without adding them to both sides. Both handle NULLs sensibly, unlike `NOT IN`. Choose by what you need in the output.

Ordering and parentheses

`ORDER BY` applies to the whole combined result and belongs at the very end, once:

```
SELECT ... FROM a
UNION ALL
SELECT ... FROM b
ORDER BY 2 DESC;
```

An `ORDER BY` inside one branch is either an error or ignored, depending on the engine. To sort or limit one branch, wrap it in a subquery or a CTE.

Precedence is worth knowing: `INTERSECT` binds tighter than `UNION` and `EXCEPT`, exactly as multiplication binds tighter than addition. Mixing them without brackets is how you get a result nobody predicted. Put the brackets in.

A practical use: the union of similar tables

Data often arrives split by month or by region — `sales_2024`, `sales_2025`, or one table per country. A `UNION ALL` view stitches them into one queryable surface:

```
CREATE VIEW sales AS
  SELECT *, 2024 AS yr FROM sales_2024
  UNION ALL
  SELECT *, 2025 AS yr FROM sales_2025;
```

This works, and it is worth knowing that most databases now offer native table partitioning that does the same thing while letting the planner skip irrelevant partitions entirely. The manual union is the portable version and the right first step; if it becomes slow, partitioning is what you are reaching for next.

BEFORE YOU MOVE ON

You stack two payment exports with `UNION` and the total comes out slightly lower than the two files' totals added together. What happened?

A. The two exports have their columns in a different order, so some amounts were added into the wrong column and silently left out of the final total.

B. `UNION` removed rows identical across every selected column, which included genuine repeat payments of the same amount on the same day.

C. `UNION` cannot combine results from different sources, so one export was silently truncated.

D. The sum skipped rows where any column was NULL, because set operations exclude NULL rows.

20 · 8 min

Subqueries: a query inside a query

THE ONE THING TO KEEP

Reach for EXISTS and NOT EXISTS over IN and NOT IN: they say what you mean, cannot be poisoned by a NULL, and let the planner turn them into a join.

Four shapes, four uses

A subquery is a SELECT inside another statement. It appears in four positions, and they behave differently enough to be worth separating.

Scalar, in the SELECT list. Returns exactly one row and one column, becoming a value on each output row:

```
SELECT o.id, o.amount,
       (SELECT full_name FROM customers c WHERE c.id = o.cus-
tomer_id) AS customer
FROM orders o;
```

It cannot change the grain, which is its great virtue. If it ever returns two rows, the query fails rather than silently duplicating — a much better failure than a join's.

In the FROM clause, as a derived table. It must have an alias:

```
SELECT city, avg(order_count) FROM (
  SELECT customer_id, city, count(*) AS order_count
  FROM orders GROUP BY customer_id, city
) per_customer
GROUP BY city;
```

This is how you aggregate an aggregate — the average number of orders per customer, by city, which no single GROUP BY can express.

In WHERE, with IN or a comparison. Filters against a list computed elsewhere:

```
SELECT * FROM students
WHERE id IN (SELECT student_id FROM payments WHERE amount >
20000);
```

Correlated, referencing the outer query. The subquery mentions a column from outside, so conceptually it runs once per outer row:

```
SELECT * FROM orders o
WHERE o.amount > (SELECT avg(amount) FROM orders WHERE cus-
tomer_id = o.customer_id);
```

"Orders larger than that customer's own average." The `o.customer_id` reference is what makes it correlated.

EXISTS, and why it is the one to reach for

```
SELECT s.* FROM students s
WHERE EXISTS (SELECT 1 FROM payments p WHERE p.student_id =
s.id);
```

`EXISTS` asks a yes/no question and stops at the first matching row. `SELECT 1` is conventional — the columns are never read, so people write the cheapest thing possible.

Three reasons to prefer it over the alternatives.

It **cannot multiply rows**. A join to `payments` would return one row per payment; `EXISTS` returns one row per student regardless of how many payments there are. When the question is "which students have paid" rather than "what did they pay", this is exactly right and no `DISTINCT` is needed.

It **handles NULL correctly**. `NOT IN` against a list containing a NULL returns zero rows, always, as the NULL lesson shows in detail. `NOT EXISTS` has no such trap.

It **stops early**. A student with 400 payments is settled by the first one.

IN versus EXISTS versus JOIN

For a simple membership test on modern Postgres, all three are usually rewritten into the same plan, and the choice is about readability. The differences that survive:

- Use `EXISTS` when you want one row per outer row and only care whether a match exists.
- Use `JOIN` when you need columns from the other table in the output.
- Use `IN` when the list is small and literal, or comes from a simple subquery, and you find it clearer.
- Never use `NOT IN` against a subquery on a nullable column.

That last one is worth writing on a wall. `NOT IN (SELECT student_id FROM payments)` returns nothing at all if a single `student_id` is NULL, and return-

ing nothing looks like a valid finding.

The performance shape to know

A correlated subquery *can* run once per outer row. Ten thousand outer rows and a subquery costing 2 ms is twenty seconds. Postgres often decorrelates these into a single join or hash aggregate, and then the cost disappears — but "often" is not "always", and other engines are less reliable about it.

The diagnostic is direct: run `EXPLAIN ANALYZE` and look for a node with `loops=10000`. That number is telling you the subquery ran ten thousand times. When you see it and the query is slow, rewrite the correlated form as a pre-aggregated CTE joined once:

```
WITH avg_per_customer AS (
  SELECT customer_id, avg(amount) AS avg_amount FROM orders
GROUP BY customer_id
)
SELECT o.* FROM orders o
JOIN avg_per_customer a ON a.customer_id = o.customer_id
WHERE o.amount > a.avg_amount;
```

One pass to build the averages, one join to use them. Same answer, and it stays fast as the table grows.

Readability

Subqueries nested three deep are the hardest SQL to read, because you have to hold the inner result in your head while parsing the outer one. Beyond one level of nesting, name the pieces with CTEs — the next lesson. A query built from four named steps is understandable by somebody who has never seen it; the same logic nested four deep is not.

BEFORE YOU MOVE ON

`SELECT * FROM customers WHERE id NOT IN (SELECT customer_id FROM blocked_customers)` returns no rows at all, although only a dozen customers are blocked. Why?

A. The subquery returns more rows than `NOT IN` can handle, so the comparison fails and returns nothing.

B. `NOT IN` requires the subquery to return a single column backed by a unique constraint, and without one the comparison is undefined and returns nothing.

C. At least one `customer_id` in `blocked_customers` is NULL, so the expanded chain of inequalities contains a comparison that can never be true.

D. The two `customer_id` columns have different types, so no value ever matches and the negation excludes everything.

21 · 7 min

CTEs: naming the steps

THE ONE THING TO KEEP

A CTE names a step so a query reads in order; it is not a saved result, and since Postgres 12 it is inlined into the main query unless you write MATERIALIZED.

A query you can read a year later

A CTE — a `WITH` clause — gives a name to an intermediate result:

```
WITH recent_orders AS (
  SELECT * FROM orders
  WHERE placed_at >= current_date - interval '90 days'
    AND status = 'paid'
),
per_customer AS (
  SELECT customer_id, count(*) AS orders, sum(amount) AS spend
  FROM recent_orders
  GROUP BY customer_id
)
SELECT c.full_name, p.orders, p.spend
FROM per_customer p
JOIN customers c ON c.id = p.customer_id
WHERE p.spend > 50000
ORDER BY p.spend DESC;
```

Read it top to bottom: take recent paid orders, summarise them per customer, attach names, keep the big spenders. Four sentences, in the order a person would say them. The same logic as nested subqueries is the same length and takes five times as long to understand, because you have to read it inside out.

Later CTEs can reference earlier ones, which is what makes them compose. This is the single biggest readability improvement available in SQL, and it costs nothing.

What a CTE is not

It is not a variable, and it is not stored. It exists for the duration of the one statement. Running the query again recomputes everything.

It is also **not necessarily evaluated once**. Older Postgres treated every CTE as an optimisation fence — always materialised, always exactly once, never merged into the outer query. Since version 12 the planner may inline a CTE that is referenced once, which is usually faster and occasionally changes behaviour you were relying on. You can force either:

```
WITH heavy AS MATERIALIZED ( ... )      -- compute once, keep
the result
WITH light AS NOT MATERIALIZED ( ... )  -- inline it into the
outer query
```

Reach for `MATERIALIZED` when a CTE is expensive and used several times. Reach for `NOT MATERIALIZED` when a CTE selects from a large table and the outer query has a filter that could have used an index — materialising it first would read the whole table before filtering.

Other engines differ. MySQL 8 and SQLite support CTEs; older MySQL does not have them at all, and derived tables in the FROM clause are the portable fallback.

Recursive CTEs

`WITH RECURSIVE` builds a result from itself, which is how you walk a tree or generate a series:

```
WITH RECURSIVE months AS (
  SELECT date '2026-01-01' AS m
  UNION ALL
  SELECT (m + interval '1 month')::date FROM months WHERE m <
date '2026-12-01'
)
SELECT * FROM months;
```

An anchor row, a rule for the next row, and a stopping condition. Leave out the stopping condition and it runs forever — or until the server runs out of memory, which is the same thing from your point of view.

In Postgres, `generate_series` does this particular job better and the next lesson uses it. Recursion earns its place on genuine hierarchies: an org chart, a category tree, a bill of materials, the reply chain of a comment thread.

CTEs that write

In Postgres a CTE can contain `INSERT`, `UPDATE` or `DELETE` with a `RETURNING` clause, which makes multi-step data changes atomic:

```
WITH moved AS (
  DELETE FROM orders WHERE placed_at < '2024-01-01' RETURNING *
)
INSERT INTO orders_archive SELECT * FROM moved;
```

The delete and the insert happen in one statement, so there is no window where the rows exist in neither table. This is a genuinely useful pattern for archiving and for audit logging, and it is Postgres-specific.

One rule that trips people: all parts of the statement see the *same snapshot* of the data. A CTE that updates a row and a later part of the same statement that reads that row will not see the update. Two statements in a transaction, not one statement with two CTEs, when you need the second step to see the first.

When a CTE is the wrong tool

Twelve CTEs chained into one enormous statement is not more readable than four, it is a program written in the wrong language. When a query reaches that size, the pieces usually deserve to be **views** — named, reusable, documented, testable on their own. A view that ten queries share is one definition to fix when the business rule changes; the same logic copied into ten CTEs is ten places to forget.

The rule of thumb: a CTE is for structuring one query. A view is for a definition that more than one query needs.

BEFORE YOU MOVE ON

A query beginning `WITH recent AS (SELECT * FROM events) SELECT … FROM recent WHERE day = '2026-03-14'` took 40 seconds on Postgres 11 and under a second on Postgres 16 against the same data and indexes. What changed?

A. Newer versions store CTE results in memory rather than on disk, so the whole events table is held in RAM and the temporary file writes that dominated the old run are avoided.

B. The newer version added an index on the events table automatically during query planning.

C. Older versions always materialised the CTE, reading the whole table before filtering; newer ones can inline it so the day filter applies during the scan.

D. The older version evaluated the CTE once per output row, and the newer one evaluates it once.

22 · 8 min

Generating the rows that are not there

THE ONE THING TO KEEP

GROUP BY cannot produce a row for a period in which nothing happened; generate the spine of expected rows, LEFT JOIN the facts to it, and count the fact column rather than `*`.

The zero that never appears

A grouped query can only report on rows that exist. In March, the Kochi branch made no sales. `GROUP BY branch, month` therefore produces no pile for Kochi in March, and no row.

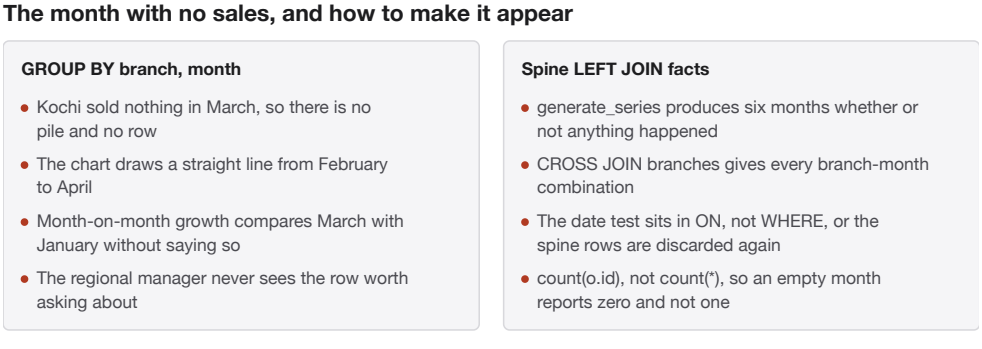
The consequences are worse than a missing line in a table. A chart draws a straight line from February to April, implying steady trade through a month

with none. A month-on-month growth calculation compares March to January without saying so. A report emailed to a regional manager simply lacks the row that would have prompted a question.

Missing rows are much harder to notice than wrong numbers, because nothing is there to look wrong.

Generate the frame first

The fix is to build the complete set of rows you want — the **spine** — and left join the data onto it. In Postgres:



A grouped query can only report on rows that exist, so a branch with no sales in March produces no pile and therefore no row. Missing rows are much harder to notice than wrong numbers, because nothing is there to look wrong.

```
SELECT generate_series(date '2026-01-01', date '2026-12-01',
interval '1 month')::date AS month;
```

Twelve rows, whether or not anything happened in them. Now cross join it with the branches to get every combination:

```
WITH months AS (
  SELECT generate_series(date '2026-01-01', date '2026-06-01',
interval '1 month')::date AS month
),
spine AS (
  SELECT b.id AS branch_id, b.name, m.month
  FROM branches b CROSS JOIN months m
)
SELECT
  s.name,
  s.month,
  coalesce(sum(o.amount), 0) AS revenue,
  count(o.id)                AS orders
FROM spine s
LEFT JOIN orders o
       ON o.branch_id = s.branch_id
      AND o.placed_at >= s.month
      AND o.placed_at <  s.month + interval '1 month'
GROUP BY s.name, s.month
ORDER BY s.name, s.month;
```

Read the three parts that make it work.

The `CROSS JOIN` is deliberate — this is the one place a Cartesian product is exactly what you want. Six months times four branches is twenty-four rows, guaranteed.

The date condition sits in the `ON` clause, not in `WHERE`. This is the LEFT JOIN trap from the joins lesson: a `WHERE o.placed_at >= ...` would discard the spine rows that matched nothing, undoing the entire exercise.

`coalesce(sum(o.amount), 0)` turns the NULL from an empty pile into a zero. And note `count(o.id)`, not `count()` — *the unmatched spine row still produces one output row, so* `count()` would report one order in a month with none.

Portable ways to make a spine

`generate_series` is Postgres and DuckDB. Elsewhere:

A recursive CTE works in SQLite, MySQL 8 and SQL Server:

```
WITH RECURSIVE months(m) AS (
  SELECT date '2026-01-01'
  UNION ALL
  SELECT date(m, '+1 month') FROM months WHERE m < date '2026-
06-01'
)
SELECT * FROM months;
```

A permanent calendar table is the approach most data warehouses take, and it is better than either. One row per date from 2015 to 2035, with columns for the month, quarter, fiscal year, weekday, and whether it is a public holiday in each market you operate in. Ten minutes to build, and every date question afterwards becomes a join instead of an argument about fiscal quarters.

```
CREATE TABLE calendar AS
SELECT d::date AS day,
       date_trunc('month', d)::date AS month_start,
       extract(isodow FROM d)       AS weekday,
       extract(isodow FROM d) >= 6  AS is_weekend
FROM generate_series(date '2015-01-01', date '2035-12-31', in-
terval '1 day') d;
```

The holiday column cannot be computed and has to be loaded, which is exactly why having the table matters: it gives holidays somewhere to live, instead of being hard-coded into whichever query needed them last.

The same trick for categories

Spines are not only for dates. Any report that must show every category, including the empty ones, is the same shape: start from the dimension table, left join the facts.

```
SELECT c.title, count(e.student_id) AS students
FROM courses c
LEFT JOIN enrolments e ON e.course_id = c.id
GROUP BY c.title;
```

Courses with nobody enrolled appear with zero — which is very often the row somebody actually needs to see.

The judgement call

Not every gap should be filled with a zero. A missing month for a branch that opened in April is not zero revenue, it is *not applicable*, and printing zero implies a business that existed and sold nothing. Bound the spine by the entity's own lifetime:

```
FROM branches b
CROSS JOIN months m
WHERE m.month >= date_trunc('month', b.opened_on)
  AND (b.closed_on IS NULL OR m.month <= date_trunc('month',
b.closed_on))
```

The distinction between "zero happened" and "the question does not apply" is real, and it is the same distinction as zero versus NULL from the earlier lesson. A spine gives you the power to assert one of them; use it to assert the one that is true.

BEFORE YOU MOVE ON

A report LEFT JOINs a generated branch-by-month spine to `orders` and shows an order count of 1 in months where a branch had no orders at all, with revenue correctly shown as 0. What is wrong?

A. The LEFT JOIN should be a FULL OUTER JOIN so unmatched order rows also appear.

B. `coalesce` was applied to the revenue but not to the order count, so NULL is being displayed as 1.

C. The count is `count(*)`, which counts the NULL-filled row from the unmatched spine entry, instead of `count(o.id)`, which counts only real orders.

D. The spine contains a duplicate row for each empty month, so one phantom order is being counted per duplicate even though nothing was sold in that month.

CASE STUDY · MODULE 2

Twelve lakh rupees that were never spent

A district health programme in Nashik, reporting drug costs and visit counts to a state funder, with the report due at ten the next morning

The programme runs 46 sub-centres. Its database has three tables that matter tonight. `visits` holds one row per patient visit: 214,000 rows for the year. `dispensations` holds one row per medicine handed out during a visit, with a quantity and a unit cost: 611,000 rows. `referrals` holds one row per onward referral made during a visit: 38,000 rows.

The funder wants one page: for each sub-centre, the number of visits, the total drug cost, and the number of referrals. Priya, the programme's data officer, writes what the question asks for.

```
SELECT v.centre_id,
       count(*)                          AS visits,
       sum(d.qty * d.unit_cost)          AS drug_cost,
       count(r.id)                       AS referrals
FROM visits v
JOIN dispensations d ON d.visit_id = v.id
LEFT JOIN referrals   r ON r.visit_id = v.id
GROUP BY v.centre_id;
```

The total drug cost comes back as 1,84,00,000 rupees. She knows, without checking anything, that the programme's entire drug budget for the year was 1,72,00,000 and that it underspent. The number is twelve lakh above a ceiling it cannot have crossed.

She checks the row count. `visits` has 214,000 rows; the joined result has 638,000. Something on the right matched more than once, which she expects — a visit has several dispensations. But the drug cost should still be right, because each dispensation appears once per visit. It is the referrals join that is doing the damage in the other direction, and the drug cost that is inflated by it: a visit with four dispensations and two referrals emits eight rows, so each of the four dispensation rows is counted twice.

It is 8pm. Priya has three options and each costs something.

The first is to pre-aggregate. Two CTEs, one summing dispensations per visit and one counting referrals per visit, then join both summaries to `visits` one row to one row. It is the correct answer, it is about fifteen minutes of typing, and it needs to be checked afterwards, which is another half hour she has.

The second is faster and is what a colleague suggests on the phone: add `DISTINCT`. `count(DISTINCT r.id)` for referrals and `count(DISTINCT v.id)` for visits both become right immediately. The drug cost does not, because there is no `sum(DISTINCT)` that means what she needs — `sum(DISTINCT d.qty * d.unit_cost)` would collapse two genuinely separate dispensations of the same medicine at the same price into one, and would under-report instead of over-reporting. The colleague's fix repairs the two numbers she can see are wrong and quietly corrupts the one she cannot.

The third is to send last year's methodology note with a figure computed the way it was computed last year, which she knows was the same query, and flag the discrepancy in a footnote. That costs nothing tonight. It also means that if last year's report used this query, last year's drug cost was inflated the same way, and the funder has been told a wrong number twice.

The funder's officer has already said, in an email Priya has open, that a resubmission after the deadline moves the district to the next quarterly cycle, which delays a tranche of about forty lakh by three months.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Priya took the first option and it took fifty minutes rather than fifteen, because the pre-aggregated version returned 46 rows against 44 the original had produced. Two sub-centres had visits and no dispensations at all, and the inner join to `dispensations` had been dropping them — the chasm trap sitting un-

derneath the fan trap, deleting two centres from a report that also inflated the rest. Both were centres that had run out of stock in February, which was the most useful finding in the whole exercise and appeared nowhere in the funder's requested columns. The corrected drug cost was 92,00,000 rupees, almost exactly half the original, which is what two referrals per visit on average will do. She sent the report at 9.40pm with a note saying that the previous year's figure had been produced by the same query and was likely to be similarly overstated. The funder's officer replied that the previous year's figure had never been reconciled against the drug budget by anyone, and asked for the corrected series. The two out-of-stock centres received a supervisory visit in March.

State the mechanism that inflated the drug cost, in one sentence, without using the words "fan trap".

Two independent one-to-many joins hang off the same parent, so every dispensation row for a visit is paired with every referral row for that visit, and the output is the Cartesian product of the two children per visit — which means each dispensation's cost is summed once per referral.

The colleague's DISTINCT fix repairs the counts. Why is applying the same instinct to the drug cost worse than leaving it wrong?

Because it fails in the opposite direction and is harder to notice. `count(DISTINCT ...)` deduplicates identifiers, which is exactly right. There is no summing equivalent: `sum(DISTINCT qty * unit_cost)` deduplicates values, so two separate dispensations that genuinely cost the same amount collapse into one and the total is understated. An overstated figure that crosses a known budget ceiling gets caught; an understated one that sits comfortably below it does not.

The corrected query returned 46 rows where the original returned 44. What check would have caught that before the totals were even looked at, and why does it belong in every multi-join query?

A row ledger: count the rows at each step and compare with the grain you claimed. She claimed one row per sub-centre and there are 46 sub-centres, so a result with 44 rows has lost two before any arithmetic happened. The same ledger catches the opposite fault — a count that rises after a join means something on the right matched more than once. It takes seconds, it is arithmetic rather than judgement, and it is the only check that sees both the inflation and the deletion.

MODULE 2 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 `FROM orders o LEFT JOIN refunds r ON r.order_id = o.id WHERE r.reason <> 'damaged'` was meant to return all orders. It returns only orders that have a refund. Why?

A. For an unmatched order r.reason is NULL, so the test is unknown, not true

B. A LEFT JOIN keeps unmatched rows only when there is no WHERE clause

C. `refunds` holds no row for those orders, so the join emitted nothing for them

D. The comparison should have been written with != rather than with <>

2 Which check tells you a query has a fan trap before you look at any total?

A. Whether the result contains rows that are duplicated in every column

B. Counting the one-to-many joins that hang off the same parent table

C. Whether the total is larger than the same figure was last month

D. Adding DISTINCT and seeing whether the number changes

3 A join on `c.id = o.customer_ref` returns zero rows, although both tables plainly contain the same customers. Which step should come first?

A. Rebuild the index on customer_ref so the comparison can use it

B. Wrap both sides in lower() and btrim() and try again

C. Print one key from each side, bracketed, with its length

D. Switch to a LEFT JOIN to see whether more rows come back

4 Two payments of 500 rupees from the same student, on the same day, are identical in every selected column. The two result sets containing them are combined with UNION. What happens?

A. The query errors, because the two rows cannot be told apart

B. Both rows survive; UNION removes only rows sharing a primary key

C. One row is removed, and the total is short by 500

D. Both rows survive, but the order they appear in is undefined

5 You want one row per student who has paid, with no duplicates and no DISTINCT. Why does `EXISTS` suit this better than a join to `payments`?

A. EXISTS evaluates its subquery once, whereas a join evaluates it per row

B. A join cannot appear inside a WHERE clause in standard SQL

C. EXISTS produces a boolean column that the outer query can group on

D. EXISTS asks whether any match exists, so it cannot multiply rows

6 A monthly report LEFT JOINs a spine of every branch-month to `orders`. Why must it use `count(o.id)` rather than `count(*)`?

A. count(*) counts the unmatched spine row, so an empty month reports one order

B. count(*) reads every column of the row and is therefore slower on a wide table

C. count(*) is not permitted in a query that also has a GROUP BY clause

D. They return the same value; the difference appears only when DISTINCT is used

MODULE 3

Summaries, and the piles behind them

A summary is a claim about a pile of rows, and most wrong summaries are right about the arithmetic and wrong about the pile. This block starts with GROUP BY and NULL, then works through the places an aggregate quietly changes meaning: integer division, averages of averages, percentiles, week boundaries, rates with a moving denominator, subtotals, wide reports built from long data, and the reconciliation habit that catches all of them before a number leaves your hands.

> BY THE END OF THIS MODULE YOU CAN
>
> Compute any count, sum, average, percentile or rate at a stated grain with NULLs and time boundaries handled deliberately, produce subtotals and a wide report from long data in one query, and reconcile every total against a figure you already trust before sending it

23 · 7 min

GROUP BY: making piles and asking about each one

THE ONE THING TO KEEP

Each aggregate answers a question about one pile, and COUNT(*) and COUNT(DISTINCT) are different questions.

Piles

`GROUP BY city` takes all your rows and sorts them into piles, one pile per distinct city. Then every aggregate you write — `COUNT`, `SUM`, `AVG`, `MAX` — is computed **inside one pile**, and you get **one output row per pile**.

```
SELECT   city, COUNT(*) AS orders, SUM(amount) AS revenue
FROM     orders
WHERE    placed_at >= '2026-03-01' AND placed_at < '2026-04-01'
GROUP BY city
ORDER BY revenue DESC;
```

Thirty thousand rows go in. Fourteen rows come out, one per city. That is the whole idea.

Why the database refuses your query

Add a column and it breaks:

```
SELECT city, customer_name, COUNT(*)
FROM orders
GROUP BY city;
-- ERROR: column "orders.customer_name" must appear in the
GROUP BY clause
--        or be used in an aggregate function
```

The error looks bureaucratic. It is not. The Lagos pile contains 900 rows with 900 different customer names, and the output has exactly one slot for Lagos. Which of the 900 should go there? There is no right answer, so the database refuses to invent one.

Every column in your SELECT must be either something you grouped by (so it is the same for the whole pile) or wrapped in an aggregate (so the pile is squeezed into a single value). MySQL in some configurations will let this slide and pick a name at random. That is worse, not better.

The three counts

These are different questions and they routinely get confused.

`COUNT(*)` counts rows in the pile. Full stop.

`COUNT(phone)` counts rows in the pile where `phone` is not NULL. Aggregates skip NULLs, always.

`COUNT(DISTINCT customer_id)` counts how many different customers appear in the pile.

So in a table where one row is one order:

- `COUNT(*)` is **how many orders**
- `COUNT(DISTINCT customer_id)` is **how many customers ordered**

Neither is more correct. They answer different questions, and the whole skill is knowing which question you were asked. A repeat customer with four orders is one customer and four orders, and both facts are true at the same time.

WHERE before, HAVING after

`WHERE` filters rows on their way into the piles. `HAVING` filters whole piles on their way out.

```
SELECT   city, SUM(amount) AS revenue
FROM     orders
WHERE    status = 'paid'          -- discard rows before piling
GROUP BY city
HAVING   SUM(amount) > 500000     -- discard piles after sum-
ming
ORDER BY revenue DESC;
```

You cannot put `SUM(amount) > 500000` in WHERE, because when WHERE runs the piles do not exist yet. You *can* put `status = 'paid'` in HAVING in some databases, but do not: it means the same thing, runs later, and reads as though it is a statement about the city rather than about individual orders.

Two things worth knowing early

No GROUP BY means one pile containing everything. `SELECT COUNT(*) FROM orders` is a group query with a single group. That is why you get one row back.

Grouping by two columns makes a pile per combination. `GROUP BY city, month` gives one row per city per month. If you have 14 cities and 12 months, expect up to 168 rows, and expect fewer, because combinations with no rows produce no pile at all. A city with zero orders in July does not appear as a zero — it simply is not there. If your report needs that zero, you have to supply the list of months yourself and join to it. Missing rows are much harder to notice than wrong numbers.

BEFORE YOU MOVE ON

In a table with one row per order, `COUNT(*)` for March returns 812 and `COUNT(DISTINCT customer_id)` returns 540. A colleague says the gap means the data needs cleaning. What is actually going on?

A. The 812 is inflated by duplicate rows, and the duplicates should be removed before either number is reported
B. COUNT(DISTINCT ...) is the safer number and should be preferred whenever the two disagree
C. Each row is one order, so a customer who ordered several times is counted once per order; both numbers are correct and answer different questions
D. The WHERE clause ran before the grouping, so rows were counted that the grouping would have excluded

24 · 8 min

NULL, and why your counts are wrong

THE ONE THING TO KEEP

WHERE keeps only rows where the test is true, and every comparison with NULL is unknown.

NULL means unknown

It does not mean zero. It does not mean empty string. It means the database has no value here, and it will not pretend otherwise.

That produces three-valued logic. A comparison can be true, false, or **unknown**. And there is one rule that explains almost every NULL surprise you will ever hit:

WHERE keeps a row only when the condition is true. Not false. Not unknown.

The comparisons that never match

```
SELECT * FROM students WHERE phone = NULL;   -- always zero
rows
```

Not an error. Zero rows, even for the students whose phone genuinely is NULL. "Unknown equals unknown" is unknown, not true. The correct form is a different operator entirely:

```
SELECT * FROM students WHERE phone IS NULL;
SELECT * FROM students WHERE phone IS NOT NULL;
```

The same rule bites on inequality, which is far less obvious. Suppose `status` is `'active'`, `'churned'`, or NULL for people who never set it.

```
SELECT COUNT(*) FROM users WHERE status <> 'churned';
```

The NULL people are not returned. `NULL <> 'churned'` is unknown, and unknown is not true, so those rows are dropped. You asked for everyone who is not churned and you got only the actives. Nothing warns you. The number is simply smaller than the truth, and it looks like a perfectly ordinary number.

```
WHERE (status <> 'churned' OR status IS NULL)   -- what you
meant
```

The denominator you did not notice changing

Aggregates skip NULLs. This is usually what you want, and occasionally it destroys your analysis.

A survey has 1,000 responses. Two hundred people skipped the monthly income question.

- `COUNT(*)` → 1000
- `COUNT(income)` → 800

- `AVG(income)` → the total of 800 values, divided by **800**

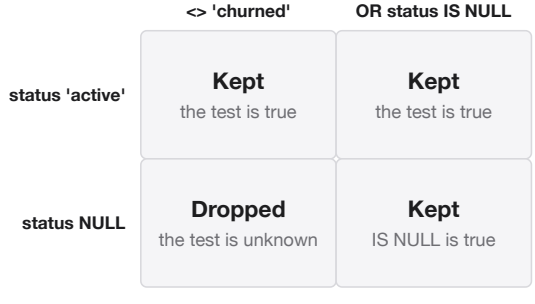
That last one is the trap. You will report "average monthly income of respondents: 47,200 pesos" and it is really the average among the 800 who answered. If people with low incomes were likelier to skip the question — which is exactly what happens in real surveys — your number is biased upward, and nothing in the output hints at it.

Also: `SUM` over a column that is entirely NULL returns NULL, not 0. A revenue tile on a dashboard showing a blank rather than a zero has usually met this.

The NOT IN disaster

This one deserves its own warning.

Why the unknown rows vanish from a count



WHERE keeps a row only when the condition is true. NULL <> 'churned' is not false, it is unknown, and unknown is not true. You asked for everyone who is not churned and you were given only the actives, with nothing to warn you.

```
SELECT * FROM students
WHERE id NOT IN (SELECT student_id FROM payments);
```

If even one row of `payments.student_id` is NULL, this returns **zero rows, always**. Because `id NOT IN (1, 2, NULL)` expands to `id <> 1 AND id <> 2 AND id <> NULL`, and that last term is unknown, so the whole AND can never be true.

Zero rows looks like a real answer. "Great, every student has paid." Use `NOT EXISTS` or the anti-join from the joins lesson instead — both handle NULLs

the way you expect:

```
SELECT s.* FROM students s
WHERE NOT EXISTS (SELECT 1 FROM payments p WHERE p.student_id =
s.id);
```

Do not paper over it

`COALESCE(income, 0)` replaces NULL with zero. It is the right tool sometimes and a lie other times.

If your column means "amount refunded" and no refund happened, zero is genuinely correct. If your column means "reported income" and someone skipped the question, replacing it with zero manufactures 200 destitute respondents who do not exist. The distinction between "it was zero" and "we do not know" is real information, and once you have flattened it you cannot get it back.

The habit

For every nullable column in a query, decide out loud what should happen to the unknowns: counted, excluded, or replaced. Then write that decision into the query so the next person can see it. Most NULL bugs are not misunderstandings of SQL. They are decisions nobody made.

BEFORE YOU MOVE ON

A `status` column holds 'active', 'churned', or NULL for people who never set it. A team wants everyone who is not churned and runs `WHERE status <> 'churned'`. Which people come back?

A. The 'active' people and the NULL people, since NULL is not equal to 'churned'

B. Only the 'active' people

C. No rows at all, because a NULL anywhere in the column makes the comparison unknown for the whole query

D. The 'active' and NULL people if status is a text column, but only the 'active' ones if it is an enum

25 · 9 min

Averages: the four ways AVG misleads

THE ONE THING TO KEEP

An average is a sum divided by a count, and both halves can be silently wrong: integers truncate, unequal rows need weights, and averages of averages answer a different question than the average of the rows.

A mean is a sum divided by a count, and each half can go wrong

`avg(amount)` looks like the safest function in SQL. It is the one most often wrong in a way nobody notices, because an average is always a plausible-looking number. There are four distinct mechanisms, and each has a different fix.

One: integer division

Type `SELECT 7 / 2` into Postgres and you get `3`. SQLite and SQL Server also say `3`. MySQL says `3.5000` and DuckDB says `3.5`. The expression is identical; the engines disagree about whether dividing two integers should produce an integer.

This matters the moment you compute an average by hand:

```
SELECT sum(qty) / count(*) AS avg_qty FROM order_items;   -- 2,
in Postgres
SELECT avg(qty)            AS avg_qty FROM order_items;   --
2.4375
```

`sum(qty)` is an integer, `count()` *is an integer, and the division truncates before you ever see it.* `avg()` *is safe because it promotes to* `numeric` *internally. If you must divide, cast one side first:* `sum(qty)::numeric / count( )`. The bug hides best in percentages, where `paid / total * 100` produces `0` for every row under 100%, because `paid / total` is `0` before the multiplication happens.

Two: the average of averages

A regional manager asks for the average order value across two branches.

- Branch A: 100 orders, average ₹500. Total ₹50,000.
- Branch B: 10 orders, average ₹5,000. Total ₹50,000.

Average the two branch averages and you get ₹2,750. Compute the true average and you get ₹100,000 over 110 orders, which is ₹909. The first number is not an approximation of the second; it is an answer to a different question ("what is the average branch's average?"), and it is nearly three times larger.

This happens whenever a summary table is summarised again. A `daily_stats` table with an `avg_order_value` column cannot be averaged to get the monthly figure. You need the daily *sum* and the daily *count* stored separately, then `sum(total) / sum(orders)`. Sums and counts combine; averages do not. The same rule breaks percentiles, which the next lesson covers, and it is why a well-designed summary table stores components rather than results.

Three: the weight that was not applied

"Average price per unit" is not `avg(unit_price)`. If one line sells 1,000 units at ₹10 and another sells 1 unit at ₹1,000, `avg(unit_price)` is ₹505, and the

real price people paid per unit is ₹11,000 over 1,001 units, which is about ₹10.99.

```
SELECT sum(qty * unit_price) / sum(qty) AS price_per_unit
FROM order_items;
```

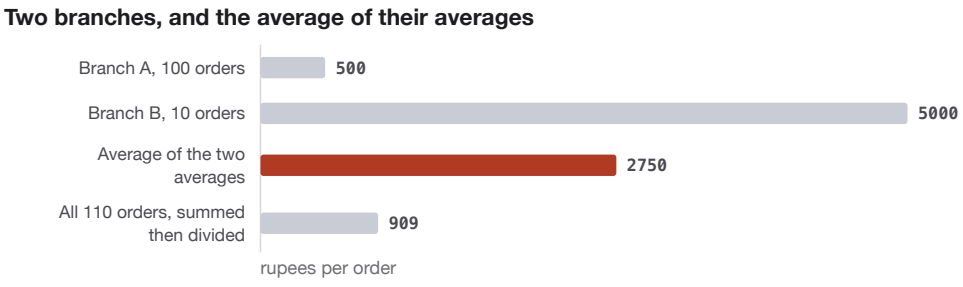
The pattern is always the same: multiply each value by its weight, sum, and divide by the sum of the weights. Any time the rows in your pile are not equally important — orders of different sizes, days with different traffic, students taking different numbers of exams — an unweighted `avg()` answers a question about rows, not about the thing you care about.

Four: rounding at the wrong moment

Round at the end, never in the middle. A query that rounds each row to two decimals and then sums 40,000 rows carries up to 40,000 half-cent errors. Postgres, when it rounds `numeric`, rounds ties away from zero, so `round(2.5)` is `3`; Python's `round(2.5)` is `2`, because it rounds ties to even. A figure computed in SQL and re-checked in a notebook can legitimately disagree by one unit in the last place, and knowing that saves an afternoon.

And the one everyone knows

The mean sits where nobody is when the distribution has a long tail. The earlier lesson on a right number versus a useful number made that point; the operational rule is that `avg()` on money, latency, or anything with outliers should be reported next to a median, and the next lesson shows how to get one.



Branch A took 100 orders and branch B took 10, so the two branch averages are not equally important. Averaging them answers a different question — what is the average branch's average — and gives an answer three times the truth. Sums and counts combine; averages do not.

The habit

Before writing `avg(x)`, ask three things. Is `x` an integer that will be divided by an integer? Is each row equally important, or does it need a weight? And is this average being computed from rows, or from a table that already contains averages? A yes to the first, a no to the second, or a yes to the third means `avg()` is the wrong tool, and the fix is a sum and a count you control yourself.

BEFORE YOU MOVE ON

A `daily_stats` table stores, for each day, `orders` and `avg_order_value`. An analyst computes last month's average order value as `avg(avg_order_value)` over the month's 31 rows. What is wrong with that figure?

A. Nothing, provided every day had at least one order, since the average of 31 daily averages is by definition the monthly average and any weighting would only introduce bias.

B. It weights a Sunday with 40 orders the same as a Monday with 900, so it answers "what was the average day's average?" rather than the average across all orders.

C. It is truncated by integer division, because `avg()` over an integer column returns an integer.

D. It double-counts, because each day's average already includes the previous day's orders in a running total.

26 · 9 min

Percentiles: describing a distribution instead of its mean

THE ONE THING TO KEEP

A percentile answers "what value does a given share of rows fall below?", and unlike sums, percentiles from two groups cannot be combined, so the raw rows or a sketch must be kept.

One number cannot describe a shape

A pile of 100,000 page-load times has a mean of 410 ms. That is true and nearly useless. It could mean every page loads in about 400 ms, or that 95% load in 200 ms and 5% take eight seconds. Those are different products, and the people who hit the eight seconds are the ones who leave.

A percentile answers a question the mean cannot: "what is the value that a given share of rows fall below?" The median is the 50th percentile, the point half the rows sit under. The 95th percentile, written p95, is the value that 95% of rows are under and 5% are over.

Getting one out of SQL

Postgres uses a syntax that reads oddly the first time:

```
SELECT
  percentile_cont(0.5)  WITHIN GROUP (ORDER BY load_ms) AS p50,
  percentile_cont(0.95) WITHIN GROUP (ORDER BY load_ms) AS p95,
  percentile_cont(0.99) WITHIN GROUP (ORDER BY load_ms) AS p99,
  avg(load_ms)                                          AS mean
FROM page_loads
WHERE loaded_at >= '2026-03-01' AND loaded_at < '2026-04-01';
```

`percentile_cont` interpolates between the two nearest rows, so it can return a value no row actually has. `percentile_disc` returns a real row's value. For latency and money the difference is cosmetic; for something like "the median number of children per household" `percentile_disc` avoids reporting 1.5 children.

DuckDB has `quantile_cont(load_ms, 0.95)` and a plain `median(load_ms)`. MySQL and SQLite have no percentile function; you sort, number the rows with a window function, and pick the row at position `ceil(0.95 * n)`, which the next module covers. Spreadsheets have `PERCENTILE` and `MEDIAN`, and LibreOffice Calc's are the same as Excel's.

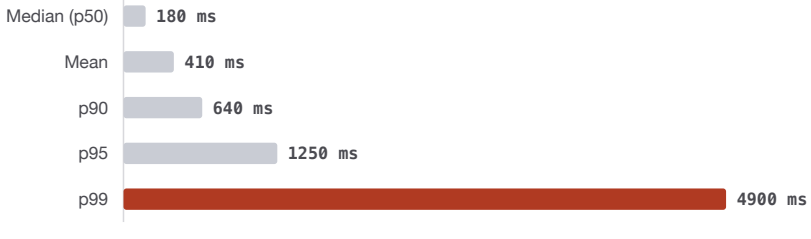
Why p99 is the number that matters most

If 1% of requests are slow, that sounds negligible. At a million requests a day it is ten thousand slow experiences, and because a page makes many requests, a user's chance of hitting at least one p99 event is far higher than 1%. Twenty requests per page at a 1% tail gives roughly an 18% chance the page contains a slow one. This is why engineering teams track p99 rather than the mean: the mean is dominated by the fast majority and hides the tail entirely.

Seeing the whole shape

Percentiles are samples of the distribution. A histogram is the distribution. `width_bucket` sorts values into equal-width bins:

One month of page loads: the mean, and the shape it hides



The mean sits between the median and the 90th percentile and describes neither. The p99 is twenty-seven times the median, and at a million requests a day that tail is ten thousand slow experiences. Report p50, p95 and p99 beside the mean and the count.

```
SELECT width_bucket(amount, 0, 5000, 10) AS bucket,
       min(amount), max(amount), count(*)
FROM orders
GROUP BY bucket
ORDER BY bucket;
```

Ten bins of ₹500 between 0 and 5,000, with bucket 11 catching everything above. Read the counts down the column and you can see whether there is one hump, two humps, or a wall at a price point. Two humps usually means two populations in one table, such as retail and wholesale customers, and that is a finding on its own.

The limitation that is a mechanism, not a policy

You can add two sums. You cannot combine two medians. If server A has a p95 of 300 ms and server B has a p95 of 900 ms, the p95 across both is not 600 ms and cannot be computed from the two figures at all. It depends on how many requests each server handled and where every request fell. An exact percentile needs the raw values, sorted, which is why the query above has to read and sort all 100,000 rows, and why a summary table holding `p95` per hour cannot give you p95 per day.

Large engines make a trade here. BigQuery's `APPROX_QUANTILES`, Spark's `approx_percentile` and DuckDB's `approx_quantile` use sketches that hold a compressed picture of the distribution in a few kilobytes, can be merged across servers and hours, and are wrong by a small stated amount, typically under 1% of the rank. For a dashboard that is the right trade. For a service-level agreement with money attached, run the exact version and accept the sort.

