KEDBYTE

SOFTWARE · AI · AUTOMATION

HOW DATA WORKS

From a Written Record to a Global Database

The plain-English guide to data systems,
followed by the engineer's version of the same thing.

WRITTEN BY

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Volume B · Chapters 7–13 · First Edition

From Records to Databases

HOW DATA WORKS

From a Written Record to a Global Database

Author	Shikhar Singh
Designation	Founder & Chairman, KedByte Technologies Private Limited
Published by	KedByte Technologies Private Limited
Imprint	KedByte Press
Edition	First Edition, September 2026
Subject	Data systems, databases, software engineering and general reference
Extent	Eight parts, fifty-two chapters

Examples. Mira's Corner and all shop records, identifiers, prices, people and events are invented teaching material. They are not customer data or observations of a live commercial system. New examples state their own assumptions and do not silently replace earlier facts.

Accuracy and currency. Technical references identify versions or consultation dates where available. Software and documentation change. Local tests exercise stated examples in the recorded environment; they do not verify every sentence, implement every production safeguard or establish compatibility with every software version.

Educational scope. This book and its code are for learning. They are not a production service, legal opinion, security certification or promise of recovery from every failure. Use isolated practice environments and obtain appropriate authority before inspecting or changing a real system.

Preparation. Prepared with AI assistance for Shikhar Singh and KedByte Press. The publisher's accompanying quality report records the checks performed and their limits. Independent technical, legal and accessibility certification is not claimed.

VOLUME B

From Records to Databases

Move from ordinary files to explicit relationships, enforceable rules and useful questions.

This volume contains Chapters 7–13 of the complete book. Chapter and section identifiers remain unchanged. Its local page numbers differ from the complete edition. The volume glossary covers its own chapter vocabulary; the source register is shared across the series.

How to read this book

The shape of every section

Every main teaching section uses the same six blocks. The labels describe ways to approach an idea, not different classes of reader.

Block	What it is for
PLAIN — in simple words	The problem and central idea in ordinary language.
PLAIN — a picture in your head	An everyday comparison, followed by its explicit limits.
PLAIN — a worked example	Concrete records, quantities or steps that can be checked.
PLAIN — what is really happening inside	The actual mechanism, explained without a jump in assumed knowledge.
TECHNICAL — the engineer's version	Precise vocabulary, implementation details, assumptions and source references.
WORDS — remember these	Each term connected to both an everyday and a technical meaning.

Two ways to read it

1. **One continuous pass.** Start at Chapter 1 and read every block. The later engineering treatment grows out of the example already introduced.
2. **Two passes.** Read the PLAIN and WORDS blocks first, then return for the TECHNICAL blocks. The browser reader provides modes for both routes. Practice and chapter summaries remain visible in either mode.
3. **Question-led reference.** Use the contents, chapter search or glossary to locate a subject. Read the nearby assumptions before borrowing a formula, query or design pattern.

Conventions used throughout

1. Main sections have stable identifiers such as 26.4. Their six learning blocks extend that identifier, for example 26.4.5 for the engineering treatment. Chapter and section numbers stay constant across the complete edition, individual volumes and website.
2. Numbered paragraphs separate ideas. An analogy includes a statement of where it breaks. The technical treatment should deepen the simple explanation rather than silently contradict it.
3. A product-specific behavior is not presented as a universal database rule. PostgreSQL examples refer to the documented version; SQLite examples identify their separate implementation and tested runtime.
4. A source marker such as [S25] points to the source register. Consultation dates belong to those records. Unversioned documentation can change; the included test report identifies the environment actually exercised.

5. Worked shop examples are synthetic. A table of amounts is not evidence of payment settlement, real customer activity or a production incident.
6. Every chapter ends with practice and worked answers, common wrong ideas, and a summary of twenty numbered entries. A summary entry may wrap onto more than one printed line.
7. Code blocks may be complete executable fragments, queries requiring the stated fixture, or explicitly labeled conceptual traces. The companion-lab guide identifies the implemented exercises and their boundaries.
8. A successful local test establishes only its asserted property. It is not independent review, complete security assurance or certification of a production service.

One shop, growing demands

Mira's Corner is a fictional stationery shop. Mira owns it and Dev helps at the counter. The canonical four agreed order lines comprise six items and 28,650 paise: O-1042 totals 17,100 and O-1043 totals 11,550. Taxes, discounts, delivery fees and settlement records are deliberately outside that initial fixture.

The later catalogue-price example does not rewrite historical agreements. The earlier expected-stock-10/observed-stock-9 discrepancy remains unexplained. Later race conditions and capstone transactions are separate demonstrations, not invented explanations of that discrepancy.

The capstone adds synthetic tenant identifiers T-A and T-B. These represent separate organizational scopes; a branch identifier is not a substitute for tenant authorization. CAP-001 is a separate order whose two notebooks and one pen happen to total the same 17,100 paise as the opening example.

