BUILD WITH IT

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.

First edition, September 2026

Addaly

Series editor: Dr Mustafa Mun

Draft, open for academic review

BUILD WITH IT

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.

Series editor: Dr Mustafa Mun

Addaly

First edition, September 2026 · Draft, open for academic review

Data, SQL and Getting to the Answer

© 2026 Addaly.

Licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0). You may copy, print, share and adapt it for any non-commercial purpose, with credit, under the same licence. creativecommons.org/licenses/by-nc-sa/4.0

First edition, September 2026, build 62a26075.

Written by AI models to a written standard, with Dr Mustafa Mun as series editor. Exam keys checked blind by a second model: 3,665 of 3,666 agreed. Academic review invited.

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.

How to cite: *Mun, M. (2026). Data, SQL and Getting to the Answer. Addaly. addaly.com/learn/data-and-sql.*

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.

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. The answers to every check, test and examination question are at the back.

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

Brief contents

1

MODULE 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.

Lesson 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.

Figure 1.1

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

The answer and why: p. 365

Lesson 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('payments.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.

The answer and why: p. 366

Lesson 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.

CASE STUDY · MODULE 1

The fee sheet that could not be trusted twice

An illustrative case, built from documented patterns.

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.

YOUR ANSWERS

1. 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?

2. 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.

3. 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?

STOP HERE

Write your answers before you turn the page. How it played out starts on p. 64.

CASE STUDY · MODULE 1

How it played out

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 permanent 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.

The questions, discussed

1. 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.

2. 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.

3. 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. The answers, and the reason behind each, are in the key on p. 364. If two go wrong, re-read the module before the exam.

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

nothing.

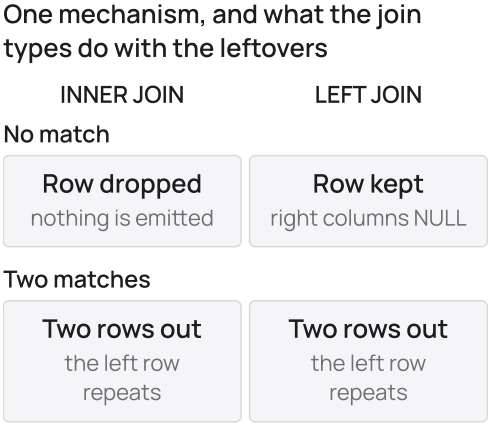
- **INNER JOIN**: unmatched rows on either side are dropped.
- **LEFT JOIN**: every left row survives; unmatched ones get NULLs on the right.
- **RIGHT JOIN**: the mirror image. Every right row survives.
- **FULL OUTER JOIN**: everything from both sides survives, with NULLs wherever there was no match.

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.

Figure 2.1



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
million rows
```

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
- changing data safely, from constraints and transactions to permissions

60 questions, the whole pool. The site draws 25 for a sitting, pass mark 70%. Work any 25 under your own conditions. Answers from page 406.

1 Module 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 Module 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 Module 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 Module 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 Module 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 Module 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

Module 1 · Tables, types and the question you are actually asking

The module starts on p. 9.

MODULE 1 TEST

Module 1 test, Q1, p. 66

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` …

Answer A: No single row of his is both cash and over 5,000

Why: 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.

Module 1 test, Q2, p. 66

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…

Answer B: Ending the ORDER BY with a column that cannot tie, such as the id

Why: 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.

Module 1 test, Q3, p. 66

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…

Answer C: Every payment over 20,000 labelled medium

Why: 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.

Module 1 test, Q4, p. 67

`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…

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

Why: 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.

Module 1 test, Q5, p. 67

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

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

Why: 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.

Module 1 test, Q6, p. 67

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?

Answer C: Listing the distinct values of every low-cardinality column

Why: `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.

THE LESSON CHECKS

Lesson 1, p. 15

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…

Answer A: Her address now sits in many rows and nothing forces those rows to agree with each other

Why: A database stores each fact once and enforces rules; a spreadsheet stores copies and enforces nothing.

B The sheet will eventually grow too…: Treats file size as the reason to move off a spreadsheet, when size is the symptom that shows up last.

C Addresses are being stored as free…: Assumes the flaw is in how the value is formatted rather than in how many times it is stored.

D Each sale has no unique identifier…: Reaches for the missing primary key, which is a real gap but not what breaks when the address changes.

Lesson 2, p. 19

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…

Answer B: Query the CSV directly with DuckDB, since it reads the file in place and is built for scanning large tables.

Why: 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.

A Set up Postgres, create a table…: 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 Open the file in a spreadsheet…: 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 Load it into SQLite, because SQLite…: 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.

Lesson 3, p. 23

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…

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

Why: 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 Rounding is being applied at display…: Display rounding produces a consistent, explainable difference; the give-away here is that the direction and size vary unpredictably.

B The column is too narrow and…: Overflow in a double happens around 10^308 and would produce enormous errors, not a few cents.

C Some rows were inserted with NULL…: Skipped NULLs would produce a consistent shortfall of whole order amounts, not a drift of cents.

BUILD WITH IT

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.

WHAT IT TEACHES	A first course in data for people who have never written a query.
WHAT IS INSIDE	Six modules and 66 lessons, 29 figures drawn from data, six case studies, six module tests, a 60-question examination, and every answer with the reason behind it.
LICENCE	CC BY-NC-SA 4.0. Copy, print, share and adapt it for any non-commercial purpose, with credit, under the same licence. There is no paid edition.
THE COURSE	Free to read, with the lessons, the films and the examination, at **addaly.com/learn/data-and-sql**

Scan to open the course

Addaly

First edition, September 2026 · build 62a26075