Reporting them

Put p50, p95 and p99 in the same row as the mean and the count. A mean far above the median says "long tail". A p99 ten times p50 says "a small group is having a very different experience". The count says how many rows the tail is, which is the number somebody will ask for next.

BEFORE YOU MOVE ON

Two web servers each report their own p95 response time for the day: 300 ms and 900 ms. A manager asks for the p95 across both servers combined. What can you say from these two figures alone?

A. It is 600 ms, the mean of the two, since each server contributed one day's worth of requests.
B. It is 900 ms, because the combined 95th percentile must equal the worse of the two.
C. Nothing exact; it lies somewhere between 300 and 900 ms and needs the raw values or a mergeable sketch to compute.
D. It is 300 ms weighted by request counts, because p95 is the value 95% of requests fall under and most requests are fast.

27 · 10 min

Grouping by day, week and month without lying about the boundaries

THE ONE THING TO KEEP

Group time with half-open ranges, a named time zone and a stated week start, because BETWEEN drops the last day, engines disagree on Monday versus Sunday, and a month is a different length from the last one.

Most time-series bugs are boundary bugs

"Revenue by month" sounds like the simplest report in the business. Every part of it hides a decision: where a day starts, which day a week starts on, whether February counts for less, and whether 31 March made it into March.

Truncating a timestamp to a period

Grouping by month means turning every timestamp into the first moment of its month, then piling:

```
SELECT date_trunc('month', placed_at) AS month,
       count(*)                       AS orders,
       sum(amount)                    AS revenue
FROM orders
WHERE placed_at >= '2026-01-01' AND placed_at < '2027-01-01'
GROUP BY 1
ORDER BY 1;
```

`date_trunc` works in Postgres and DuckDB and takes `'day'`, `'week'`, `'month'`, `'quarter'`, `'year'`. SQLite has no such function; use `strftime('%Y-%m', placed_at)`, which produces a text key like `2026-03`. MySQL has `DATE_FORMAT(placed_at, '%Y-%m')`. The text-key versions sort correctly only because the format puts year before month; `'%m/%Y'` would put March 2026 after February 2027.

Notice the WHERE clause. It uses a half-open range: greater than or equal to the start, strictly less than the day after the end. `BETWEEN '2026-01-01' AND '2026-12-31'` looks equivalent and is not, because on a timestamp column `'2026-12-31'` means midnight at the start of that day, so every order placed during 31 December is excluded. Half-open ranges are the habit that prevents this, and they also work for any period length without needing the last day's date.

Which day a week starts on

Postgres and DuckDB start `date_trunc('week', ...)` on Monday, following the ISO standard. Google Sheets and Excel's `WEEKNUM` start on Sunday by default. MySQL's `WEEK()` has eight modes and the default is Sunday-start. Two people computing "weekly revenue" from the same table can disagree on every single week, and both be internally consistent.

ISO weeks also cross year boundaries. 1 January 2027 is a Friday, so ISO week 1 of 2027 starts on Monday 4 January, and 1, 2 and 3 January 2027 belong to

week 53 of 2026. A report grouped by `extract(week ...)` without the ISO year beside it will fold those three days into the wrong year's week 1. Group by `date_trunc('week', ...)` and show the week's start date instead of a week number, and the ambiguity disappears.

Months are not the same length

February 2026 has 28 days. March has 31. A business with flat daily sales shows a 10.7% jump from February to March that means nothing. Either report revenue per day for each month, or compare each month to the same month a year earlier, when the length matches. Fridays matter too: a month with five Fridays outsells a month with four for a bar or a cinema, and that difference is calendar, not performance.

Four decisions hidden inside "revenue by month"

Written the quick way

- BETWEEN '2026-03-01' AND '2026-03-31' — everything after midnight on the 31st is gone
- extract(week FROM placed_at) — three days of January 2027 land in the wrong year's week 1
- date_trunc('day', placed_at) on a timestamptz — the boundary is whoever's midnight the server keeps
- March is 10.7% above February, on flat daily sales

Written so the boundaries are stated

- >= '2026-03-01' AND < '2026-04-01' — half-open, and the last day counts
- date_trunc('week', ...), showing the week's start date rather than its number
- AT TIME ZONE 'Asia/Kolkata' before truncating, and the zone named in the title
- Revenue per day, or March against last March, so the lengths match

Every time-grouped query should answer four questions from its text alone: which zone, which day the week starts on, whether the range is half-open, and what happens to a period with no rows. If it cannot, somebody reading the chart will guess, and they will guess differently from you.

Whose midnight

A `timestamptz` column stores an instant. Which day that instant belongs to depends on a time zone. An order at 01:30 on 1 April in Mumbai was placed at 20:00 UTC on 31 March. Group by UTC and it lands in March; group by local time and it lands in April. Neither is wrong, but only one matches what the shop manager saw.

```
SELECT date_trunc('day', placed_at AT TIME ZONE 'Asia/Kolkata')
AS local_day,
       sum(amount)
FROM orders
GROUP BY 1;
```

Decide the zone once, write it into the query, and say it in the report title. A business operating in one country uses that country's zone. A business across several usually reports in UTC and accepts that "yesterday" means something slightly odd to everyone, because the alternative is a report whose days overlap.

The month with no rows

A month in which a branch sold nothing produces no pile and therefore no row, which the earlier lesson on generating a spine solves. Time grouping is the most common place that lesson is needed, because a chart with a missing month draws a line straight across the gap and hides that anything was wrong.

The habit

Every time-grouped query should be able to answer four questions from its text alone: which zone, which day the week starts, whether the range is half-open, and what happens to a period with no rows. If the query cannot answer one of them, somebody reading the chart will guess, and they will guess differently from you.

BEFORE YOU MOVE ON

A query filters `WHERE placed_at BETWEEN '2026-03-01' AND '2026-03-31'` on a timestamp column and reports March revenue. Finance says the figure is about 3% low. What is the most likely cause?

A. BETWEEN excludes both endpoints, so orders on 1 March and 31 March were dropped from the total.

B. ``'2026-03-31'`` on a timestamp column means midnight at the start of that day, so every order placed during 31 March is excluded.

C. The timestamps are stored in UTC, and BETWEEN cannot apply a time zone, so about a day of orders at the end of the month has shifted into April.

D. The planner used an index on `placed_at` and index scans skip rows that arrived after the index was built.

28 · 9 min

Rates: a numerator, a denominator, and the same grain for both

THE ONE THING TO KEEP

Compute a rate's numerator and denominator in one query from one filtered pile, sum each before dividing, guard the division with NULLIF, and show the denominator beside the result.

A rate is two counts pretending to be one number

Conversion rate, refund rate, pass rate, open rate: each is a numerator divided by a denominator, and every way a rate goes wrong is a way one of the two halves was computed differently from the other. This lesson is about the mechanics; the earlier lesson on right versus useful numbers made the point that the denominator is where the meaning lives.

The division that stops the query

Postgres refuses to divide by zero and aborts the whole query with `ERROR: division by zero`. MySQL, SQLite and DuckDB return NULL for the same expression and carry on. So a rate query that runs on your laptop's DuckDB can fail in production Postgres on the first day a branch has no visits. Guard the denominator explicitly:

```
SELECT branch,
       count(*) FILTER (WHERE converted)
AS conversions,
       count(*)
AS visits,
       count(*) FILTER (WHERE converted)::numeric
         / nullif(count(*), 0)
AS rate
FROM visits
GROUP BY branch;
```

`nullif(x, 0)` returns NULL when `x` is zero, so the division yields NULL instead of an error. NULL is the honest result: a branch with no visits does not have a conversion rate of zero, it has no conversion rate. Note the `::numeric` cast, which the lesson on averages explained; without it two integer counts divide to `0`.

The average of rates is not the rate

You have a daily table with a `rate` column. Last month's rate is not `avg(rate)`. Sunday had 12 visits and 6 conversions, a rate of 50%. Monday had 4,000 visits and 120 conversions, a rate of 3%. The average of the two rates is 26.5%. The actual rate across both days is 126 out of 4,012, which is 3.1%. Days with almost no traffic swing an average of rates violently and barely move the true rate.

The rule is the same as for averages: keep the numerator and denominator as separate columns, sum each, and divide the sums.

```
SELECT sum(conversions)::numeric / nullif(sum(visits), 0) AS
pooled_rate
FROM daily_stats
WHERE day >= '2026-03-01' AND day < '2026-04-01';
```

Both halves must be drawn from the same pile

The most common rate bug is a numerator from one query and a denominator from another, each with its own filters. Conversions counted in local time and visits in UTC. Conversions including test accounts and visits excluding them. Orders placed in March over visitors seen in March, where a visitor arriving on 31 March and buying on 1 April is in the denominator but not the numerator. Each mismatch moves the rate by a few percent, which is exactly the size of change somebody will present as a result.

The defence is to compute both halves in one query from one filtered set, as the first example does. When the two halves genuinely come from different tables, join them at the same grain first and check that each side has the same row count you expect, then divide.

The trick with booleans

When the numerator is "rows where something is true", the rate is the average of a 0/1 column:

```
SELECT avg(CASE WHEN converted THEN 1 ELSE 0 END) AS rate FROM
visits;
-- DuckDB and Postgres also accept avg(converted::int)
```

It is compact, it is portable, and it fails safely on an empty group by returning NULL.

Percentage change has its own trap

"Up 40% on last month" is `(this - last) / last`. When `last` is zero the expression divides by zero; when `last` is negative, as a net figure can be, the

sign flips and "up" becomes "down". Guard with `nullif` and, for figures that can be negative, report the absolute change alongside the percentage or instead of it.

Small denominators

One out of two is 50%. It is also meaningless. Any rate reported without its denominator invites the reader to trust a coin flip as much as a census. Put `n` in the next column, always, and consider hiding rates where `n` is under some floor, say 30, or marking them. A grouped report in which the highest and lowest rates both belong to tiny groups is the normal case, not the exception, because small groups have the most room to land at the extremes by chance.

The habit

Write the numerator and the denominator as two columns before writing the division. Look at both. Then divide, guarded, with the count beside the result. A rate that arrives with its two parts can be argued with; a rate that arrives alone cannot.

BEFORE YOU MOVE ON

A weekly report computes conversion rate as `avg(daily_rate)` from a table with one row per day. Traffic is heavy on weekdays and very light at weekends. Compared with the true weekly rate, what does the report show?

A. The same value, since every day contributes equally to a week.

B. A value that is too low, because weekend days with few visits post a rate of zero whenever they have no conversions, and two such days drag the seven-day mean down.

C. A value pulled towards the weekend rates, which barely affect the true rate; the fix is to sum conversions and visits separately and divide.

D. A value that is too high, because `avg()` skips NULL days and the weekend rows are NULL.

29 · 7 min

Listing the members of a pile: string_agg and array_agg

THE ONE THING TO KEEP

string_agg and array_agg keep a pile's members instead of counting them; order inside the aggregate, watch MySQL's silent 1,024-character cut, and never store the joined list back into a column.

"How many" is not always the question

`count(*)` tells you a customer placed 4 orders. The person asking often wants to know which four. GROUP BY seems to forbid that, since the pile has to collapse into one row and there are four order numbers. The way out is an aggregate that keeps the values instead of summarising them.

string_agg: one row, one readable list

```
SELECT customer_id,
       count(*)                                      AS orders,
       string_agg(order_ref, ', ' ORDER BY placed_at) AS or-
der_list
FROM orders
GROUP BY customer_id;
```

The output is one row per customer with a column like `A-1041, A-1187, A-1302, A-1350`. The `ORDER BY` inside the aggregate is what makes the list predictable; without it the order is whatever the engine happened to do, and it changes between runs. `string_agg(DISTINCT city, ', ')` collapses repeats, which is what you want for "which cities has this customer ordered from".

The same function exists nearly everywhere under a different name. Postgres and DuckDB say `string_agg`; SQLite and MySQL say `group_concat`; BigQuery says `STRING_AGG`, and its `STRING_AGG(order_ref, ', ' ORDER BY placed_at)` matches Postgres almost character for character. The syntax for ordering inside the aggregate varies, and SQLite's `group_concat` cannot order at all unless you feed it from an ordered subquery.

The silent truncation in MySQL

MySQL's `group_concat` stops at 1,024 characters by default and does not raise an error. It simply returns the first 1,024 characters of the list. A customer with 200 orders gets a list that ends mid-reference, and nothing in the result says so. Raise the limit for the session with `SET SESSION group_concat_max_len = 1000000;` before any query that could produce a long list, and treat any exported list whose length is exactly 1,024 as suspect.

array_agg: keeping the values as values

A comma-joined string is for people to read. When the list is for a program, or you will need to filter or count it later, keep it as an array:

```
SELECT customer_id,
       array_agg(product_id ORDER BY placed_at) AS products
FROM order_items
GROUP BY customer_id;
```

Postgres returns a real array, `{88,12,88,301}`, with functions to match: `cardinality(products)` for its length, `products[1]` for the first, `88 = ANY(products)` to test membership. DuckDB calls this `list()` and returns a list, with `len(l)`, `l[1]`, `list_contains(l, 88)` and `list_position` as the matching functions. SQLite has no array type at all; `json_group_array` builds a JSON array you can read back with `json_each`, which is the nearest thing it offers. `unnest(products)` turns the array back into rows, so you can aggregate, join to it, and re-aggregate without a temporary table.

```
-- customers who bought product 88 and later bought 301
SELECT customer_id
FROM (SELECT customer_id, array_agg(product_id ORDER BY
placed_at) AS p
      FROM order_items GROUP BY customer_id) t
WHERE array_position(p, 88) < array_position(p, 301);
```

That is a "did A happen before B" question answered without a self-join, which the earlier lesson showed can be awkward.

Where this stops being a good idea

An aggregated list in a query result is fine. An aggregated list stored in a table column is the mistake the lesson on keeping each fact in one place warned about. `tags = 'red, sale, new'` cannot be indexed usefully, cannot be joined, cannot be counted per tag without splitting the string, and will one day contain `'red,sale'` without the space. Aggregate for display and export; store one value per row.

There is also a size question. `string_agg` over a million-row group builds a string of tens of megabytes in memory. If the list would be too long for a human to read, the human does not want the list; they want a count, a sample, or the first ten. `array_agg(x ORDER BY placed_at)[1:10]` in Postgres, or `list(x)[1:10]` in DuckDB, gives the first ten without building the rest.

The habit

Reach for `string_agg` when a person will read the result and `array_agg` when a program will. Always order inside the aggregate. In MySQL, raise the length limit first. And if you find yourself storing the result, stop and add a table instead.

BEFORE YOU MOVE ON

A MySQL report lists each customer's order references with `group_concat(order_ref)`. For the biggest customers the list appears to end part-way through a reference, and no error is shown. What is happening?

A. `group_concat` cannot handle groups with more than a few hundred rows and stops aggregating once its internal buffer of rows is full, keeping what it has.

B. The result is being cut at `group_concat_max_len`, 1,024 characters by default, and MySQL truncates rather than raising an error.

C. The order references contain commas, so the separator is splitting references into pieces that look truncated.

D. Some references are NULL, and `group_concat` stops at the first NULL it meets in the group.

30 · 9 min

The row behind the MAX: which order was the biggest

THE ONE THING TO KEEP

MAX returns a value with no memory of its row; to get the row, join back to the summary, use DISTINCT ON or arg_max, and compare output rows against group count to catch ties and all-NULL groups.

MAX gives you a value, not a row

"What was each customer's largest order?" is easy: `max(amount)` grouped by customer. "Which order was it, and when?" is the question people actually ask, and the obvious query does not work:

```
SELECT customer_id, max(amount), order_id, placed_at
FROM orders
GROUP BY customer_id;
-- ERROR: column "orders.order_id" must appear in the GROUP BY
clause
```

The database is right to refuse. `max(amount)` squeezes the pile to one number; `order_id` has four values in that pile and no rule says which one to show. MySQL in a permissive mode will pick one at random and make you think it worked, which is the worst outcome. You need a way to say "the row where the amount is the max".

Join back to the summary

The portable approach in two steps: compute the maxima, then join them back to find the rows that hit them.

```
WITH peak AS (
  SELECT customer_id, max(amount) AS max_amount
  FROM orders GROUP BY customer_id
)
SELECT o.customer_id, o.order_id, o.placed_at, o.amount
FROM orders o
JOIN peak p ON p.customer_id = o.customer_id AND p.max_amount =
o.amount;
```

It works everywhere, including SQLite and older MySQL. It has one honest flaw: ties. A customer with two orders of exactly ₹5,000 gets two rows, and a report that expected one row per customer now has more rows than customers. Sometimes that is the truth you want. When it is not, you need a tiebreaker, and a join cannot express "the earliest of the tied ones" without another round of the same trick.

Pick one row per group directly

Postgres has `DISTINCT ON`, which is the clearest answer to "one row per group, chosen by this ordering":

```
SELECT DISTINCT ON (customer_id)
       customer_id, order_id, placed_at, amount
FROM orders
ORDER BY customer_id, amount DESC, placed_at ASC;
```

Read it as: sort within each customer by amount descending, then by date, and keep the first row of each customer. The tiebreaker is just the next sort key. It is fast, it returns exactly one row per group, and it is a Postgres extension that no other engine spells the same way.

DuckDB and ClickHouse have `arg_max(order_id, amount)`, which returns the `order_id` from the row where `amount` is largest, as an ordinary aggregate you can put next to `count(*)`. BigQuery gets there with `ARRAY_AGG(order_id ORDER BY amount DESC LIMIT 1)[OFFSET(0)]`. Each is the same idea: an aggregate that remembers which row won.

The correlated form

The version that reads most like the question is a subquery that fetches the top row per customer:

```
SELECT c.id,
       (SELECT order_id FROM orders o
        WHERE o.customer_id = c.id
        ORDER BY amount DESC, placed_at LIMIT 1) AS
biggest_order
FROM customers c;
```

It returns one column, so a second column means a second subquery, and it runs once per customer unless the planner is clever. Postgres's `LATERAL` join lets the subquery return several columns at once and is the tool when you need the whole row, but the mechanism is the same.

Which to use

When the tie is meaningful and you want to see both rows, join back to the summary. When you want exactly one row and have a tiebreaker, use

`DISTINCT ON` in Postgres, `arg_max` in DuckDB, or the array trick in BigQuery. When you need the top three per customer rather than the top one, none of these is comfortable, and the next module's `row_number()` is the general tool that makes the top-N question a two-line filter.

Whichever form you pick, put a comment above the query naming the tiebreaker. "Latest wins" and "earliest wins" are both defensible, and the next reader cannot tell from the SQL alone which one you chose or whether you chose at all.

The check that catches the wrong one

Whatever form you choose, count the output rows and compare them with the number of groups. More rows than groups means ties leaked through. Fewer means some groups had only NULL amounts, which `max()` ignores, so they had no maximum and no row. Both are findings; neither should be a surprise you discover in a meeting.

BEFORE YOU MOVE ON

Using the join-back pattern (a CTE of `max(amount)` per customer joined to `orders` on customer and amount), a report of "each customer's biggest order" returns 1,046 rows for 1,000 customers. What explains the extra rows?

A. The join condition is missing the customer, so amounts are matching across customers.

B. `max()` returned NULL for customers with no orders in the period, and NULL matched every row of `orders` in the join, adding rows for them.

C. Forty-six customers have two or more orders that tie at their maximum amount, and the join returns every tied row.

D. The CTE was materialised and read twice, once for each side of the join, duplicating some rows.

31 · 8 min

Subtotals and grand totals in one query: ROLLUP and GROUPING

THE ONE THING TO KEEP

ROLLUP stacks several grains in one result, so use GROUPING() to tell a subtotal's NULL from a real one and never sum across the output, which counts each row once per level.

Three grains in one report

A sales report wants revenue by city, a subtotal for each region, and a grand total at the bottom. That is three queries with three different GROUP BY clauses, and three scans of the table. Or it is one:

```
SELECT region, city, sum(amount) AS revenue
FROM sales
GROUP BY ROLLUP (region, city)
ORDER BY region, city;
```

`ROLLUP (region, city)` produces the piles for `(region, city)`, then for `(region)`, then for `()`, which is everything. The output stacks them. The subtotal row for a region has `city` set to NULL, and the grand total row has both columns NULL.

`CUBE (region, channel)` goes further and produces every combination: by region and channel, by region alone, by channel alone, and the total. `GROUPING SETS ((region, city), (channel), ())` lets you name exactly which piles you want and skip the rest, which is what you reach for when CUBE would produce thirty subtotals nobody reads.

Postgres, DuckDB, SQL Server, Oracle and BigQuery support all three. MySQL has `GROUP BY region, city WITH ROLLUP` and nothing else. SQLite

has none of them, and the fallback is `UNION ALL` of the separate grouped queries, which is more typing and the same result.

The NULL that is not a NULL

The subtotal rows use NULL to mean "all cities". That collides head-on with a real NULL: a sale whose `city` was never recorded also produces a row with `city` NULL, at the detail grain. In the output the two rows look identical, and a report that labels every NULL city as "Total" has just relabelled its missing data as a subtotal.

`GROUPING()` tells them apart. It returns 1 when the column was rolled up in that row, and 0 when the row is at the detail grain and the NULL is genuine:

```
SELECT
  CASE WHEN GROUPING(region) = 1 THEN 'All regions' ELSE region
END AS region,
  CASE WHEN GROUPING(city)   = 1 THEN 'All cities'  ELSE city
END AS city,
  sum(amount) AS revenue
FROM sales
GROUP BY ROLLUP (region, city);
```

Now a real unknown city stays NULL and a subtotal is labelled as one. In a report with any nullable grouping column, `GROUPING()` is not optional.

What the output is not

The result of a ROLLUP is not a table at one grain. It is three tables stacked, and any arithmetic across the whole result is wrong. `SELECT sum(revenue) FROM (...rollup...)` counts every sale three times: once in its city, once in its region, once in the total. Anybody who exports the result to a spreadsheet and drags a SUM over the column makes this mistake by reflex.

So a ROLLUP result is for reading, not for further computation. If a downstream step needs the numbers, give it the detail grain and let it compute its own subtotals, or give it the subtotal rows only, filtered with `WHERE GROUPING(city) = 1`.

Ordering the rows

The natural output order puts subtotals wherever the engine's hash table happened to leave them. To get the layout people expect, city rows followed by their region's subtotal and then the grand total last, sort on the grouping flags:

```
ORDER BY GROUPING(region), region, GROUPING(city), city
```

Detail rows have flag 0 and sort first within each region; the subtotal has flag 1 and follows; the grand total has both flags set and comes last.

When to use it

ROLLUP earns its place in reports that will be read as a document: a printed monthly summary, a PDF for a board, an exported sheet. It costs one scan instead of three and keeps the subtotals consistent with the detail by construction, because they are computed from the same filtered rows in the same statement. It does not belong in a query that feeds a chart or another query, where a single grain is what the consumer expects and a stacked one will be silently misread.

GROUPING SETS is the version to remember for the common request of "totals by region and, separately, totals by channel": it produces exactly those two breakdowns from one scan without the full region-by-channel grid that CUBE would add in between.

BEFORE YOU MOVE ON

A ROLLUP report by region and city is exported to a spreadsheet, and a colleague sums the revenue column to get a company total. The figure is roughly three times the true total. Why?

A. ROLLUP adds a row per city per region, so cities that appear in more than one region are counted again for each.
B. The output stacks the city grain, the region subtotals and the grand total, so each sale is counted once at every level.
C. The NULL city rows for unrecorded cities were treated as subtotals and included twice.
D. Spreadsheets treat NULL as zero and the exported subtotals lost their grouping flags, so the sum silently included blank rows as well as real ones.

32 · 9 min

Wide and long: pivoting rows to columns and back

THE ONE THING TO KEEP

Store and compute in long form and pivot only for display, because a query's columns are fixed at plan time and cannot be discovered from the data, and a wide table needs a schema change for every new period.

Two shapes for the same facts

A revenue-by-city-by-month result can be long, with one row per city per month and three columns, or wide, with one row per city and a column for each month. Long is how a database stores and groups facts. Wide is how a human reads them across a page. Most reporting is the act of turning one into the other, and most of the friction comes from asking SQL to do the human-shaped half.

Long to wide, when you know the columns

The tool is conditional aggregation, which the CASE lesson introduced: one aggregate per output column, each filtered to its slice.

```
SELECT city,
       sum(amount) FILTER (WHERE month = '2026-01-01') AS jan,
       sum(amount) FILTER (WHERE month = '2026-02-01') AS feb,
       sum(amount) FILTER (WHERE month = '2026-03-01') AS mar
FROM monthly_sales
GROUP BY city;
```

Or `sum(CASE WHEN month = '2026-01-01' THEN amount END)` where FILTER is unsupported. A city with no sales in February gets NULL in that column, and `coalesce(..., 0)` decides whether the report shows a blank or a zero, which is a decision about meaning, not formatting.

Why the columns must be typed by hand

You will want to write "one column per month, whatever months exist". SQL cannot do it, and the reason is structural rather than a missing feature. A query's output columns are part of its type, fixed when the query is planned, before any row is read. The engine has to know there will be a column called `apr` before it can decide how to produce one. Data-driven columns would mean the shape of the result depends on the result, which the planner cannot resolve.

The escape hatches confirm the rule. Postgres's `crosstab()` in the `tablefunc` extension still needs the output column list declared in the call. DuckDB's `PIVOT monthly_sales ON month USING sum(amount)` genuinely discovers the months, and does it by running a first query to list them and then building the second query from that list; BigQuery's `PIVOT` requires the values in the statement. Where a report must pivot on values that change, the usual answer is to pivot in the reporting tool, or to generate the SQL from a query of the distinct values, which is what DuckDB does for you.

Wide to long

The reverse arrives whenever a spreadsheet lands in the database: `city, jan, feb, mar` as four columns. Grouping by month is impossible in that shape. Unpivot it:

```
SELECT city, month, amount
FROM wide_sales
CROSS JOIN LATERAL (VALUES ('2026-01-01', jan), ('2026-02-01',
feb), ('2026-03-01', mar))
  AS v(month, amount);
```

Each source row becomes three rows, one per month. The portable version is a `UNION ALL` of three `SELECT city, '2026-01-01', jan FROM wide_sales` queries. DuckDB has `UNPIVOT wide_sales ON jan, feb, mar INTO NAME month VALUE amount`, which reads as what it does.

Why wide is the wrong storage shape

Store the wide table and every new month is an `ALTER TABLE` to add a column, every query that groups by time has to be rewritten, and the table is mostly NULL for any city that started trading recently. Store long, and a new month is just more rows. The spreadsheet habit of one column per period is exactly the habit the first lesson of this course described as a spreadsheet stopping working, and the fix is the same: long in the table, wide only at the moment of display.

The free path

A pivot table in LibreOffice Calc or Google Sheets performs this operation with a drag, and for a one-off report on a few thousand rows it is the right tool. Export the long result as CSV, open it, and pivot there. Keep the SQL that produced the long result, because the pivot is display and the query is the record of what the numbers mean.

The habit

Write the query long, check the numbers long, and pivot last, in the query only when the column list is genuinely fixed, otherwise in the tool that shows the result. A wide result whose columns you had to type is a report; a wide table somebody stores is a problem waiting for its thirteenth month.

BEFORE YOU MOVE ON

A monthly pivot query uses one `sum(amount) FILTER (WHERE month = ...)` per month. April's sales are in the table, but the report has no April column. What is the direct cause?

A. FILTER ignores months that were absent when the table was created, so April needs a schema refresh before the expression can see it.

B. The query names its output columns by hand, and nobody added an April expression, so the April rows are read and then discarded.

C. April's rows are still NULL-typed because they arrived after the query was last planned.

D. GROUP BY city collapsed April into the March column because both fall in the same quarter.

33 · 8 min

Reconciling: making a total tie out before you send it

THE ONE THING TO KEEP

Before a total leaves your hands, make the grouped parts sum to the ungrouped whole, keep a row-count ledger across each join, tie the figure to an external number with the difference explained, and check one row by hand.

A total nobody has checked is a rumour

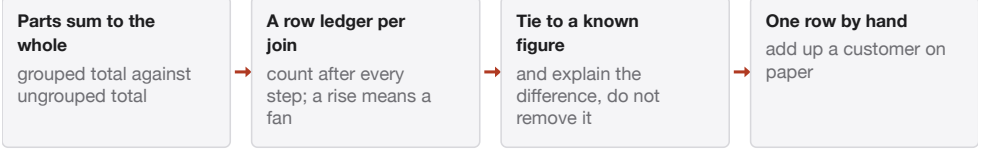
Every number in this module can be produced by a query that runs without error and returns something plausible. The only way to know it is right is to compare it with something else that ought to agree with it. Accountants call this reconciliation, and they have done it for five centuries because it works.

There are four comparisons, and a query that has passed all four is one you can put your name under.

One: the parts must sum to the whole

Run the grouped query and the ungrouped one:

Four comparisons before a number leaves your hands



Each of these is a query, not a ritual, and each returns nothing when all is well. They catch the mechanical errors so that errors of meaning are the only ones left to argue about — which is a much better argument to be having.

```
SELECT sum(amount) FROM orders WHERE placed_at >= '2026-03-01'
AND placed_at < '2026-04-01';
-- 4,182,300

SELECT city, sum(amount) FROM orders
WHERE placed_at >= '2026-03-01' AND placed_at < '2026-04-01'
GROUP BY city;
-- 14 rows totalling 4,161,900
```

The grouped rows add up to 20,400 less than the whole. That gap is orders with a NULL city. GROUP BY puts them in their own pile, so they should appear as a row with a NULL key, unless a filter such as `WHERE city IS NOT NULL` or an inner join to a cities table removed them. Either way the report now silently excludes ₹20,400, and the reader will assume the fourteen rows are everything.

A one-line version of the check, which should return zero:

```
SELECT (SELECT sum(amount) FROM orders WHERE ...) -
       (SELECT sum(city_total) FROM (SELECT sum(amount) AS
city_total FROM orders WHERE ... GROUP BY city) t);
```

Two: the row ledger across joins

Write down the count at each step: 31,940 orders in the table; 31,940 after the LEFT JOIN to customers; 31,940 after the LEFT JOIN to the pre-aggregated items summary. The moment a count rises, a join has fanned; the moment it falls, an inner join or a WHERE on the right-hand side has dropped rows. The lesson on joins made this the check for a single join. Across a real query with four joins, the ledger is the only way to see which one did it.

Three: tie to an external figure

Somewhere there is a number produced by a different system: the finance ledger, the payment provider's dashboard, last month's report that went to the board. Compare. The figures will not match exactly, and the useful work is explaining the difference, not making it go away. Refunds recorded on a differ-

ent date. A currency conversion applied at a different rate. Timing: a payment that settled on 1 April for an order placed on 31 March. A reconciliation that ends with "the ₹38,200 difference is 41 refunds processed after month end" is a result. One that ends with "close enough" is not.

Set a tolerance in advance. For a monthly revenue figure a tenth of a percent is typical; anything larger needs a named cause.

Four: one row by hand

Pick a customer. Look up their orders in the source system. Add them up on paper. Compare with the report's row. This catches the class of error that no total can: a join that pairs the right sum with the wrong name, a time zone that moved one order into the wrong month, a status filter that excluded a legitimate order. It takes three minutes and it is the check most often skipped.

Making it a query, not a ritual

Each of these can be a query that returns rows only when something is wrong:

```
-- rows present in the total but absent from the breakdown
SELECT count(*), sum(amount) FROM orders
WHERE placed_at >= '2026-03-01' AND placed_at < '2026-04-01'
  AND city IS NULL;
```

Put these in a file next to the report query. Run them together. A report that arrives with its reconciliation queries and their zero-row results has done something a bare number cannot: it has shown its work.

What reconciliation cannot do

It cannot tell you the definition was right. If "revenue" should have excluded cancelled orders and the query included them, every check above passes and the number is still wrong, because the external figure was computed the same wrong way or was never computed at all. That failure belongs to the question, and the final module of this course is about it. Reconciliation catches the me-

chanical errors so that the meaning errors are the only ones left to argue about.

BEFORE YOU MOVE ON

A grouped revenue-by-city report sums to ₹20,400 less than the ungrouped monthly total from the same table and date filter. What is the most likely explanation?

A. GROUP BY discards rows whose grouping column is NULL, so orders with no recorded city are dropped by the grouping itself before any sum runs.

B. `sum()` rounds each city's subtotal to the nearest hundred and the rounding losses accumulate.

C. Orders with a NULL city were excluded from the grouped query by a filter or an inner join, and they carry the missing ₹20,400.

D. The grouped query ran a moment later and new orders were inserted in between, so the totals cover different data.

CASE STUDY · MODULE 3

The waiting time everybody was happy with

A city bus operator in Coimbatore with 340 buses, preparing the quarterly service review for the transport committee

The operator's contract with the city contains one performance number: average waiting time at a stop, which must stay under nine minutes. Every quarter, Karthik's team reports it. Every quarter it comes in between seven and eight minutes, and every quarter the committee's public consultation produces a stack of complaints about waiting half an hour at particular stops on particular evenings.

The number is computed from `arrivals`, one row per bus per stop per arrival, with a timestamp. Waiting time is derived as the gap since the previous bus at that stop. There are 4.1 million rows a quarter.

This quarter Karthik is asked by a new committee member for the median as well, and for the 90th percentile. He runs them. The mean is 7.8 minutes. The median is 5.2. The 90th percentile is 19 minutes. The 99th is 54.

Then he looks at how the mean has been computed for the last three years, and finds two things.

The first is that the quarterly figure has always been the average of the daily figures held in a `daily_stats` table, which stores one row per stop per day with an `avg_wait` column. Sunday has a tenth of the traffic of a Tuesday and much longer gaps, so its long waits sit in the average with the same weight as a weekday's short ones — but the far bigger effect runs the other way, because a stop with four buses a day and one with two hundred contribute equally to an average of averages. Recomputing from the raw gaps, weighted properly, gives a mean of 11.4 minutes, not 7.8.

The second is the denominator. The gap is only recorded when two consecutive arrivals are logged at the same stop. When a bus is cancelled, no arrival is logged, so the long wait it caused is not a row in the table at all. The rows that exist are the rows where a bus turned up. The measure is a measure of waiting time among people whose bus came.

Karthik has two weeks and a decision that is not really technical.

He can report the number the way it has always been reported, with the median and the percentiles added as supplementary information, and note the weighting question in an annexe. That keeps the contract compliant, keeps three years of published figures internally consistent, and lets the committee see the tail for the first time. It also means the headline number in the minutes will be 7.8 minutes, and the headline number is what the newspaper prints.

Or he can report 11.4 minutes as the mean, with the method change explained. That breaches the contractual ceiling for the first time in three years, triggers a review clause, and — because the previous figures cannot be recomputed for stops where the arrival data has been archived — produces a series with a step in it that no one can smooth. The operator's managing director has already asked him, not unreasonably, whether a number that changes because the analyst changed his mind is a number at all.

The cancellation problem breaks both options equally, and fixing it needs the scheduled timetable joined against the arrivals — a spine of expected buses, LEFT JOINed to what actually arrived — which is three days of work he does not have this quarter.

> **STOP HERE**
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Karthik reported both. The headline stayed 7.8 minutes, labelled "as previously computed: unweighted mean of daily stop averages", with 11.4 beside it labelled "weighted mean over all recorded gaps" and the p50, p90 and p99 under both. The annexe said in one paragraph that neither figure counts a wait caused by a cancelled bus, because a cancelled bus logs no arrival, and estimated from the timetable that this excluded roughly 4% of the quarter's ex-

pected services. The committee spent almost none of its meeting on the mean and most of it on the p90 of 19 minutes, broken down by stop, which identified eleven stops accounting for a third of all waits over twenty minutes. The review clause was not triggered, because the committee accepted the old definition as the contractual one and asked for the new one to be adopted from the following year with a renegotiated ceiling of thirteen minutes. The spine against the timetable was built the following quarter and moved the p90 to 24 minutes. Karthik's note to himself afterwards was that the three years of compliant reports had never been wrong, exactly; they had answered a question about the average day at the average stop, and nobody rides the average stop.

Why is the average of the daily `avg_wait` values not the average waiting time, and what should `daily_stats` have stored instead?

Because an average of averages weights each daily row equally regardless of how many gaps it summarises, so a stop-day with four buses counts as much as one with two hundred. Averages do not combine; sums and counts do. The summary table should store the components — the total waiting minutes and the number of gaps per stop per day — so any period's mean is `sum(total_minutes) / sum(gaps)`. This is the same rule that makes it impossible to recover a quarterly percentile from daily percentiles.

The measure only sees stops where two consecutive arrivals were logged. Name the failure and explain why adding a filter or a date range cannot repair it.

It is survivorship: the population is defined by having survived the thing being measured. The long waits caused by cancellations are absent from the table before any filter runs, so no WHERE clause can bring them back. The only repair is to supply the rows that should exist — a spine of scheduled services from the timetable, LEFT JOINed to the arrivals — so that a missing bus becomes a row with a known gap rather than no row at all.

The committee spent its meeting on the p90 rather than on either mean. What does that suggest about which statistics to put in front of a decision-maker?

That a single central figure describes a distribution nobody experiences, and that the tail is where the decisions live. Reporting p50, p90 and p99 beside the mean, with the count, turns "is the service acceptable" into "which stops, on which

evenings, produce the waits people complain about" — a question with an operational answer. The gap between the mean and the median is itself the finding, and it is more useful than either number alone.

MODULE 3 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 In a table where one row is one order, a city's pile returns `COUNT(*)` of 900 and `COUNT(DISTINCT customer_id)` of 340. What is true?

A. The table has duplicate rows, since a customer should appear once

B. There were 900 orders, placed by 340 different customers

C. 560 of the orders are missing a customer id and were skipped

D. COUNT(DISTINCT ...) is the correct figure and COUNT(*) has overcounted

2 `status` holds 'active', 'churned', or NULL for people who never set it. `SELECT COUNT(*) FROM users WHERE status <> 'churned'` comes back smaller than expected. Why?

A. COUNT(*) skips rows whose status column is NULL

B. The NULL rows are counted as churned, because unknown defaults to false

C. <> cannot be used on a text column that permits NULL

D. NULL <> 'churned' is unknown, and WHERE keeps only rows that are true

3 Branch A took 100 orders averaging 500 rupees; branch B took 10 orders averaging 5,000. Averaging the two branch averages gives 2,750, and the true average is 909. What is the general rule?

A. Sums and counts combine across groups; averages do not

B. The mean is unreliable when one group is more than ten times another

C. AVG must be given a weight argument when groups differ in size

D. The two branches should be reported separately, never combined

4 Server A has a p95 of 300 ms and server B a p95 of 900 ms. What is the p95 across both?

A. 600 ms, the mean of the two, if both handled the same number of requests

B. 900 ms, because a percentile across groups takes the larger value

C. It cannot be computed from the two figures at all

D. Between 300 and 900 ms, and closer to whichever server was busier

5 A `daily_stats` table has one row per day with a `rate` column. Sunday: 12 visits, 6 conversions. Monday: 4,000 visits, 120 conversions. What is last week's conversion rate?

A. 26.5%, the average of the daily rates, which is what the column is for

B. 3.1%, from the summed conversions over the summed visits

C. 50%, since Sunday's rate is the highest observed in the period

D. It depends on whether the rate column was rounded when it was stored

6 A report uses `GROUP BY ROLLUP (region, city)`, and some sales have no recorded city. Why is `GROUPING()` not optional here?

A. It is required whenever ROLLUP is combined with an ORDER BY clause

B. Without it, the subtotal rows are computed from the detail rows twice

C. It converts the subtotal rows to a single grain the report can sum

D. A subtotal's NULL and a genuinely missing city look identical

MODULE 4

Windows, sequences and time

GROUP BY collapses a pile into one row. A window function looks at the pile and leaves the rows where they are, which is what every "compared with the previous one", "rank within the group", "running total" and "seven-day average" question needs. This block builds the window from its three parts, partition, order and frame, works through the questions it answers, and finishes with the two time problems that defeat plain joins: what was true as of a date, and how a cohort of customers behaves month by month.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Answer any "compared with what came before" or "which one within each group" question — rank in group, top N per group, running total, change from the previous row, moving average, streaks, sessions and cohort retention — with a window function whose frame you can state, and join a fact to the value that was true at its own moment in time

34 · 9 min

A window: GROUP BY that keeps the rows

THE ONE THING TO KEEP

A window function computes an aggregate over a row's partition and writes it beside the row instead of collapsing the rows, and because it runs after WHERE you filter on its result in a second layer.

The question GROUP BY cannot answer

"Show every order, and next to it, the share of its city's revenue that it represents." GROUP BY can give you the city totals. It cannot give you the orders, because it has collapsed them into the totals. You could compute the totals in a CTE and join them back, which works and which you have done in earlier lessons. A window function does the same thing in one expression, and once you can read one, most of the awkward SQL you meet turns out to be somebody working around not knowing it.

```
SELECT order_id, city, amount,
       sum(amount) OVER (PARTITION BY city)             AS
city_total,
       amount / sum(amount) OVER (PARTITION BY city)    AS
share_of_city
FROM orders;
```

Every row stays. Each one gains a column that looks at the other rows in its city, adds them up, and writes the answer beside it. `OVER` is the word that turns an aggregate into a window function. `PARTITION BY city` says which rows count as "the others": the window a row looks through is its own partition.

`OVER ()` with nothing inside means the window is the whole result. `amount / sum(amount) OVER ()` is each order's share of everything.

Three parts, in order

A window has up to three parts, and the later lessons in this module are each about one of them.

- **PARTITION BY** decides which rows a row can see. Like GROUP BY, without the collapsing.
- **ORDER BY**, inside the OVER, puts the visible rows in sequence, which is what "previous row", "rank" and "running total" need. Without it there is no previous row.
- **The frame** narrows the sequence further, to "the six rows before this one" or "everything up to here". It only exists once there is an ORDER BY, and it has a default that surprises people, which gets its own lesson.

When it runs, and why that matters

A window function is computed after `WHERE`, `GROUP BY` and `HAVING` have finished and before `ORDER BY` and `LIMIT`. That order explains the first error everybody hits:

```
SELECT *, row_number() OVER (ORDER BY amount DESC) AS rn
FROM orders
WHERE rn <= 10;
-- ERROR: column "rn" does not exist
```

`WHERE` runs before the window is computed, so the column is not there yet. The fix is to compute in one layer and filter in the next:

```
WITH ranked AS (
  SELECT *, row_number() OVER (ORDER BY amount DESC) AS rn FROM
orders
)
SELECT * FROM ranked WHERE rn <= 10;
```

The same order explains a quieter fact: a window function sees the rows after `WHERE` has applied. `sum(amount) OVER ()` in a query with `WHERE city = 'Pune'` is Pune's total, not the company's. If you need the company total on

Pune's rows, the filter has to move into a later layer, or the total has to come from a subquery.

Several windows at once

You can mix windows and plain columns freely, and use different partitions in the same SELECT:

The three parts of a window, and what each one decides

PARTITION BY	which rows this row is allowed to see
ORDER BY, inside the OVER	puts those rows in a sequence
The frame	how much of that sequence the aggregate covers
Computed after WHERE, GROUP BY and HAVING	so filter its result in a second layer
Computed before ORDER BY and LIMIT	the window sees the rows WHERE already kept

PARTITION BY is GROUP BY without the collapsing. The ORDER BY inside OVER is what makes "previous row" and "running total" mean anything. The frame is the part with a default that surprises people, and it exists only once there is an ORDER BY.

```
SELECT order_id, city, amount,
       sum(amount) OVER (PARTITION BY city)  AS city_total,
       sum(amount) OVER ()                    AS grand_total,
       count(*)    OVER (PARTITION BY customer_id) AS cus-
tomer_orders
FROM orders;
```

When one window is reused many times, name it once with a `WINDOW` clause: `WINDOW w AS (PARTITION BY city ORDER BY placed_at)` at the end of the query, then `sum(amount) OVER w`. It keeps a ten-column query readable.

Where it works, and what it costs

Window functions are standard SQL and present in Postgres, DuckDB, BigQuery, SQL Server, SQLite from version 3.25 and MySQL from version 8. MySQL 5.7 does not have them, which is why older tutorials do the self-join contortions the earlier lesson showed. In spreadsheets the equivalent is a

helper column with a formula that references other rows, which works until the sheet is sorted.

The cost is a sort. To compute a window with `PARTITION BY city ORDER BY placed_at`, the engine sorts the rows by city and date, then walks them. On a million rows that is fast; on a billion it is the thing to look at in the plan. Two windows with different orderings mean two sorts. Where you can, give every window in a query the same partition and order, so the engine sorts once.

The habit

When you catch yourself writing a CTE that aggregates and a join that brings the aggregate back to the rows, stop and ask whether a window does it in one line. Usually it does, it is easier to read, and it is one fewer place for the grain to go wrong.

BEFORE YOU MOVE ON

A query computes `count(*) OVER () AS total_rows` and filters `WHERE city = 'Pune'`. The analyst expected `total_rows` to be the size of the whole table but gets Pune's row count on every row. Why?

A. The window function runs after WHERE, so its partition contains only the rows that survived the filter.

B. `OVER ()` with an empty partition defaults to partitioning by the first column in the WHERE clause, which here is city.

C. `count(*)` inside a window counts distinct values of the ordering column rather than rows, and Pune has that many distinct dates.

D. The database rewrote the window into a GROUP BY on city because the WHERE clause referenced that column, collapsing the count to the group.

35 · 8 min

ROW_NUMBER, RANK and DENSE_RANK: three answers to "what position"

THE ONE THING TO KEEP

row_number breaks every tie arbitrarily unless you give it a tiebreaker, rank shares positions and skips, dense_rank shares and does not skip, and the row count of a filtered ranking tells you which one you actually needed.

Four students, two of them tied

Scores: Asha 90, Bilal 85, Chen 85, Dev 80. Ask for their positions and there are three defensible answers, and SQL gives you all three as separate functions.

```
SELECT name, score,
       row_number() OVER (ORDER BY score DESC) AS rn,
       rank()       OVER (ORDER BY score DESC) AS rnk,
       dense_rank() OVER (ORDER BY score DESC) AS drnk
FROM results;
```

- `row_number()` gives 1, 2, 3, 4. Every row gets a distinct number. Bilal and Chen are tied, and one of them gets 2 and the other 3 by an order the query did not specify.
- `rank()` gives 1, 2, 2, 4. Ties share a position, and the next position skips ahead by the number of tied rows, because two people came second and so nobody came third.
- `dense_rank()` gives 1, 2, 2, 3. Ties share a position and nothing is skipped.

None of these is more correct. A sports league uses `rank()`: two teams level on points are joint second, and the next is fourth. A "top three products" list that should show every product that made the top three uses `dense_rank()`.

A pagination key or a deduplication step uses `row_number()`, because it needs every row to have a different number.

row_number needs a complete order

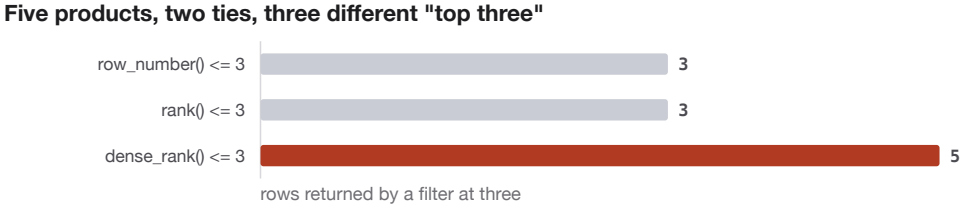
Because `row_number()` must break every tie, it will break them arbitrarily if you let it. Two runs of the same query can give Bilal 2 then 3. In a report that is a cosmetic wobble; in a query that keeps the row where `rn = 1` and deletes the rest, it decides which row survives. Always add a tiebreaker that cannot tie, usually the primary key:

```
row_number() OVER (PARTITION BY customer_id ORDER BY placed_at
DESC, order_id DESC)
```

Now "the customer's latest order" is the same row every time, and when two orders share a timestamp the higher id wins, which is a rule you can explain.

Where NULLs rank

Sorting descending puts NULLs first in Postgres, because NULL is treated as larger than every value, and a student with no score would be ranked top of the class. Say what you mean: `ORDER BY score DESC NULLS LAST`. MySQL and SQLite treat NULL as smallest, so the same query ranks the missing score last, which is a difference that shows up only when the data has gaps.



Sales are 900, 700, 700, 500 and 500. row_number numbers them 1 to 5 and breaks the ties on nothing in particular. rank gives 1, 2, 2, 4, 4. dense_rank gives 1, 2, 2, 3, 3, so a filter at three lets both 500s through. Count the rows a ranking picked; that count tells you which function you actually needed.

Cutting into equal groups

`ntile(4) OVER (ORDER BY spend DESC)` assigns each customer to a quartile: the top quarter of spenders get 1, the next quarter 2, and so on. When the rows do not divide evenly the earlier groups get the extra: ten rows into four tiles gives groups of 3, 3, 2, 2. It is the tool for "split customers into deciles by revenue" and it is also how a lot of segmentation gets done badly, because `ntile` produces groups of equal *count*, not equal *value*. The top decile by count may hold 60% of revenue. If you want groups of equal revenue, you need a running total and a cut at each tenth of the sum, which the running-totals lesson makes straightforward.

Position as a fraction

`percent_rank()` returns 0 for the first row and 1 for the last, with each row placed by how many rows fall before it. `cume_dist()` returns the share of rows with a value at or below this one. A student at `cume_dist` 0.9 scored as well as or better than 90% of the class. These are how "you are in the top 5%" is computed, and they inherit the tie behaviour of `rank()`, so tied students get the same fraction.

Rank within a group

Add a partition and the ranking restarts in each group:

```
rank() OVER (PARTITION BY class_id ORDER BY score DESC)
```

Now every class has its own first place. This is the pattern under every "best in category" question, and the next lesson turns it into a filter.

The check that catches the wrong function

If your ranking is used to pick rows, count the rows it picked. `rank() <= 3` can return five rows when there are ties; `row_number() <= 3` returns exactly three and may have excluded a tied row on the basis of nothing. Neither is

wrong. One of them is what was asked for, and the count is how you find out which you built.

BEFORE YOU MOVE ON

Five products have monthly sales of 500, 400, 400, 300 and 300. A report filters on `rank() OVER (ORDER BY sales DESC) <= 3` to show the top three. How many rows does it return?

A. Exactly three, because rank stops at position 3 and the tied 300s are ranked 4 and 5.

B. Five, because rank never skips positions, so the 300s are ranked 3 and the filter includes everything.

C. Two, because rank assigns tied rows the same position and then discards all but one of them.

D. Three: the 500 at rank 1 and both 400s at rank 2, with the 300s at rank 4 and excluded.

---

36 · 8 min

Top N per group in two lines

THE ONE THING TO KEEP

Top N per group is a ranking window in a CTE and a filter on the rank outside it; the function you rank with is a statement about ties, and the ORDER BY inside the window must end in a column that cannot tie.

The question that used to be hard

"The three best-selling products in each category." Before window functions this took a correlated subquery that counted how many products in the same category outsold this one, and it ran once per row. With a window it is a rank and a filter:

```
WITH ranked AS (
  SELECT category, product_id, sales,
         row_number() OVER (PARTITION BY category
                            ORDER BY sales DESC, product_id) AS
rn
  FROM product_sales
)
SELECT category, product_id, sales
FROM ranked
WHERE rn <= 3
ORDER BY category, rn;
```

Read the window aloud: within each category, order by sales descending and by product id to break ties, and number the rows from 1. Then keep the first three of each. The previous lesson explained why the filter has to live in a second layer, and why `product_id` is in the ORDER BY.

Which N, and which ties

`row_number` returns exactly three per category, and when two products tie for third, one is dropped by the tiebreaker. That is right for "show me three". It is wrong for "show me everything that made the top three", where `dense_rank() <= 3` returns both tied products, and possibly four or five rows for that category. `rank() <= 3` is the third option: it returns the tied rows but, if two products tie for first, it returns only ranks 1, 1 and 3. Decide which the reader wants before choosing the function, and say which you chose in the column name or a comment.

The one-row case

When N is 1 the previous module's `DISTINCT ON`, `arg_max` and join-back patterns all compete with this one. The window version is the most portable and the only one that extends to N of 2 without rewriting, so it is the default; `DISTINCT ON` is worth keeping for Postgres-only code where it reads more clearly.

Percent of the group, not a fixed count

Sometimes the ask is "the top 10% of customers in each region". `ntile(10)` gets close but produces equal-count buckets, which is what "top 10% by count" means. For "customers who together make up the top 10% of revenue" you want a running share:

```
WITH s AS (
  SELECT region, customer_id, revenue,
         sum(revenue) OVER (PARTITION BY region ORDER BY rev-
enue DESC)
           / sum(revenue) OVER (PARTITION BY region) AS
cum_share
  FROM customer_revenue
)
SELECT * FROM s WHERE cum_share <= 0.10;
```

Two windows with the same partition, one ordered and one not, and the division gives each customer's position in the cumulative curve. The engine sorts once for both because the partition matches.

Performance, briefly

The window has to sort each category's rows by sales. For a product table that is nothing. For "the latest three events per device" over a billion-row events table it is the whole cost, and there are two things that help. An index on `(device_id, event_time DESC)` lets the engine read rows already in window order and skip the sort. And Postgres from version 15 recognises a `row_number() <= N` filter on the outer query and stops scanning each partition after N rows, so the query no longer numbers a million events per device to keep three. On older versions, and on engines without that optimisation, a `LATERAL` join that fetches `LIMIT 3` per device from the index is faster, at the cost of reading less naturally.

The trap in the tiebreaker

If the ORDER BY inside the window is not complete, the rows that make it into the top N can change between runs, and a daily report will show a differ-

ent "third-best product" on days when nothing changed. The fix is the same as before: end the window's ORDER BY with a column that cannot tie. If the report's audience will notice a product appearing and disappearing, this is not a nicety.

The habit

Every top-N query should be able to say, in one sentence, what happens to ties, and that sentence should match the function used. "Three per category, ties broken by product id" is `row_number`. "Everything in the top three positions" is `dense_rank`. If you cannot say the sentence, the report is going to surprise somebody.

BEFORE YOU MOVE ON

A daily report uses `row_number() OVER (PARTITION BY category ORDER BY sales DESC) <= 3` and, on a day when no sales changed, shows a different third product in one category than it did the day before. What is the most likely cause?

A. The window recomputed ranks from a different starting row because the table was vacuumed overnight and its physical order changed the ranking.

B. row_number is non-deterministic by design and cannot be used for reports; rank should be used instead.

C. Two products tie on sales in that category and the window's ORDER BY has no tiebreaker, so which one gets number 3 is arbitrary.

D. The CTE was materialised on one day and inlined on the next, and the two execution paths order partitions differently.

37 · 9 min

Running totals, and the frame you did not know you set

THE ONE THING TO KEEP

An ORDER BY inside a window sets a frame whose default, RANGE up to the current row, includes tied rows; write ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW explicitly when you mean one row at a time.

Adding ORDER BY to a window changes what it sums

`sum(amount) OVER (PARTITION BY customer_id)` gives each row its customer's total. Add an ORDER BY inside the window and the same function becomes a running total:

```
SELECT customer_id, placed_at, amount,
       sum(amount) OVER (PARTITION BY customer_id ORDER BY
placed_at) AS running_total
FROM orders;
```

Each row now shows the sum of its customer's orders up to and including itself. The ORDER BY did not just sort; it introduced a **frame**, a rule about which of the ordered rows the aggregate covers. With no ORDER BY the frame is the whole partition. With an ORDER BY, the default frame is "from the start of the partition up to the current row", which is what makes it a running total.

The default is RANGE, and RANGE includes ties

The default frame is written out as `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, and the word RANGE hides a surprise. In RANGE mode, "current row" means "the current row and every row that ties with it in the

ORDER BY". Take four orders on days 1, 2, 2 and 3 with amounts 10, 20, 30 and 40:

- `RANGE` (the default): running totals 10, 60, 60, 100. Both day-2 rows see each other, so both show 60.
- `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`: 10, 30, 60, 100. Each row sees only the rows physically before it.

Which is right depends on the question. "Cumulative revenue by the end of each day" wants RANGE, because both rows on day 2 belong to a day whose total is 60. "A running balance after each transaction" wants ROWS, because a statement lists one line at a time. Most people want ROWS, get RANGE, and only notice on a day with two orders, when the running total appears to jump.

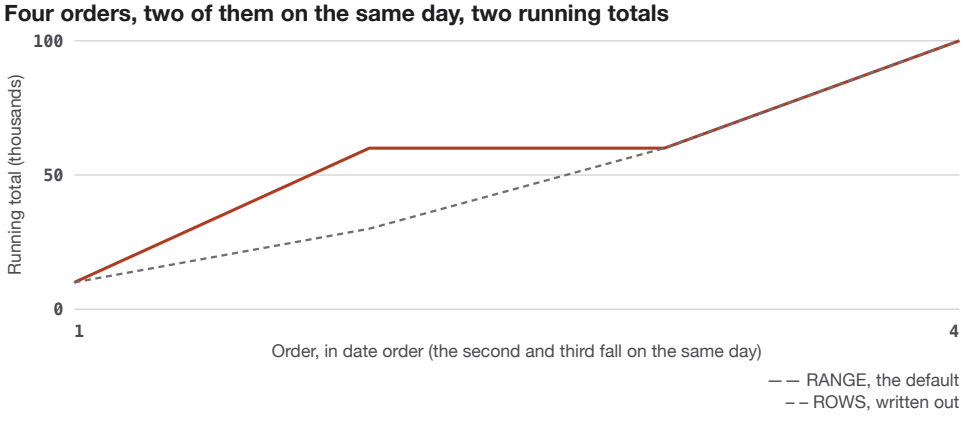
Say the frame explicitly whenever the ORDER BY can tie. If you order by a unique key there are no ties and the two agree, which is another reason to end every window ORDER BY with the primary key.

Running totals that reset

The partition is the reset. `PARTITION BY customer_id` restarts at zero per customer; `PARTITION BY customer_id, date_trunc('year', placed_at)` restarts per customer per year, which is how "year-to-date spend" is built.

Cumulative share, and the 80/20 that is usually 80/15

Divide a running total by the partition total and you have a cumulative share, which turns a sorted list into a curve:



The amounts are 10, 20, 30 and 40 thousand, on days 1, 2, 2 and 3. In the default RANGE frame the two day-2 rows tie in the ORDER BY, so each sees the other and both read 60. A ROWS frame counts one row at a time. Both are right for some question; only one is right for yours.

```
WITH c AS (
  SELECT customer_id, revenue,
         sum(revenue) OVER (ORDER BY revenue DESC, customer_id)
AS running,
         sum(revenue) OVER ()
AS total,
         row_number() OVER (ORDER BY revenue DESC, customer_id)
AS rn,
         count(*)     OVER ()
AS n
  FROM customer_revenue
)
SELECT rn::numeric / n AS share_of_customers, running / total
AS share_of_revenue
FROM c;
```

Find the row where `share_of_revenue` first exceeds 0.8 and read `share_of_customers` from it. Marketing folklore says 20%; on real retail data the answer is often between 10% and 30%, and knowing your own number is worth more than the folklore. This is also the way to cut customers into groups of equal revenue rather than equal count.

Running counts and running distinct counts

`count(*) OVER (ORDER BY placed_at)` is a running count and behaves exactly like the running sum. `count(DISTINCT customer_id) OVER (ORDER BY placed_at)` is not allowed in Postgres and most engines: distinct aggregates do not work as windows, because a running distinct count cannot be computed by adding one row's contribution to the previous total. The workaround is to number each customer's first order with `row_number()` partitioned by customer, keep the rows where it is 1, and take a running count of those. That is the "customers acquired to date" line on every growth chart.

The habit

Whenever a window has an ORDER BY, write the frame out. `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` is eleven words that remove an entire category of bug, and a reader who does not know the default will thank you. The next two lessons use frames that look backwards a fixed distance rather than to the start, and the same rule applies there.

BEFORE YOU MOVE ON

A running balance is computed with `sum(amount) OVER (ORDER BY txn_date)`. On days with two transactions, both rows show the same balance, which already includes both amounts. What is happening?

A. The two rows are duplicates produced by a fanning join upstream, so each carries the other's amount as well as its own.

B. `sum()` as a window sums the whole partition when the ORDER BY column has repeated values, falling back to the no-ORDER-BY behaviour for those rows only.

C. The transactions were inserted in the wrong order, so the second row's amount is being counted before the first.

D. The default RANGE frame treats rows tied on `txn_date` as one position, so each sees both amounts; a ROWS frame gives one line at a time.

38 · 9 min

LAG and LEAD: the previous row, and the change since it

THE ONE THING TO KEEP

LAG and LEAD read a fixed number of rows back or forward in the window's order, not a fixed distance in time, and last_value returns the current row unless the frame is extended to UNBOUNDED FOLLOWING.

Reading the row before this one

Month-over-month growth, days since the last order, whether a price went up or down: every one of these needs a row to see its predecessor. `lag()` does exactly that.

```
SELECT month, revenue,
       lag(revenue) OVER (ORDER BY month)              AS pre-
v_revenue,
       revenue - lag(revenue) OVER (ORDER BY month)    AS
change,
       (revenue - lag(revenue) OVER (ORDER BY month))
         / nullif(lag(revenue) OVER (ORDER BY month), 0) AS pc-
t_change
FROM monthly_revenue;
```

`lag(revenue)` returns the value of `revenue` from the row one position earlier in the window's order. The first row has no predecessor and gets NULL, which is why the percentage change is guarded with `nullif`. `lag(revenue, 12)` reaches twelve rows back, which is the same month last year if, and only if, every month is present. `lag(revenue, 1, 0)` supplies a default instead of NULL. `lead()` is the mirror image and looks forward.

Per-entity, with a partition

Add a partition and each entity gets its own sequence:

```
SELECT customer_id, placed_at,
       placed_at - lag(placed_at) OVER (PARTITION BY cus-
tomer_id ORDER BY placed_at, order_id)
         AS days_since_last_order
FROM orders;
```

A customer's first order gets NULL, every later one gets the gap. `avg(days_since_last_order)` over that result is the average reorder interval, and `lead()` in the same shape gives "days until the next order", which is what you compute when the question is whether a customer has gone quiet compared with their own habit.

The gap is in rows, not in time

`lag()` looks back one *row*. If a customer has months with no orders, the previous row is not the previous month; it is whenever they last ordered. For "revenue in the previous calendar month" on a table with gaps, either join to a spine first so that every month exists as a row, or look the value up by date with a self-join on `month - interval '1 month'`. The window over a spine is cleaner and is the usual answer. The mistake to avoid is computing month-over-month change with `lag()` on a table that quietly skips a month, which reports a two-month change as if it were one.

first_value, last_value, and a trap

`first_value(price) OVER (PARTITION BY product_id ORDER BY changed_at)` gives every row the product's original price. `last_value()` should, by symmetry, give the latest, and it does not:

```
last_value(price) OVER (PARTITION BY product_id ORDER BY
changed_at)
-- returns each row's OWN price
```

The reason is the default frame from the previous lesson. It runs from the start of the partition to the current row, so the "last" value in the frame is always the current one. To see the genuine last value the frame has to extend to the end:

```
last_value(price) OVER (PARTITION BY product_id ORDER BY
changed_at
                        ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING)
```

This is the most searched-for window function bug there is, and it is the frame every time. `nth_value(price, 2)` has the same behaviour and the same fix.

Direction and streaks

Comparing a row with its predecessor is how you classify movement: `CASE WHEN price > lag(price) OVER w THEN 'up' WHEN price < lag(price) OVER w THEN 'down' ELSE 'flat' END`. Count the 'up' rows and you have how often the price rose. Flag the rows where the direction changed and take a running sum of the flags, and you have numbered the runs, which is the next lesson's method for streaks and sessions.

What lag cannot do

It looks back a fixed number of rows in one ordering. "The previous order from the same branch" when the window is partitioned by customer needs a different partition and therefore a second window, and "the previous order over ₹1,000" needs a filter before the window, because `lag()` cannot skip rows conditionally. A `lag()` on a filtered CTE handles the second case; the first is two windows in one query, which the first lesson of this module showed is fine.

The habit

Every `lag()` answers two questions you should be able to state: ordered by what, and partitioned by what. And any `last_value()` without an explicit frame is a bug until proved otherwise.

BEFORE YOU MOVE ON

`last_value(status) OVER (PARTITION BY ticket_id ORDER BY changed_at)` is meant to show every row of a ticket's history alongside the ticket's final status, but each row shows its own status instead. What fixes it?

A. Replace `last_value` with `first_value` and reverse the ORDER BY to `changed_at DESC`, because last_value is only reliable on the first row of a partition.

B. Add `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`, because the default frame ends at the current row.

C. Remove the ORDER BY, because last_value only looks at the whole partition when the rows are unordered.

D. Use `max(changed_at)` in a GROUP BY and join back, since window functions cannot see rows after the current one.

39 · 8 min

Moving averages: seven rows is not seven days

THE ONE THING TO KEEP

A ROWS frame counts rows, so a seven-row average over a table with missing days is not a seven-day average; join to a spine or use a RANGE frame measured in time, and mark the first partial windows.

Smoothing a noisy line

Daily revenue jumps around. Weekends dip, a promotion spikes, a bank holiday empties the chart. A seven-day moving average replaces each day's figure with the mean of that day and the six before it, and the weekly rhythm disappears, leaving the trend.

```
SELECT day, revenue,
       avg(revenue) OVER (ORDER BY day
                          ROWS BETWEEN 6 PRECEDING AND CURRENT
ROW) AS ma7
FROM daily_revenue;
```

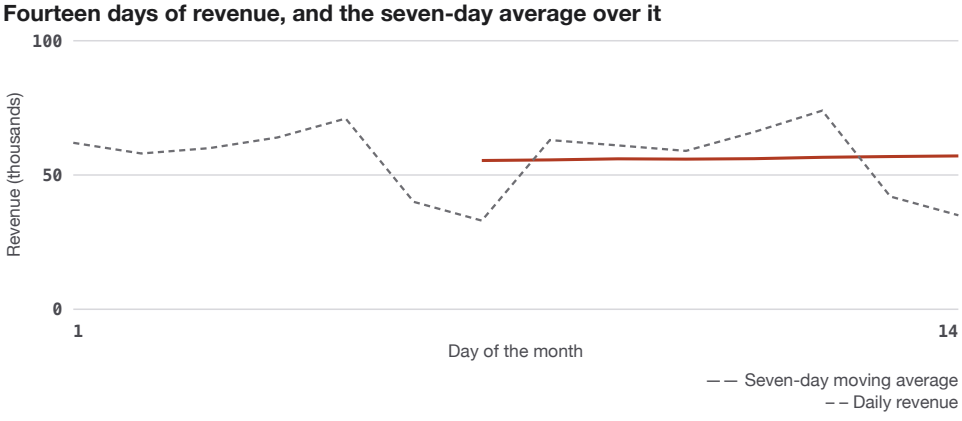
The frame is the whole idea: instead of running from the start of the partition, it runs from six rows back to here. `ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING` is a centred seven-day window, which has no lag against the trend but cannot be computed for the most recent three days. `ROWS BETWEEN 29 PRECEDING AND CURRENT ROW` is the 30-day version.

The first six rows

On the first day of the series the frame contains one row, on the second two, and so on. `avg()` divides by the rows it has, so the first six values are averages of shorter windows and are systematically noisier. Either drop them, mark them, or report the count alongside:

```
count(*) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND
CURRENT ROW) AS days_in_window
```

and filter `WHERE days_in_window = 7` in an outer layer. A chart that starts with a partial window shows a false step on day seven, when the average suddenly has its full complement of rows.



The weekend dips on days 6, 7, 13 and 14 disappear into the average, leaving a trend that barely moves. The smoothed line starts on day seven, because the first six frames hold fewer than seven rows; drop them, mark them, or report the row count beside them.

Seven rows is only seven days if every day is there

A `ROWS` frame counts rows. If the table has no row for a day with no revenue, "6 PRECEDING" reaches back further than six days, and the moving average is over whatever seven trading days happened to exist. The line looks smooth and means something different at each point. There are two fixes.

The first is the spine from the earlier module: generate every date, LEFT JOIN the revenue to it, and coalesce missing days to zero (if zero is what a missing day means) before the window. The second is a `RANGE` frame measured in time rather than rows, which Postgres supports from version 11 and DuckDB supports:

```
avg(revenue) OVER (ORDER BY day
                   RANGE BETWEEN interval '6 days' PRECEDING
AND CURRENT ROW)
```

This frame includes every row whose `day` is within six days of the current one, however many rows that is. It is honest about gaps, and it changes the meaning slightly: a missing day now contributes nothing rather than a zero, so the average is over the days that traded. Decide which meaning you want, because the two charts will differ exactly where the business was closed.

A moving average lags the truth

Averaging the last seven days means the line responds to a change about three and a half days after it happened. A trailing 30-day average responds two weeks late. That is the price of smoothing, and it is why a moving average is a bad early-warning signal and a good after-the-fact description. When you need to react quickly, plot the raw series beside the smoothed one, or use a shorter window and accept the noise.

Moving sums and moving distinct counts

The same frame works for `sum()`, and "revenue in the trailing 30 days" is one of the most useful columns in any daily table. `count(DISTINCT customer_id)` over a moving frame is not supported, for the reason the running-totals lesson gave: a sliding distinct count cannot be updated one row at a time. For "customers active in the trailing 30 days" the honest query is a self-join from each day to the orders within its window, or a daily table of first-and-last-active dates.

Weighted and exponential

A simple moving average weights each of the seven days equally. An exponentially weighted average gives recent days more weight and never fully forgets old ones, and it cannot be written as a single window frame, because each value depends on the previous smoothed value. A recursive CTE can do it and is unpleasant; the free path is to pull the daily series into Python and call `pandas.Series.ewm(span=7).mean()`, or to use LibreOffice Calc with a column formula. Know that the option exists, and reach for SQL's plain moving average first, because it is the one a reader can check by hand.

The habit

State three things for every moving average: the window length, whether it counts rows or days, and what happens on days with no row. If the chart's caption cannot say all three, the smoothing has hidden something rather than revealed it.

BEFORE YOU MOVE ON

A shop closes on Sundays and its `daily_sales` table has no row for those days. Its "7-day moving average" uses `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW`. What is each value actually an average of?

A. The last seven calendar days, with Sundays treated as zero, because the window's ORDER BY on `day` fills the missing date with a NULL row that averages as zero.

B. The last seven trading days, which span eight calendar days whenever a Sunday falls inside the window.

C. Six days, because `6 PRECEDING` excludes the current row and the Sunday gap removes one more.

D. The last seven calendar days, because the window measures the distance in the ORDER BY column's units when that column is a date.

40 · 9 min

Streaks and sessions: gaps and islands

THE ONE THING TO KEEP

Label each run of consecutive values with date minus row_number, or each session with a running sum of gap flags, then group by the label; the gap threshold is a definition that changes every downstream count.

Two shapes of the same problem

"How long is this user's current daily-login streak?" and "split this stream of clicks into visits" look unrelated. Both are the same operation: walk the rows in order, decide where one run ends and the next begins, and number the runs. SQL people call the runs *islands* and the breaks *gaps*, and there are two tricks, one for each shape.

Islands of consecutive values

A user logged in on 3, 4, 5, 9, 10 and 14 March. That is three streaks. Number the rows and subtract:

```
SELECT user_id, login_day,
       login_day - (row_number() OVER (PARTITION BY user_id
ORDER BY login_day))::int AS grp
FROM logins;
```

For the six days the row numbers are 1 to 6, and `login_day - row_number` gives 2 March, 2 March, 2 March, 5 March, 5 March, 8 March. Within a streak, the date and the row number advance together, so their difference stays constant. At a gap the date jumps and the row number does not, so the difference changes. That constant is an arbitrary label for the streak, and grouping by it finishes the job:

```
WITH g AS (... the query above ...)
SELECT user_id, min(login_day) AS streak_start, max(login_day)
AS streak_end,
       count(*) AS days
FROM g
GROUP BY user_id, grp
ORDER BY user_id, streak_start;
```

The longest streak per user is a `max(days)` over that; the current streak is the island whose `streak_end` is today. The trick works on any evenly spaced sequence: consecutive invoice numbers, consecutive weeks, consecutive months if you convert months to a running integer first. It needs each value to appear once per partition, so deduplicate a login table with several logins a day before applying it.

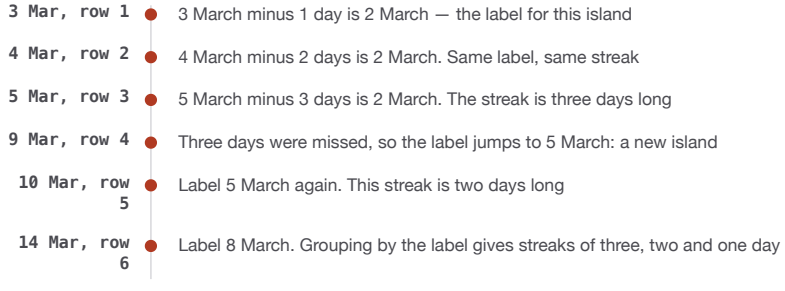
Sessions defined by a gap in time

Clicks are not evenly spaced, so the row-number trick does not apply. Instead, decide the gap that separates two sessions, flag each row that starts a new one, and take a running count of the flags:

```
WITH flagged AS (
  SELECT user_id, clicked_at,
         CASE WHEN clicked_at - lag(clicked_at) OVER (PARTITION
BY user_id ORDER BY clicked_at)
                   > interval '30 minutes'
              OR lag(clicked_at) OVER (PARTITION BY user_id
ORDER BY clicked_at) IS NULL
              THEN 1 ELSE 0 END AS new_session
  FROM clicks
),
numbered AS (
  SELECT *, sum(new_session) OVER (PARTITION BY user_id ORDER
BY clicked_at
                                   ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS session_no
  FROM flagged
)
SELECT user_id, session_no,
       min(clicked_at) AS started, max(clicked_at) AS ended,
       max(clicked_at) - min(clicked_at) AS duration,
       count(*) AS clicks
FROM numbered
GROUP BY user_id, session_no;
```

Three layers: compare each row with its predecessor and flag a break; running-sum the flags so that every row carries its session number; group by that number. The first row of each user has no predecessor and is flagged as a session start. Every layer uses something from earlier in this module, and this is the query where they all meet.

Six logins, three streaks, and the label that finds them



Within a streak the date and the row number advance together, so their difference holds still. At a gap the date jumps and the row number does not, so the difference changes. That constant is an arbitrary name for the streak, and grouping by it finishes the job.

The threshold is a definition, not a fact

Thirty minutes of inactivity is the convention web analytics tools inherited from the 1990s. It is not a property of the data. Change it to 10 minutes and a user who reads a long article becomes two sessions; change it to two hours and an afternoon of intermittent browsing becomes one. The number of sessions per user, the average session length and every "conversion per session" rate move with the threshold, sometimes by a factor of two. Write the threshold into the query as a named constant, put it in the report's caption, and be suspicious of any session count compared across two systems that may not share it.

Where the islands trick pays for itself

- Machine uptime from a status log: islands of `status = 'up'` give outage windows and their lengths.
- Membership history: islands of consecutive paid months give each tenure, and the gaps are churn-and-return events.
- Data quality: islands in a sequence that should be unbroken (invoice numbers, sensor readings every minute) surface the exact ranges that are missing.

Each of these is a `min`, `max` and `count` per island, once the island has a label.

The habit

Draw six rows on paper before writing either query. Mark where the runs break. Then decide whether the break is "the next value is not adjacent" (row-number trick) or "the next value is too far away" (lag-and-flag trick). The wrong choice produces a query that runs and returns something plausible, which is the recurring theme of this course.

BEFORE YOU MOVE ON

A user logged in on 3, 4, 5, 9 and 10 March. Using `login_day - row_number() OVER (ORDER BY login_day)` to label streaks, what labels do the five rows get, and why does the method work?

A. 3, 3, 3, 9, 9 March; the label is each streak's first date, because row_number restarts at 1 whenever the previous day is missing from the partition.

B. 1, 1, 1, 2, 2; row_number restarts whenever the date is not adjacent to the previous one, producing a streak counter directly.

C. 2, 2, 2, 4, 4 March; the three-day gap between 5 and 9 March advances the label by the number of missing days, so later streaks are labelled by their gap length.

D. 2, 2, 2, 5, 5 March; inside a streak the date and the row number rise together, so their difference only changes where a day is skipped.

41 · 10 min

Cohorts and retention: how a month of new customers behaves afterwards

THE ONE THING TO KEEP

A cohort is a group defined by a first event, retention is the share of it active at each age, and the grid is a triangle you read down a column only as far as every cohort has reached.

The question a total hides

Revenue went up. Was it because you are keeping customers, or because you are acquiring new ones fast enough to hide that the old ones leave? A total cannot say. A cohort analysis can: take everyone who started in the same month, and watch what share of them is still active one, two, three months later.

Step one: assign each customer to a cohort

A cohort is defined by a first event, and the first event is a `min()`:

```
WITH first_order AS (
  SELECT customer_id, date_trunc('month', min(placed_at)) AS
cohort_month
  FROM orders
  GROUP BY customer_id
)
```

One row per customer, with the month they first bought. This is a decision dressed as a query: "first order" could equally be "first sign-up" or "first paid order", and a customer who bought once in 2023 and returned in 2026 belongs to the 2023 cohort forever. Say which event defines the cohort in the report title.

Step two: each customer's active months, as an age

```
activity AS (
  SELECT o.customer_id,
         f.cohort_month,
         date_trunc('month', o.placed_at) AS active_month
  FROM orders o
  JOIN first_order f USING (customer_id)
  GROUP BY 1, 2, 3
),
aged AS (
  SELECT cohort_month,
         customer_id,
         (extract(year  FROM active_month) - extract(year  FROM
cohort_month)) * 12
       + (extract(month FROM active_month) - extract(month FROM
cohort_month)) AS months_since
  FROM activity
)
```

The `GROUP BY` in `activity` collapses several orders in one month to one row, because retention counts customers, not orders. `months_since` is the cus-

tomer's age in months at the time of that activity: 0 for the cohort month itself, 1 for the following month, and so on. The arithmetic with `extract` is the portable way; DuckDB has `date_diff('month', cohort_month, active_month)` and does it in one call.

Step three: the matrix

```
SELECT cohort_month,
       months_since,
       count(DISTINCT customer_id) AS active,
       count(DISTINCT customer_id)::numeric
         / first_value(count(DISTINCT customer_id))
             OVER (PARTITION BY cohort_month ORDER BY months_since) AS retention
FROM aged
GROUP BY cohort_month, months_since
ORDER BY cohort_month, months_since;
```

Two grains in one query, and worth reading slowly. The GROUP BY produces one row per cohort per age, with the number of customers active at that age. The window then runs over those grouped rows, and `first_value` ordered by age within each cohort picks the age-0 count, which is the cohort's size. Divide by it and the `retention` column reads "of the customers who started in January, 41% bought something in month 3".

Pivoting `months_since` into columns, with the earlier lesson's technique, gives the triangle-shaped grid that cohort reports are known for.

The triangle, and the comparison it forbids

The grid is a triangle because a cohort that started three months ago has ages 0 to 3 and nothing further. That shape carries the most common misreading: comparing month-6 retention across cohorts when only the older cohorts have reached month 6. The newer cohorts are not worse; they are absent. Read the grid down a column, comparing the same age across cohorts, and only as far as every cohort in the comparison has reached. A "retention is falling" headline is often a triangle read along the wrong edge.

What "retained" means here

This query counts a customer as retained in a month if they ordered in that month. A customer who orders in months 0, 1 and 3 is retained at age 3 and not at age 2, and the curve can rise from month 2 to month 3. That is correct for purchase behaviour and different from subscription retention, where "retained" means "had not cancelled by then" and the curve can only fall. Both are called retention; a chart that does not say which will be compared with one of the other kind.

The habit

A cohort report should state four things in its caption: the event that defines the cohort, the event that counts as active, whether the measure can rise, and the age beyond which cohorts are not comparable. Every one of them is a decision in the query above, and every one changes the shape of the triangle.

BEFORE YOU MOVE ON

A cohort grid shows month-6 retention of 38% for the January cohort and blank for the June cohort, and the report is dated end of September. A reader concludes the June cohort has "collapsed to zero by month 6". What is wrong?

A. The June cohort's month-6 figure was excluded by a `count(DISTINCT ...)` that returns NULL for empty groups, so it should be read as zero after all.

B. The retention window used RANGE instead of ROWS and merged June's ages, leaving month 6 with no row to display.

C. Retention measured by orders cannot fall to zero, so the blank must be a data-loading failure for June.

D. June's cohort is only three months old at the end of September, so its month-6 cell does not exist yet and no comparison at that age is possible.

42 · 9 min

Dates, intervals and the two kinds of timestamp

THE ONE THING TO KEEP

Store timestamptz and convert with a named zone at display time, because a plain timestamp discards where the clock was, and treat month arithmetic as a rule you chose, since engines disagree on what 31 January plus a month is.

Adding a month is not one operation

What is 31 January plus one month? Postgres says 28 February. Add another month and it says 28 March, not 31 March, because the intermediate result has already lost the 31. Month arithmetic is not associative: `'2026-01-31' + interval '2 months'` is 31 March, but `('2026-01-31' + interval '1 month') + interval '1 month'` is 28 March.

SQLite makes a different choice. `date('2026-01-31', '+1 month')` produces `2026-03-03`, because it adds one to the month number to get 31 February, then normalises the overflow forward into March. Neither engine is wrong; each has picked a rule, and a schedule that says "bill on the same day next month" has to pick one too, and write it down.

The safe patterns: work from the first of the month and add months to that, or anchor to "the last day of the month" explicitly with `date_trunc('month', d) + interval '1 month' - interval '1 day'`. Both survive being applied repeatedly.

Days between two dates

Subtracting two `date` values in Postgres gives an integer number of days. Subtracting two timestamps gives an `interval`, which prints as `3 days 04:12:00` and has to be converted to be summed or averaged: `extract(epoch FROM (b - a)) / 86400` gives fractional days. `age(b, a)` gives a human-

shaped interval in years, months and days, which is right for "how old is this account" on a screen and wrong for arithmetic, because a month has no fixed length. DuckDB's `date_diff('day', a, b)` and SQLite's `julianday(b) - julianday(a)` are the equivalents.

timestamp and timestamptz are different types

Postgres has both, and the names mislead. `timestamp` (without time zone) is a wall-clock reading with no information about where the clock was: `2026-03-29 02:30:00` and nothing more. `timestamptz` (with time zone) is an *instant*, stored internally as UTC and converted to the session's time zone on the way out. The second one is nearly always what you want, because an instant can be compared with any other instant, while a wall-clock reading from Mumbai and one from London cannot be ordered without knowing which is which.

The trap is that both accept the same input text, so a column created as `timestamp` will happily store values that were actually instants, quietly discarding the zone. A database that received `2026-03-29T02:30:00+05:30` into a `timestamp` column has kept `02:30` and thrown away the `+05:30`, and no query can get it back.

The rule that follows: store `timestamptz`, keep the server's session zone at UTC, and convert for display with `AT TIME ZONE 'Asia/Kolkata'`. Use the named zone, never a fixed offset like `+05:30`, because named zones carry the history of when a region changed its rules and an offset does not.

Daylight saving, for the readers who have it

India has no daylight saving, so a business there can go years without meeting this. A business with users in Europe or North America meets it twice a year. On 29 March 2026 the clocks in London go from 00:59:59 GMT to 02:00:00 BST, so the wall-clock time `01:30` does not exist that day, and on the last Sunday of October `01:30` happens twice. A `timestamp` column cannot represent which of the two it means; a `timestamptz` can, because it stores the instant. Daily totals grouped in local time will have one 23-hour day and one 25-

hour day a year, and a report that compares those days with their neighbours will show a dip and a bump that are calendar, not business.

The clock inside a transaction

`now()` in Postgres returns the time the current transaction started, and returns the same value for every call inside it, however long it runs. A batch that stamps ten thousand rows with `now()` over four minutes gives them all the same timestamp. `clock_timestamp()` reads the real clock each time. Most of the time the frozen value is what you want, because it makes a transaction's writes look simultaneous; for measuring how long steps took, it is useless.

Store what you were given

When data arrives with a zone, keep it. When it arrives without one, record where it came from in a second column, because "the export was in the branch's local time" is a fact that lives in someone's head until the day they leave. Everything in the earlier lesson on grouping by time depends on knowing whose midnight a value refers to, and that knowledge has to be written down somewhere the query can reach.

BEFORE YOU MOVE ON

A subscription system stores each customer's next billing date by repeatedly adding one month to the previous one. A customer who first paid on 31 January is billed on 28 February, then 28 March, then 28 April. What explains the drift?

A. The database rounds every interval addition down to the nearest 28 days so that months are comparable, which permanently shortens the cycle.

B. Each addition starts from the previous result, which lost the 31 when it hit February, so month arithmetic applied step by step is not the same as adding several months at once.

C. The billing column is a `timestamp` without time zone, so the +05:30 offset was discarded on every write and each cycle lost half a day until the drift added up to a whole day per month.

D. `interval '1 month'` in Postgres is defined as 28 days for compatibility with SQLite, which normalises overflowing dates the same way.

43 · 10 min

As-of joins: the value that was true at the time

THE ONE THING TO KEEP

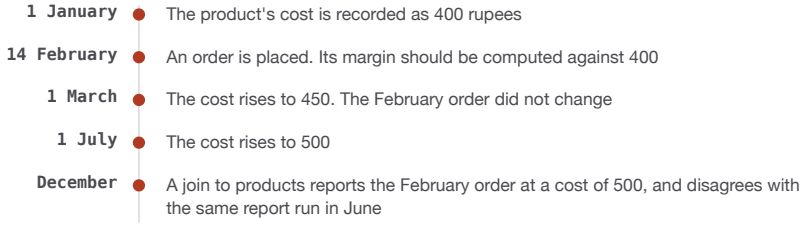
A fact that changes must be joined as of the moment the other row happened, through an effective-dated table with half-open periods that you have checked for overlaps and gaps, or through a latest-before lookup.

The report that rewrites history

A product cost ₹400 in January, ₹450 from March, and ₹500 from July. A report of margin by month joins each order to the `products` table, reads `cost`, and every January order is now shown with a cost of ₹500. Run the same report in December and it disagrees with the version run in June. Nothing was wrong with the join. The join was to the *current* value of a fact that changes, and the question was about the value at the time of the order.

The lesson on keeping each fact in one place showed one answer: copy the value onto the order row at the moment it happens, the way an invoice carries its own tax rate. That is right when the row is a legal record. It is not available when the history lives in a separate table of changes, which is the usual case for prices, exchange rates, tax rules, staff assignments and customer segments. Then the join itself has to be time-aware.

A cost that changed twice, and a report that rewrote history



Nothing was wrong with the join. It was a join to the current value of a fact that changes, asked about a moment in the past. Either copy the cost onto the order when it happens, or join to an effective-dated table through the period that contains the order's date.

An effective-dated table

The clean shape stores each version of the fact with the period it was true for:

```
CREATE TABLE product_costs (
  product_id  bigint  NOT NULL,
  valid_from  date    NOT NULL,
  valid_to    date,             -- NULL means "still current"
  cost        numeric NOT NULL
);
```

Half-open periods, as with every date range in this course: a row is in force from `valid_from` up to but not including `valid_to`. The join then asks for the version whose period contains the order's date:

```
SELECT o.order_id, o.placed_at, pc.cost
FROM orders o
JOIN product_costs pc
  ON  pc.product_id = o.product_id
  AND o.placed_at >= pc.valid_from
  AND (o.placed_at < pc.valid_to OR pc.valid_to IS NULL);
```

Each order matches exactly one cost row, provided the periods for a product neither overlap nor leave gaps. That proviso is the whole risk, and it gets its own check below.

When there is only a start date

Often the history table records only when each value began: a `price_changes` table with `changed_at` and the new price. There is no `valid_to` to test against. The version in force at a moment is then "the latest change at or before that moment", which is a top-1-per-group question with a date condition:

```
SELECT o.order_id, o.placed_at, pc.price
FROM orders o
JOIN LATERAL (
  SELECT price FROM price_changes p
  WHERE p.product_id = o.product_id AND p.changed_at <= o.-
placed_at
  ORDER BY p.changed_at DESC
  LIMIT 1
) pc ON true;
```

`LATERAL` lets the subquery refer to the order it is being run for. It is correct and it runs once per order, which is fine for a hundred thousand and worth measuring for a hundred million. The window alternative is to compute `valid_to` once with `lead(changed_at) OVER (PARTITION BY product_id ORDER BY changed_at)`, turning the start-only table into an effective-dated one, and then use the range join above.

DuckDB has this join as a first-class construct: `FROM orders o ASOF JOIN price_changes p ON o.product_id = p.product_id AND o.placed_at >= p.changed_at` picks the latest matching row per order without the subquery, and it is one of the strongest reasons to reach for DuckDB on a laptop. Pandas offers `merge_asof` for the same operation in Python.

Validating the history table

An effective-dated table that overlaps gives an order two cost rows, and the join multiplies; one with gaps gives an order none, and an inner join drops it. Both failures produce a plausible total. Check the table before trusting any join to it:

```
-- overlaps: a version starts before the previous one ends
SELECT product_id, valid_from
FROM (SELECT *, lag(valid_to) OVER (PARTITION BY product_id
ORDER BY valid_from) AS prev_to
      FROM product_costs) t
WHERE valid_from < prev_to;

-- gaps: a version starts after the previous one ended
... WHERE valid_from > prev_to;
```

Both should return no rows. Run them whenever the history table changes, because the person adding a new price does not think of themselves as editing a join.

Which is right: copy the value, or join as of?

Copying at write time makes the row self-contained, survives the history table being wrong later, and is what auditors expect on a document. Joining as of read time keeps one source of truth, lets a mistaken price be corrected everywhere at once, and is what analysis usually wants. Many systems do both: the invoice carries the rate it was issued with, and the analytics query joins as of the date to ask what the rate *should* have been. The disagreement between the two is itself a report worth running.

BEFORE YOU MOVE ON

A margin report joins each order to `products.cost` and is re-run every month. The January margin figure it shows in December is lower than the figure it showed in February for the same January orders. What is the most likely cause?

A. The join is fanning because `products` has gained duplicate rows over the year, inflating costs in proportion.

B. `cost` in `products` holds only the current value, so every re-run prices January's orders at whatever the cost is now.

C. The December run used a different session time zone, moving some January orders into February and out of the calculation.

D. Postgres inlines the CTE differently after ANALYZE has run, and the two execution plans apply the cost column at different stages.

CASE STUDY · MODULE 4

The retention curve that was falling because it was young

A spoken-English learning app in Indore, 90,000 registered learners, three days before a board meeting about a forty-lakh content investment

Ananya runs analytics for the app. Every month she produces a cohort grid: learners grouped by the month of their first completed lesson, and for each cohort the share still active one, two, three and more months later. Active means completing at least one lesson in the month.

The grid for the last fourteen months, pivoted the way the board likes it, shows month-6 retention of 31% for the January cohort of last year, 29% for March, 26% for June, 22% for September and 14% for November. Read across that row and the story is unmistakable: retention has almost halved in a year. The board's paper, drafted by someone else from her grid, says the content is stale and recommends the forty-lakh investment in new material.

Ananya checks the numbers and they are all arithmetically correct. Then she checks the shape of the grid, and finds that the November cohort is four months old. Its month-6 cell is not 14%; it is empty. What the pivot has produced for November is the last cell that had any data in it, which is month 4, and the board paper has read a triangle along an edge that does not exist.

The genuinely comparable comparison is a column, not a row: month-3 retention for every cohort that has reached month 3. That series is 34%, 33%, 31%, 32%, 30%, 29% — a slow decline of about five points over a year, not a halving.

That is still a decline, and now Ananya has a second problem. Two of the cohorts contain a promotion. In June and July the app ran a free-trial campaign with a telecom partner that brought in 21,000 registrations, of whom a large share completed one lesson and never returned. Those learners are in the June and July cohorts by the definition being used — first completed lesson — and they drag those cohorts' curves down without anything having changed

about the content. Excluding them is defensible and is also the kind of exclusion that can be made to produce any number you like.

She has three days, and the decision is what to put in front of the board.

If she sends the corrected column-wise reading, the recommendation loses most of its evidence. The head of content, who has spent two months preparing the plan, will point out — correctly — that a five-point decline is still a decline and that the app's competitors ship new material monthly. Killing the plan on Ananya's chart and then losing ground would be an expensive way to be right about a pivot table.

If she says nothing and lets the paper go, the board approves forty lakh on the strength of a number that means the November cohort is young. Ananya will know. And the same grid goes to the board every quarter, so the error will be repeated, will eventually be found by somebody, and will make every future number she produces arguable.

There is a third path with its own cost: present both readings and the promotion effect, and ask the board to decide on a smaller pilot. That is honest, it delays the content plan by a quarter in a market where a quarter matters, and it hands a room of eleven people a methodological argument three days before they were expecting a recommendation.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Ananya took the third path and made one change to the artefact rather than to the argument: she reissued the grid with every cell that lay beyond a cohort's age blanked out rather than left as the last available value, so that the triangle looked like a triangle. The board paper's headline could not be written from the new grid at all, which settled the discussion faster than her memo did. She showed the month-3 column, the five-point decline, and the June and July co-

horts with and without the telecom registrations — 29% against 33% for June, which put most of the recent decline down to who had been acquired rather than what they were taught. The board approved a pilot of about a quarter of the proposed spend, aimed at the two levels where the month-1 drop was steepest, and asked for the cohort grid to carry a caption. The caption is now four lines: what event defines a cohort, what counts as active, that the measure can rise because a learner who returns in month 4 after missing month 3 is counted again, and the age beyond which cohorts are not comparable. The head of content's own view six months later was that the pilot had been the better plan anyway, because it had a measurable edge and the full programme did not.

Explain precisely why reading the cohort grid across a row produced a halving, and what reading replaces it.

The grid is a triangle: a cohort that started four months ago has ages 0 to 4 and nothing beyond, so its month-6 cell does not exist. Reading a row compares cohorts at different ages, or worse, compares a real month-6 value with whatever the pivot put in an empty cell. The comparable reading is down a column — the same age across cohorts — and only as far as every cohort in the comparison has actually reached.

The measure counts a learner as retained in a month if they completed a lesson in it, so the curve can rise. Why does that matter when this chart is shown next to a subscription retention chart?

Because they are different measures with the same name. Purchase- or activity-based retention asks whether the learner was active in that month, so someone active in months 0, 1 and 3 is retained at 3 and not at 2, and the curve can go up. Subscription retention asks whether the learner had not cancelled by then, which can only fall. Put side by side without captions, the activity curve looks noisy or the subscription curve looks better, and neither comparison means anything.

Excluding the telecom registrations improved the June cohort by four points. What makes that a legitimate adjustment rather than a convenient one?

Legitimacy comes from the rule being stated before the result is seen, applied to every cohort rather than to the inconvenient ones, and reported alongside the unadjusted figure rather than instead of it. Ananya showed June with and without,

and the definition — registrations arriving through a specific partner campaign — is a fact about acquisition that anybody can check, not a threshold tuned until the line looked right. The same adjustment made quietly, on one cohort, would be indistinguishable from choosing the answer.

MODULE 4 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 `SELECT *, row_number() OVER (ORDER BY amount DESC) AS rn FROM orders WHERE rn <= 10` fails with "column rn does not exist". Why, and what is the fix?

A. Window functions run after WHERE, so filter the result in a second layer

B. row_number() must be given a PARTITION BY before it can be referenced

C. Aliases can never be used in a WHERE clause, so repeat the whole expression

D. The window needs a frame, which is what makes the column materialise

2 Four orders fall on days 1, 2, 2 and 3, for 10, 20, 30 and 40. `sum(amount) OVER (ORDER BY day)` returns 10, 60, 60, 100 rather than 10, 30, 60, 100. What is happening?

A. The two day-2 rows were summed before the window was applied

B. The default frame is RANGE, in which tied rows see each other

C. The window's ORDER BY sorted the rows but did not accumulate them

D. A running total requires ROWS; with RANGE it returns the partition total

3 `last_value(price) OVER (PARTITION BY product_id ORDER BY changed_at)` returns each row's own price instead of the latest one. What is wrong?

A. last_value ignores its ORDER BY and reads the physical row order

B. The partition must be dropped for last_value to see the whole product

C. The default frame ends at the current row, so last means this row

D. last_value needs DESC ordering, which makes the newest row come first

4 Five products sell 900, 700, 700, 500 and 500. A filter of `<= 3` on the ranking returns five rows. Which function was used?

A. rank(), which shares a position and then skips ahead
B. row_number(), which numbers every row separately
C. ntile(3), which cuts the list into three groups of equal count
D. dense_rank(), which shares a position and skips nothing

5 A seven-row moving average is computed over a daily revenue table in which closed days have no row. What does the line actually show?

A. An average over the last seven days on which the shop traded
B. An average over seven days, with closed days counted as zero
C. A seven-day average that lags the trend by three and a half days
D. Nothing: the window returns NULL wherever a day is missing

6 A margin report joins each order to `products` for a cost. Run in June and again in December, it disagrees about January's margins. What is the fault?

A. A join key mismatch that appeared when new products were added
B. The join reads the current cost, not the cost at the order's date
C. date_trunc placed some January orders into December's pile
D. The products table gained rows, so an inner join multiplied the orders

MODULE 5

Data that arrives dirty

No table you are handed was made for the question you have. This block is the craft of getting a file into a database without losing a row, turning text that people typed into typed columns, deciding what a duplicate and a missing value mean, and checking the result with queries that fail loudly. It ends with the three lessons that Google's and Kaggle's curricula treat as most of the job: encoding categories, building a feature table for a model, and keeping the future out of it.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Load an exported CSV with mixed encodings, ambiguous dates and duplicate rows into a typed, deduplicated table by re-runnable query with every dropped row counted, decide and document what each missing value means, and produce a feature table for a model in which no column knows anything that happened after its row's timestamp

44 · 9 min

Cleaning real data without destroying it

THE ONE THING TO KEEP

Keep the raw data, clean it in a query you can re-run, and count every row you drop.

What a real export looks like

Here is a payments file, 12,400 rows, straight from a provider's dashboard. Every one of these is real and common:

- The city column contains `Lagos`, `lagos`, `LAGOS `, `Lagos State` and `Lagos, NG`
- Dates read `03/04/2026`. That is 3 April or 4 March depending on which country exported it, and the file does not say
- Amounts appear as `₦12,500.00`, `12500`, `12,500` and `(1,200)` — that last one is accounting notation for negative
- Phone numbers with `+234`, without it, and one with a trailing tab character
- 380 rows that look like duplicates, because someone re-ran the export and pasted it underneath
- The string `N/A` sitting in a column that is otherwise numbers
- A name rendered `Ren√©`, which is UTF-8 read as something else

You cannot fix this with a list of tricks. You need a method.

Profile before you fix

Count things first. This takes four minutes and tells you what you are dealing with.

```
SELECT COUNT(*), COUNT(DISTINCT reference), COUNT(amount) FROM
raw_payments;

SELECT city, COUNT(*)
FROM raw_payments
GROUP BY city
ORDER BY COUNT(*) DESC;
```

Then — and this is the part people skip — **look at the tail of that list, not the head.** The top five values are the ones you already knew about. The errors live among the values that appear twice. Sort ascending and read the bottom forty rows. That is where `Lagos, NG` and the empty string and the row where someone typed the amount into the city column are hiding.

Clean in a query, not in the file

Keep the raw table exactly as it arrived. Never edit it. Do your cleaning in a view or a query that you can re-run:

```
CREATE VIEW payments_clean AS
SELECT
  reference,
  lower(btrim(city))                              AS city_key,
  to_date(paid_on_text, 'DD/MM/YYYY')             AS paid_on,
  replace(replace(amount_text, '₦', ''), ',', '')::numeric AS
amount
FROM raw_payments;
```

The reason is not tidiness. It is that next month the provider sends another file, and a view runs again in one second while forty minutes of manual find-and-replace has to be done again from memory, differently, by whoever is free. It is also the only way to answer "what did you actually change?" six months later, when somebody disputes your number.

Deciding what a duplicate means is not a technical question

Two rows with the same reference, the same amount, four seconds apart is almost certainly a retry, and you should keep one.

Two rows with the same customer and the same amount two weeks apart is probably a real repeat payment, and collapsing them steals money from your report.

Only someone who knows the business can tell you which rule applies. Ask, write the answer into a comment above the query, and do not let the shape of the data decide for you.

This is also why `SELECT DISTINCT *` is not a deduplication strategy. It removes only rows identical in every single column, so a retry that differs by one timestamp survives, while two genuine separate transactions that happen to match get silently merged. It is wrong in both directions at once.

Never drop silently

When 62 rows have an unparseable date, the tempting move is a filter that excludes them so the query runs. Do not. Route them somewhere you can count:

```
SELECT COUNT(*) FROM raw_payments
WHERE to_date(paid_on_text, 'DD/MM/YYYY') IS NULL;   -- 62
```

Sixty-two out of 12,400 is 0.5% and is probably fine. But you must know the number, and you must look at ten of the rows. Half the time the unparseable rows are not random — they are all from one branch, or all from one week when a system was misconfigured, and dropping them removes an entire segment from your analysis while the totals still look reasonable.

A pipeline that fails loudly is better than one that quietly discards three percent of your business.

The one habit

At the end of any cleaning job, write down two numbers: how many rows came in, and how many came out. If you cannot explain the difference row-for-row, you are not finished.

BEFORE YOU MOVE ON

An export has 12,400 rows and around 380 look like duplicates. An analyst runs `SELECT DISTINCT *`, gets 12,020 rows, and moves on. What is wrong with that?

A. DISTINCT is slow on large tables and GROUP BY on the key columns would perform better

B. DISTINCT only removes rows identical in every column, so retries that differ by a timestamp survive while genuine repeat transactions that happen to match are merged

C. The duplicates should be deleted from the source table so the fix does not have to be repeated next month

D. Nothing is wrong, since the row count dropped by 380 exactly as expected

45 · 9 min

Getting a file in without losing a row

THE ONE THING TO KEEP

Count lines before and rows after every load, and state the delimiter, quoting, header and encoding rather than letting the loader guess, because a wrong guess drops or splits rows without an error.

The row count is the first test, and it usually fails

A CSV arrives with 12,400 lines. You load it and the table has 12,317 rows. Or 12,912. Nothing errored. Both numbers mean the loader disagreed with you about where a row starts and ends, and everything downstream is now computed on a different file from the one you were sent.

Count before you load, count after, and do not proceed until you can explain the difference.

```
wc -l payments.csv        # 12401 lines, one of them the header
```

```
SELECT count(*) FROM raw_payments;   -- must be 12400, or you
know why not
```

The tools, and where each puts the file

Postgres has `COPY raw_payments FROM '/path/payments.csv' WITH (FORMAT csv, HEADER true)`, which reads a file on the *server's* disk, and `\copy` in psql, which reads a file on *your* machine and streams it. On a laptop they are the same disk; against a hosted database only `\copy` will find your file. DuckDB reads the file in place: `SELECT * FROM read_csv('payments.csv')`, and `CREATE TABLE raw_payments AS SELECT * FROM read_csv(...)` keeps it. SQLite has `.import --csv payments.csv raw_payments` in its shell. MySQL has `LOAD DATA LOCAL INFILE`. Every one of them takes options for delimiter, quoting, header and encoding, and every one guesses when you do not say.

Where the rows go missing

Embedded newlines. A quoted field may contain a line break: an address, a comment, a product description. `wc -l` counts that as two lines; a CSV parser counts it as one row. When the line count exceeds the row count by a few dozen, this is usually why, and it is not a fault. When the parser is *not* told the field is quoted, the reverse happens: one row becomes two broken ones, and the second half lands in the wrong columns.

The wrong delimiter. Excel in a German, French or Brazilian locale writes semicolons between fields, because the comma is the decimal separator there. A loader expecting commas reads each line as a single wide column and reports one column and the right row count, which is the one case where the row count passes and the load is still useless. Look at the first line of the file before loading it.

Ragged rows. A line with fewer fields than the header. Postgres stops with `missing data for column "amount"` and loads nothing, which is the honest behaviour. DuckDB can be told `ignore_errors = true` and will skip the row silently, or `store_rejects = true` to put the rejected lines in a table you can count and read. Skipping silently is how 83 rows vanish; use the version that keeps them.

The characters that are not what they look like

Text files carry no label saying which encoding they use, and the loader has to guess. The symptom of a wrong guess is a name like `RenÃ©e` or `Ren√©e`. The mechanism is worth knowing once: in UTF-8 the letter `é` is two bytes, `C3 A9`. Read those two bytes as Windows-1252 and they decode as `Ã` and `©`; read them with the old Mac Roman table and they come out as `√` and `©`. The bytes were fine. The table used to interpret them was wrong.

On a Mac or Linux, `file -I payments.csv` reports the encoding it detects. Tell the loader: `COPY ... WITH (ENCODING 'WIN1252')` in Postgres, or convert first with `iconv -f WINDOWS-1252 -t UTF-8`. Loading with the wrong encoding and "fixing" the names afterwards with find-and-replace is the path to a table where some names are fixed and some are doubly broken.

One more invisible character: Excel writes a byte-order mark, `EF BB BF`, at the start of UTF-8 files. Loaders that do not strip it produce a first column named `id`, which looks like `id` and cannot be referenced as `id`. If your first column is oddly unqueryable, this is why.

Numbers that are text

`₹12,500.00`, `12,500`, `(1,200)`: none of these is a number to a loader. Let them arrive as text. The next lesson is about turning them into numbers in a query you can re-run, rather than in the loader, where a failed cast on row 9,000 either aborts the load or, worse, drops the row.

The habit

Look at the first three lines of the file with your own eyes. Count the lines. Load with the delimiter, quoting, header and encoding stated rather than guessed. Count the rows. Explain the difference. Only then write a query.

BEFORE YOU MOVE ON

A CSV shows 12,401 lines in `wc -l`, and after loading with `HEADER true` the table has 12,358 rows. No errors were reported. What is the most likely explanation?

A. The loader deduplicated 42 identical rows during the import, which is standard behaviour for CSV readers.

B. Some quoted fields contain line breaks, so 42 rows span two lines each; the line count is inflated and the load is probably correct.

C. The file uses semicolons as delimiters, so the loader merged some adjacent lines into single rows to fill the expected number of columns.

D. The file is Windows-1252 encoded and rows containing accented characters were rejected by the UTF-8 loader as malformed and dropped without notice.

46 · 8 min

Stage as text, then cast in a query you can re-run

THE ONE THING TO KEEP

Load every column as text into a raw table that is never updated, then cast in a view beside a query that lists and counts every value the cast would refuse, because a typed load has to be right about every row before any row is in.

Let the loader guess and it will guess on the first thousand rows

DuckDB's CSV reader looks at a sample of the file, decides that a column full of digits is `BIGINT`, and loads it. The postcode column contains `400001`, `110001`, `560001` for the first twenty thousand rows and then `SW1A 1AA` for a customer in London. Depending on the version and the options, that row fails, the whole column falls back to text, or the load stops. Postgres's `COPY` does not guess at all; it insists on a table with declared types, and it aborts the entire load on the first value that does not fit, leaving you with nothing.

Both behaviours come from the same fact: a typed load has to be right about every row before any row is in. The way around it is to make the load trivially right by declaring every column as text.

```
CREATE TABLE raw_payments (
  reference   text,
  paid_on     text,
  amount      text,
  city        text,
  phone       text,
  loaded_at   timestamptz DEFAULT now(),
  load_file   text
);
```

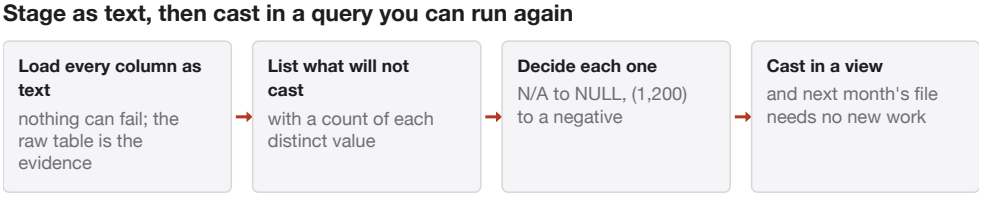
Nothing can fail. Every row lands exactly as it was in the file, and the `loaded_at` and `load_file` columns say when and from where, so next month's file can be appended without confusion. This table is never updated. It is the evidence.

Cast in a view, and count what refuses

The typed version is a query over the raw table:

```
CREATE VIEW payments_typed AS
SELECT reference,
       to_date(paid_on, 'DD/MM/YYYY')
AS paid_on,
       replace(replace(amount, '₹', ''), ',', '')::numeric
AS amount,
       city, phone, loaded_at
FROM raw_payments;
```

Run it and Postgres stops at the first amount that is not a number: `invalid input syntax for type numeric: "N/A"`. That is useful information delivered in the least useful way, because it says nothing about how many rows are affected or which. Find them first, with a test that cannot fail:



A typed load has to be right about every row before any row is in. Loading as text makes the load trivially right and moves every decision into SQL, next to a count of the rows that needed it. When that count is not zero, it is a list of decisions still to make, not a reason to add a filter.

```
SELECT amount, count(*)
FROM raw_payments
WHERE replace(replace(amount, '₹', ''), ',', '') !~ '^-?[0-9]+
(\.[0-9]+)?$'
GROUP BY amount
ORDER BY count(*) DESC;
```

The regular expression describes what a well-formed number looks like, and the query lists every value that does not match, with its frequency. `N/A` 62 times, an empty string 14 times, `(1,200)` 3 times. Now each of those is a decision: `N/A` becomes NULL, the empty string becomes NULL, and the accounting parentheses become a negative. The decisions go into the view with `CASE` and `nullif`, and the count query is kept beside it to prove that nothing is left over.

DuckDB has `try_cast(amount AS DECIMAL(12,2))`, which returns NULL instead of erroring, so the failures can be found with `WHERE try_cast(...) IS NULL AND amount IS NOT NULL`. Postgres 16 added `pg_input_is_valid(amount, 'numeric')` for the same purpose. Both are cleaner than the regex and do the same job: separate the rows that convert from the rows that need a decision.

What a wrong type destroys

Some casts lose information and cannot be undone. A phone number loaded as an integer loses its leading zero: `0221234567` becomes `221234567`, and no later query can tell whether the zero was there. Account numbers, postcodes, product codes with leading zeros, and anything with a `+` at the front all belong in text for the same reason. A number with more than fifteen significant digits, such as a 16-digit card or transaction reference, loaded as a `double` is rounded to the nearest representable value, so two different references can become the same number. A DuckDB sniffer or a spreadsheet will make both of these mistakes for you without asking, and the raw-as-text stage is the only defence.

Re-runnable is the point

When the provider sends March's file, it is appended to `raw_payments` with a new `load_file` value, and the view already handles it. When someone discovers in June that `(1,200)` was a refund and not a negative sale, the view changes in one place and every report built on it changes together. When an auditor asks what the original value was, it is in the raw table, untouched. None of that is possible if the cleaning happened in a spreadsheet before the load.

The habit

Two tables, or a table and a view: one that holds exactly what arrived, and one that holds what you decided it meant. The decisions live in SQL, next to a count of the rows that needed them. When the count is not zero, that is a list of decisions still to make, not a reason to add a filter.

BEFORE YOU MOVE ON

A phone column was loaded with the loader's guessed type, `bigint`. Later, a join to another system's phone numbers, stored as text, matches only a fraction of rows. What has most likely happened, and can the raw table fix it?

A. The bigint column overflowed for numbers longer than ten digits, wrapping them to negative values; re-casting to numeric in the view restores them.

B. Leading zeros were discarded when the digits became a number, and only a raw table that kept the column as text still holds them.

C. The text side has spaces and dashes in the numbers, and normalising both sides to digits-only in the join will make every row match.

D. bigint stores phone numbers exactly and the mismatch must come from the other system using a different country code convention.

47 · 9 min

Parsing dates that were typed by people

THE ONE THING TO KEEP

A slash-separated date is ambiguous unless some value has a day above 12, so test the column before choosing a pattern, parse in a view, and check the minimum, maximum and month histogram afterwards.

03/04/2026 is two dates

A file from a British or Indian system means 3 April. A file from an American one means 4 March. The file does not say which, and neither does the column name. Parsing it with the wrong pattern does not produce errors for the first twelve days of every month, because 03/04 is valid either way. It produces wrong dates, silently, for about forty percent of the rows, and correct ones for the rest.

Let the data tell you, when it can

If any value in the column has a first part greater than 12, the first part cannot be a month, so the format is day-first:

```
SELECT count(*) FILTER (WHERE split_part(paid_on, '/', 1)::int
> 12) AS first_part_over_12,
       count(*) FILTER (WHERE split_part(paid_on, '/', 2)::int
> 12) AS second_part_over_12
FROM raw_payments;
```

One of those counts should be zero and the other should not. If the first is 3,400 and the second is 0, the file is day-first. If both are zero, which happens on a small file or one covering only the first twelve days of a month, the data cannot tell you, and the only honest step is to ask the person who exported it. A mixed file, where both counts are non-zero, means two formats in one col-

umn, usually because two people exported from two systems, and the `load_file` column from the previous lesson is how you find which rows came from where.

The parsing functions

Postgres: `to_date(paid_on, 'DD/MM/YYYY')`. The pattern letters are `DD` day, `MM` month, `YYYY` four-digit year, `YY` two-digit year, `HH24:MI:SS` for times, and `to_timestamp` when a time is present. Modern Postgres rejects an impossible date such as 31 February with an error rather than rolling it forward, which is what you want.

DuckDB: `strptime(paid_on, '%d/%m/%Y')`, with C-style codes. MySQL: `STR_TO_DATE(paid_on, '%d/%m/%Y')`. SQLite has no parser at all; it expects ISO text and you rearrange the pieces with `substr`: `substr(d, 7, 4) || '-' || substr(d, 4, 2) || '-' || substr(d, 1, 2)`, which is ugly and works.

Two-digit years

`26` is 2026 to a human reading a recent file and 1926 to a birth-date column. Postgres's `YY` maps 00 to 69 onto 2000 to 2069 and 70 to 99 onto 1970 to 1999. A customer born in 1968 is parsed as born in 2068. For any column that can hold dates more than fifty years apart, do not accept two-digit years; find the source that has four.

Spreadsheet serial numbers

A date column exported from Excel or Sheets sometimes arrives as `45000`. That is not a code; it is the number of days since 30 December 1899, which is how spreadsheets store dates internally. `date '1899-12-30' + 45000` is 15 March 2023. The anchor is 30 December rather than 31 because of a leap-year bug in Lotus 1-2-3 that Excel reproduced on purpose in 1987 and has kept ever since. A column that mixes serial numbers and typed text needs a `CASE` that tests whether the value is all digits and parses each kind its own way.

Times, and the parts people leave out

`14:30` and `2:30 PM` are the same time in two formats; `HH24:MI` and `HH12:MI AM` in Postgres's pattern language. A value like `2026-03-04` with no time is midnight, and a range filter on it behaves as the grouping-by-time lesson described. A value like `24:00` is rejected by Postgres and accepted by some systems as the end of the day. A value with a trailing `Z` is UTC; one with `+05:30` is an offset; one with neither is a wall-clock reading whose zone lives in somebody's head, which the earlier lesson on timestamps covered.

The check that stays

After parsing, look at the distribution:

```
SELECT min(paid_on), max(paid_on), count(*) FILTER (WHERE
paid_on IS NULL)
FROM payments_typed;
```

A minimum of 1 January 1970 means a zero was parsed as a date. A maximum in the future means a day-first file was read month-first, or a two-digit year went the wrong way. A NULL count above zero is the list of values still to be decided. Add a histogram by month and read it: a file covering March with a dozen rows scattered across the other eleven months is a file with a format problem on those dozen rows, and they are usually all from one source.

BEFORE YOU MOVE ON

A file's `order_date` column contains values like `03/04/2026`. A test shows no value with a first part above 12 and no value with a second part above 12. The analyst parses it as day-first because the customer is in Mumbai. What is the status of that decision?

A. Correct and verified, because the absence of values above 12 in either position proves both parts are within valid month range and the locale settles the order.

B. Wrong, because a file with no day above 12 must be month-first, since day-first files always contain days in the teens and twenties.

C. Unverified; the data cannot decide it, and it should be confirmed with whoever exported the file before any date-based figure is reported.

D. Irrelevant, because to_date raises an error for any row parsed with the wrong pattern, so a successful parse confirms the format.

48 · 9 min

Text that looks identical and is not

THE ONE THING TO KEEP

Two strings that look identical can differ by invisible whitespace, case, or the Unicode form of an accent; build a normalised key column for grouping and joining and keep the original for display.

The GROUP BY that shows the same city twice

```
SELECT city, count(*) FROM raw_payments GROUP BY city ORDER BY
city;
-- Lagos      4,120
-- Lagos      2,893
```

Two rows, identical on screen. They differ in a character you cannot see. This lesson is the short list of ways two strings look the same and compare unequal, and the one-line fix for each.

Whitespace that trim() does not remove

`trim()` removes ordinary spaces from both ends. It does not remove a tab, a non-breaking space (the character a web page uses to stop a line wrapping, code point 160), or a zero-width space (code point 8203, which copy-and-paste from some apps inserts). All three are invisible in every tool you will look at the data with.

```
SELECT city, length(city), encode(city::bytea, 'hex')
FROM raw_payments
GROUP BY city;
```

`length` exposes the difference: one `Lagos` is five characters and the other is six. `encode(..., 'hex')` shows the bytes, and `c2a0` at the end is the non-breaking space. Remove them explicitly:

```
btrim(translate(city, chr(160) || chr(8203) || chr(9), '   '))
```

`translate` maps each listed character to a space, and `btrim` then removes the spaces. `regexp_replace(city, '\s+', ' ', 'g')` collapses runs of internal whitespace to one, so `Lagos State` and `Lagos State` agree.

Case

`lower()` makes `LAGOS`, `Lagos` and `lagos` equal. It is not always safe: the Turkish dotless i and the German `ß` both misbehave under naive lowering, and a name-matching system for Turkish customers needs the locale-aware version. For a city key in most datasets `lower()` is fine, and the display column keeps the original.

One accent, two spellings

`é` can be stored as a single code point, U+00E9, or as the letter `e` followed by a combining acute accent, U+0065 U+0301. They render identically. They are different strings, of different lengths, and `=` says so. Files from macOS often carry the second form; files from Windows the first; a database with both has customers who cannot be found by their own name.

Unicode defines a normal form so that equal-looking text has one representation. Postgres 13 and later has `normalize(name, NFC)`; DuckDB has `nfc_normalize(name)`. Apply it to every text column in the clean view, once, and the problem disappears for everything downstream. Where accents should not matter at all for matching, the `unaccent` extension in Postgres turns `Renée` into `Renee`, and the matching key uses that while the display column keeps the accent.

Build a key, keep the original

The pattern that comes out of all this is two columns per messy text field:

```
city                                                    AS
city_display,
lower(btrim(unaccent(normalize(
  translate(city, chr(160) || chr(8203) || chr(9), '   '),
NFC)))) AS city_key
```

Group, join and filter on `city_key`. Show `city_display`. Never overwrite the original, because the day you need to know that the source wrote `Lagos, NG` is the day after you deleted it.

Pulling structure out of text

A phone column with `+234 803 123 4567`, `08031234567` and `803-123-4567 (home)` has one useful thing in it, the digits. `regexp_replace(phone, '[^0-9]', '', 'g')` strips everything else, and a `CASE` on `length()` and the leading digits decides whether a country code is present. `substring(notes from '[A-Z]{2}[0-9]{6}')` pulls a reference number out of free text. Regular ex-

pressions are the tool for this, they are the same in Postgres and DuckDB for anything simple, and the honest limit is that they match shapes, not meanings: a ten-digit number in a notes column might be a phone, an order, or a date written without separators.

What matching cannot fix

After all of the above, `Lagos State` and `Lagos` still differ, and so do `Bombay` and `Mumbai`. Those are not encoding problems; they are the same place under two names, and no function knows that. The categories lesson later in this module handles them with a mapping table, which is a list of facts, not a transformation. Keep the two kinds of problem apart: normalisation makes identical things compare equal, and mapping declares that different things are the same.

BEFORE YOU MOVE ON

`GROUP BY customer_name` shows `Renée Dubois` twice with separate counts, and `length()` reports 12 for one and 13 for the other. Which explanation fits and what fixes it?

A. One row has a trailing space, and `trim()` in the grouping key will merge them.

B. One row stores `é` as one code point and the other as `e` plus a combining accent; `normalize(name, NFC)` in the key merges them.

C. The two rows have different letter case that the terminal is hiding, and `lower()` in the grouping key will merge them into a single row.

D. One value is stored in Windows-1252 and the other in UTF-8, and re-encoding the table to UTF-8 will merge them.

49 · 8 min

Duplicates: finding them, and deciding which row survives

THE ONE THING TO KEEP

Find duplicates with GROUP BY and HAVING, keep one with row_number over a stated ordering and a tiebreaker, count what you dropped, and treat fuzzy matching as a judgement whose false merges cost more than its misses.

Three kinds of duplicate

The earlier cleaning lesson made the point that whether two rows are "the same" is a business question. This lesson is about the mechanics once the question is answered, and there are three different mechanics because there are three different kinds of duplicate.

Exact duplicates are rows identical in every column. They come from a file pasted twice or a job that ran twice. **Key duplicates** share the identifying columns and differ elsewhere: the same payment reference with two time-stamps, because a retry recorded a second row. **Near duplicates** share nothing exactly and are still the same thing: `ada@example.com` and `Ada@Example.com`, or `+91 98765 43210` and `9876543210`.

Finding exact and key duplicates

```
-- exact: group by everything
SELECT reference, paid_on, amount, city, count(*)
FROM raw_payments
GROUP BY reference, paid_on, amount, city
HAVING count(*) > 1;

-- by key: group by what should be unique
SELECT reference, count(*), min(loaded_at), max(loaded_at)
FROM raw_payments
GROUP BY reference
HAVING count(*) > 1;
```

The second query's `min` and `max` of the load time tell you whether the duplicates came in one file or across two, which is usually the whole diagnosis. Look at ten of them before deciding anything.

Two kinds of duplicate, and what each tool does with them

DISTINCT keeps any row that differs in any column, so a retry that differs by one timestamp survives while two genuinely separate payments that happen to match are merged. It is wrong in both directions at once. A ranking with a stated ordering is a decision you can explain.

Keeping one, by a rule you can state

The rule is nearly always "the latest" or "the first", and a window function applies it:

```
CREATE VIEW payments_deduped AS
SELECT reference, paid_on, amount, city
FROM (
  SELECT *,
         row_number() OVER (PARTITION BY reference
                            ORDER BY loaded_at DESC, ctid DESC)
AS rn
  FROM raw_payments
) t
WHERE rn = 1;
```

Within each reference, rows are numbered from the most recently loaded, and only number 1 survives. The `ctid` tiebreaker (the physical row address in Postgres; use a serial id elsewhere) makes the choice deterministic when two rows share a load time, which the ranking lesson explained is not optional. `DISTINCT` cannot do this job, because it keeps rows that differ in any column and has no notion of "latest".

The rows that did not survive are `WHERE rn > 1`, and a count of them, by load file, goes into the load log. A deduplication that cannot say how many rows it removed is a filter, not a decision.

Deleting in place

If the duplicates are in a real table rather than a raw staging table, and the decision is to remove them permanently:

```
DELETE FROM payments
WHERE id IN (
  SELECT id FROM (
    SELECT id, row_number() OVER (PARTITION BY reference ORDER
BY created_at DESC, id DESC) AS rn
    FROM payments
  ) t WHERE rn > 1
);
```

Run the inner SELECT first and count it. Then run the DELETE inside a transaction, compare the row count it reports with that count, and only then commit. The module on changing data safely has the full discipline; the short

version is that a DELETE without a rehearsal is how a year of payments disappears.

Near duplicates

Normalising the key, as the previous lesson showed, turns most near duplicates into exact ones: lower-case the email, strip the phone to digits, and the `GROUP BY` finds them. What remains is genuinely fuzzy: `Priya Sharma` and `Priya Sharmaa`, `Bombay Traders` and `Bombay Trading Co`. Postgres's `pg_trgm` extension gives `similarity(a, b)`, a score from 0 to 1 based on shared three-letter fragments, and `levenshtein(a, b)` from `fuzzystrmatch` counts the edits between them.

```
SELECT a.name, b.name, similarity(a.name, b.name)
FROM customers a JOIN customers b ON a.id < b.id
WHERE similarity(a.name, b.name) > 0.6;
```

This is a self-join over every pair, which is a million comparisons for a thousand customers; on a hundred thousand it needs the extension's index or a blocking step that only compares names sharing a first letter and a city. And the threshold is a judgement with a cost on both sides. Set it high and `Sharma` and `Sharmaa` stay apart; set it low and two different people called Priya Sharma are merged, which loses a customer's history into somebody else's and is far harder to undo than a missed match. Review a sample of the proposed merges by hand, log every merge with both original ids, and merge fewer rather than more.

The habit

Say the rule out loud before the query: "one row per reference, the most recently loaded one wins, ties broken by id". If you cannot say the rule, you are not deduplicating; you are guessing with a window function.

BEFORE YOU MOVE ON

A team removes duplicate payments with `SELECT DISTINCT * FROM raw_payments`. Retried payments, which share a reference but differ by a few seconds in `paid_at`, all survive. Why, and what should replace it?

A. DISTINCT compares only the first column, so it keeps rows whose references match but whose later columns differ; adding `reference` to an ORDER BY clause before the DISTINCT fixes it.

B. DISTINCT keeps every row that differs in any column, and the retries differ in `paid_at`; a `row_number()` partitioned by reference and filtered to 1 keeps one per reference.

C. DISTINCT works but the retries were loaded from a second file, and DISTINCT only removes duplicates within a single load batch.

D. The `paid_at` column is a timestamptz, and DISTINCT treats timestamps that differ in time zone display as unequal even when they are the same instant.

50 · 8 min

Assertions: queries that return no rows when everything is right

THE ONE THING TO KEEP

Encode each thing that must be true as a query that returns its violations, run the whole file after every load, and include a size-and-distribution comparison with the previous load, because row-level checks cannot see a file that is silently half missing.

A test you can run is worth ten you remember to do

After the load, the casts and the deduplication, you have a clean view. You believe it is right. Next month a new file arrives, the same view runs on it, and something that was true in March is not true in April: a reference appears

twice, an amount is negative, a city is spelt a new way. Nothing errors. The report goes out.

The defence is a set of queries, each of which encodes one thing that must be true and returns the rows that violate it. When all of them return nothing, the table passes. When one of them returns rows, those rows are the problem, already found.

The checks worth writing first

The key is unique.

```
SELECT reference, count(*) FROM payments_clean GROUP BY refer-
ence HAVING count(*) > 1;
```

Required columns are present.

```
SELECT * FROM payments_clean WHERE reference IS NULL OR amount
IS NULL OR paid_on IS NULL;
```

Values are in range.

```
SELECT * FROM payments_clean
WHERE amount <= 0 OR amount > 10000000
   OR paid_on < date '2020-01-01' OR paid_on > current_date;
```

Every reference to another table resolves.

```
SELECT p.* FROM payments_clean p
LEFT JOIN customers c ON c.id = p.customer_id
WHERE c.id IS NULL;
```

The categories are ones we know.

```
SELECT status, count(*) FROM payments_clean
WHERE status NOT IN ('paid', 'refunded', 'failed') GROUP BY
status;
```

Each takes a minute to write and is a permanent record of what "clean" was supposed to mean.

The check that catches the most

None of the above notices when a file is silently half the size it should be. A truncated export, a source system that dropped a branch, a job that ran before the day's data landed: all pass every check above, because every row that *is* there is fine. The check that catches them compares this load with the last one:

```
WITH by_load AS (
  SELECT load_file, count(*) AS n, sum(amount) AS total
  FROM payments_clean GROUP BY load_file
)
SELECT * FROM by_load
WHERE n < 0.7 * (SELECT avg(n) FROM by_load)
   OR n > 1.5 * (SELECT avg(n) FROM by_load);
```

A load that is less than 70% or more than 150% of the typical size is flagged. The thresholds are arbitrary and should be, because the point is to be asked, not to be blocked. The same shape works for the share of NULL cities, the number of distinct customers, and the total amount. Distribution checks catch the failures that row-level checks are blind to.

Run them as one thing

Put every check in a file, `checks.sql`, each one labelled, and run the file after every load. A shell loop that runs each query and prints its row count is enough; a check with a non-zero count fails the load. Free tools exist that do exactly this with less typing. `dbt`, which is open source, lets you declare `unique`, `not_null`, `accepted_values` and `relationships` tests on a column in a YAML file and runs them as SQL. Great Expectations and Soda Core

do the same from Python. They are worth adopting once you have more than a dozen checks; before that, a file of queries and a loop is the same thing without the dependency.

Promote the ones that should never be violated

Some checks are not warnings but laws: a payment reference must be unique, an amount must be positive. Those belong in the table as constraints, `UNIQUE (reference)` and `CHECK (amount > 0)`, so that a bad row is refused at write time rather than found at check time. The next module builds tables that way. The queries stay useful for the rules that are usually true, such as "a load is about the size of the last one", which no constraint can express.

The habit

When you find a data problem, write the query that would have found it, add it to the file, and re-run the whole file. A month later you have a test suite that describes every way this data has ever gone wrong, and the next new way is the only one that can surprise you.

BEFORE YOU MOVE ON

A monthly payments file is silently truncated by the exporting system at 50,000 rows out of 80,000. Uniqueness, not-null, range and referential checks all pass. Which check would have caught it, and why do the others miss it?

A. A `count(*) FILTER (WHERE amount IS NULL)` check, because truncation leaves the missing rows present with NULL amounts that the not-null check should have seen.

B. A referential check against the customers table, because 30,000 customers now have no payments and appear as unmatched rows.

C. A comparison of this load's row count with previous loads, because every row that did arrive is individually valid and only the total reveals the loss.

D. A CHECK constraint on `paid_on` restricting it to the current month, because the missing rows fall outside the month and would be rejected.

51 · 9 min

Missing values: NULL, empty, N/A, zero and minus one

THE ONE THING TO KEEP

Sentinels like empty strings, N/A, zero and minus one are values a query will happily average, so convert them to NULL, keep a flag for missingness, and fill only in the feature table, because every fill shrinks variance or invents a cluster.

Five spellings of "we do not know"

The NULL lesson covered how a database compares an unknown. This one is about what to do when a column full of unknowns arrives, and it starts with the fact that most exports do not use NULL for missing at all. They use the empty string, or `N/A`, or the literal text `NULL`, or a dash, or a zero, or `-1`, or `9999`, or `1900-01-01`. Every one of these is a value that a query will happily compare, sum and average.

Find them by looking at the tail of the value list, as the earlier cleaning lesson showed, and then at the round numbers: a spike of `0` in an income column, a cluster at `-1` in an age column, a date of 1 January 1900 in a column of birthdays. Then convert each to a real NULL in the clean view, and count each kind separately:

```
CASE WHEN income_text IN ('', 'N/A', 'NULL', '-', '0', '-1',
'9999') THEN NULL
     ELSE income_text::numeric END AS income
```

The counts matter because the sentinels often mean different things. In one system `-1` means "customer refused to say" and the empty string means "the field did not exist when this account was created". Those are different facts, and the next section is about why.

Four reasons a value is missing

- **Not applicable.** A spouse's name for someone unmarried. There is no value and never will be.
- **Not collected.** A field added to the form in 2025; every account from before has nothing. Missingness is a function of account age.
- **Refused or skipped.** The customer left it blank. People who skip an income question are not a random sample of people.
- **Lost.** A migration dropped it, an integration failed for a week. Missingness is a function of a date range.

The treatment differs by reason. Not-applicable values should stay NULL and be handled with `CASE`. Not-collected values can often be recovered from another source. Lost values can be identified by date and the affected period excluded or flagged. Refused values are the dangerous ones, because they are missing *because of* what they would have been, and no arithmetic on the rows that remain can undo that.

What filling in does to the numbers

A model or a chart needs a number in every cell, so the temptation is to fill the gaps. Each fill has a mechanical consequence.

Fill with **zero** and the average falls, and a histogram grows a spike at zero that describes nobody. Fill with the **mean** and the average is unchanged, but the spread shrinks, because every filled row sits exactly on the centre, and anything that depends on variability (a confidence interval, a correlation, a model's certainty) is now overstated. Fill with the **median** and the same happens, slightly less. Fill by **carrying the last value forward** in a time series and a sensor that died looks like it is reporting a steady reading.

The fill that loses least is the one that keeps the fact of the missingness:

```
coalesce(income, (SELECT percentile_cont(0.5) WITHIN GROUP (OR-
DER BY income) FROM customers))
  AS income_filled,
(income IS NULL)::int AS income_missing
```

Two columns instead of one. The model can learn that the missing flag matters, which it often does, since customers who skip the income question behave differently from those who answer. A report can show the rate of missingness next to the average. The information was never thrown away.

Dropping rows and dropping columns

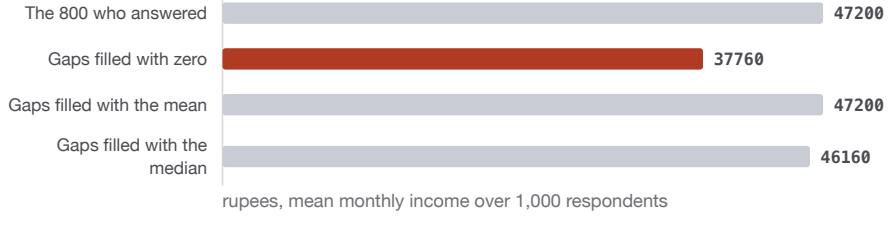
A column that is 90% missing is usually a column to drop, unless the 10% is a specific, meaningful group. A row with one missing value is usually a row to keep, with the flag above. Dropping every row that has any missing value is the default in many tools and the worst option in most datasets, because missingness is rarely random. If customers who churn stop filling in the survey, dropping the incomplete rows drops the churners, and the retention figure computed on what remains is a figure about the loyal.

The test is simple: compare the rows you would drop with the rows you would keep, on a column you care about. If the two groups differ, dropping is not neutral.

Where each decision lives

The raw table keeps the sentinels. The clean view converts them to NULL and keeps NULL. The feature table, which is two lessons away, is where filling happens, next to a flag, because that is the only place a filled value is needed and the only place the filling rule can be stated once. Reports read the clean view and decide, per query, what an unknown should do, as the NULL lesson said.

Two hundred people skipped the income question. Four ways to report it



Eight hundred people answered, with a mean of 47,200 and a median of 42,000. Filling with zero invents two hundred destitute respondents and drags the mean down by a fifth. Filling with the mean leaves the mean alone and quietly shrinks the spread, because every filled row sits exactly on the centre.

BEFORE YOU MOVE ON

A team fills every missing `income` with the column's mean before training a model and building a report. Which consequence follows from the mechanism of that choice?

A. The average income in the report rises, because the mean of a partially filled column is always above the mean of the values that were present, since the blanks were disproportionately low earners.

B. The model gains accuracy, because the filled rows now carry the most representative value the column can have.

C. The spread of income shrinks, since every filled row sits exactly at the centre, so anything relying on variability is overstated, and the fact of missingness is lost unless a flag is kept.

D. The database refuses the update, because `coalesce` with an aggregate subquery is not allowed inside a view that feeds a model.

52 · 9 min

Categories: canonical values, mapping tables and encoding for a model

THE ONE THING TO KEEP

Clean a category with a mapping table that a LEFT JOIN can expose gaps in, choose the encoding by cardinality and by whether the values have an order, and never give a model integer codes for categories that are not ordered.

A category is a value from a list, and the list is the asset

`status`, `city`, `plan`, `channel`: columns whose values come from a set rather than a range. A number column is cleaned by casting; a category column is cleaned by deciding what the set is and mapping everything onto it. The set is the thing to write down, because the day it exists only inside a long

`CASE` expression is the day two reports disagree about how many plans there are.

A mapping table instead of a CASE

`Bombay`, `Mumbai`, `MUMBAI`, `Mumbai, MH` and `Bombay (Mumbai)` are one city. Normalising case and whitespace, as the text lesson showed, collapses three of them. The other two are a fact somebody has to declare, and the place to declare it is a table:

```
CREATE TABLE city_map (
  variant   text PRIMARY KEY,   -- the normalised key as it ap-
pears in the data
  canonical text NOT NULL
);
INSERT INTO city_map VALUES ('bombay', 'Mumbai'), ('mumbai',
'Mumbai'), ('mumbai, mh', 'Mumbai');
```

```
SELECT p.*, coalesce(m.canonical, p.city_key) AS city
FROM payments_clean p
LEFT JOIN city_map m ON m.variant = p.city_key;
```

The LEFT JOIN is the important part. A variant that is not in the table comes through unchanged, and a query for `WHERE m.variant IS NULL` lists every value the map has not seen yet, with its frequency. That list is the to-do list, it gets shorter every month, and adding a row to a table is something a non-technical colleague can do while editing a `CASE` is not.

Cardinality decides what you can do with it

`count(DISTINCT city)` is the first thing to know about a category. A `status` with 4 values and a `postcode` with 19,000 values are both categories and nothing else about handling them is the same. Low-cardinality columns can be pivoted into one column each, listed in a report, and used as-is by most tools. High-cardinality columns cannot be pivoted, cannot be listed, and need a different treatment before a model sees them.

What a model needs

Most models take numbers. A column of city names has to become numbers, and there are three ways, each with a mechanism worth understanding.

One-hot encoding makes one 0/1 column per value: `is_mumbai`, `is_delhi`, `is_pune`. In SQL that is a `CASE` per value, or `count(*) FILTER (WHERE city = 'Mumbai')` in a grouped query. It is faithful and it does not scale: 19,000 postcodes become 19,000 columns, nearly all zero on every row. The free path for the mechanics is `pandas.get_dummies` or scikit-learn's `OneHotEncoder`, but the decision about which values to encode is a SQL decision about cardinality.

Integer codes replace each value with a number: Mumbai 1, Delhi 2, Pune 3. Compact, and wrong for most models, because a linear model or a distance-based one will treat Pune as "more" than Mumbai and halfway between Delhi and something else. The numbers have an order that the cities do not. Integer codes are correct only for genuinely ordered categories (small, medium, large; bronze, silver, gold), and then the order must be the one you assign, not the alphabetical one a tool picks.

Frequency and target encoding replace each value with a statistic about it: how often it appears, or the average outcome for rows with that value. Both are one SQL query.

```
SELECT city, count(*) AS city_freq, avg(churned::int) AS
city_churn_rate
FROM customers GROUP BY city;
```

Frequency encoding is safe. Target encoding is powerful and is the most common source of leakage in tabular work, because a row's own outcome contributes to the average it is then given as a feature. The rate must be computed on training rows only, excluding the row itself, or the model learns to read its own answer. The leakage lesson returns to this.

Rare values and new values

A category with 300 values where 280 of them appear fewer than ten times is better encoded as the top 20 plus `other`. Decide the threshold in SQL, from counts, and write the list of kept values into a table so that next month's data is encoded the same way. A value that appears for the first time at prediction time, a new city, must map to `other` and not crash the pipeline, which the mapping-table approach handles by construction and a hard-coded `CASE` does not.

The habit

For every category column, write down three things: the canonical list, the mapping from variants to it, and the encoding rule for a model. All three live in tables or in one view, so that the report and the model agree on what a city is.

BEFORE YOU MOVE ON

A model is trained with `city` replaced by an integer code assigned alphabetically, so Ahmedabad is 1, Bengaluru is 2, Chennai is 3, and so on. Why is that a problem for a linear or distance-based model?

A. Integer codes overflow for high-cardinality columns, so cities beyond the first few thousand share codes and are confused with each other.

B. The codes carry an order and spacing that the cities do not have, so the model treats Chennai as greater than Ahmedabad and two steps away from it.

C. Alphabetical codes change whenever a new city is added, so the model cannot be retrained without re-encoding, which is fine for the first training run.

D. Linear models cannot accept integer columns at all and require every feature to be a floating-point value between 0 and 1.

53 · 10 min

A feature table: one row per entity, as of a moment

THE ONE THING TO KEEP

A feature table has one row per entity per as-of moment, every feature computed from strictly before that moment and the label from strictly after, built by one parameterised query that also serves predictions so training and live features cannot drift apart.

The table a model is trained on has a very specific shape

One row per thing you are predicting about, at a moment in time. Every column except one is something you knew at that moment. The one exception is the label: what happened afterwards, which is what the model is learning to predict. This lesson builds that table in SQL, because most of the work of a model is this query, and most of the ways a model goes wrong are ways this query was wrong.

Pick the moment first

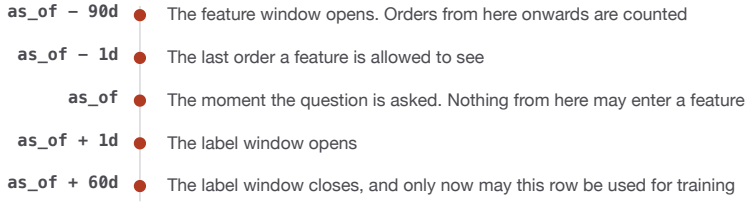
Say the prediction is "will this customer place no orders in the next 60 days". The moment is an `as_of` date. The features are computed from orders strictly before it. The label is computed from orders in the 60 days after it. A customer can appear many times, once per `as_of` date, and each appearance is a separate training example: the same customer on 1 January and 1 February are two rows with different features and possibly different labels.

Generate the moments with a spine, as the earlier module did for months:

```
WITH snapshots AS (
  SELECT c.id AS customer_id, d::date AS as_of
  FROM customers c
  CROSS JOIN generate_series(date '2025-01-01', date '2026-03-
01', interval '1 month') d
  WHERE c.created_at < d       -- only customers who existed at
that moment
)
```

Ten thousand customers and fifteen monthly snapshots give 150,000 rows, each a moment somebody could have been asked the question.

One row of a feature table, and which side of the moment each column comes from



Read the finished query and, for every column, say which side of as_of its data comes from. Every feature: before. The label: after. The last event is the one people forget, and forgetting it labels every recent customer as churned because their future has not happened yet.

Features from the past only

```
features AS (
  SELECT s.customer_id, s.as_of,
         count(o.id)                                      AS or-
ders_90d,
         coalesce(sum(o.amount), 0)                       AS
spend_90d,
         s.as_of - max(o.placed_at)::date                 AS
days_since_last_order,
         s.as_of - c.created_at::date                     AS
tenure_days
  FROM snapshots s
  JOIN customers c ON c.id = s.customer_id
  LEFT JOIN orders o
    ON  o.customer_id = s.customer_id
    AND o.placed_at <  s.as_of
    AND o.placed_at >= s.as_of - interval '90 days'
  GROUP BY s.customer_id, s.as_of, c.created_at
)
```

Every condition on `o.placed_at` is strictly before `as_of`. The LEFT JOIN keeps customers with no recent orders, which are precisely the ones most likely to churn and would be missing from an inner join. `days_since_last_order` is NULL for a customer who never ordered in the window, and the missing-values lesson says what to do with that: keep a flag, fill in this table, and nowhere else.

The label from the future only

```
labels AS (
  SELECT s.customer_id, s.as_of,
         (count(o.id) = 0)::int AS churned_60d
  FROM snapshots s
  LEFT JOIN orders o
    ON  o.customer_id = s.customer_id
    AND o.placed_at >= s.as_of
    AND o.placed_at <  s.as_of + interval '60 days'
  GROUP BY s.customer_id, s.as_of
)
SELECT f.*, l.churned_60d
FROM features f JOIN labels l USING (customer_id, as_of)
WHERE l.as_of + interval '60 days' <= current_date;   -- the
label window must have closed
```

The final WHERE is easy to forget and matters: a snapshot taken 30 days ago does not yet know whether the customer churned, and including it labels every such customer as churned because their future has not happened yet.

The same query at prediction time

When the model is used, somebody has to compute the same features for today. If the training query used a column that a nightly job fills in three hours after the prediction runs, the live features are different from the trained ones, and the model's accuracy in production falls without any error being raised. This is called train/serve skew, and the defence is structural: one query, parameterised by `as_of`, produces both the training rows (with many historical dates) and the live rows (with today's date and no label). If the query cannot run for today, it should not be the training query.

Where it lives

Write the result to a table with `as_of` as part of its key, and never overwrite it. Next month's run appends new snapshots. A model trained in March can be compared with one trained in June on the same rows, and a mysterious change in accuracy can be traced to a change in the features or in the data

rather than in both at once. DuckDB on a laptop handles this comfortably for millions of rows, and `COPY features TO 'features.parquet'` hands the result to Python or to whatever trains the model.

The habit

Read the finished query and, for every column, say which side of `as_of` its data comes from. Every feature: before. The label: after. Anything you cannot place is the subject of the next lesson.

BEFORE YOU MOVE ON

A churn feature table is built with monthly snapshots up to 1 March 2026, a 60-day label window, and the query is run on 20 March 2026. What is wrong with the 1 March rows?

A. Their features include orders from March, because the 90-day window is measured backwards from the run date rather than from `as_of`.

B. Their 60-day label window has not closed, so customers who have not yet ordered are labelled churned when their future simply has not happened.

C. They duplicate the 1 February rows, since a customer with no orders in either month produces identical features and the join cannot separate them.

D. They are missing entirely, because `generate_series` stops before the last date when the interval does not divide the range evenly.

54 · 10 min

Leakage: when a feature knows the future

THE ONE THING TO KEEP

Leakage enters a feature table through an unbounded aggregate, a join to a current value, a column written after the outcome, a target encoding that includes the row itself, or a split that puts one entity on both sides; a timestamp check catches the first two and a person who knows the source systems catches the rest.

The model that scored 99% and failed on Monday

A churn model is trained, evaluated on held-out rows, and reports 0.99 accuracy. It goes live and predicts almost nothing useful. Nothing in the model is broken. One of its features contained the answer, in a form that was available in the training table and is not available on the day a prediction is needed. That is leakage, and in tabular work it almost always enters through a join or an aggregate in the feature query.

Five ways it gets in

An aggregate over all history. `sum(amount) AS total_spend` computed over every order the customer ever placed, including the ones after `as_of`. A customer whose total spend keeps rising after the snapshot is, by definition, not churning. The model learns "high total spend means not churned" and is right in training and useless live, where the future orders do not exist yet. The previous lesson's `o.placed_at < s.as_of` is the fix, on every join, every time.

A join to a current dimension value. `customers.segment` is joined in for every snapshot, and the segment is set to `'lapsed'` by a nightly job when a customer has not ordered for 60 days. For a snapshot from last year, the segment now says what the customer became, which is the label. The as-of join

from the time module is the fix: the segment that was true on `as_of`, from a history table, or no segment at all if no history exists.

A column that is updated after the outcome. `last_contacted_at`, `account_status`, `refund_count`, `support_tickets`. Each is edited by a process that happens because of the outcome: the retention team contacts customers who look like leaving, the refund is issued when they complain on the way out. Any column that can change after `as_of` is suspect unless its history is stored and joined as of.

Target encoding on the whole table. The categories lesson computed `avg(churned)` per city and used it as a feature. If that average includes the row it is then attached to, the row's own label has leaked into its feature, a little for a big city and a lot for a city with three customers. Compute it on training rows only, and exclude the current row: `(sum(churned) - churned) / (count(*) - 1)` over the group.

The same customer on both sides of the split. Rows for one customer at January and February snapshots are near-duplicates. Put one in training and one in test, and the model is evaluated on rows it has effectively seen. The split must be by customer, or by time with all training snapshots before all test snapshots, which is also the split that resembles production.

Detecting it

Some leaks announce themselves. A feature with an almost perfect correlation with the label is a leak until proved otherwise; real signals are weaker than that. A model that is far better than the previous one after a change to the feature query is suspect in proportion to the improvement.

Others can be found mechanically. For each feature, the query should be able to state the source table and the timestamp column it was bounded by. A check query asserts it:

The same four features, written twice

Knows the future	Knows only the past
• sum(amount) over every order the customer ever placed	• sum(amount) where placed_at is strictly before as_of
• customers.segment, which last night's job set to 'lapsed'	• The segment that was true on as_of, from a history table
• A city's churn rate whose average includes this row's own label	• The rate over training rows only, with this row subtracted out
• January and February snapshots of one customer, split across training and test	• A split by customer, or by time with every training snapshot first

A feature with an almost perfect correlation with the label is a leak until proved otherwise; real signals are weaker than that. A timestamp check finds the first two of these. The other two are found by a person who knows which process writes each column, and when.

```
-- must return no rows: no feature source row is at or after
its snapshot
SELECT s.customer_id, s.as_of, max(o.placed_at) AS latest_used
FROM snapshots s JOIN orders_used_for_features o USING (cus-
tomer_id, as_of)
GROUP BY 1, 2
HAVING max(o.placed_at) >= s.as_of;
```

Run it beside the feature query, in the same file as the validation checks from earlier in this module.

The leaks a timestamp check cannot see

A `notes` field that a support agent fills in only when a customer calls to cancel. A `data_entry_timestamp` that is populated only for accounts that went through a manual review, which happens mostly to accounts about to be closed. The timestamps of these columns are before `as_of`, and they still carry the future, because the *fact that they exist* is caused by the outcome. No query can catch this. It is found by a person who knows how the data is produced reading the feature list and asking, of each column, "what process writes this, and does that process know the outcome?" That conversation is the most valuable hour in a modelling project, and it is not a technical hour.

The habit

Before a feature table is used, two things happen. The timestamp check runs and returns nothing. And somebody who understands the source systems reads the column list and signs off that no column is written by a process downstream of the outcome. The second is the one that gets skipped, and it is the one that would have caught the 99%.

BEFORE YOU MOVE ON

A churn model's strongest feature is `total_lifetime_spend`, computed as `sum(amount)` over all of a customer's orders. Evaluation accuracy is 0.98; live performance is poor. What is the mechanism?

A. Lifetime spend is dominated by a few large customers, and the model overfits to them; scaling the column to a 0 to 1 range will fix it.

B. Customers who churn have their orders deleted by a retention process, so their lifetime spend is artificially low in the training table and the model learns that low spend means churn.

C. The sum includes orders placed after each snapshot, so a customer whose spend continues to grow is one who did not churn, and the feature encodes the label.

D. `sum(amount)` skips NULL amounts, so customers with incomplete order data appear to spend less and are labelled as churners.

55 · 8 min

Why a query is slow, and what an index does

THE ONE THING TO KEEP

An index lets the database skip rows; anything that hides a column's raw value takes that ability away.

Slow almost always means reading too much

A database with no help reads every row of the table and checks each one. Four million payment rows at 200 bytes each is about 800 MB of reading to answer a question whose answer is six rows. Even from memory that is not free. From disk it is seconds.

Ask the database what it is doing rather than guessing:

```
EXPLAIN ANALYZE
SELECT id FROM payments WHERE student_id = 4412;

-- Seq Scan on payments  (cost=0.00..84210.00 rows=1 width=8)
--   Filter: (student_id = 4412)
--   Rows Removed by Filter: 3999994
--   Planning Time: 0.1 ms
--   Execution Time: 612.4 ms
```

`Seq Scan` means it read the whole table. `Rows Removed by Filter: 3999994` means it threw away almost all of it. That line is the confession.

What an index actually is

An index is a second structure — normally a B-tree — holding the values of one or more columns in sorted order, each with a pointer to where the full row lives. Because it is sorted, the database can binary search it. Four million rows becomes about 22 comparisons instead of four million.

```
CREATE INDEX ON payments (student_id);

-- Index Scan using payments_student_id_idx on payments
--   (actual time=0.021..0.033 rows=6 loops=1)
--   Execution Time: 0.08 ms
```

612 ms to 0.08 ms. That is the whole trick, and it is a big trick.

Indexes are not free. Every INSERT, UPDATE and DELETE must update every index on the table. They take disk space. A table with twelve indexes that is written constantly will be slow to write and its indexes will mostly go unused. Index the columns you filter and join on, not every column you have.

When the index sits there unused

You hid the raw value behind a function.

```
WHERE date(created_at) = '2026-03-14'   -- index on created_at
is useless
WHERE lower(email) = 'ada@example.com'  -- index on email is
useless
```

The index stores `created_at`, not `date(created_at)`. To compare, the database must compute the function for every row first, which means reading every row. Rewrite as a range, or build an index on the expression itself:

```
WHERE created_at >= '2026-03-14' AND created_at < '2026-03-15'
CREATE INDEX ON users (lower(email));
```

A leading wildcard. `LIKE '%kumar'` cannot use a B-tree, because a sorted list does not help you find things by their ending. `LIKE 'kumar%'` can.

The filter matches most of the table. `WHERE is_active = true` when 95% of rows are active. Following the index and then fetching 95% of the rows one at a time is slower than reading the table straight through. When the planner ignores your index here, it is right and you are wrong.

The table is small. Scanning 500 rows takes microseconds. No index will beat that.

The wins that are usually bigger than an index

Return less. Add the missing filter. Add a LIMIT. A query that returns 400,000 rows to an application that displays 20 is slow because of the 399,980, not because of the plan.

Stop using `SELECT *` on wide tables. If one column holds a 40 KB JSON blob, selecting it for every row means moving hundreds of megabytes across the network to display a name and a date.

Do not run one query per row. An application that fetches 900 orders and then runs one query per order to get the customer has made 901 round trips. At 3 ms each that is nearly three seconds, and the database will report every individual query as fast. One join does it in 12 ms. This is the most common performance bug in real software and it never shows up as a slow query in the logs.

Reading a plan without being an expert

Find the node with the largest `actual time`. That is your problem, not the scary-looking one at the top. Then compare the estimated `rows=` against the `actual rows=`. If the planner expected 3 rows and got 900,000, it chose its whole strategy on a bad guess — usually stale statistics (run `ANALYZE`) or a filter it cannot reason about, like a function on a column.

BEFORE YOU MOVE ON

A table has an index on `created_at`, a timestamp. The query `WHERE date(created_at) = '2026-03-14'` still takes eight seconds. Why is the index not helping?

A. The index has gone stale after so many inserts and needs to be rebuilt

B. The index stores the raw timestamps, so the database must compute date() on every row before it can compare anything

C. A timestamp is being compared to a string, and the type mismatch prevents the index from being used

D. The planner decided the table was too small to be worth an index lookup

CASE STUDY · MODULE 5

The eleven hundred rows that were not random

Sahyog Diagnostics, a chain of nine collection centres around Coimbatore, loading a year of partner referral records for a payment reconciliation

The partner hospital pays Sahyog a fee per referred test. Once a year the hospital sends a CSV and the two finance teams reconcile it against Sahyog's own records. This year's file has 28,900 rows and one column that will not behave: `referred_on`, which arrives as text like `14/01/2026`, `2026-01-14`, `14-Jan-26` and, in a few hundred rows, `45671`.

Meena, who does the analysis, stages the file exactly as she was taught: every column as `text`, into a `raw_referrals` table that is never updated, with a `load_file` column recording where each row came from. Then she writes the casting view and counts what refuses.

`to_date(referred_on, 'DD/MM/YYYY')` parses 27,720 rows and fails on 1,180. She lists the failures with their frequencies. Six hundred and ten are `14-Jan-26` style, three hundred and twelve are ISO, two hundred and fifty-eight are five-digit numbers — spreadsheet serials, days since 30 December 1899 — and none of the three is difficult to parse once identified. It is an hour's work to write a `CASE` that tests the shape of each value and applies the right parser.

The reconciliation is due to the partner's finance office in two days, and Meena's manager points out that 1,180 out of 28,900 is 4.1%, that the fee is a flat 340 rupees per referral, and that the difference is about four lakh on a settlement of just under a crore. His proposal is to load the 27,720 rows that parse, settle on those, and reconcile the remainder next quarter when there is time. Four lakh is real money and it is also within the tolerance the two organisations have used before.

Meena runs one more query before answering: the 1,180 failing rows grouped by centre and by month.

They are not spread across the year and the nine centres. One thousand and four of them are from the Pollachi centre, and every one of those is from 14 January onwards. Pollachi changed its practice-management software in the second week of January, and the new software writes dates in a different format and exports spreadsheet serials for anything entered on a mobile device.

So the 4.1% is not a random 4.1%. It is most of one centre's second half of the year. Dropping it does not lose four lakh evenly; it removes Pollachi from the analysis from mid-January, and Pollachi is the centre whose referral volume the partner hospital has been questioning. A settlement computed on the parsing rows would under-pay Sahyog for that centre specifically, and — worse for the relationship — would produce a referral count for Pollachi that supports the partner's argument that the centre is not referring.

Meena's manager is not being careless. His point is that an hour of parsing work invites a second hour of checking, and a third when the parsed dates turn out to move rows between months, and that the deadline is real: the partner's own quarter closes on Friday and a late reconciliation waits three months for the next cycle. He has also seen a colleague spend a week on a date column and arrive at the same settlement figure, and he is weighing a certain cost in staff time against an uncertain gain of a few lakh. What he has not seen is Meena's query grouping the failures by centre, because she ran it after he made his proposal and before she answered him.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Meena did the hour of parsing and it cost most of a day, as her manager predicted. Two of the three formats were straightforward. The spreadsheet serials were not: 258 rows parsed correctly with the 30 December 1899 anchor, but a histogram by month afterwards showed 41 of them landing in 1970, which turned out to be zeros in the source — a genuinely empty field exported as `0`

rather than as an empty string. Those 41 went to NULL with a flag, and the count of them went into the load note. The reparsed file gave Pollachi 1,004 additional referrals worth 3.41 lakh, and moved 63 rows across a month boundary because the day-first and month-first readings of the same string had been landing them in different months. The reconciliation went out on Friday with a one-page note listing the three date formats found, the counts of each, the 41 rows set to NULL and why. The partner hospital's finance office replied that they had been dropping unparseable rows from their own side for two years and had no idea how many, which turned the settlement into a joint exercise to rebuild both sides from the raw files. The parsing view is now run monthly rather than annually, and a check that fails the load when a new date format appears was added the same week.

The manager's proposal to drop 4.1% of rows would have been reasonable under one condition that turned out not to hold. What was it, and what query established that it did not?

The condition is that the dropped rows are a random sample of the file, so that removing them shifts the total without changing its shape. The query that tested it was a count of the failing rows grouped by centre and by month, which showed 1,004 of 1,180 coming from one centre from one date onwards. Unparseable rows are very often not random: they are one branch, one week, or one system change, and dropping them removes a whole segment while the totals still look plausible.

Why did staging every column as text make this recoverable, when a typed load would not have?

A typed load has to be right about every row before any row is in: Postgres would have aborted the whole COPY on the first unparseable date, and a loader that guesses would have silently skipped or mangled the offending rows. Text staging cannot fail, so all 28,900 rows are on disk exactly as they arrived, and every decision about what they mean happens afterwards in a view, next to a count of the rows that needed the decision. When the spreadsheet serials turned out to include zeros, the original values were still there to look at.

After reparsing, 63 rows moved across a month boundary. Why is that worth reporting rather than quietly fixing?

Because it changes figures that have already been sent: a monthly referral count for December or January that somebody has seen will now be different, and the

difference has a cause anyone can check. Reporting it is also the only way the other side can find the same rows in their own file. A cleaning step that cannot say what it changed is indistinguishable from a cleaning step that changed the wrong thing, and the whole point of cleaning in a re-runnable query is to be able to answer "what did you actually change?" six months later.

MODULE 5 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Why load every column of a messy CSV as text before casting anything?

A. Text columns are faster to load, since no conversion work is done
B. A typed load has to be right about every row before any row is in
C. The engine cannot infer a type until the whole file has been read once
D. Casting later lets the database choose a narrower type than you would

2 A date column holds values like `03/04/2026` and the file does not say which country produced it. What tells you whether it is day-first?

A. Whether any value has a first part greater than 12
B. Whether the file also contains times in a 24-hour format
C. Whether more values parse successfully under one pattern than the other
D. Whether the maximum parsed date falls in the future

3 Why is `SELECT DISTINCT *` not a deduplication strategy for a payments export?

A. DISTINCT cannot be used together with a GROUP BY on the same query
B. It removes rows differing in any column and merges genuine repeats
C. It keeps the last row of each group, which is rarely the row you want
D. It is slower than a window function on any table above a million rows

4 Two hundred of a thousand respondents skipped the income question. The gaps are filled with the column's mean. What has changed?

A. The mean has risen, because the filled rows sit above the median
B. Nothing has changed, which is why mean-filling is the neutral choice
C. The mean is unchanged and the spread has shrunk
D. The mean is unchanged and the count of respondents has fallen

5 A nightly export arrives with half its usual rows because a source system dropped a branch. Every row-level check passes. What catches it?

A. A NOT NULL constraint on the branch column of the target table
B. A unique key check, since the missing branch breaks referential integrity
C. Comparing this load's row count and totals with previous loads
D. A foreign key from the load table to the list of branches

6 A churn model scores 0.99 in testing and predicts almost nothing useful in production. Its feature `total_spend` is `sum(amount)` over every order the customer ever placed. What went wrong?

A. The training set was too small for a feature with that much variance
B. The feature is on a different scale from the others and dominates the model
C. The sum includes orders placed after the snapshot, which is the answer
D. Customers with no orders were dropped, so the model never saw them

MODULE 6

Changing data safely

Everything so far has read. This block writes: tables whose constraints refuse bad rows, inserts that can be run twice without duplicating, updates and deletes rehearsed as SELECTs, transactions that make a mistake reversible, two people writing the same row at once, schema changes on a table that is in use, views, JSON columns, history tables, and who is allowed to do any of it. It ends by designing a small schema from a paragraph of requirements.

BY THE END OF THIS MODULE YOU CAN

Design and create a small schema whose keys and constraints refuse bad data, insert, update and delete inside a transaction you have rehearsed and can roll back, explain what happens when two writers touch the same row, alter a table that is in use without an outage, and give an analyst or an AI a read-only path into the database

56 · 9 min

CREATE TABLE: a constraint is a test that runs on every write

THE ONE THING TO KEEP

A constraint is a test the database runs on every write, refusing the row rather than flagging it later; name each one, and know that CHECK sees one row at a time, so multi-row rules need UNIQUE, EXCLUDE or a trigger.

The table is the first line of defence

Every check in the previous module ran after the data was in. A constraint runs before, on every insert and every update, and a row that fails it never exists. That is the difference between finding a negative payment in next month's audit and never having one.

```
CREATE TABLE payments (
  id           bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  reference    text        NOT NULL,
  student_id   bigint      NOT NULL REFERENCES students (id),
  amount       numeric(12, 2) NOT NULL,
  paid_on      date        NOT NULL,
  method       text        NOT NULL,
  note         text,
  created_at   timestamptz NOT NULL DEFAULT now(),
  CONSTRAINT payments_reference_unique  UNIQUE (reference),
  CONSTRAINT payments_amount_positive   CHECK (amount > 0),
  CONSTRAINT payments_method_known      CHECK (method IN
('cash', 'upi', 'card', 'transfer')),
  CONSTRAINT payments_not_in_future     CHECK (paid_on <= cur-
rent_date)
);
```

Read it as a list of promises. Every payment has a reference and it is unique. Every payment belongs to a student who exists. The amount is money, to the

paisa, and positive. The method is one of four. The date is not in the future. `note` is the only thing allowed to be missing, and that is a decision, not an omission.

What each kind refuses

`NOT NULL` refuses an unknown. Pair it with `DEFAULT` for columns that should be filled in automatically, as `created_at` is; a `NOT NULL` column with no default and no value in the INSERT is an error, which is what you want for `amount` and not for `created_at`.

`UNIQUE` refuses a repeat. It builds an index to do it, which the performance module returns to, and it treats NULLs as distinct from each other, so a nullable unique column can hold many NULLs. `PRIMARY KEY` is `UNIQUE` plus `NOT NULL` plus the declaration that this is the identity, as the keys lesson described.

`CHECK` refuses any row for which its expression is not true. It can reference several columns of the same row: `CHECK (valid_to IS NULL OR valid_to > valid_from)` is the guard for every effective-dated table in the time module. It cannot look at other rows or other tables, and that limit matters below.

`REFERENCES` refuses a row that points at nothing, and refuses (or cascades) the deletion of a row that is pointed at.

Name them

An unnamed constraint produces an error like `new row violates check constraint "payments_check1"`. A named one says `payments_amount_positive`, and the person reading the error at two in the morning knows what happened without opening the schema. Names cost nothing.

The rule a CHECK cannot express

"A student may not have more than one payment per day" is a rule about two rows. "A booking may not overlap another booking of the same room" is a rule about two rows. `CHECK` sees one row at a time and cannot enforce either. The options, in order of preference:

A `UNIQUE` constraint expresses "at most one row per combination": `UNIQUE (student_id, paid_on)` enforces the first rule exactly. Postgres's `EXCLUDE` constraint expresses "no two rows may overlap", which is the second: `EXCLUDE USING gist (room_id WITH =, during WITH &&)` on a range column, and two bookings for the same room whose ranges intersect cannot both exist. Beyond those, the rule needs a trigger, which is code that runs on every write and can query anything, or it lives in the application, which means every application. A rule that lives in the database is enforced for the admin fixing data by hand at midnight; a rule in the application is not.

Six refusals, from the shape of one value to a rule about two rows

The type	refuses a value of the wrong shape
NOT NULL	refuses an unknown
CHECK	refuses a value this row may not hold
UNIQUE	refuses a repeat, and states the grain
REFERENCES	refuses a row that points at nothing
EXCLUDE, or a trigger	the only way to state a rule about two rows

A constraint costs a comparison per write. A missing one costs the query that finds the bad rows a year later, the decision about what to do with them, and every report that ran in between. Note where the ladder stops: CHECK sees one row at a time and cannot express a rule about two.

The engines differ in what they enforce

SQLite accepts a `REFERENCES` clause and, by default, ignores it. Foreign keys are enforced only when the connection has run `PRAGMA foreign_keys = ON`, and every connection has to do it. A SQLite table with foreign keys declared and never enforced is the norm, not the exception, and orphan rows are the usual discovery. MySQL's older MyISAM engine has the same property; InnoDB enforces. DuckDB enforces primary and unique keys and checks, and is built for analysis rather than for a thousand concurrent writers, which is the right trade for what it is.

Constraints are cheap to add and expensive to lack

A constraint costs a comparison per write. A missing constraint costs the query that finds the bad rows a year later, the decision about what to do with them, and every report that ran in between. When in doubt, add it. When the data already violates it, Postgres refuses to add it, which is itself the list of rows to fix, and `NOT VALID` lets you add it for future rows while you work through the old ones.

BEFORE YOU MOVE ON

A team wants to guarantee that no student has two payments recorded on the same date. They write `CHECK (count(*) = 1)` on the payments table and get an error. What is the right tool and why does the CHECK fail?

A. A trigger is the only option, because constraints can never involve more than one column and this rule involves two.

B. `UNIQUE (student_id, paid_on)`, because a CHECK evaluates one row at a time and cannot count other rows.

C. `CHECK (student_id IS DISTINCT FROM paid_on)`, since the rule is that the two columns must not coincide within the table.

D. An `EXCLUDE` constraint on the `paid_on` range, because two payments on the same date have overlapping day-long intervals and EXCLUDE is built for overlaps.

57 · 8 min

INSERT, and the INSERT you can run twice

THE ONE THING TO KEEP

An INSERT that must be safe to re-run needs a unique key and an ON CONFLICT clause, because the database detects the duplicate by inserting into the unique index, and without that index it cannot detect one at all.

The nightly job that ran twice

A script inserts the day's payments from the provider's file. It crashes halfway, somebody re-runs it, and now 3,000 payments exist twice. Nothing in `INSERT` prevents that; it does what it says. The property you want is that running the job again produces the same table as running it once, which has a name, idempotence, and a specific way to get it.

The shapes of INSERT

```
INSERT INTO payments (reference, student_id, amount, paid_on,
method)
VALUES ('P-10441', 812, 4500.00, '2026-03-14', 'upi');

INSERT INTO payments (reference, student_id, amount, paid_on,
method)
VALUES ('P-10442', 813, 4500.00, '2026-03-14', 'upi'),
       ('P-10443', 814, 3000.00, '2026-03-14', 'cash');

INSERT INTO payments (reference, student_id, amount, paid_on,
method)
SELECT reference, student_id, amount, paid_on, method
FROM payments_clean
WHERE loaded_at > '2026-03-14';
```

Name the columns every time. `INSERT INTO payments VALUES (...)` with no column list depends on the order columns were created in, and breaks the

day someone adds one. The multi-row form is far faster than one statement per row, because each statement is a round trip to the server; the `INSERT ... SELECT` form moves rows from a staging table into a real one and is how the previous module's clean view becomes a table. `RETURNING id` at the end of any of them gives back the generated keys, which is how an application learns the id of the row it just created without a second query.

Making it safe to repeat

The reference is unique, so the second run fails on the first duplicate with a constraint error, and loads nothing. That is safe but unhelpful. `ON CONFLICT` says what to do instead:

```
INSERT INTO payments (reference, student_id, amount, paid_on,
method)
SELECT ... FROM payments_clean
ON CONFLICT (reference) DO NOTHING;
```

Rows whose reference already exists are skipped; the rest go in. Run it ten times and the table is the same as after once. Or, when the new row should replace the old:

```
ON CONFLICT (reference) DO UPDATE
  SET amount = EXCLUDED.amount,
      method = EXCLUDED.method;
```

`EXCLUDED` is the row that would have been inserted. This is the upsert, insert-or-update in one statement, and it is how a daily feed of customer records is kept current without first checking whether each one exists.

What it needs underneath

`ON CONFLICT (reference)` works only because there is a unique index on `reference`. The database detects the conflict by trying to insert into that index. Without one you get `there is no unique or exclusion constraint matching the ON CONFLICT specification`, and the fix is the constraint, not

a different statement. This is also why an upsert on a column that is *usually* unique but not declared so is not possible: the database will not guess.

SQLite uses the same `ON CONFLICT` syntax. MySQL says `INSERT ... ON DUPLICATE KEY UPDATE amount = VALUES(amount)`. SQL Server and Oracle, and Postgres from version 15, have `MERGE`, which can also delete and is more general and more verbose. All of them rest on a unique key.

Two rows in the same statement with the same key

An `INSERT ... SELECT ... ON CONFLICT` whose source contains the same reference twice fails with `ON CONFLICT DO UPDATE command cannot affect row a second time`. The source has to be deduplicated first, with the `row_number` pattern from the previous module. The error is doing you a favour: it has found a duplicate in the file before it found its way in.

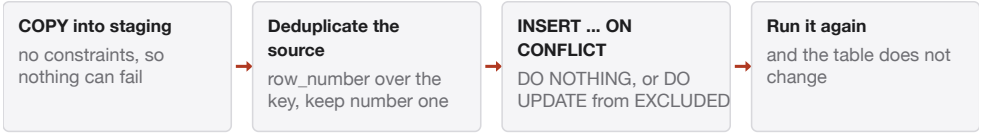
Bulk loads

For a million rows, `COPY` is ten to a hundred times faster than any form of `INSERT`, because it streams the data in one operation instead of parsing a statement per batch. The pattern for a big idempotent load is `COPY` into a staging table with no constraints, then one `INSERT ... SELECT ... ON CONFLICT` from staging into the real table. The constraints are checked once, at the boundary between the two.

The habit

Ask of every job that writes: what happens if this runs twice? If the answer is "duplicates", the job needs a unique key and an `ON CONFLICT`. If the answer is "an error", the job is safe but will page somebody. If the answer is "the same table either way", the job is finished.

A load that produces the same table whether it runs once or ten times



ON CONFLICT works only because a unique index exists to detect the conflict in; without one the database cannot guess which column made the row a duplicate. Deduplicating the source first is not optional either — the same key twice in one statement is an error, and a useful one.

BEFORE YOU MOVE ON

A nightly load uses `INSERT ... SELECT ... ON CONFLICT (email) DO UPDATE` into a customers table and fails with "there is no unique or exclusion constraint matching the ON CONFLICT specification". Emails are in fact unique in the data. What is the fix?

A. Change `DO UPDATE` to `DO NOTHING`, since DO UPDATE requires a primary key and DO NOTHING works on any column.

B. Deduplicate the source with `DISTINCT` first, because the error means the incoming rows contain the same email twice.

C. Add `UNIQUE (email)` to the table, because the conflict is detected by the unique index and the data being unique in practice is not enough.

D. Replace `ON CONFLICT` with a `WHERE NOT EXISTS` subquery, since Postgres can only detect conflicts on the primary key column and email is not the key.

58 · 8 min

UPDATE and DELETE: rehearse it as a SELECT

THE ONE THING TO KEEP

Run the WHERE clause as a SELECT, read the rows and the count, then change the first word; for UPDATE FROM, also prove that no target row matches more than once, because the database will pick one silently.

The statement that changes every row

```
UPDATE orders SET status = 'cancelled';
```

There is no WHERE clause. Every order in the table is now cancelled. The database reports `UPDATE 31940` and waits for the next command. It has done exactly what it was told, and what it was told was a typo: a WHERE clause on the next line that was never sent, or a query pasted without its last line.

Every person who works with databases has a version of this story, and the ones who only have one version learned the habit in this lesson.

Rehearse

Before any UPDATE or DELETE, run the same WHERE clause as a SELECT, and look at what comes back:

```
SELECT count(*), min(placed_at), max(placed_at)
FROM orders
WHERE status = 'pendng';
-- 3 rows, all from last Tuesday
```

Three rows, last Tuesday. That is what you expected: a typo in the data from one bad import. Now, and only now, convert the SELECT into the write:

```
UPDATE orders SET status = 'pending' WHERE status = 'pendng';
-- UPDATE 3
```

The row count after the write must match the count from the rehearsal. If it says `UPDATE 40000`, stop, because something between the rehearsal and the write changed, and the transaction lesson next explains how to take it back.

`RETURNING` shows what was changed, which is the rehearsal's other half:

```
DELETE FROM orders WHERE status = 'test' RETURNING id, cus-
tomer_id, amount;
```

Read the returned rows. If one of them is a real customer, you know before the transaction is committed.

UPDATE with a join

Correcting one table from another is `UPDATE ... FROM`:

```
UPDATE orders o
SET    city = c.city
FROM   customers c
WHERE  c.id = o.customer_id
  AND  o.city IS NULL;
```

Each order with a missing city takes it from its customer. This has the join hazard the whole course has been about. If the FROM matches more than one row per order, Postgres updates the order with one of them and does not say which, and MySQL and SQL Server have their own versions of the same silence. The rehearsal for an UPDATE FROM is therefore a count of matches per target row:

```
SELECT o.id, count(*) FROM orders o JOIN customers c ON c.id =
o.customer_id
GROUP BY o.id HAVING count(*) > 1;
```

It must return nothing. `DELETE ... USING` is the same shape for deletes, with the same rehearsal.

Deleting in batches

`DELETE FROM events WHERE created_at < '2024-01-01'` on a table with 200 million old rows will run for an hour, hold locks the whole time, and generate a transaction log the size of the table. Delete in slices instead:

```
DELETE FROM events
WHERE id IN (SELECT id FROM events WHERE created_at < '2024-01-
01' LIMIT 10000);
```

in a loop until the count is zero. Each slice commits quickly, other users get a turn between slices, and a mistake discovered after the third slice has cost thirty thousand rows rather than two hundred million. The partitioning lesson in the performance module shows the version that costs nothing: dropping a whole partition of old rows in one metadata operation.

Tools that help, and one that does not

DBeaver and pgAdmin, both free, can be set to ask for confirmation before running an UPDATE or DELETE without a WHERE. Turn that on. `psql` has no such setting; it is the tool that trusts you, and it is the one usually connected to production. Some teams alias `psql` to open with `--set AUTOCOMMIT=off`, so that every write is inside a transaction until you say `COMMIT`, which is the subject of the next lesson and the strongest protection available.

The habit

Never type UPDATE or DELETE first. Type SELECT, read the rows, then edit the first word. The count in the rehearsal and the count in the write are the same number, or the write does not get committed.

BEFORE YOU MOVE ON

`UPDATE orders o SET city = c.city FROM customer_addresses c WHERE c.customer_id = o.customer_id` runs without error and reports the expected row count, but some orders now show a city the customer moved away from years ago. What happened?

A. UPDATE FROM applies rows in reverse chronological order, so the oldest address is always the last one written and wins.

B. Customers have several addresses, so each order matched more than one row and the database used one of them without saying which.

C. The WHERE clause is missing an `AND o.city IS NULL`, so orders that already had the right city were overwritten with the customer's current one.

D. `customer_addresses` was joined without an index, and the sequential scan returned rows in physical order, which favours older addresses.

59 · 9 min

Transactions: making a mistake reversible

THE ONE THING TO KEEP

A transaction makes several writes one unit and makes any of them reversible until COMMIT; know whether your tool autocommits, and remember that in Postgres one failed statement aborts the whole block until ROLLBACK.

All of it, or none of it

A fee payment is recorded as two writes: insert a row into `payments`, and reduce `balance_due` on the student's row. If the first succeeds and the second fails, the student has paid and still owes. If the second succeeds and the first fails, the debt is gone and there is no record of the money. Neither half is acceptable alone.

```
BEGIN;
INSERT INTO payments (reference, student_id, amount, paid_on,
method)
VALUES ('P-10500', 812, 4500, '2026-03-14', 'upi');
UPDATE students SET balance_due = balance_due - 4500 WHERE id =
812;
COMMIT;
```

Between `BEGIN` and `COMMIT` the two writes are one unit. If the connection drops after the INSERT, the database throws the INSERT away when it recovers. If the UPDATE fails a constraint, nothing from the INSERT survives. Other users see neither change until the COMMIT, and then they see both. That is what a transaction is: a boundary inside which partial results do not exist.

The undo button

The same mechanism is the safety net for the previous lesson:

```
BEGIN;
UPDATE orders SET status = 'pending' WHERE status = 'pendng';
-- UPDATE 40000   <- not the 3 you rehearsed
ROLLBACK;
```

`ROLLBACK` discards everything since `BEGIN`. The 40,000 rows are as they were. Nothing happened, as far as anybody else can tell. This is why the habit is to open a transaction before any write on a database that matters: the cost is one word, and it turns a catastrophe into a `ROLLBACK`.

Autocommit, and which mode you are in

When you do not write `BEGIN`, most tools wrap each statement in its own transaction and commit it immediately. That is autocommit, and it means a bare `UPDATE` cannot be rolled back, because it was committed before the prompt returned. `psql` is in autocommit by default. DBeaver can be switched to manual commit, where nothing is saved until you press the commit button, and shows a counter of uncommitted statements. Application libraries vary: Python's `psycopg` opens a transaction implicitly and needs `commit()` called; some others autocommit. Before writing anything on a new tool, find out which mode it is in, because the two modes produce opposite mistakes: work that vanished because nobody committed, and work that cannot be undone because everything did.

The error that aborts the whole thing

In Postgres, an error inside a transaction puts the transaction into an aborted state. Every subsequent statement fails with `current transaction is aborted, commands ignored until end of transaction block`, even a harmless `SELECT 1`, until you `ROLLBACK`. People meet this in a script that continues after a failed statement and cannot understand why everything after it is failing too. The behaviour is deliberate: a transaction with one failed step is not in a state anyone can reason about, so Postgres refuses to let you commit it.

`SAVEPOINT` gives a partial undo. After `SAVEPOINT before_risky;`, a failure can be answered with `ROLLBACK TO before_risky;`, which discards only the work since the savepoint and lets the transaction continue. Batch loaders use it to skip a bad row and keep going.

What a transaction costs

While it is open, its locks are held. An `UPDATE` inside a transaction that then waits for a human to read the result and type `COMMIT` holds a lock on those rows for as long as the human takes, and every other writer that needs them waits. A transaction left open over lunch is a common cause of a queue of stuck requests, and the performance module's lesson on waiting shows how to find one. Keep transactions short: rehearse outside the transaction, then open it, write, check the count, commit.

Schema changes inside a transaction

Postgres treats `CREATE TABLE`, `ALTER TABLE` and `DROP TABLE` as transactional. A migration that creates two tables and fails on the third leaves nothing behind, and can be tested with a `ROLLBACK` at the end. MySQL commits implicitly before and after every schema statement, so a failed migration there leaves whatever it got through, and has to be cleaned up by hand. Oracle and SQL Server sit between. It is one of the larger practical differences between engines and it decides how you write a migration.

The habit

Open a transaction before a write you have not done a hundred times. Check the row count. Commit or roll back. If a tool commits for you, know that it does before you press enter.

BEFORE YOU MOVE ON

A Postgres script runs `BEGIN`, an INSERT that fails on a constraint, and then several SELECT statements that each fail with "current transaction is aborted, commands ignored until end of transaction block". What is going on?

A. The failed INSERT locked the table, and the SELECTs are being refused because they cannot obtain a read lock while the failed write holds it.

B. The SELECTs reference the row the INSERT tried to create, which does not exist, so each one errors on the missing row.

C. Postgres requires a COMMIT after every failed statement to clear the error before continuing.

D. The error put the transaction into an aborted state, and Postgres refuses every further statement until a ROLLBACK ends the block.

60 · 9 min

Two people, one row: lost updates and how the database prevents them

THE ONE THING TO KEEP

A read followed by a write is a gap another transaction can enter; close it with an atomic UPDATE that reads and writes in one statement, a FOR UPDATE lock, or a version check that detects the collision and retries.

The counter that goes wrong under load

A course has 1 seat left. Two people click "enrol" at the same moment. Each request runs the same code: read `seats_left`, see 1, subtract one, write 0. Both read 1. Both write 0. Two students are enrolled in one seat, and the table says nothing is wrong.

This is a lost update, and it is the fundamental problem of concurrent writes: a read and a write that are two statements can have another transaction slip be-

tween them. Everything in this lesson is a way of closing that gap.

Make the read and the write one statement

```
UPDATE courses
SET    seats_left = seats_left - 1
WHERE  id = 42 AND seats_left > 0
RETURNING seats_left;
```

The database reads and writes the row under a lock it takes for the duration of the statement, so the second request cannot read between the first's read and write. It sees the row after the first request's write, `seats_left` is 0, the WHERE clause fails, and it updates nothing. `RETURNING` tells the application whether it won: a row means enrolled, no row means full. This is the first thing to reach for, and it handles most counters, balances and stock levels with no further machinery.

Lock the row for a read-then-write

When the logic between the read and the write is too complicated for one statement, lock the row explicitly:

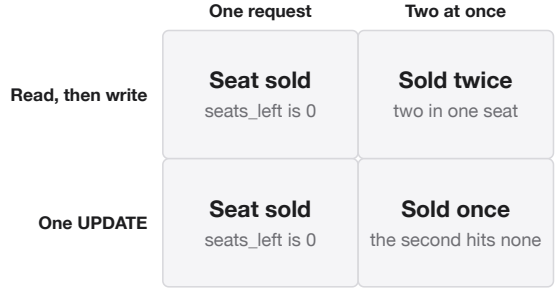
```
BEGIN;
SELECT seats_left FROM courses WHERE id = 42 FOR UPDATE;
-- application decides what to do
UPDATE courses SET seats_left = seats_left - 1 WHERE id = 42;
COMMIT;
```

`FOR UPDATE` makes the second transaction's `SELECT ... FOR UPDATE` wait until the first commits, and then it sees the updated value. The gap is closed by making the second reader queue. The cost is that the row is locked for the length of the transaction, which is why the previous lesson said to keep transactions short: a `FOR UPDATE` held while a human thinks is a course nobody can enrol in.

Or detect the collision instead of preventing it

Optimistic concurrency skips the lock and checks at write time whether anyone else got there first, using a version number:

One seat left, and the gap between a read and a write



A read and a write in two statements leave a gap another transaction can enter. One UPDATE that reads and writes in the same statement closes it: the second request sees the row after the first has written, the WHERE fails, no row is returned, and the application learns it lost.

```
UPDATE courses
SET    seats_left = 0, version = version + 1
WHERE  id = 42 AND version = 7;
```

The application read version 7. If another transaction has already updated the row, the version is 8, the WHERE fails, zero rows are updated, and the application re-reads and tries again. No lock is held between the read and the write, so it scales to many readers, at the cost of retries when collisions are common. Web applications use this pattern constantly, and the `version` column, or an `updated_at` timestamp used the same way, is the tell.

What a transaction can see

Postgres's default isolation level is Read Committed: each statement sees the data as committed at the moment the statement started. Two SELECTs in one transaction can therefore see different data if someone committed in between. `REPEATABLE READ` freezes the view for the whole transaction, and `SERIALIZABLE` goes further and refuses to commit any transaction whose result could not have come from running the transactions one after another. The refusal appears as an error, `could not serialize access`, and the applica-

tion must retry. Stricter levels trade retries for correctness; the atomic-update pattern above is correct at every level, which is another reason to prefer it.

Deadlock

Transaction A locks row 1 and then wants row 2. Transaction B locks row 2 and then wants row 1. Each waits for the other forever. Postgres notices within a second, kills one of them with `deadlock detected`, and the other proceeds. The killed one must retry. The prevention is to lock rows in a consistent order, for instance by id ascending, so that two transactions wanting the same rows always queue rather than cross. A transfer between two accounts that always locks the lower id first cannot deadlock with another transfer.

The engines differ, again

SQLite allows one writer at a time for the whole database file. A second writer gets `database is locked` and must wait or retry, which is fine for a phone app and is the reason SQLite is not the choice for a busy web service. DuckDB is similar, and built for a single analyst rather than a crowd. Postgres and MySQL lock rows, not files, so thousands of writers touching different rows proceed together, and the patterns in this lesson only matter when they touch the same one.

BEFORE YOU MOVE ON

Two requests each run `SELECT balance FROM accounts WHERE id = 7` (both see 500), then `UPDATE accounts SET balance = 400 WHERE id = 7`, intending to withdraw 100 each. The final balance is 400. Which change makes the result correct?

A. Wrap each request's two statements in `BEGIN ... COMMIT`, because a transaction guarantees the other request cannot run until it finishes.

B. Write `UPDATE accounts SET balance = balance - 100 WHERE id = 7`, so the read and the write happen in one statement under the row lock.

C. Add an index on `accounts.id`, since the second request only overtook the first because the row lookup was slow.

D. Change the isolation level to REPEATABLE READ, which makes the second UPDATE see the first one's result and subtract from 400.

61 · 9 min

Changing a table that is in use

THE ONE THING TO KEEP

Add nullable, backfill in batches, then constrain, because adding NOT NULL or an expression default rewrites or scans a live table under the strongest lock; set a lock timeout so a queued ALTER cannot take the site down behind one long report.

The schema is not finished, and the table is busy

The `orders` table has forty million rows, a web application reading from it every second, and it needs a new column. `ALTER TABLE orders ADD COLUMN channel text;` looks like nothing. Whether it is nothing depends on what the database has to do to every existing row, and on who is holding a lock at the moment you run it.

What is instant and what rewrites the table

Adding a nullable column with no default is a metadata change in every major engine: the catalogue is updated, existing rows are untouched and read as NULL for the new column, and it completes in milliseconds regardless of table size.

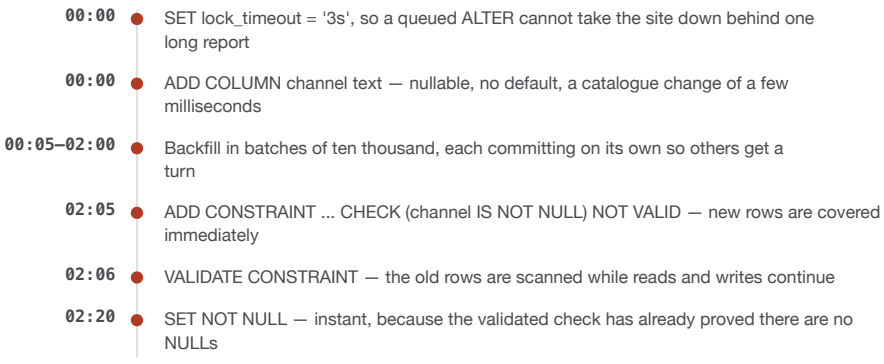
Adding a column with a constant `DEFAULT` used to rewrite every row in Postgres, minutes to hours on a big table. Since Postgres 11 a constant default is also metadata only, with the default applied on read for old rows. A default that is an expression, such as `DEFAULT now()`, still rewrites. Changing a column's type rewrites unless the new type is a strict widening, `varchar(50)` to `text` for instance. Adding `NOT NULL` to an existing column scans the whole table to prove that no NULL exists, and holds a lock while it does.

The pattern for a new required column on a live table is therefore three steps rather than one:

```
ALTER TABLE orders ADD COLUMN channel text;             -- in-
stant, nullable
-- backfill in batches of 10,000 with UPDATE ... WHERE channel
IS NULL LIMIT-style loops
ALTER TABLE orders ADD CONSTRAINT orders_channel_nn CHECK
(channel IS NOT NULL) NOT VALID;
ALTER TABLE orders VALIDATE CONSTRAINT orders_channel_nn;   --
scans without the heavy lock
ALTER TABLE orders ALTER COLUMN channel SET NOT NULL;       --
instant, since the check proves it
```

`NOT VALID` adds the rule for new rows immediately; `VALIDATE` checks old rows while allowing reads and writes to continue. Postgres 12 and later recognises the validated check and sets `NOT NULL` without rescanning.

Adding a required column to a table that is read every second



The danger is not the ALTER, which takes milliseconds. It is the queue behind it: a long report holds a share lock, the ALTER waits, and every new query waits behind the ALTER. The site looks down. A lock timeout means the change fails instead, and you try again when the report has finished.

The lock queue

`ALTER TABLE` takes the strongest lock there is, `ACCESS EXCLUSIVE`, which waits for every current reader and writer to finish, and blocks every new one until it is done. On a busy table the danger is not the ALTER itself but the queue behind it: a long-running report holds a share lock, the ALTER waits behind it, and every ordinary query that arrives waits behind the ALTER. The site appears to be down. It is not; it is queued behind a metadata change that would have taken two milliseconds if it had been allowed to start.

Set a lock timeout before any schema change on a live database: `SET lock_timeout = '3s';`. If the ALTER cannot get its lock in three seconds, it fails, nothing is queued, and you try again when the report has finished. Creating an index has its own version of the problem and its own fix, `CREATE INDEX CONCURRENTLY`, which builds the index without blocking writes, at the cost of taking longer and needing to be run outside a transaction.

Rename nothing that is in use

Renaming a column breaks every query, view and application that uses the old name at the instant the rename commits. The safe sequence is additive: add the new column, have the application write both, backfill, switch reads to the new column, stop writing the old one, and drop it in a later release. It is more steps and it means no moment at which anything is broken. The same applies

to dropping a column: first confirm nothing reads it, which a quick grep of the code and a look at the views that depend on it will tell you.

Migrations are files, not commands

A schema change typed into a GUI against production is a change nobody can review, repeat on the test database, or reverse. The discipline is a numbered file per change, `0042_add_orders_channel.sql`, applied in order, with a table in the database recording which have run. The free tools that do exactly this are Flyway, Liquibase, dbmate and sqitch, and most application frameworks ship their own; the earlier lesson on Postgres transactions is why they can apply a multi-statement migration atomically there and not in MySQL. The file is also the documentation: six months later, the reason a column exists is the commit message on the migration that added it.

The habit

Before a schema change, answer three questions in writing: does this rewrite the table, what lock does it take, and what breaks if the old name disappears. Then set a lock timeout, and run it from a migration file that has already run on a copy.

BEFORE YOU MOVE ON

A team runs `ALTER TABLE orders ADD COLUMN channel text` on a busy production table during the day. The statement itself should take milliseconds, yet the site stops responding for four minutes. What is the most likely mechanism?

A. Adding a text column rewrites every row to make space for it, and forty million rows take four minutes.
B. The ALTER waited behind a long-running query for its exclusive lock, and every new query queued behind the ALTER until that query finished.
C. The application's connection pool refreshed its cached column list and reconnected every client at once, overwhelming the server for the duration of the reconnect storm.
D. Postgres automatically rebuilt every index on the table to include the new column, holding a lock throughout.

62 · 8 min

Views: a saved question, not a saved answer

THE ONE THING TO KEEP

A view stores a question and re-runs it on every reference, so it is always current and never faster than its query; a materialised view stores the answer, is as stale as its last refresh, and must say so.

One definition, used everywhere

Three reports each define "active customer" in their own WHERE clause, and the three definitions have drifted. The fix is not to align the three clauses; it is to have one, in one place, that the three reports read from:

```
CREATE VIEW active_customers AS
SELECT c.*
FROM customers c
WHERE c.deleted_at IS NULL
  AND EXISTS (SELECT 1 FROM orders o
              WHERE o.customer_id = c.id
                AND o.placed_at >= current_date - interval '90
days');
```

`SELECT count(*) FROM active_customers` now means the same thing in every report, and when the business changes the definition to 60 days, it changes in one line and every report follows.

What a view is, mechanically

A view stores the query, not its result. Every time it is referenced, the engine substitutes the view's definition into the calling query and plans the whole thing together, which means a view is exactly as fast or slow as the query it wraps, and it always reflects the current data. Filters applied to the view are pushed inside it where the planner can: `SELECT * FROM active_customers WHERE city = 'Pune'` uses an index on city if one exists, because after substitution it is a plain query over `customers`.

That last point has a limit. A view containing `DISTINCT`, an aggregate, or a window function is a barrier that outer filters cannot always be pushed through, and a view over a view over a view can produce a plan nobody can read. Views are for naming, not for stacking.

Views as an interface

A view can hide columns: `CREATE VIEW customers_public AS SELECT id, display_name, city FROM customers` leaves out phone, email and date of birth, and an analyst role granted access to the view and not the table cannot reach them. It can present a stable shape over tables that change: when `orders` is split into two tables next year, a view named `orders` with the old columns keeps every existing report working. And it can rename the awkward: a

view over a vendor's `tbl_cust_mstr` called `customers` is a small kindness to everyone who comes after.

Materialised views: when the answer is worth storing

A view whose query takes forty seconds is a report that takes forty seconds every time somebody opens it. A materialised view runs the query once and stores the rows:

```
CREATE MATERIALIZED VIEW monthly_revenue AS
SELECT date_trunc('month', placed_at) AS month, city,
sum(amount) AS revenue
FROM orders GROUP BY 1, 2;

REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_revenue;
```

Reads are now instant, and the data is as old as the last refresh. That staleness is a property, not a defect, and it belongs in the report's title: "as of 06:00 today". `CONCURRENTLY` lets reads continue during the refresh and requires a unique index on the materialised view; without it the refresh locks readers out for its duration. Schedule the refresh, and put its timestamp somewhere a query can read, because "the dashboard is wrong" and "the dashboard is six hours old" are different conversations.

DuckDB and SQLite have views and not materialised ones; the equivalent is a table rebuilt by a scheduled `CREATE TABLE ... AS`. BigQuery has materialised views that refresh themselves within limits.

Writing through a view

A simple view over one table, with no aggregate or join, accepts `INSERT`, `UPDATE` and `DELETE` and passes them to the table. `WITH CHECK OPTION` refuses a write through the view that would produce a row the view cannot see, so that an insert through `active_customers` cannot create a deleted customer. Anything more complicated needs a trigger, and at that point a view is the wrong tool.

Dependencies

`ALTER TABLE customers DROP COLUMN city` fails if a view uses `city`, with an error naming the view. That is a feature: the database knows what depends on what and will not let a schema change silently break a report. `DROP ... CASCADE` overrides it and drops the views too, which is the option to read twice before typing.

The habit

When a definition matters to more than one query, make it a view. When a view is slow and its data can be minutes old, make it materialised and print the refresh time. When a view has a view inside it, ask whether one of them should be a table.

BEFORE YOU MOVE ON

A dashboard reads from a materialised view refreshed nightly at 02:00. At 15:00 a manager sees a revenue figure that does not include a large order placed at 10:00. Which statement is true?

A. The view's query has a bug that excludes large orders, since a view always reflects the current table contents.

B. The order was inserted inside an uncommitted transaction and is invisible to every reader, including the dashboard.

C. The figure is correct as of 02:00, and the dashboard should show its refresh time so the reader knows what "as of" applies.

D. `REFRESH MATERIALIZED VIEW CONCURRENTLY` only picks up rows inserted since the previous refresh, and the order will appear after two more refreshes.

63 · 9 min

JSON in a column: when it helps and what it costs

THE ONE THING TO KEEP

A JSON column trades every constraint for flexibility, so keys can be misspelt and types can drift on every insert; keep the document, promote the keys you query into generated columns with real types, and profile the keys before trusting them.

Attributes that refuse to be columns

A shop sells shirts, chargers and books. A shirt has a size and a colour; a charger has a wattage and a plug type; a book has an author and a page count. A `products` table with a column for every attribute of every kind of product is mostly NULL and grows a column every time a new kind arrives. A `product_attributes` table with one row per product per attribute works and makes every query a pivot. The third option is a column that holds structured data:

```
CREATE TABLE products (
  id     bigint PRIMARY KEY,
  name   text NOT NULL,
  price  numeric(10,2) NOT NULL,
  attrs  jsonb NOT NULL DEFAULT '{}'
);

INSERT INTO products VALUES
  (1, 'Linen shirt', 1800, '{"size": "M", "colour": "white"}'),
  (2, 'USB-C charger', 1200, '{"watts": 65, "plug": "type-
D"}');
```

Postgres's `jsonb` stores the document in a parsed binary form and lets queries reach inside it: `attrs ->> 'colour'` returns the text `white`, `attrs -> 'watts'` returns the JSON value, and `attrs @> '{"plug": "type-D"}'`

asks whether the document contains that fragment. MySQL has a `JSON` type with `->>` and `JSON_EXTRACT`; SQLite has `json_extract(attrs, '$.colour')`; DuckDB reads JSON natively and can unnest it. BigQuery has a `JSON` type too.

The other honest use is keeping what an API sent, whole. A `raw_payload jsonb` column beside the parsed columns means that when the parser turns out to have missed a field, the field is still there.

What you gave up

Every promise the first lesson of this module made, a JSON column breaks.

Nothing checks the keys. Half the shirts say `"colour"` and half say `"color"`, because two developers wrote two forms, and a query for `attrs ->> 'colour'` finds half the shirts. A table column cannot be misspelt on insert; a JSON key can, on every insert.

Nothing checks the types. `"watts": 65` in one row and `"watts": "65W"` in the next. A sum over `(attrs ->> 'watts')::int` fails on the second row, or, if the cast is guarded, silently drops it.

Nothing checks presence. A shirt with no size is a valid document.

The schema has moved into every query. With columns, the database knows the shape of a product and every tool can display it. With JSON, the shape lives in the heads of the people who wrote the inserts, and the first task of anyone new is to discover it:

```
SELECT key, count(*), count(DISTINCT jsonb_typeof(attrs ->
key))
FROM products, jsonb_object_keys(attrs) AS key
GROUP BY key ORDER BY count(*) DESC;
```

That query lists every key in use, how often, and how many different types it has been given. It is the profiling step from the cleaning module, applied to a column that was supposed to be structured.

Keep the document, promote what you query

The pattern that gets the flexibility without the chaos is to keep the JSON and pull the keys you actually filter and join on into real columns, with real types and real constraints:

```
ALTER TABLE products
  ADD COLUMN colour text GENERATED ALWAYS AS (attrs ->>
'colour') STORED,
  ADD COLUMN watts  int  GENERATED ALWAYS AS ((attrs ->>
'watts')::int) STORED;
CREATE INDEX ON products (colour);
```

A generated column is computed from the document on every write, can be indexed like any other column, and fails the insert if the cast fails, which is the type check the JSON did not have. Queries use `colour`; the document remains for the attributes nobody has needed to promote yet. When a key is queried often enough to matter, it is promoted; the JSON column is where attributes wait to be found important.

Indexing the whole document

Postgres can index a `jsonb` column with a GIN index, after which `attrs @> '{"plug": "type-D"}'` uses the index rather than scanning every document. It is the right tool for "find documents containing this fragment" across unknown keys, and it is large and slow to update, which the performance module's lesson on index types weighs. For a known key, the generated column and its ordinary index are smaller and faster.

The habit

Use JSON for what varies and for what you want to keep whole. The moment a key appears in a WHERE clause, a JOIN or a GROUP BY, give it a column. And run the key-profiling query on any JSON column you inherit, before believing anything about what it contains.

BEFORE YOU MOVE ON

A `products.attrs` JSON column has been written by two services for a year. A query for `attrs ->> 'colour' = 'white'` returns about half the white shirts the merchandising team expects. What is the most likely cause, and how is it found?

A. The jsonb type lower-cases values on storage, so `White` no longer matches; comparing with `ilike` fixes it.

B. Half the rows store the document as text rather than jsonb, and the `->>` operator returns NULL for those rows; casting the column to jsonb fixes it.

C. One service writes the key as `color`, and a query listing the keys in use with `jsonb_object_keys` and their counts will show both spellings.

D. The GIN index on `attrs` is stale and returns only rows indexed before the second service started writing; reindexing fixes it.

64 · 9 min

Keeping history: soft deletes, audit rows and who changed what

THE ONE THING TO KEEP

An UPDATE destroys the old value and a DELETE destroys the row, so decide up front whether a table needs a deleted_at flag behind a filtering view, an audit table filled by a trigger, or an effective-dated design for as-of questions.

An UPDATE destroys the old value

A customer's address changes. `UPDATE customers SET address = ... WHERE id = 812` writes the new one, and the old one is gone. Next month someone asks where the March invoice was sent, and the table cannot say. A `DELETE` is the same one step further: the row is gone, and with it the fact that it ever existed.

Most businesses need some of the past. This lesson is the three ways to keep it, and what each costs.

Soft delete: the row stays, marked

```
ALTER TABLE customers ADD COLUMN deleted_at timestamptz;
UPDATE customers SET deleted_at = now() WHERE id = 812;   --
instead of DELETE
```

The row remains, with a timestamp saying when it was "deleted". Every query that should not see deleted customers must now say `WHERE deleted_at IS NULL`, and every query that forgets will count them. This is the containment problem in miniature: the filter has to be everywhere, and the safe way to make it everywhere is a view, `customers_live`, that has the filter built in, with the raw table read only by the code that needs the deleted rows.

Two things break quietly. A `UNIQUE (email)` constraint now prevents a deleted customer's email from ever being reused; the fix is a partial unique index, `CREATE UNIQUE INDEX ON customers (email) WHERE deleted_at IS NULL`, which enforces uniqueness only among the living. And a foreign key from `orders` still points at the soft-deleted customer, which is what you want for an order that happened and is confusing for a report that joins to it without the filter.

An audit table: what changed, when, by whom

Soft delete keeps the row but not its earlier versions. An audit table keeps every version:

```
CREATE TABLE customers_audit (
  audit_id    bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  customer_id bigint NOT NULL,
  op          text NOT NULL CHECK (op IN ('I', 'U', 'D')),
  changed_at  timestamptz NOT NULL DEFAULT now(),
  changed_by  text NOT NULL DEFAULT current_user,
  old_row     jsonb,
  new_row     jsonb
);

CREATE OR REPLACE FUNCTION customers_audit_fn() RETURNS trigger
AS $$
BEGIN
  INSERT INTO customers_audit (customer_id, op, old_row,
new_row)
  VALUES (coalesce(NEW.id, OLD.id), left(TG_OP, 1),
to_jsonb(OLD), to_jsonb(NEW));
  RETURN NULL;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER customers_audit AFTER INSERT OR UPDATE OR DELETE
ON customers
FOR EACH ROW EXECUTE FUNCTION customers_audit_fn();
```

A trigger is code the database runs on every write to the table. This one copies the row before and after each change into the audit table as JSON, with the operation, the time and the database user. Nothing in the application has to remember to do it, which is the point: an audit that depends on every code path calling it is an audit with gaps. SQLite and MySQL have triggers with their own syntax; the previous lesson's JSON column is what makes storing "the whole row, whatever its columns" straightforward.

The audit table answers "who changed this and what did it say before". It does not answer "what was true on 14 March" conveniently, because that means finding, for each customer, the latest audit row before that date, which is the as-of pattern from the time module. If as-of questions are the common ones, the effective-dated table from that module is the better shape, with `valid_from` and `valid_to` on each version. Many systems keep both: the effective-dated table for queries, the audit log for accountability.

The minimum, on every table

`created_at` and `updated_at`, both `timestamptz NOT NULL DEFAULT now()`, with a trigger or application rule that sets `updated_at` on every change. They cost nothing and they answer half of all questions about history: how old is this, when did it last change, what changed today. A table without them is a table where "when" cannot be asked.

What history costs

An audit table grows faster than the table it watches, because every update adds a row. On a table updated a hundred times a second it becomes the largest table in the database within months, and needs its own retention rule, usually "keep two years, then archive to files". Retention is also where the law enters: some records must be kept for a fixed period, some must be deleted on request, and the two requirements can apply to the same row. The moderation-and-legal material elsewhere on this site covers the legal-hold case; the database design point is that a soft delete and a legal hold are different flags, and merging them makes the audit log lie about which one applied.

The habit

Before the first `UPDATE` or `DELETE` on a table, decide which of the three it needs: nothing, a `deleted_at`, or a full audit trail. The decision is much cheaper to make before the history you wanted has already been overwritten.

BEFORE YOU MOVE ON

After adding a `deleted_at` column and switching to soft deletes, a customer who closed their account tries to sign up again with the same email and is refused because the address "already exists". What is the correct fix?

A. Hard-delete the old row after all, because a unique constraint and soft deletes are fundamentally incompatible.

B. Drop the unique constraint on email, since the application layer can check for duplicates among live customers.

C. Append the deletion timestamp to the old row's email so the address becomes free again, and strip it back off if the account is ever restored.

D. Replace the constraint with a partial unique index on email `WHERE deleted_at IS NULL`, so uniqueness is enforced only among live rows.

65 · 8 min

Who may do what: roles, GRANT and a read-only door

THE ONE THING TO KEEP

Create a role for each kind of access with the least it needs, remember ALTER DEFAULT PRIVILEGES for future tables, and give an analyst or an AI a read-only role with a statement timeout and views that hide personal columns.

Everyone connects as the owner, until the day that matters

A small project starts with one database user, the one that created the tables, and every tool, script and person uses it. It works until an analyst's exploratory query includes a `DELETE` by mistake, or an AI assistant generating SQL decides that the cleanest way to answer a question is to create a temporary table, or a laptop with the password on it is lost. Permissions are the difference between "that could not have happened" and "we are restoring from backup".

Roles and grants

A role is a named set of permissions, and a user is a role that can log in:

```
CREATE ROLE analyst LOGIN PASSWORD '...';
GRANT CONNECT ON DATABASE school TO analyst;
GRANT USAGE ON SCHEMA public TO analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO analyst;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON
TABLES TO analyst;
```

The last line is the one people forget. `GRANT ... ON ALL TABLES` covers the tables that exist today; `ALTER DEFAULT PRIVILEGES` covers the ones created next month, so the analyst does not lose access to every new table until somebody remembers. With this, `analyst` can read everything and write nothing: an `UPDATE` fails with `permission denied for table orders`, which is the error you want them to see.

A belt to go with the braces: `ALTER ROLE analyst SET default_transaction_read_only = on;` makes every transaction the role opens read-only at the session level, so that even a table they have somehow been granted write access to cannot be changed without an explicit override.

Three users, not one

The usual shape for a real system:

- **The migration user** owns the tables and can alter them. Used only by the migration tool, from a deployment pipeline, never interactively.
- **The application user** can `SELECT`, `INSERT`, `UPDATE` and `DELETE` on the tables the application needs, and cannot alter anything. If the application is compromised, the attacker cannot drop a table.
- **The analyst user** can `SELECT`, and often only from views that omit personal data.

The superuser, `postgres`, is used by nobody day to day. Its password lives in a password manager and its use is an event.

Hiding columns and rows

`GRANT SELECT (id, display_name, city) ON customers TO analyst` grants three columns and not the phone or email beside them. A view, as the earlier lesson said, does the same more readably and can also transform: `left(email, 3) || '***'` for a partly masked address. Row-level security goes further:

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;
CREATE POLICY branch_only ON orders FOR SELECT TO branch_manag-
er
  USING (branch_id = current_setting('app.branch_id')::int);
```

A branch manager's session sets `app.branch_id` and sees only their branch's orders, in every query, without any WHERE clause. The policy is enforced by the database rather than by the application remembering, which is the same argument as for constraints.

The read-only door for an AI

An assistant that writes SQL from a question is a user that types fast, does not read the schema carefully, and cannot be embarrassed. Give it the analyst role and nothing more. Add `SET statement_timeout = '30s'` for the role, so that a query it writes with an accidental cross join is killed rather than left to run for an hour. Point it at a read replica if one exists, so that its scans do not compete with the application. And keep personal columns behind views, because a model that has seen a phone number may repeat it. The final module of this course is about getting good SQL out of an AI; this lesson is about making sure the bad SQL cannot do harm.

The engines that have no roles

SQLite has no users. Access is the file's permissions on disk, and a read-only connection is opened with `?mode=ro` in the connection string or by giving the process a file it cannot write. DuckDB attaches a database read-only with `ATTACH 'school.duckdb' (READ_ONLY)`. Both are the right level of control for a single machine and no substitute for roles on a shared server.

The habit

When a new person, tool or program needs the database, create a role for it with the least it needs, and write down why. When a role has not been used for a quarter, revoke it. The list of roles is the list of everyone who could have changed the data, and it should be short enough to read aloud.

BEFORE YOU MOVE ON

An analyst role was granted `SELECT ON ALL TABLES IN SCHEMA public` in January. In March the analyst reports "permission denied" on a table that was created in February, while every older table still works. Why?

A. Postgres revokes grants on tables that have not been queried for 30 days, and the new table has not yet been read by the role.

B. `GRANT ... ON ALL TABLES` applies only to tables that existed at the time, and no `ALTER DEFAULT PRIVILEGES` was set for tables created afterwards.

C. The February table was created by the migration user, and tables can only be read by the role that created them until ownership is transferred to the reader.

D. The new table has row-level security enabled by default, which blocks every role without a policy, including analysts with SELECT.

66 · 11 min

From a paragraph to a schema: a tuition centre

THE ONE THING TO KEEP

Turn requirements into a schema by giving every noun a table with a stated grain, every many-to-many its bridge table with its own facts, every implied rule a constraint, and every derivable number a query rather than a column.

The paragraph

> *We run a tuition centre. Students enrol in batches; a batch is a course (say, Class 10 Maths) running over a term with a fixed weekly timetable and a fee. A student can be in several batches and a batch has many students. Fees are paid in instalments, and we need to know who owes what. We take attendance at every session. Teachers are assigned to batches, and a teacher can teach several.*

That is the whole brief, and it is enough. Designing the schema is the act of reading it slowly.

Nouns become tables, one grain each

Student, batch, course, teacher, session, enrolment, payment, attendance record. Say the grain of each: one row is one student; one row is one batch (a course in a term); one row is one session of a batch on a date; one row is one instalment paid. "Enrolment" is not in the paragraph as a noun, but "a student can be in several batches and a batch has many students" is a many-to-many relationship, and the earlier lesson said those need a table in the middle. It arrives with facts of its own: the date the student joined, and the fee agreed for that student, which may differ from the batch's list fee.

The tables

```
CREATE TABLE students (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  full_name   text NOT NULL,
  phone       text NOT NULL,
  created_at  timestamptz NOT NULL DEFAULT now(),
  CONSTRAINT students_phone_unique UNIQUE (phone)
);

CREATE TABLE teachers (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  full_name   text NOT NULL
);

CREATE TABLE batches (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  course_name text NOT NULL,                -- 'Class 10
Maths'
  term_start  date NOT NULL,
  term_end    date NOT NULL,
  list_fee    numeric(10,2) NOT NULL CHECK (list_fee >= 0),
  teacher_id  bigint NOT NULL REFERENCES teachers (id),
  CONSTRAINT batches_term_valid CHECK (term_end > term_start)
);

CREATE TABLE enrolments (
  student_id  bigint NOT NULL REFERENCES students (id),
  batch_id    bigint NOT NULL REFERENCES batches (id),
  joined_on   date NOT NULL,
  agreed_fee  numeric(10,2) NOT NULL CHECK (agreed_fee >= 0),
  PRIMARY KEY (student_id, batch_id)
);

CREATE TABLE sessions (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  batch_id    bigint NOT NULL REFERENCES batches (id),
  held_on     date NOT NULL,
  CONSTRAINT sessions_one_per_day UNIQUE (batch_id, held_on)
);

CREATE TABLE attendance (
  session_id  bigint NOT NULL REFERENCES sessions (id),
  student_id  bigint NOT NULL REFERENCES students (id),
```

```
  present     boolean NOT NULL,
  PRIMARY KEY (session_id, student_id)
);

CREATE TABLE payments (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  student_id  bigint NOT NULL REFERENCES students (id),
  batch_id    bigint NOT NULL REFERENCES batches (id),
  amount      numeric(10,2) NOT NULL CHECK (amount > 0),
  paid_on     date NOT NULL,
  reference   text NOT NULL UNIQUE,
  FOREIGN KEY (student_id, batch_id) REFERENCES enrolments
(student_id, batch_id)
);

CREATE INDEX ON enrolments (batch_id);
CREATE INDEX ON sessions (batch_id, held_on);
CREATE INDEX ON payments (student_id, batch_id);
```

The decisions inside it

`agreed_fee` is copied onto the enrolment rather than read from the batch, for the reason the lesson on keeping facts in one place gave: it is the fee that was true for this student at this moment, and a later change to `list_fee` must not rewrite it.

`payments` references the enrolment through a two-column foreign key, so a payment can only be recorded against a batch the student is actually in. That is a rule the paragraph implied and nobody stated; the schema states it.

`sessions_one_per_day` says a batch meets at most once per day, which is what "a fixed weekly timetable" implies. If evening and morning sessions of the same batch ever exist, the constraint is the thing that will tell you the model has changed.

There is no `balance_due` column on `students` or `enrolments`. It is tempting, it is what the paragraph asks for, and it is derived: agreed fee minus payments. Stored, it goes stale the first time a payment is recorded without the update, and the concurrent-writes lesson showed how that happens. Computed, it is always right:

```
SELECT s.full_name, b.course_name,
       e.agreed_fee - coalesce(sum(p.amount), 0) AS balance_due
FROM enrolments e
JOIN students s ON s.id = e.student_id
JOIN batches  b ON b.id = e.batch_id
LEFT JOIN payments p ON p.student_id = e.student_id AND p.-
batch_id = e.batch_id
GROUP BY s.full_name, b.course_name, e.agreed_fee
HAVING e.agreed_fee - coalesce(sum(p.amount), 0) > 0;
```

That query is the "who owes what" report, and it is the test of the schema: it needs no tricks, no DISTINCT, no fan-trap defence, because there is exactly one child table in it. Attendance percentage per batch is a second such query, and the earlier lessons on ratios and on spines say how to write it so that a student who joined late is not counted absent for sessions before they joined.

What was left out on purpose

No `deleted_at` yet, because nothing in the brief says students are removed; when it does, the history lesson says how. No JSON, because every attribute here is known and typed. No teacher-to-batch bridge table, because the paragraph says a batch has a teacher, singular; if two teachers ever share a batch, the change is a new table and a moved column, and the foreign key will make the change visible rather than silent. A schema is a set of claims about the world, and leaving a claim out is also a claim.

The habit

Read the requirements twice. List the nouns, give each a grain. Find the many-to-many sentences and give each its table. For every number that could be derived, do not store it. For every rule the paragraph implies, write a constraint. Then write the two or three reports the schema exists to produce, and if any needs a workaround, the schema is not finished.

BEFORE YOU MOVE ON

A colleague proposes adding `balance_due numeric` to the `enrolments` table so the "who owes what" report becomes a single-table query. What is the strongest objection?

A. A numeric column cannot hold a negative balance for overpayments, so the report would miss refunds owed.

B. Adding a column to a live table rewrites every row, which is unacceptable on a table this size.

C. The balance is derived from agreed fee minus payments, and a stored copy goes stale the first time a payment is recorded without the matching update.

D. The report would still need a join to `students` for the name, so the single-table promise is not kept and the column adds nothing except ongoing maintenance.

CASE STUDY · MODULE 6

Four thousand marks, and the update that could not be rehearsed

The examination cell of a polytechnic in Bhopal, the evening before semester results are published to 6,200 students

Results go live at nine the next morning. At six in the evening the exam controller discovers that one paper — Applied Mathematics II, taken by 4,100 students — has been entered against the wrong scheme. The paper was moderated with a 5-mark grace applied to anyone who scored between 35 and 39, and the grace was applied twice by two different clerks working from the same list, so those students carry marks 5 higher than they should.

Ravi, who maintains the college's ERP database, has the `marks` table in front of him: one row per student per paper, with `marks_obtained`, `entered_by` and `updated_at`. He also has a `moderation_log` table listing which students received grace and by whom, with 1,180 rows for this paper, of which some students appear twice.

The correction is arithmetically simple: subtract 5 from the students whose grace was applied twice. Identifying them is a group and a count. The statement writes itself:

```
UPDATE marks m
SET    marks_obtained = m.marks_obtained - 5
FROM   moderation_log l
WHERE  l.student_id = m.student_id
  AND  l.paper_code = m.paper_code
  AND  m.paper_code = 'AM-II';
```

Ravi does not run it, for the reason the module gives: an `UPDATE ... FROM` whose right-hand side matches more than once per target row updates that row with one of the matches and does not say which. Here the right-hand side matches twice for exactly the students he is trying to fix — that duplication is the bug — and Postgres will apply the subtraction once per target row, not once per match. The statement would take 5 off every student with any grace

at all, including the 1,180 minus the duplicates who received it correctly. The rehearsal query he runs instead, counting matches per target row, returns 612 rows with a count above one. That is the population.

Which leaves the decision, at half past six.

The first option is a corrected statement: subtract 5 only from the 612 students identified by the group-and-count, inside a transaction, with the row count checked against 612 before committing. Twenty minutes including the rehearsal. It requires Ravi to be confident that `moderation_log` is a complete record of what the clerks did — and the clerks were working from a printed list, entering marks directly into the ERP screen, with the log written by a trigger that was added eight months ago and has never been audited.

The second is what the exam controller wants: hold publication, re-enter the paper from the moderated answer sheets tomorrow, and publish at five in the evening. That is unarguably correct and it costs a day in which 6,200 students, most of whom need the result for internship applications closing that week, refresh a page that says nothing. The college did this two years ago and it made the local newspaper.

The third is to publish on time with the paper's results withheld, and issue that one paper later. It affects only the affected students, and it also tells 4,100 of them, publicly, that something went wrong with their marks — before anyone knows whether the fix is right.

What none of the three options addresses is that the ERP's application user owns the schema and can alter it, that every clerk shares one login, and that `updated_at` is set by the application rather than by the database, so a row changed directly in the database tonight will carry a timestamp that says nothing about who changed it.

> **STOP HERE**
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Ravi took the first option and added one step: before the UPDATE, he took a copy of the affected rows into a table called `marks_ammii_before_20260714`, with the student id, the old mark and the time. The whole correction then ran inside one transaction — the copy, the update, and a count — and he read `UPDATE 612` before typing COMMIT. The check that mattered came afterwards, and it was not a row count: he picked six students from the 612, pulled their moderated answer sheets from the exam cell, and added the marks by hand. Five matched. One did not, because that student's grace had been applied twice by the same clerk in the same minute and the trigger had logged it once, so the double application was invisible to `moderation_log` entirely. That single row turned a settled question back into an open one, and the exam controller held the paper — the third option — while a full reconciliation against the answer sheets ran the next morning. It found nine more. Results for the other papers went out at nine as planned. The lasting changes were smaller and more useful than the incident: separate database roles for the application, the migration tool and the exam cell's read-only reporting; an audit trigger on `marks` recording the old row, the new row, the database user and the time; and a standing rule that no correction to a published mark is made by an UPDATE without a before-copy in the same transaction.

Why would the first version of the UPDATE have been wrong, and what rehearsal exposes that class of fault in any `UPDATE … FROM`?

Because the join matches twice for exactly the students being fixed, and an `UPDATE … FROM` applies the SET once per target row rather than once per match, choosing one matching row silently. The rehearsal is a count of matches per target row — join the two tables, group by the target's key, and keep the groups with a count above one. It must return nothing before the UPDATE is safe. Here it returned 612 rows, which was both the proof that the statement was wrong and the definition of the population that needed correcting.

The row count after the UPDATE was 612, exactly as rehearsed, and the statement was still not known to be right. What did the hand check add that no count could?

A count proves the statement touched the rows it was aimed at. It cannot prove the aim was right, because both the rehearsal and the update read the same source — `moderation_log` — and that source was incomplete. Checking six students against the answer sheets tested the assumption underneath the whole correction, which is the one thing internal consistency can never test. This is the same principle as reconciling a total against an external figure: an error shared by the query and its check is invisible to the check.

Name two of the changes made afterwards and explain which specific risk each one removes.

Separate database roles remove the risk that an ordinary reporting session, or a compromised application, can alter or drop a table: the exam cell's reporting role can only SELECT, and the schema can only be changed by the migration user from a deployment pipeline. An audit trigger on `marks` removes the risk that a corrected mark leaves no trace of what it was before or who changed it — the trigger runs on every write regardless of which code path or which person made it, which is exactly the property an audit that depends on the application remembering does not have. The before-copy rule removes the risk that a correct-looking UPDATE cannot be reversed after COMMIT.

MODULE 6 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 `INSERT ... ON CONFLICT (reference) DO NOTHING` fails with "there is no unique or exclusion constraint matching the ON CONFLICT specification". What does that mean?

A. The reference column is nullable, and NULLs cannot be compared
B. There is no unique index on reference to detect a conflict in
C. DO NOTHING requires a WHERE clause naming the rows to skip
D. The source query returns the same reference more than once

2 You are about to run `UPDATE orders o SET city = c.city FROM customers c WHERE c.id = o.customer_id AND o.city IS NULL`. What must the rehearsal prove?

A. That the join produces no order matching more than one customer row
B. That every order in the table has a customer id that is not NULL
C. That the number of rows to be changed is smaller than a stated batch size
D. That the city column on customers contains no values that are NULL

3 Inside a transaction in Postgres, one statement fails. Every statement afterwards, including `SELECT 1`, returns "current transaction is aborted". Why is that deliberate?

A. Postgres has released the transaction's locks and cannot re-acquire them
B. The connection has been reset, so the session must be opened again
C. A transaction with one failed step is not in a coherent state
D. The failed statement is retried in the background until it succeeds

4 A course has one seat left. Two requests read `seats_left`, both see 1, both write 0, and two students are enrolled. Which fix removes the fault most simply?

A. Run both requests at the SERIALIZABLE isolation level and retry on failure

B. Add a UNIQUE constraint on the pair of student and course

C. Retry the write until the read and the write agree on the value

D. UPDATE ... SET seats_left = seats_left - 1 WHERE seats_left > 0

5 A forty-million-row table needs a new required column. Which sequence keeps the site up?

A. Add the column with NOT NULL and a default in one ALTER, off-peak

B. Add it nullable, backfill in batches, add NOT VALID, then validate

C. Create a new table with the column, copy the rows, then rename both

D. Add it nullable and leave it so, enforcing the rule in every application

6 A dashboard reads a materialised view refreshed at six each morning. What must the dashboard say?

A. Which columns of the underlying tables the view selects from

B. That the figures come from a view rather than from a table

C. The time of the last refresh, because the data is that old

D. That the numbers may be recomputed if a query is run again

EXAMINATION

Data, SQL and Getting to the Answer — final examination

This paper covers the whole course: reading a schema and stating the grain of a result; joins, including the ones that multiply; aggregates, NULLs and the boundaries of time; window functions, streaks, cohorts and as-of joins; loading and cleaning data that arrives dirty; and changing data safely, from constraints and transactions to permissions. Twenty-five questions are drawn each time from a larger pool, so a retake is a different paper. The pass mark is 70 per cent. There is no timer: read the question twice and work the mechanism out rather than hurrying. If you do not pass, you may retake after thirty minutes with fresh questions, as often as you need. A teacher can open the next course for you at any time, whatever this paper says.

On the website a sitting is 25 questions drawn from this pool and the pass mark is 70%. There is no time limit, there and here. Passing opens the next course in the track.

1 1. Tables, types and the question you are actually asking

A shop keeps 3,000 rows in a spreadsheet, edited by one person, to answer occasional questions. When does the move to a database become necessary?

A. When the file grows beyond about ten thousand rows

B. When more than one person writes, or a fact appears in two rows

C. As soon as any question requires more than one calculation column

D. When the file takes longer than a minute to open on the office laptop

2 1. Tables, types and the question you are actually asking

You practise on SQLite. `INSERT INTO students VALUES (1, 'Ade', 'Grade Nine')` succeeds although `class_level` was declared `int`. What is going on?

A. SQLite has type affinity rather than type constraints

B. The insert omitted a column list, so all types are ignored

C. Integer columns in SQLite accept any value that has a digit in it

D. The value was silently converted to zero and stored as an integer

3 1. Tables, types and the question you are actually asking

Your first `COPY payments FROM 'payments.csv'` against a hosted Postgres fails with permission denied, though the file is plainly on your desktop. Why?

A. The database role lacks the pg_read_server_files privilege on the table

B. The file must be converted to UTF-8 before the server will accept it

C. COPY reads a file on the server's disk; \copy reads one on yours

D. Hosted databases disable bulk loading and require INSERT statements

4 1. Tables, types and the question you are actually asking

`enrolments` has `PRIMARY KEY (student_id, course_id)`. Students can re-enrol in the same course next term. What is wrong?

A. A composite key needs its own surrogate id before it can be referenced

B. The key states that a student enrols in a course once, which is now false

C. Two-column keys cannot be used as the target of a foreign key

D. Nothing: the key is correct and the second enrolment will simply update it

5 1. Tables, types and the question you are actually asking

`payments.student_id REFERENCES students(id)`. Which ON DELETE behaviour suits a table of money received?

A. CASCADE, so tidying a student record leaves no orphan payments behind
B. SET NULL, so the payments survive without pointing at a missing student
C. RESTRICT, so the student cannot be deleted while payments exist
D. No clause at all, since the default keeps the payments untouched

6 1. Tables, types and the question you are actually asking

`order_items` stores `unit_price` while `products` stores `current_price`. Is that duplication a normalisation fault?

A. Yes, and a join to products at report time removes it
B. Yes, but it is tolerated because joins are slow at reporting scale
C. Only if the two values are allowed to differ from one another
D. No: it records what was charged, which must never change again

7 1. Tables, types and the question you are actually asking

`WHERE country = 'NG' OR country = 'KE' AND amount > 5000` returns every Nigerian payment, whatever the amount. Why?

A. AND binds more tightly than OR, so the amount test applies only to KE
B. The two country tests should have been written as a single IN list
C. OR is evaluated left to right, so the first true test ends the condition
D. A text comparison and a numeric comparison cannot be combined with AND

8 1. Tables, types and the question you are actually asking

A funnel query aborts with "division by zero" on Postgres for branches with no attempts. What is the right repair?

A. Add `WHERE attempts > 0`, which removes the offending rows
B. Wrap the division in COALESCE so the result becomes zero
C. Divide by `NULLIF(attempts, 0)`, so the rate becomes NULL
D. Cast both sides to numeric, which makes the division safe

9 1. Tables, types and the question you are actually asking

`SELECT DISTINCT city, id FROM customers` was written to get one row per city and returns the whole table. Why?

A. DISTINCT applies to the whole row, and the id makes each unique
B. DISTINCT must come after the column list, not before it
C. An id column is always excluded from DISTINCT because it is the key
D. DISTINCT needs an ORDER BY on the same column before it can group

10 1. Tables, types and the question you are actually asking

A subscription business computes churn as the share of rows in `customers` whose status is 'cancelled', and gets 4%. Cancelled accounts are purged after a year. What is wrong?

A. The status column should have been counted with COUNT rather than COUNT(*)
B. Purged customers are gone from the denominator
C. The figure needs a date filter, since it currently spans every year at once
D. Churn should be computed per month and then averaged across the year

11 2. Putting tables together

Which query lists orders that have no refund at all?

A. An inner join to refunds, with `WHERE r.reason IS NULL` on the result
B. A LEFT JOIN to refunds, keeping the rows where `r.id IS NULL`
C. `SELECT ... FROM orders WHERE refund_count = 0` on the orders table
D. A join to refunds with `NOT EXISTS` written inside the ON clause

12 2. Putting tables together

A report using `NATURAL JOIN` has worked for a year and today returns no rows. Nothing in the query changed. What is the likeliest cause?

A. A column with a matching name was added to both tables
B. One of the tables gained rows whose key does not match
C. The join now runs before the WHERE clause rather than after it
D. A collation change reordered the values so equality no longer holds

13 2. Putting tables together

`SELECT * FROM orders, customers` runs for an hour and produces hundreds of millions of rows. What happened, and what prevents it?

A. The tables were joined on the wrong column; name the columns explicitly
B. Every left row was paired with every right one; use JOIN ... ON
C. The query lacked a LIMIT, which every exploratory query should carry
D. The planner chose a nested loop; an index on customer_id fixes it

14 2. Putting tables together

You must find every discrepancy between a ledger and a bank statement, in both directions, in one result. Which join?

A. LEFT JOIN from the ledger, since the ledger is the system of record
B. INNER JOIN, then compare the amounts on the matched rows
C. FULL OUTER JOIN, keeping unmatched rows from either side
D. CROSS JOIN, then filter to the pairs with equal references

15 2. Putting tables together

How do you find students enrolled in both course 11 and course 22, given a bridge table with one row per student per course?

A. `WHERE course_id = 11 AND course_id = 22` on the enrolments table
B. A self-join of enrolments on student_id with a different course on each side
C. `WHERE course_id IN (11, 22)` with SELECT DISTINCT on the student id
D. Group by student and require two distinct courses

16 2. Putting tables together

A table holds `students.courses` containing text like 'stats,algebra,physics'. Which problem is intrinsic rather than cosmetic?

A. Counting how many students take algebra requires searching strings
B. The column will eventually exceed the maximum length of a text field
C. Sorting the list alphabetically has to be done in the application
D. The values cannot be displayed without splitting them for the reader

17 2. Putting tables together

A recursive CTE walking an org chart runs until the server runs out of memory. What is the fault, and the guard?

A. The anchor row is missing; every recursive CTE needs a starting row
B. The UNION should be UNION ALL, which stops the duplicate expansion
C. The recursion is unindexed; an index on manager_id ends it quickly
D. The data contains a cycle; add a depth limit or track the path

18 2. Putting tables together

You are comparing a ledger with a bank file using EXCEPT. Why run it in both directions?

A. EXCEPT deduplicates, so one direction may hide repeated rows
B. EXCEPT is not symmetric; the two directions differ
C. Running it twice confirms that the result is stable between executions
D. The second run uses the first as a cache and is therefore much faster

19 2. Putting tables together

A CTE selects from a very large table, and the outer query filters on an indexed column. The query is slow. What is worth trying?

A. `WITH ... AS MATERIALIZED`, which computes the CTE exactly once
B. Replacing the CTE with a temporary table populated beforehand
C. `WITH ... AS NOT MATERIALIZED`, so the filter can reach the index
D. Adding an ORDER BY inside the CTE so the planner keeps the rows sorted

20 2. Putting tables together

Why can a scalar subquery in the SELECT list not cause the row multiplication a join can?

A. It runs before the FROM clause, so no rows exist yet to multiply
B. It is cached per outer row, so repeated matches return one value
C. The planner rewrites it into a LEFT JOIN, which never multiplies
D. It must return one row, or the query fails outright

21 3. Summaries, and the piles behind them

Why can `SUM(amount) > 500000` not be written in a WHERE clause?

A. WHERE runs before the piles exist, so there is nothing to sum yet
B. WHERE accepts only comparisons between two columns of the same row
C. SUM is computed after ORDER BY, which WHERE precedes
D. It can, but only when the query also has a HAVING clause

22 3. Summaries, and the piles behind them

`SELECT city, customer_name, COUNT(*) FROM orders GROUP BY city` is refused. Why is that refusal correct rather than bureaucratic?

A. COUNT(*) may not appear beside an ungrouped text column
B. The Lagos pile holds 900 names and one output slot
C. customer_name belongs in the customers table, not in orders
D. GROUP BY accepts at most one column unless ROLLUP is used

23 3. Summaries, and the piles behind them

A survey has 1,000 responses; 200 people skipped the income question. What does `AVG(income)` return?

A. The total of 800 values divided by 800
B. The total of 800 values divided by 1,000
C. NULL, because some rows in the column are NULL
D. The total of 1,000 values, with the missing ones treated as zero

24 3. Summaries, and the piles behind them

`WHERE id NOT IN (SELECT student_id FROM payments)` returns zero rows, and you know some students have never paid. What is happening?

A. The subquery returns too many rows for NOT IN to evaluate

B. NOT IN compares only the first value returned by the subquery

C. One student_id in payments is NULL, so nothing can be true

D. NOT IN requires a correlated subquery when the tables share a key

25 3. Summaries, and the piles behind them

In Postgres, `paid / total * 100` returns 0 for every row where paid is less than total. Why?

A. The multiplication happens first and overflows the integer type

B. Percentages need `round()`, which also performs the type promotion

C. `total` contains zeros, and division by zero silently yields zero

D. Two integers divide to an integer, truncating the fraction

26 3. Summaries, and the piles behind them

Two analysts compute weekly revenue from one table and disagree about every week. Both queries are internally consistent. What is the likeliest cause?

A. One truncated to the week and the other extracted a week number

B. One used SUM and the other used a running total over the window

C. The engines disagree about which day a week starts on

D. One filtered with BETWEEN and the other with a half-open range

27 3. Summaries, and the piles behind them

A MySQL export lists each customer's order references with `group_concat`. Several lists end mid-reference. What happened?

A. group_concat truncates at 1,024 characters and does not warn

B. The references contain commas, which broke the CSV export

C. The aggregate ran out of memory and returned what it had built

D. An ORDER BY inside the aggregate cut the list at the sort boundary

28 3. Summaries, and the piles behind them

A "largest order per customer" report joins back to a per-customer `max(amount)` and returns more rows than there are customers. Why?

A. The join matched customers with no orders and produced NULL rows
B. max() was computed over the joined result rather than the base table
C. Some customers have two orders at exactly the maximum amount
D. The CTE was materialised, so its rows were counted twice

29 3. Summaries, and the piles behind them

Why can SQL not produce "one column per month, whatever months exist in the data"?

A. Column names must be literals, so they cannot contain a date
B. A query's columns are fixed when it is planned
C. The GROUP BY would need to reference a column that does not yet exist
D. It can, but only in a materialised view, where the shape is stored

30 3. Summaries, and the piles behind them

An ungrouped total is 4,182,300 and the fourteen city rows add up to 4,161,900. What is the most likely explanation?

A. Rounding differences between the grouped and ungrouped queries
B. A join in the grouped version dropped a small number of rows at random
C. Orders with a NULL city were removed by a filter or an inner join
D. The grouped query used a different date range from the ungrouped one

31 4. Windows, sequences and time

What does `sum(amount) OVER (PARTITION BY city)` do that `GROUP BY city` cannot?

A. It computes the total without reading every row of the city
B. It writes the city's total beside every order, keeping the rows
C. It allows several different aggregates over the same set of rows
D. It orders the result by city without a separate ORDER BY clause

32 4. Windows, sequences and time

A query filtered to `WHERE city = 'Pune'` includes `sum(amount) OVER ()` as a grand total. What does that column contain?

A. Pune's total, because the window sees the rows WHERE kept

B. The company's total, because an empty OVER ignores the filter

C. NULL, because a window with no partition has no rows to sum

D. The running total up to each row, since no frame was specified

33 4. Windows, sequences and time

A deduplication keeps the row where `row_number() OVER (PARTITION BY reference ORDER BY loaded_at DESC) = 1`. Why add a second sort key?

A. Without it the window cannot use an index and will scan the table

B. row_number needs at least two columns to establish a partition

C. When two rows share a load time, which one survives is arbitrary

D. The DESC keyword requires a following column to define its direction

34 4. Windows, sequences and time

Customers are segmented into deciles with `ntile(10) OVER (ORDER BY spend DESC)`, and the top decile turns out to hold 60% of revenue. Is the segmentation wrong?

A. Yes: ntile requires the rows to divide evenly into ten groups

B. Yes: ORDER BY DESC inverts the tiles, so decile 1 is the smallest

C. No: ntile makes groups of equal count, not of equal value

D. No: 60% in the top decile is the expected result of any ntile

35 4. Windows, sequences and time

A report must show every product that reached the top three in its category, including all products tied at third. Which filter?

A. `dense_rank() <= 3`, which shares positions and skips none

B. `row_number() <= 3`, with the product id as the tiebreaker

C. `rank() <= 3`, which shares positions and then skips ahead

D. `ntile(3) = 1`, which places the leading third in the first bucket

36 4. Windows, sequences and time

A cohort grid shows month-6 retention falling from 31% to 14% across the last year's cohorts. The most recent cohort is four months old. What is the fault?

A. The cohorts were defined by sign-up rather than by first purchase
B. Retention was computed as a share of all customers, not of the cohort
C. The window's first_value read the wrong row, so the denominators are wrong
D. Recent cohorts have not reached month 6

37 4. Windows, sequences and time

`count(DISTINCT customer_id) OVER (ORDER BY placed_at)` is rejected by Postgres. What is the standard way to get "customers acquired to date"?

A. Compute it in a subquery per day and join the results back
B. Replace it with `count(customer_id)` and divide by the average orders
C. Use a RANGE frame, which supports distinct aggregates
D. Flag each customer's first order and run a running count

38 4. Windows, sequences and time

Month-over-month change is computed with `lag(revenue) OVER (ORDER BY month)` on a table where a branch had no revenue in March, so March has no row. What does April's change show?

A. NULL, because the previous month is missing from the ordering
B. The change since February, presented as a one-month change
C. Zero, because lag treats a missing period as an unchanged one
D. The change since March, with revenue coalesced to zero

39 4. Windows, sequences and time

A column declared `timestamp` receives the value `2026-03-29T02:30:00+05:30`. What is stored?

A. The instant, converted to the server's configured time zone
B. 02:30, with the offset discarded and unrecoverable
C. 21:00 on 28 March, the equivalent moment in UTC
D. The text as given, since the type keeps the original string

40 4. Windows, sequences and time

Before trusting a join to an effective-dated cost table, which two checks must return no rows?

A. That every cost is positive, and that no product has more than one row
B. That valid_to is never NULL, and that valid_from is never in the future
C. That the table has an index on product_id, and that costs are numeric
D. That no version overlaps the previous one, and none leaves a gap

41 5. Data that arrives dirty

`wc -l` reports 12,401 lines and the loaded table has 12,380 rows. The header accounts for one line. What is the likeliest cause?

A. Twenty rows failed a type check and were skipped in silence
B. Twenty quoted fields contain a line break
C. The file uses Windows line endings, which are counted twice
D. The loader deduplicated twenty rows that were exactly identical

42 5. Data that arrives dirty

A CSV from a colleague in Brazil loads with the right number of rows and one very wide column. What happened?

A. The file is separated by semicolons, not commas
B. The file is encoded in Latin-1, so the delimiters were misread
C. The header row was missing, so all columns were merged into one
D. The quoting character is a single quote rather than a double quote

43 5. Data that arrives dirty

A name arrives in the table as `RenÃ©e`. What has gone wrong?

A. The name was entered twice and the two spellings were merged
B. The database collation cannot represent accented characters
C. The accent is a combining character that the terminal cannot draw
D. UTF-8 bytes were read with a single-byte encoding

44 5. Data that arrives dirty

After loading a CSV written by Excel, the first column looks like `id` but cannot be referenced as `id`. Why?

A. Excel exports the first column name in upper case regardless of the header

B. The column name carries a byte-order mark the loader did not strip

C. The loader reserved `id` and renamed the column silently

D. A trailing space in the header became part of the column name

45 5. Data that arrives dirty

Which of these is destroyed permanently by loading a column with the wrong type?

A. A phone number's leading zero, cast into an integer column

B. An amount's currency symbol, stripped before a numeric cast

C. A date's original text, parsed into a date column

D. A category's capitalisation, lower-cased into a key column

46 5. Data that arrives dirty

`GROUP BY city` returns two rows that both read `Lagos`. Which query identifies the difference fastest?

A. Compare the two with `=` and look at whether the result is NULL

B. Select the city with its length and its bytes in hex

C. Count the rows in each group and keep the smaller one

D. Apply `lower()` to both and see whether the groups then merge

47 5. Data that arrives dirty

Two customer records both display as `Renée` and do not match on `=`. Neither has stray whitespace. What is the cause and the fix?

A. One row has a trailing zero-width space; use translate to remove it

B. The database collation is case-sensitive; compare with ILIKE instead

C. The accent is stored in two Unicode forms; normalise both to NFC

D. One value is stored as bytes and the other as text; cast before comparing

48 5. Data that arrives dirty

City variants are mapped to canonical names with a `city_map` table and a LEFT JOIN. What does the LEFT JOIN buy that a CASE expression does not?

A. Unmapped variants come through and can be listed with their counts
B. It is faster, because the mapping can be indexed on the variant
C. It applies the mapping to new rows automatically as they are inserted
D. It guarantees that every value in the column has a canonical form

49 5. Data that arrives dirty

A model is given city as an integer code: Mumbai 1, Delhi 2, Pune 3. What does the model conclude that is false?

A. That there are exactly three cities in the data
B. That the codes will remain stable when new data arrives
C. That Pune is greater than Mumbai and lies beyond Delhi
D. That a city with a higher code appears more often in the data

50 5. Data that arrives dirty

A model's accuracy in production is far below its accuracy in testing, and no error is raised anywhere. Which structural defence would have prevented it?

A. Retraining the model weekly rather than monthly
B. Holding out a larger test set from the same training table
C. Scaling every feature to the same range before training
D. One query, parameterised by as_of, used for both

51 6. Changing data safely

"A booking may not overlap another booking of the same room." Why can a CHECK constraint not enforce it?

A. CHECK sees one row at a time and cannot look at other rows
B. CHECK cannot reference a range type or compare two timestamps
C. CHECK runs only on INSERT, so an UPDATE could still create an overlap
D. CHECK is evaluated after the row is written, when it is already too late

52 6. Changing data safely

A tuition centre's brief asks to know who owes what. Why should `balance_due` not be a column on `enrolments`?

A. Numeric columns cannot be updated safely inside a transaction
B. It is derived, and a stored copy goes stale
C. The value changes too often for an index on it to stay useful
D. It belongs on the students table, since a student owes the money

53 6. Changing data safely

A `jsonb` column stores product attributes. `colour` is queried in every report. What should happen?

A. Move every attribute into its own column and drop the JSON
B. Add a GIN index on the document so the key lookup is fast
C. Split the products table by kind, so each has its own columns
D. Promote colour to an indexed generated column

54 6. Changing data safely

Two hundred million old rows must be deleted from a busy table. Why delete in slices of ten thousand?

A. A single DELETE cannot exceed the maximum statement length
B. Each slice commits quickly, so others get a turn
C. Slices let the planner use an index that one large DELETE cannot
D. The transaction log is written only once per slice rather than per row

55 6. Changing data safely

A transaction runs `SELECT ... FOR UPDATE`, then waits for a person to read the result and type COMMIT. What is the cost?

A. The lock escalates to the whole table after a few seconds of waiting
B. Postgres cancels the transaction once its statement timeout expires
C. Every other writer needing that row waits too
D. The row is read from a snapshot, so the value shown may be stale

56 6. Changing data safely

Two transfer transactions deadlock against each other regularly. Which change prevents it?

A. Lock the rows in a consistent order, such as by ascending id

B. Raise the isolation level to SERIALIZABLE for both transactions

C. Retry the failed transaction immediately rather than after a delay

D. Take a table-level lock so that only one transfer runs at a time

57 6. Changing data safely

A `customers` table gains a `deleted_at` column for soft deletes. It has `UNIQUE (email)`. What quietly breaks?

A. Foreign keys from orders begin to fail for soft-deleted customers

B. The unique index stops working once any row has a NULL deleted_at

C. A returning customer cannot reuse their old email

D. Queries that filter on deleted_at can no longer use the email index

58 6. Changing data safely

An audit trail must record who changed a row and what it said before. Why put it in a database trigger rather than in the application?

A. A trigger can write the old row, which the application cannot read

B. Application code cannot know the database user that made the change

C. Triggers run inside the transaction and are therefore faster to write

D. The trigger runs on every write, whoever made it

59 6. Changing data safely

An analyst was granted SELECT on all tables last year and cannot read this year's new tables. What was forgotten?

A. ALTER DEFAULT PRIVILEGES, which covers tables created later

B. GRANT USAGE ON SCHEMA, without which nothing at all is visible

C. A second GRANT CONNECT, because the database was recreated

D. Membership of a group role, which is what carries the privilege forward

60 6. Changing data safely

An AI assistant will be given a database connection to answer questions. Which combination is the right level of care?

A. The application's role, with a nightly backup taken before each session

B. A read-only role, a statement timeout, and views hiding personal columns

C. A superuser connection to a replica, so production cannot be affected

D. The migration role with DELETE revoked, so schema questions can be answered

Answers

Each entry gives the correct option, then what each wrong one reveals about how somebody was thinking. The second part is worth more than the first: a wrong answer here is a named misreading, not a mistake.

Lesson checks

1. When a spreadsheet stops working — A

`B` — Treats file size as the reason to move off a spreadsheet, when size is the symptom that shows up last.

`C` — Assumes the flaw is in how the value is formatted rather than in how many times it is stored.

`D` — Reaches for the missing primary key, which is a real gap but not what breaks when the address changes.

2. Getting a database onto the machine you have — B

`A` — It is not wrong, but it front-loads schema design onto a question you have not yet decided is worth answering; the import alone costs more than the answer.

`C` — Most spreadsheets stop near a million rows and will either refuse the file or become unusable, and the operation is not repeatable next month.

`D` — SQLite will do it, but it is optimised for small transactional reads and writes rather than scanning a whole column, and you still pay the import step first.

3. What a column type actually buys you — D

`A` — Display rounding produces a consistent, explainable difference; the give-away here is that the direction and size vary unpredictably.

`B` — Overflow in a double happens around 10^308 and would produce enormous errors, not a few cents.

`C` — Skipped NULLs would produce a consistent shortfall of whole order amounts, not a drift of cents.

4. Keys: how a database knows one thing from another — C

A — A real hazard, but it exists whether or not email is the primary key, and it is fixed with normalisation on write.

B — That would be a visible, immediate error the team can handle; the damaging failure here is the one that succeeds.

D — Data protection governs how personal data is handled, not which column is a primary key; this states a legal conclusion the situation does not support.

5. Where each fact should live — C

A — Applies the third-normal-form test to a column that records history rather than a current attribute, which is the standard way this rule gets misused.

B — Justifies the column by performance when the real justification is correctness; that framing invites someone to remove it during a later cleanup.

D — Reaches the right verdict by an invented rule; reference data is normally stored once, and this column is an exception for a different reason.

6. SELECT and WHERE: asking for rows — B

A — Reads the conditions as applying to the person rather than to a single row.

C — Believes grouping by student is implicit whenever a query mentions a student column.

D — Invents a restriction on AND rather than noticing that filtering happens per row.

7. Sorting, limits, and the rows that move between runs — C

A — This is a real problem with offset pagination, but it happens with brand new rows, not with the ties described here.

B — Assumes an index imposes a stable secondary order; the planner may not use that index, and equal keys have no guaranteed order within it.

D — Misstates the order of operations: OFFSET is applied after ORDER BY, not before.

8. CASE, COALESCE and computing a column that does not exist — D

`A` — CASE returns exactly one value per row; it cannot emit two labels for the same row.

`B` — Assumes CASE picks the best-matching branch the way some rule engines do; it evaluates strictly top to bottom.

`C` — Gets the boundary wrong — `> 1000` excludes 1000 — and misses the larger problem with the branch order.

9. Reading a database somebody else built — D

`A` — Worth doing, but it changes the result at the level of cents rather than materially altering the figure.

`B` — Concerns how long the query takes, not whether the number it produces is right.

`C` — Affects presentation order only; revenue is unchanged by how the rows are sorted.

10. Grain: the sentence to say before you write the query — B

`A` — The SUM reads the joined result, not the original table; the join has already duplicated the customer rows before any aggregation happens.

`C` — Only the one side is duplicated; each order still appears exactly once, so the order total is unaffected.

`D` — `SUM(DISTINCT ...)` would drop genuinely repeated values — two customers with the same limit, two orders of the same amount — replacing one error with another.

11. A right number and a useful number are not the same — C

`A` — Assumes a time window repairs the number, when the people who would change it are absent from the table entirely.

`B` — Locates the error in how cancellation is labelled rather than in who made it into the table at all.

`D` — Treats the problem as a units conversion, which would be a fair critique of a number that was measuring the right population.

12. Asking an AI for SQL without being lied to — B

A — Worries about staleness, which produces loud errors rather than quietly wrong totals.

C — Confuses slowness with incorrectness; a database returns the complete result or an error, never a partial sum.

D — Distrusts SQL for financial arithmetic in general instead of examining what this particular join did to the rows.

13. Joins: reason about them, do not memorise them — C

A — Reaches for row multiplication, the other famous join bug, which would raise the count rather than collapse it.

B — Trusts the LEFT JOIN's promise without noticing that WHERE runs after the join and can undo it.

D — Treats the problem as a choice of join type when the join itself was already doing the right thing.

14. The four joins, and what each one is for — B

A — LEFT JOIN keeps unmatched rows but does not duplicate them; NULL keys would leave the count unchanged at 31,940.

C — A cross join of 31,940 orders against a customer table would produce hundreds of millions of rows, not 2,160 extra.

D — Invents a rule; a LEFT JOIN returns at least as many rows as the left table, and exactly that many when each key matches at most once.

15. When the join key looks right and matches nothing — B

A — An index changes how fast rows are found, never which rows match.

C — Both databases treat a bare JOIN as an inner join; there is no such default difference.

D — A genuine difference between the two, but it would still leave some exact-case matches in SQLite rather than exactly zero.

16. Many-to-many, and the table in the middle — B

`A` — Close to the mechanism but misdescribes it: there is only ever one output row per course-enrolment pair, not a separate row for the course.

`C` — `count(DISTINCT *)` is not valid SQL, and distinctness is not the issue — there is genuinely one row.

`D` — Attributes the off-by-one to bad source data rather than to the join, which would also make the counts wrong for popular courses.

17. Two child tables, one very wrong total — D

`A` — DISTINCT removes rows that are identical in every column, so two shipments of the same quantity on the same day would be merged while genuinely duplicated rows with differing ids survive.

`B` — Changes what happens to unmatched rows and does nothing about matched ones; the multiplication is unaffected.

`C` — Counts each combination once, but there are still four rows per shipment in the output and any sum over them is still four times too large.

18. Joining a table to itself — B

`A` — NULL never equals NULL, so students with no phone number match nothing and drop out of an inner join entirely.

`C` — That would be a cross join; this join does have a condition, so only rows with equal phone numbers are paired.

`D` — Whitespace differences would prevent matches rather than create them, and would reduce the count rather than inflate it.

19. UNION, INTERSECT and EXCEPT — B

`A` — A column-order mismatch with compatible types would corrupt the data visibly or raise a type error, not shave a small amount off the total.

`C` — Invents a limitation; a set operation cares about column count and types, not where the rows came from.

`D` — Set operations keep NULL rows and treat two NULLs as duplicates of each other; they do not drop them.

20. Subqueries: a query inside a query — C

A — There is no such row limit; large IN lists are slow rather than empty.

B — Invents a requirement; duplicate values in the list are harmless.

D — A type mismatch would make nothing match, which for a negation would return *all* rows rather than none.

21. CTEs: naming the steps — C

A — Materialised CTEs spill to disk in both versions; the change was in whether they are materialised at all.

B — Databases do not create indexes as a side effect of planning a query.

D — Reverses the historic behaviour: the old rule was exactly-once materialisation, which is why it was called an optimisation fence.

22. Generating the rows that are not there — C

A — Would add rows for orders with no matching spine entry, which is a different problem and does not affect the count in empty months.

B — COALESCE turns NULL into a chosen value; without it the count would show NULL or 0, never 1.

D — A CROSS JOIN of distinct branches and distinct months cannot produce duplicates, and duplicates would inflate the revenue as well.

23. GROUP BY: making piles and asking about each one — C

A — Reads a repeated customer id as a data-quality defect rather than as the normal shape of repeat business.

B — Turns a choice between two questions into a rule about which function is more trustworthy.

D — Blames execution order for a gap that has nothing to do with filtering.

24. NULL, and why your counts are wrong — B

A — Treats NULL as an ordinary value that participates normally in comparisons.

C — Generalises the NOT IN behaviour, where one NULL really does poison everything, to ordinary comparisons.

D — Assumes NULL handling depends on the column's data type rather than being a rule of the logic itself.

25. Averages: the four ways AVG misleads — B

A — Treats averages as combinable like sums; they only coincide when every day has exactly the same number of orders, which never happens.

C — `avg()` promotes to a decimal type in every major engine; integer truncation is a hazard of `sum() / count()`, not of `avg()`.

D — Invents a running total that is not in the table; a daily average is computed from that day's rows only.

26. Percentiles: describing a distribution instead of its mean — C

A — Averages percentiles as if they were sums; the combined p95 depends on how many requests each server handled and where every request fell.

B — Would be true only if server B handled at least 5% of all traffic and every one of its slow requests outranked A's; nothing here says so.

D — Uses the request-count weighting that works for a mean, but a percentile is a position in a sorted list, not a weighted sum.

27. Grouping by day, week and month without lying about the boundaries — B

A — BETWEEN is inclusive at both ends; the problem is what the second endpoint means on a timestamp column, not exclusion.

C — A zone shift moves a few evening hours, not a full day, and it would affect the start of the month as much as the end.

D — Indexes are maintained on every write; an index scan returns the same rows as a table scan.

28. Rates: a numerator, a denominator, and the same grain for both — C

A — Days contribute equally to the average of rates precisely because their traffic is ignored, which is the error.

B — Assumes the direction of the error; a two-visit day can just as easily post 50% from one sale, and the fault is the equal weighting, not the sign.

D — Assumes weekend rows are NULL; the scenario says they are light, not missing, and NULL skipping would not explain a rate drawn towards them.

29. Listing the members of a pile: string_agg and array_agg — B

A — There is no row limit; the limit is on the length of the output string, and it is configurable.

C — Commas inside values would produce extra separators throughout the list, not a clean cut for only the largest customers.

D — Aggregates skip NULLs entirely rather than stopping at them, and a skipped value would not cut a reference in half.

30. The row behind the MAX: which order was the biggest — C

A — Joining on amount alone would multiply rows enormously across a thousand customers, not add 4.6%.

B — NULL never matches anything in a join condition; customers with no orders produce zero rows, not extra ones.

D — Materialisation changes how a CTE is executed, never how many rows a join emits.

31. Subtotals and grand totals in one query: ROLLUP and GROUPING — B

A — A city belongs to one region in the grouping; the multiplication comes from the extra grains, not from cities repeating.

C — Mislabelled NULLs affect a handful of rows and cannot triple a total; the tripling is structural.

D — Blank cells add nothing to a sum; the extra amount comes from real subtotal rows carrying real revenue.

32. Wide and long: pivoting rows to columns and back — B

A — FILTER evaluates a condition per row at run time; nothing about the table's creation date affects it.

C — Rows are not typed by arrival time, and re-planning happens on every execution.

D — Each FILTER tests an exact month; a quarter grouping would need to be written explicitly.

33. Reconciling: making a total tie out before you send it — C

A — GROUP BY places NULL keys in their own pile and shows them as a row; they only vanish when a filter or an inner join removes them.

B — sum() does not round; rounding, if any, happens in a display step and could not lose ₹20,400 across fourteen rows.

D — New rows would make the grouped total larger than the earlier ungrouped one, not smaller.

34. A window: GROUP BY that keeps the rows — A

B — An empty OVER means one partition holding every visible row; there is no implicit partition drawn from WHERE.

C — There is no ordering column in this window, and count(*) counts rows in every context.

D — Windows are never rewritten into GROUP BY; the rows remain and the count is simply over the filtered set.

35. ROW_NUMBER, RANK and DENSE_RANK: three answers to "what position" — D

A — The two 300s share rank 4 after the tied 400s took ranks 2 and 2, so neither is in the top three; this gets the count right by accident of the wrong mechanism and is still wrong about the 400s.

B — Describes dense_rank; rank skips position 3 entirely because two products occupy position 2.

C — Rank never discards rows; every tied row keeps its shared position and is returned.

36. Top N per group in two lines — C

A — Physical order affects which tied row the engine happens to see first, but that only matters because the ORDER BY allows the tie; the cause is the incomplete ordering, not the vacuum.

B — row_number is deterministic once the ORDER BY is complete; switching to rank changes tie handling rather than fixing the ordering.

D — Materialisation changes how a CTE is executed but never which row satisfies an ORDER BY; an incomplete ordering is arbitrary under either path.

37. Running totals, and the frame you did not know you set — D

A — Duplicated rows would double the day's contribution to every later balance, not merely show the same value on both rows of the day.

B — The window still accumulates from the start of the partition; only the rows tied at the current position are pulled in together.

C — Insertion order is irrelevant to a window with an explicit ORDER BY; the issue is the frame's treatment of equal dates.

38. LAG and LEAD: the previous row, and the change since it — B

A — Reversing the order makes first_value return the final status, which works, but for the wrong stated reason; last_value is reliable once its frame reaches the end of the partition.

C — Without ORDER BY the frame is the whole partition but "last" is then undefined, so the value returned is arbitrary.

D — Windows can see following rows whenever the frame includes them; the join-back works but is a workaround for a one-clause fix.

39. Moving averages: seven rows is not seven days — B

A — Windows do not invent rows for missing dates, and even if a NULL row existed, avg() skips NULLs rather than counting them as zero.

C — The frame is 6 PRECEDING AND CURRENT ROW, so it holds seven rows; the gap changes which dates those rows cover, not how many there are.

D — Describes a RANGE frame with an interval; ROWS counts physical rows regardless of the ordering column's type.

40. Streaks and sessions: gaps and islands — D

A — row_number never restarts inside a partition; the labels are date minus a running count, which happens to land on the day before each streak began.

B — row_number never restarts within a partition; the labels are dates, and the counter effect comes from the subtraction.

C — Missing days do not advance the row number, so the label jumps by the whole gap: 9 March minus row 4 is 5 March, not 4 March.

41. Cohorts and retention: how a month of new customers behaves afterwards — D

A — An empty group produces no row rather than a NULL count, and the group is empty because month 6 has not happened, not because nobody was active.

B — The window here only fetches the age-0 count per cohort; frame mode cannot remove an age from the grouped result.

C — Purchase-based retention can certainly reach zero; the blank is a matter of elapsed time, not of the measure's range.

42. Dates, intervals and the two kinds of timestamp — B

A — Intervals are not rounded to a fixed day count; the 28 came from February's length, not from a rounding rule.

C — An offset discarded at storage time is a one-off error of hours, not a three-day shift that appears only at the February boundary.

D — Postgres's month interval is a calendar month clamped to the month's last day, and SQLite normalises the other way, into the following month.

43. As-of joins: the value that was true at the time — B

A — Duplicate product rows would change the row count and the revenue as well as the cost; the symptom here is a shifted cost on the same rows.

C — A zone shift changes which orders are in January by a few hours' worth, not the cost applied to the ones that stay.

D — Execution plans change speed, never results; the same join on the same data yields the same rows and values.

44. Cleaning real data without destroying it — B

A — Objects to the performance of the approach rather than to what it does to the data.

C — Fixes the problem in place, destroying the raw data and making the cleaning impossible to re-run or audit.

D — Treats a matching total as verification, when the same total can be reached by removing the wrong rows.

45. Getting a file in without losing a row — B

A — No mainstream loader removes duplicates on import; deduplication is a deliberate query step, not a side effect of COPY.

C — A delimiter mismatch produces one wide column per line and the full line count, not merged rows.

D — An encoding mismatch mangles characters or raises an error; it does not silently discard whole rows.

46. Stage as text, then cast in a query you can re-run — B

A — Numbers that overflowed would have failed the load rather than wrapping, and casting a stored wrong value cannot recover digits that were never kept.

C — Normalising the text side helps, but it cannot reinstate a leading zero that the numeric side has already lost.

D — The storage is exact for the digits it keeps, but a leading zero is not a digit an integer can hold, and it is gone.

47. Parsing dates that were typed by people — C

A — The test proved only that the data cannot distinguish the formats; the locale is an assumption about the exporting system, not evidence from the file.

B — A short file, or one covering the first twelve days of a month, can be day-first and still contain no day above 12.

D — Both patterns produce valid dates for these values, so neither raises an error and a clean parse proves nothing.

48. Text that looks identical and is not — B

A — A trailing space would show the same one-character difference, but it is not the only cause and the clue points elsewhere; trim() also leaves non-breaking spaces and combining characters alone.

C — Case differences do not change a string's length; both spellings would report 12.

D — A Postgres text column has one encoding for the whole database; a mixed-encoding value would show as mangled characters, not as a clean one-character length difference.

49. Duplicates: finding them, and deciding which row survives — B

A — DISTINCT compares every selected column, not the first; ORDER BY changes presentation order, not which rows are kept.

C — DISTINCT has no notion of load batches; it compares the rows it is given regardless of origin.

D — timestamptz equality compares instants, and in any case the retries are genuinely a few seconds apart, not the same instant.

50. Assertions: queries that return no rows when everything is right — C

A — Truncated rows are absent, not present with NULLs; a not-null check can only inspect rows that exist.

B — That check looks for payments without a customer, not customers without payments, and a customer with no payment this month is normal.

D — Constraints evaluate rows being written; rows that never arrived are not written and cannot be rejected.

51. Missing values: NULL, empty, N/A, zero and minus one — C

A — Filling with the mean leaves the mean exactly where it was; that is the one statistic it does not move.

B — Filled rows carry no information about the customer; they carry the column's centre, and the model cannot tell a real average earner from a blank.

D — A scalar subquery inside coalesce is ordinary SQL and runs anywhere; the concern is statistical, not syntactic.

52. Categories: canonical values, mapping tables and encoding for a model — B

A — Integers hold billions of values; no realistic city list approaches a limit.

C — Code instability is a real maintenance problem but does not make the first model wrong; the fabricated order does.

D — Linear models accept integers freely; scaling is a separate concern and does not address the invented ordering.

53. A feature table: one row per entity, as of a moment — B

A — The feature join is bounded by `as_of`, not by the run date; the problem is on the label side.

C — `as_of` is part of the key, so the rows differ on that column even when every feature matches.

D — generate_series includes the end date when it falls on a step, and 1 March is a step from 1 January.

54. Leakage: when a feature knows the future — C

A — Skew affects model quality, but it does not produce near-perfect evaluation followed by collapse; that pattern is information available in training and absent live.

B — Invents a deletion process; the feature is wrong even when no row is deleted, because it includes orders placed after the snapshot.

D — NULL skipping changes a total slightly; it does not make a feature nearly equal to the label.

55. Why a query is slow, and what an index does — B

A — Assumes indexes drift out of date, when in fact they are maintained on every write.

C — Reaches for a type mismatch, a real cause of index problems elsewhere, when the literal here would be cast fine.

D — Applies a genuine planner behaviour to a case where the table is large and the filter is highly selective.

56. CREATE TABLE: a constraint is a test that runs on every write — B

A — Constraints can span several columns of a row; the limit is on rows, not columns, and a simpler tool exists here.

C — Compares two unrelated columns of the same row, which is meaningless here; the rule is about two different rows sharing both values.

D — EXCLUDE would work but is heavier than needed; an equality on two columns is exactly what UNIQUE expresses, and the CHECK fails for a reason this option does not state.

57. INSERT, and the INSERT you can run twice — C

A — Both forms need a unique index on the conflict column; the action taken on conflict is irrelevant to detecting it.

B — That situation produces a different error, about affecting a row a second time; this message is about the target table's constraints.

D — ON CONFLICT works on any unique index, and the subquery version is not safe under concurrent inserts; the constraint is the fix.

58. UPDATE and DELETE: rehearse it as a SELECT — B

A — There is no ordering in an UPDATE FROM; which matching row is used is unspecified, not reversed.

C — Would explain orders changing, but not why some show an address the customer no longer has; the multiple-match problem does.

D — Scan order can influence which of several matches is used, but the fault is that several matches exist; an index would not make the choice correct.

59. Transactions: making a mistake reversible — D

A — A failed statement holds no lock that blocks reads, and the message names the transaction's state, not a lock.

B — Selecting a row that does not exist returns zero rows, not an error; the message here is about the transaction, not the data.

C — COMMIT on an aborted transaction performs a rollback anyway; the required action is ROLLBACK, and the error is a state, not a counter to clear.

60. Two people, one row: lost updates and how the database prevents them — B

A — A transaction alone does not block the other request's SELECT under Read Committed; both still read 500 and one update still overwrites the other.

C — Speed does not remove the gap between a read and a write; two fast requests can still interleave in the same way.

D — Under REPEATABLE READ the second transaction would fail with a serialization error rather than see the new value; it still needs the atomic form or a retry.

61. Changing a table that is in use — B

A — A nullable column with no default is metadata only; existing rows are not touched.

C — Clients adapt to a new column without reconnecting; the outage pattern here is a lock queue, not a reconnection storm.

D — Indexes contain only the columns they were defined on; a new column is not added to them.

62. Views: a saved question, not a saved answer — C

A — A plain view does; a materialised view stores rows from its last refresh, and 10:00 is after 02:00.

B — Possible in principle, but the stated refresh schedule explains the gap without any assumption about the order's transaction.

D — A refresh re-runs the whole query and replaces the stored rows; the order will appear at the next refresh.

63. JSON in a column: when it helps and what it costs — C

A — jsonb preserves values exactly; it neither lower-cases nor alters text.

B — A column has one type; a jsonb column cannot hold some rows as plain text.

D — Indexes are updated on every write and cannot return a subset of matching rows; a stale index is not a Postgres failure mode.

64. Keeping history: soft deletes, audit rows and who changed what — D

A — They are compatible with a partial index; hard deletion throws away the history the soft delete was introduced to keep.

B — Moves an invariant out of the database, where it is enforced for every writer, into code that only some paths run.

C — Mutates historical data to work around a constraint, and every join or audit on the old email now fails to match.

65. Who may do what: roles, GRANT and a read-only door — B

`A` — Grants never expire on their own; there is no inactivity rule.

`C` — Ownership does not restrict reads to the owner; a grant to another role is all that is needed, and it was simply never made for that table.

`D` — Row-level security is off unless explicitly enabled, and the error for a missing policy is an empty result, not a permission error.

66. From a paragraph to a schema: a tuition centre — C

`A` — numeric holds negatives freely; the objection is about the value being derived, not about its type.

`B` — A nullable column, or one with a constant default, is a metadata change; and the table is small in any case.

`D` — True but minor; the real hazard is a stored value that can disagree with the payments it summarises.

Module test — 1. Tables, types and the question you are actually asking

1. A

WHERE walks the table one row at a time and evaluates the condition against the values in that row alone. It cannot see the other rows belonging to the same student. Ade's cash row fails the amount test and his large row fails the method test, so neither survives. "Students who have paid cash and who have some payment over 5,000" is a different question about a pile of rows, and it needs the grouping from a later module.

2. B

With an ORDER BY on a column that ties, the order among the tied rows is undefined, so the twenty you get are an arbitrary twenty of the forty and can change between runs. Adding a unique final sort key makes the ordering total, and nothing can move. An index does the opposite of fixing it: adding or rebuilding one is a common reason a result that had looked stable for months suddenly comes back in a different order.

3. C

CASE tests its branches strictly top to bottom and returns the first one that is true. There is no fall-through and no error. A payment of 30,000 satisfies `amount >= 5000`, that branch matches, and the evaluation stops there. This is the most common CASE bug precisely because it produces a plausible-looking result rather than a failure, so the order the conditions are written in is part of the logic, not a matter of style.

4. B

A join emits one output row per match, so a customer with three orders appears three times and her credit limit appears three times with her. The grain of the result is one row per order, and any customer-level column summed at that grain is inflated once per order. Nothing errors, because the query is valid; the defence is to say what one row of the result is before writing it, and to aggregate the many side first when the answer must stay at customer grain.

5. A

Binary floating point represents numbers as sums of powers of two, and one tenth is not one. Each of the three values is stored as the nearest representable number instead, and the sum of the first two lands a fraction away from the third. It is not a fault in the database. It is why money uses `numeric`, which stores decimal digits exactly, and why summing a hundred thousand float prices produces a total a few paise from the truth in a direction nobody can predict.

6. C

`SELECT status, count(*) FROM orders GROUP BY status` takes seconds and routinely reveals that the table contains abandoned baskets, cancellations, test orders and soft-deleted rows, none of which anybody will mention. Those columns silently filter, or fail to filter, everything. A diagram is a document somebody stopped updating; DISTINCT hides a multiplication rather than finding it; and a thirty-day window answers a different question from a calendar month.

Module test — 2. Putting tables together

1. A

The LEFT JOIN did its job and kept every order, filling the refund columns with NULL. Then WHERE asked whether NULL differs from 'damaged', which is unknown, and WHERE keeps only rows where the test is true. The unmatched orders are discarded one step after the join carefully preserved them. A condition on the right-hand table of a LEFT JOIN belongs in ON; a condition on the left-hand table belongs in WHERE.

2. B

The reliable detection is structural rather than numerical. Two independent one-to-many joins from the same parent multiply each other, whatever the numbers look like, and the query text shows this if you list the tables and write "one" or "many" beside each. Duplicate output rows need not appear, because the children's columns differ; a bigger total may be real growth; and DISTINCT changing the answer tells you something multiplied without telling you which join did it.

3. C

Bracketing a value and printing its length exposes a trailing space, a carriage return, a leading zero and a prefix in one look, and comparing the two sides side by side usually names the cause immediately. Cleaning both sides blindly may work and hides what was wrong; a LEFT JOIN changes what is returned without diagnosing anything; and an index affects speed, never whether two values are equal.

4. C

UNION concatenates and then removes rows that are identical in every column, which is a data-destroying operation when the rows are genuine repeats rather than accidental copies. Nothing indicates that it happened. UNION ALL keeps everything and is the one to reach for by default; if the rows carry an id, including it also removes the problem, because the ids differ.

5. D

A join emits one output row per match, so a student with four payments appears four times and the count of students is wrong unless you deduplicate afterwards. EXISTS answers a yes-or-no question about the outer row and stops at the first match, so the grain is untouched and no DISTINCT is needed. It also handles NULLs correctly, which `NOT IN` against a nullable column does not.

6. A

The spine row for a month with nothing in it still produces one output row, with every column from `orders` set to NULL. `count(*)` counts rows and so reports 1. `count(o.id)` counts non-NULL values of that column and reports 0, which is the truth. This is the same distinction that makes an empty category appear to have one member in every left-joined coverage report.

Module test — 3. Summaries, and the piles behind them

1. B

The two counts answer different questions and both are correct. `COUNT(*)` counts rows in the pile, which at this grain is orders. `COUNT(DISTINCT customer_id)` counts how many different customers appear in it. A repeat customer with four orders is one customer and four orders at the same time. Neither figure is more right; the skill is knowing which question you were asked.

2. D

Every comparison with NULL yields unknown rather than true or false, and WHERE keeps a row only when its condition is true. The people who never set a status are silently dropped, and the number that comes back is simply smaller than the truth with nothing to signal it. Write `WHERE (status <> 'churned' OR status IS NULL)` when the unknowns belong in the answer. Note that COUNT(*) itself counts every row in the pile, NULLs included; it is COUNT(column) that skips them.

3. A

An average of averages weights each group equally regardless of how many rows it summarises, so it answers a different question — what is the average branch's average. A summary table that stores an `avg_order_value` column cannot be averaged up to a monthly figure; it has to store the daily sum and the daily count so that the period's mean is `sum(total) / sum(orders)`. Groups can of course be combined; they have to be combined from components.

4. C

An exact percentile needs the raw values, sorted. It depends on how many requests each server handled and on where every individual request fell, and none of that survives in the two summary numbers. This is a mechanism, not a policy: it is why a summary table holding an hourly p95 cannot give you a daily p95, and why the large engines offer approximate quantile sketches, which are mergeable and wrong by a small stated amount.

5. B

126 conversions over 4,012 visits is 3.1%. Averaging the two daily rates gives 26.5%, because a day with twelve visits is given the same weight as a day with four thousand. Keep the numerator and denominator as separate columns, sum each, then divide — guarded with `nullif` so an empty day yields NULL rather than an error — and print the denominator beside the result.

6. D

ROLLUP marks a rolled-up column with NULL, and a sale whose city was never recorded also produces NULL at the detail grain. `GROUPING(city)` returns 1 for the subtotal row and 0 for the detail row, so a CASE can label one "All cities" and leave the other as the unknown it is. Without it, a report relabels its missing data as a subtotal. Note also that a ROLLUP result is three grains stacked, so summing its revenue column counts every sale three times.

Module test — 4. Windows, sequences and time

1. A

A window function is computed after WHERE, GROUP BY and HAVING have finished, so when WHERE runs the column does not exist yet. Compute the ranking in a CTE and filter outside it. The same ordering explains a quieter fact: because the window sees the rows WHERE has already kept, `sum(amount) OVER ()` in a query filtered to Pune is Pune's total and not the company's.

2. B

Adding an ORDER BY inside a window introduces a frame, and the default is written `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. In RANGE mode "current row" means the current row and every row tying with it in the ORDER BY, so both day-2 rows read 60. "Cumulative revenue by the end of each day" wants exactly that; "a running balance after each transaction" wants ROWS. State the frame whenever the ordering can tie.

3. C

The default frame runs from the start of the partition up to the current row, so the last value inside the frame is always the current one. Extend the frame with `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` and it returns what you meant. `nth_value` behaves the same way for the same reason, and `first_value` happens to work only because the frame's start is already where you want it.

4. D

dense_rank gives 1, 2, 2, 3, 3, so a filter at three admits both 500s and returns all five rows. rank gives 1, 2, 2, 4, 4 and returns three; row_number gives 1 to 5 and returns three, having dropped one of the tied 700s on the strength of a tiebreaker. None is more correct: "show me three" is row_number, "everything in the top three positions" is dense_rank, and counting the rows a ranking picked is how you find out which you built.

5. A

A ROWS frame counts rows, not days, so "6 PRECEDING" reaches back to whatever seven rows exist and the window silently spans a different number of days at each point. Either build a spine of every date and coalesce the missing ones to zero before the window, or use a RANGE frame measured in an interval, which includes every row within six days however many that is. The two produce different lines exactly where the business was closed, and the caption has to say which.

6. B

`products.cost` is what the product costs now, and the question was about what it cost then. Every time the cost changes, every historical margin the report computes changes with it. The two honest answers are to copy the cost onto the order row when the order happens, the way an invoice carries its own tax rate, or to keep an effective-dated cost table with half-open periods and join through the period containing the order's date — having first checked that table for overlaps and gaps.

Module test — 5. Data that arrives dirty

1. B

Postgres aborts the entire COPY on the first value that does not fit, leaving you with nothing; a loader that guesses from a sample fails or silently changes its mind when row 20,001 breaks the pattern. Loading as text cannot fail, so every row is on disk exactly as it arrived, and each decision then happens in a view beside a query counting the rows that needed it. It also stops a phone number losing its leading zero, which no later query can recover.

2. A

A first part above 12 cannot be a month, so the column is day-first. If neither part exceeds 12 — a small file, or one covering only the first twelve days of a month — the data cannot tell you and the only honest step is to ask whoever exported it. Counting successful parses proves nothing, because 03/04 is valid under both readings; a future maximum is a good after-the-fact alarm but not a test you can run before choosing.

3. B

It is wrong in both directions at once. A retry recorded four seconds later differs by one timestamp, so DISTINCT keeps both; two genuinely separate payments of the same amount by the same customer are identical in every selected column, so DISTINCT merges them and steals money from the report. Deduplication needs a stated key and a stated rule for which row survives, which is `row_number()` over a partition with a tiebreaker, and a count of the rows it dropped.

4. C

Every filled row sits exactly on the centre, so the mean survives untouched and the variance falls. Anything that depends on variability — a confidence interval, a correlation, a model's certainty — is now overstated, and nothing in the output shows it. The fill that loses least keeps the fact of the missingness: fill in the feature table, next to an `income_missing` flag, so that both a model and a reader can see how much of the column was invented.

5. C

Row-level checks examine the rows that are there, and every one of them is fine; the fault is in the rows that are not. A distribution check — flag a load whose count is below 70% or above 150% of the recent average, and do the same for the total, the distinct customer count and the share of NULLs — is the only thing that sees a file that is silently half missing. Thresholds should be loose: the point is to be asked, not to be blocked.

6. C

A customer whose total spend kept rising after the snapshot date is, by definition, not one who churned, so the model learns to read its own label. In production the future orders do not exist and the feature means something else entirely. The fix is a bound on every join: `o.placed_at < s.as_of`, every time, plus a check query that returns no rows when no feature source row is at or after its own snapshot.

Module test — 6. Changing data safely

1. B

The database detects the conflict by attempting to insert into a unique index, so without one there is nothing to conflict with and it will not guess which column made a row a duplicate. The fix is the constraint, not a different statement. A source containing the same reference twice is a different and equally useful error — "cannot affect row a second time" — which tells you the file has a duplicate before it gets in.

2. A

If the FROM matches more than one row per target row, Postgres updates the target with one of them and does not say which, and other engines are equally silent. So the rehearsal is a count of matches per target row: join the two tables, group by the order's id, and keep the groups whose count is above one. It must return nothing. The other half of the rehearsal is running the WHERE clause as a SELECT and comparing its count with the count the UPDATE reports.

3. C

Rather than let you commit a unit of work whose middle went missing, Postgres refuses everything until ROLLBACK ends the block. Scripts that continue past a failed statement meet this and appear to be broken everywhere at once. `SAVEPOINT` gives a partial undo: `ROLLBACK TO before_risky` discards only the work since the savepoint and lets the transaction carry on, which is how batch loaders skip a bad row and keep going.

4. D

The database reads and writes the row under a lock held for the statement, so nothing can slip in between. The second request sees the row after the first has written, the WHERE clause fails, no row is returned, and RETURNING tells the application it lost. SERIALIZABLE also works but converts the collision into an error the application must catch and retry; a unique constraint prevents a double enrolment for one student and not two students taking one seat.

5. B

Adding a nullable column with no default is a catalogue change of a few milliseconds. Adding NOT NULL to an existing column scans the whole table under the strongest lock, and a default that is an expression rewrites every row. `NOT VALID` applies the rule to new rows at once; `VALIDATE CONSTRAINT` checks the old ones while reads and writes continue; `SET NOT NULL` is then instant. Set a lock timeout first, so a queued ALTER cannot take the site down behind one long report.

6. C

A view stores a question and re-runs it, so it is always current and never faster than its query. A materialised view stores the answer, so reads are instant and the data is exactly as old as the last refresh. That staleness is a property rather than a defect, and it belongs on the page: "the dashboard is wrong" and "the dashboard is six hours old" are different conversations, and only one of them needs an engineer.

Data, SQL and Getting to the Answer — final examination

1. B ***1. Tables, types and the question you are actually asking***

Size is the least important of the reasons and the only one people notice. The three that matter are concurrent writers, the same fact stored in more than one place, and needing the data to be trustworthy in a year. None of them produces an error message when it is crossed; they produce a number that is quietly wrong. Below those lines a spreadsheet is faster than a database and is the right tool.

2. A ***1. Tables, types and the question you are actually asking***

A SQLite column suggests a type rather than enforcing one, so the refusal a type is supposed to make does not happen. Everything else in this course transfers; that one protection does not, until you declare the table with `STRICT` in a recent version, which you should. It is the reason a query that works against SQLite can meet a wall of type errors on Postgres.

3. C ***1. Tables, types and the question you are actually asking***

`COPY` runs as the database server and looks on the server's disk, which on a hosted database is a machine you have never seen. `\copy` runs in the psql client, reads your file, and streams it to the server. On a laptop they are the same disk and the distinction is invisible, which is why almost every first import meets this error somewhere else.

4. B ***1. Tables, types and the question you are actually asking***

A primary key is a statement of fact about the world. This one claims the pair is unique, so the second term's enrolment is refused or, worse, quietly folded into the first. The real key includes the term. Getting this decision right removes a whole family of duplicate-row bugs before they exist; getting it wrong is discovered when somebody cannot enrol.

5. C ***1. Tables, types and the question you are actually asking***

You should not be able to make a year of payments vanish by tidying up a student record, so the delete is refused until somebody decides what should happen to the money. CASCADE is right for rows with no meaning alone, such as the items of a deleted draft order. SET NULL is right when the fact outlives its owner, and it forces every later query to handle the NULL.

6. D ***1. Tables, types and the question you are actually asking***

`current_price` is what the product costs now; `unit_price` is what this customer was charged, which is a fact about that line of that order. Normalise it away and every historical invoice rewrites itself the next time somebody runs a sale. The same reasoning covers a delivery address copied onto a shipped order and a tax rate recorded on an invoice: anything that was true at a moment and must stay true belongs to the event.

7. A ***1. Tables, types and the question you are actually asking***

Exactly as multiplication binds more tightly than addition, the condition means `country = 'NG'` OR `(country = 'KE' AND amount > 5000)`. The database is not wrong; you did not say what you meant. Put brackets in whenever AND and OR appear together, even when you are sure, because the cost of being wrong is a plausible number rather than an error.

8. C ***1. Tables, types and the question you are actually asking***

`NULLIF(a, b)` returns NULL when the two are equal, so the division yields NULL rather than aborting. NULL is the honest answer: a branch with no attempts does not have a rate of zero, it has no rate. Filtering the rows away hides the branches with no activity, which are often the ones worth seeing, and COALESCE to zero states something false.

9. A ***1. Tables, types and the question you are actually asking***

DISTINCT deduplicates the entire selected row, not the first column, so adding a unique column defeats it completely. This is worth knowing for a second reason: reaching for DISTINCT to make a count look right hides a join that matched twice, and while it is doing so it also merges genuinely different rows that happened to look alike.

10. B ***1. Tables, types and the question you are actually asking***

The people who left hardest are the ones no longer in the table, so the population is defined by having survived and cannot be used to measure survival. A date filter does not help, because the missing rows are missing before any filter runs. The repair is a population defined by who existed at the start of the period, from a table that keeps them.

11. B ***2. Putting tables together***

This is the anti-join: join everything, then keep the failures. An inner join has already discarded exactly the rows you are looking for, so no WHERE can bring them back. Once you have seen the pattern you will use it constantly — students with no payment this month, products never sold, users who never opened the app — and `NOT EXISTS` says the same thing in a different shape.

12. A ***2. Putting tables together***

NATURAL JOIN joins on every column whose name appears in both tables. Add `created_at` to both and the join silently begins requiring those timestamps to be equal, so the result empties out with no error. It reads beautifully and it is a trap; use explicit `JOIN ... ON`, or `USING (customer_id)`, in anything that has to keep working.

13. B ***2. Putting tables together***

A comma join with no WHERE is a cross join: 31,940 orders against 8,200 customers is 261 million rows. Explicit `JOIN ... ON` syntax puts the condition next to the join, so a missing condition is a syntax error rather than a query that runs for an hour. A cross join is a legitimate tool — it is how you build a spine of every branch-month — but it should be the one you asked for.

14. C ***2. Putting tables together***

Reconciliation has three failure modes — in the bank and not the ledger, in the ledger and not the bank, in both but disagreeing — and only a full outer join can show all three at once. A left join from the ledger cannot see a payment the bank has and the ledger never recorded, which is usually the interesting one. MySQL has no FULL OUTER JOIN; the workaround is a LEFT JOIN unioned with a RIGHT JOIN.

15. D ***2. Putting tables together***

No single row contains both courses, so a row-level AND can never be true. `GROUP BY student_id HAVING count(DISTINCT course_id) = 2` after filtering to the two courses says exactly what is meant, and it generalises: change the list and the number and it answers "all of these" for any number of courses. A self-join works for two and becomes unmanageable at four.

16. A ***2. Putting tables together***

A list crammed into a cell means a missing table. Every question becomes a string search, `LIKE '%stats%'` also matches 'biostats', adding a course means rewriting a string transactionally in every affected row, and no index helps. The fix is to unpack it into a bridge table once — one row per student per course — after which both directions of the question are an ordinary GROUP BY.

17. D ***2. Putting tables together***

Somebody is transitively their own manager, which happens in real HR systems after a reorganisation, so the rule keeps producing rows forever. Add `WHERE depth < 20`, or carry the path travelled so far and refuse to revisit a node. Add the guard when you write the query, not after the incident; a missing anchor is a syntax problem you find immediately, which is a much kinder failure.

18. B ***2. Putting tables together***

`A EXCEPT B` returns rows in A that are not in B, and says nothing about rows in B that are not in A. Those are two different findings, and a reconciliation that ran only one direction has answered half the question. Empty results in both directions are a proof rather than an impression, which is what makes this pair the fastest verification tool for a migration.

19. C ***2. Putting tables together***

Since Postgres 12 a CTE referenced once may be inlined, but if it is materialised the whole large table is read before the outer filter runs. `NOT MATERIALIZED` pushes the query together so the index can be used. `MATERIALIZED` is the opposite lever, for a CTE that is expensive and referenced several times. Older Postgres treated every CTE as a fence, which is why old advice on this point is confidently wrong.

20. D ***2. Putting tables together***

A scalar subquery returning two rows is an error, not a silent duplication, which is a far better failure than a join's. That property is why `(SELECT sum(amount) FROM payments WHERE order_id = o.id)` is a safe answer to the fan trap when there are only one or two values to attach. The cost is that on some engines it runs once per outer row, which is fine for a thousand orders and painful for a million.

21. A ***3. Summaries, and the piles behind them***

WHERE filters rows on their way into the piles; HAVING filters whole piles on their way out. The sum of a pile does not exist when WHERE runs. The converse is worth stating too: a row-level test such as `status = 'paid'` can legally sit in HAVING on some engines, and should not, because it reads as a statement about the group rather than about an individual order.

22. B ***3. Summaries, and the piles behind them***

There is no right answer to "which of the 900 names should go there", so the database refuses to invent one. Every column in the SELECT must be either grouped by, and so the same for the whole pile, or wrapped in an aggregate that squeezes the pile into one value. MySQL in some configurations picks a name at random instead, which is worse rather than more convenient.

23. A ***3. Summaries, and the piles behind them***

Aggregates skip NULLs, so the denominator quietly becomes the number of people who answered. The figure is the average among respondents who chose to answer, and if low earners were likelier to skip — which is what happens in real surveys — it is biased upward with nothing in the output to hint at it. Report the count of non-NULL values beside the average, always.

24. C ***3. Summaries, and the piles behind them***

`id NOT IN (1, 2, NULL)` expands to `id <> 1 AND id <> 2 AND id <> NULL`, and that last term is unknown, so the whole conjunction can never be true for any row. Zero rows looks like a real finding — "every student has paid" — which is what makes this the most expensive NULL trap. Use `NOT EXISTS` or the anti-join, both of which behave as you expect.

25. D ***3. Summaries, and the piles behind them***

`paid / total` is integer division and truncates to 0 before the 100 is applied. Cast one side — `paid::numeric / total * 100` — or use `avg()`, which promotes internally. The engines disagree about this: `SELECT 7 / 2` is 3 in Postgres, SQLite and SQL Server, and 3.5 in MySQL and DuckDB, so a query checked on one and run on another can change meaning without changing text.

26. C ***3. Summaries, and the piles behind them***

Postgres and DuckDB start a week on Monday, following ISO; MySQL's WEEK() defaults to Sunday and has eight modes; Excel and Sheets default to Sunday. Two people can therefore disagree about every single week and both be right by their own tool. Group with `date_trunc('week', ...)` and show the week's start date rather than a week number, which also avoids the ISO week that straddles the new year.

27. A ***3. Summaries, and the piles behind them***

MySQL's `group_concat` stops at `group_concat_max_len`, which defaults to 1,024 characters, and returns the truncated string with no error. Raise it for the session before any query that could produce a long list, and treat any exported list whose length is exactly 1,024 as suspect. It is also a signal that a person does not want this list at all: they want a count, a sample, or the first ten.

28. C ***3. Summaries, and the piles behind them***

Joining back on the maximum value returns every row that hit it, so ties produce two rows for one customer. Sometimes that is the truth you want; when it is not, you need a stated tiebreaker, which a join cannot express and `DISTINCT ON`, `arg_max` or `row_number()` can. Counting output rows against the number of groups is the check: more means ties leaked through, fewer means some groups had only NULL amounts.

29. B ***3. Summaries, and the piles behind them***

The result's shape is part of the query's type and is settled before execution, so data-driven columns would mean the shape of the result depended on the result. The escape hatches confirm the rule: Postgres's `crosstab()` still needs the column list declared, and DuckDB's `PIVOT` discovers the months by running one query to list them and then building a second query from that list. Store and compute long; pivot only for display.

30. C ***3. Summaries, and the piles behind them***

GROUP BY gives NULL its own pile, so the missing 20,400 should have appeared as a row with a NULL key unless something removed it — a `WHERE city IS NOT NULL`, or an inner join to a cities table. Either way the report now silently excludes that money and the reader assumes the fourteen rows are everything. Making the parts sum to the whole is the first of the four reconciliation checks, and it is one query.

31. B ***4. Windows, sequences and time***

PARTITION BY is GROUP BY without the collapsing: each row keeps its identity and gains a column computed over the other rows in its partition. That is what makes "this order's share of its city's revenue" a single expression rather than a CTE and a join back. GROUP BY can also produce several aggregates at once; what it cannot do is return the individual orders.

32. A ***4. Windows, sequences and time***

Window functions are computed after WHERE, GROUP BY and HAVING have finished, so the window's universe is whatever survived the filter. If you need the company total on Pune's rows, the filter has to move to a later layer or the total has to come from a subquery. The same ordering explains why a window's alias cannot be referenced in WHERE.

33. C ***4. Windows, sequences and time***

row_number must give every row a different number, so it breaks a tie on whatever the engine happens to do, and two runs can disagree. In a report that is a cosmetic wobble; in a deduplication it decides which row survives and which is deleted. End the window's ORDER BY with something that cannot tie — the primary key, or `ctid` in Postgres — so the rule is one you can state and repeat.

34. C ***4. Windows, sequences and time***

`ntile` cuts by count, so the top tenth of customers can hold any share of revenue at all, and on real retail data it usually holds a lot. That is a finding, not a fault. If you want groups of equal revenue, compute a running total, divide by the partition total, and cut at each tenth of the cumulative share. When the rows do not divide evenly, the earlier tiles simply take the extra.

35. A ***4. Windows, sequences and time***

dense_rank shares a position without skipping, so every product at the third distinct sales level is admitted however many there are. row_number returns exactly three and drops a tied product on the strength of a tiebreaker; rank returns the tied rows but, if two products tie for first, admits only positions 1, 1 and 3. Say the sentence about ties before choosing the function.

36. D ***4. Windows, sequences and time***

The grid is a triangle: a four-month-old cohort has ages 0 to 4 and nothing beyond. Reading along a row compares cohorts at different ages, or reads whatever the pivot put in an empty cell. Read down a column — the same age across cohorts — and only as far as every cohort in the comparison has actually reached. A "retention is falling" headline is very often a triangle read along the wrong edge.

37. D ***4. Windows, sequences and time***

A running distinct count cannot be computed by adding one row's contribution to the previous total, which is why the window form does not exist. Flagging each customer's first order with `row_number() OVER (PARTITION BY customer_id ORDER BY placed_at) = 1` turns the question into an ordinary running count over those rows, and that line is the acquisition curve on every growth chart.

38. B ***4. Windows, sequences and time***

`lag` looks back one row, not one month. With March absent, April's predecessor is February, and a two-month change is reported as if it were one. The fix is to join to a spine so that every month exists as a row before the window runs, which is the usual answer, or to look the value up by date. The same trap turns a seven-row moving average into an average over seven trading days.

39. B ***4. Windows, sequences and time***

`timestamp` without time zone is a wall-clock reading with no information about where the clock was, so the offset is thrown away on the way in and no query can get it back. `timestamptz` stores an instant and converts on the way out. Store `timestamptz`, keep the session zone at UTC, and convert for display with a named zone rather than a fixed offset, because named zones carry the history of when the rules changed.

40. D ***4. Windows, sequences and time***

Overlapping periods give an order two cost rows and the join multiplies; a gap gives it none and an inner join drops it. Both produce a plausible total. `lag(valid_to) OVER (PARTITION BY product_id ORDER BY valid_from)` compared with `valid_from` finds both, and the checks belong beside the load, because the person adding a new price does not think of themselves as editing a join. A NULL `valid_to` is normal: it means still current.

41. B ***5. Data that arrives dirty***

A quoted field may contain a line break — an address, a comment, a product description — and `wc -l` counts that as two lines while a CSV parser counts one row. That is not a fault. The reverse is: when the parser is not told the field is quoted, one row becomes two broken ones and the second half lands in the wrong columns. Count before, count after, and explain the difference before writing any query.

42. A ***5. Data that arrives dirty***

Excel in a German, French or Brazilian locale writes semicolons between fields. A loader expecting commas reads each line as one field, which is the one case where the row count passes and the load is still useless. Look at the first line of the file with your own eyes before loading it, and state the delimiter, quoting, header and encoding rather than letting the loader guess.

43. D ***5. Data that arrives dirty***

In UTF-8 the letter é is the two bytes C3 A9. Read those with Windows-1252 and they decode as Ã and ©. The bytes were fine; the table used to interpret them was wrong. Detect the encoding with `file -I`, tell the loader, or convert with `iconv` first. Fixing the names afterwards with find-and-replace produces a table where some are repaired and some are doubly broken.

44. B ***5. Data that arrives dirty***

Excel writes the three bytes EF BB BF at the start of a UTF-8 file. A loader that does not strip them makes the first column's name begin with an invisible character, so it looks like `id` and is not `id`. It is a small, specific fault with a specific cure — strip the mark, or tell the loader the encoding is UTF-8 with BOM — and it is worth recognising on sight.

45. A ***5. Data that arrives dirty***

`0221234567` becomes `221234567`, and no later query can tell whether the zero was there. The same applies to postcodes, account numbers and any code with a leading zero or a plus sign, and to a sixteen-digit reference loaded as a double, where two different references can round to the same number. The others are recoverable because the raw text staging table still holds the original.

46. B ***5. Data that arrives dirty***

`length` shows one Lagos of five characters and one of six; the hex dump shows `c2a0` at the end, which is a non-breaking space. `trim()` does not remove that, nor a tab, nor a zero-width space, so the diagnosis has to come before the fix. Lower-casing tells you only whether case was the problem, and it will not have been if both strings read `Lagos`.

47. C ***5. Data that arrives dirty***

é can be one code point, U+00E9, or `e` followed by a combining acute accent. They render identically, have different lengths, and are different strings to `=`. Files from macOS often carry the second form and files from Windows the first. `normalize(name, NFC)` in Postgres, applied once in the clean view to every text column, removes the problem for everything downstream.

48. A ***5. Data that arrives dirty***

`WHERE m.variant IS NULL` is the to-do list: every value the map has not yet seen, with its frequency, and it gets shorter every month. A new city at prediction time maps to nothing and comes through unchanged rather than crashing the pipeline. It is also a table, so a colleague who cannot read SQL can add a row, which a CASE expression buried in a query does not allow.

49. C ***5. Data that arrives dirty***

A linear or distance-based model reads the numbers as a scale, and the cities have no such order. Integer codes are correct only for genuinely ordered categories — small, medium, large; bronze, silver, gold — and then the order must be the one you assign rather than the alphabetical one a tool picks. For unordered categories use one-hot encoding when the cardinality is small, and frequency encoding when it is not.

50. D ***5. Data that arrives dirty***

This is train/serve skew: a feature computed one way for training and another way live — often because a nightly job fills a column three hours after the prediction runs. If the same parameterised query cannot run for today, it should not be the training query. A larger test set drawn from the same table cannot see the difference, because both sides of it were built the same wrong way.

51. A ***6. Changing data safely***

A CHECK may reference several columns of the same row — `valid_to IS NULL OR valid_to > valid_from` is fine — and nothing else. A rule about two rows needs `UNIQUE` when it is "at most one per combination", Postgres's `EXCLUDE USING gist` when it is "no two may overlap", or a trigger beyond those. A rule in the database is enforced for the administrator fixing data by hand at midnight; a rule in the application is not.

52. B ***6. Changing data safely***

Balance is the agreed fee minus the payments, and every derivable number stored is a number that can disagree with its source. The concurrent-writes lesson shows exactly how: a payment recorded without the matching update, or two updates racing, and afterwards nobody can say which figure is real. Computed in a query it is always right, and the query is three lines because the schema has one child table in it.

53. D ***6. Changing data safely***

A generated column is computed from the document on every write, can be indexed like any other column, and fails the insert when the cast fails — which is the type check the JSON never had. The document stays for the attributes nobody has needed yet. A GIN index is the right tool for finding documents containing an unknown fragment, and is larger and slower to update than an ordinary index on a known key.

54. B ***6. Changing data safely***

One enormous DELETE runs for an hour, holds its locks throughout and writes a transaction log the size of the table. In slices, other users proceed between them, and a mistake noticed after the third slice has cost thirty thousand rows rather than two hundred million. The version that costs nothing at all is dropping a partition of old rows as a metadata operation.

55. C ***6. Changing data safely***

`FOR UPDATE` closes the gap between a read and a write by making the second reader queue, and the queue lasts as long as the transaction. A lock held while somebody thinks is a course nobody can enrol in, and a transaction left open over lunch is a common cause of a pile of stuck requests. Rehearse outside the transaction, then open it, write, check the count and commit.

56. A ***6. Changing data safely***

A deadlock is two transactions holding what the other wants. If every transfer touches its two accounts in the same order, they queue instead of crossing, and the cycle cannot form. Postgres detects a deadlock within a second and kills one, so retrying works; it just does not stop it happening. A table lock also works and turns concurrent transfers into a single queue.

57. C ***6. Changing data safely***

The deleted row is still in the table, so the address is still taken, forever. The fix is a partial unique index — `CREATE UNIQUE INDEX ON customers (email) WHERE deleted_at IS NULL` — which enforces uniqueness only among the living. The other thing soft deletes break is that the `WHERE deleted_at IS NULL` filter now has to appear in every query, which is why it belongs in a view instead.

58. D ***6. Changing data safely***

An audit that depends on every code path remembering to call it is an audit with gaps, and the gaps are exactly the unusual changes worth auditing. A trigger fires for the application, for a migration, for a script and for an administrator at a prompt. Storing the old and new rows as JSON keeps it working when the table gains a column, which the previous lesson's JSON type makes straightforward.

59. A ***6. Changing data safely***

`GRANT ... ON ALL TABLES` covers the tables that exist at that moment and nothing afterwards. `ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO analyst` covers the ones created next month. It is the line people forget, and the symptom is an analyst who loses access to each new table until somebody remembers.

60. B ***6. Changing data safely***

Treat it as a user that types fast, does not read the schema carefully and cannot be embarrassed. Read-only removes the write accidents; a statement timeout kills the query with the accidental cross join rather than leaving it to run for an hour; views hiding phone numbers and email addresses matter because a model that has seen a personal detail may repeat it. A replica with superuser rights is still superuser rights.