Where to find things

1. Chapters 1–6 establish meaning, records, representations, measurements, identity and history.
2. Chapters 7–13 develop files, databases, schemas, constraints, SQL and relationships.
3. Chapters 14–20 follow queries through scans, indexes, plans, reports and performance measurements.
4. Chapters 21–26 examine transactions, overlapping work, isolation, locks, versions and idempotency.
5. Chapters 27–32 trace physical storage, logs, compaction, durability, backup and repair.
6. Chapters 33–39 explain replication, failover, partitioning, consistency, consensus, distributed transactions, caches and queues.
7. Chapters 40–46 develop pipelines, streams, analytical storage, text and vector search, and careful interpretation of results.
8. Chapters 47–52 cover permissions, retention, migrations, operation, the integrated case study and the reference desk.
9. The A–Z glossary, companion-lab guide and source register follow the chapters. The browser edition also offers keyword and full chapter-text search.

Edition and evidence

This first edition contains all 52 chapters and was prepared with AI assistance for Shikhar Singh and KedByte Press. The quality report records local checks, not independent certification. The PDF fixes the layout; Word and browser pages reflow. Use stable chapter and section identifiers across formats.

Contents

PART B

From Records to Databases

Move from ordinary files to explicit relationships, enforceable rules and useful questions.

7

CHAPTER

Files, Folders and Data Formats

7.0 What this chapter gives you

1. You will separate the location of a file from the identity of its contents and the identity of the records inside it.
2. You will follow bytes through decoding, parsing and validation without treating those as the same operation.
3. You will read a CSV field containing a comma, a quotation mark or a line break without accidentally creating extra records.
4. You will distinguish a JSON number, string, Boolean, null, object and array, then apply a separate business contract.
5. You will explain what a binary format must specify and why changing a filename does not convert the contents.
6. You will design a bounded import that accounts for accepted rows, repeated deliveries and rejected material.
7. You will know when a parser must stop because it cannot safely identify the next record.

Part A established what the shop's records mean. Part B asks how those meanings travel between programs and become enforceable structures. We begin with files because the simplest exchange is often an exported document, not a database connection. Mira sends Dev a file containing agreed order lines. The canonical orders remain O-1042 and O-1043: four lines, six units and a combined value of 28,650 paise. The deliberately defective rows in this chapter are separate synthetic import inputs; they do not change either original order.

7.1 Paths and file identity

7.1.1 PLAIN — in simple words

1. A filename tells a program where to look. It does not, by itself, tell the program what it will find there.
2. Imagine that Mira saves `orders.csv` on Monday and replaces it on Tuesday. The same name can now lead to different bytes. Conversely, two differently named files can contain exactly the same bytes.
3. The records inside a file have another kind of identity. Order O-1042 remains the same identified order whether its representation arrives in an email attachment, an export folder or a database response.

4. Therefore ask three separate questions: "Which location?", "Which captured contents?" and "Which business records?" A useful import log answers all three rather than using the filename as a substitute for everything else.
5. A folder is a way of organising names. The folder called `approved` does not prove that a human approved its contents. Naming conventions can help people work, but their meaning needs an actual process behind it.
6. A suffix such as `.csv` is a useful hint. It is not an independent validation of the data. A program should still check that the contents satisfy the format and contract it expects.

7.1.2 PLAIN — a picture in your head

1. Picture a numbered shelf slot containing an envelope. The shelf address is the path. The sheets in the envelope are the file contents. The order numbers written on the sheets identify the shop's orders.
2. Move the envelope to another shelf and its contents need not change. Replace the envelope in the original slot and the address remains familiar even though the contents are new.
3. Photocopy the sheets into a second envelope and the two physical envelopes are different, while the copied text is the same. A digital byte-for-byte copy creates the analogous distinction between file objects and contents.
4. Now imagine a sign saying "use the envelope on shelf B instead". This is a useful starting picture for a symbolic link: resolving one name can lead the program to another location. [S42]
5. **Where the comparison breaks:** real filesystems have rules about links, access, open handles and replacement that depend on the platform. A paper shelf does not model those rules. In particular, a path checked a moment ago need not still name the same object when opened later.
6. The practical lesson is not to memorise shelf metaphors. It is to stop treating a convenient label as permanent evidence of identity or authority.

7.1.3 PLAIN — a worked example

1. Mira has the following four captured inputs. The letters A and B stand for two different byte sequences in this exercise; they are not actual checksum values.

Captured input	Name at capture time	Contents	Business records claimed
I-01	`exports/orders.csv`	A	Four canonical order lines
I-02	`archive/orders-copy.csv`	A	The same four lines
I-03	`exports/orders.csv`	B	A changed export requiring review
I-04	`exports/orders.json`	C	A JSON representation of the four lines

2. I-01 and I-02 have different captured names but identical contents. Importing both without checking record identity could apply the same data twice.
3. I-01 and I-03 have the same captured name but different contents. "I already processed orders.csv" is therefore not a complete explanation of what the system processed.
4. I-01 and I-04 can describe equal domain records even though their bytes differ. A byte checksum alone cannot identify this kind of semantic equality.

5. For each input, the proposed log records an import-run identifier, the displayed source name, the captured byte length, a digest, the selected contract and the outcome. Source authentication, where required, is a separate concern.
6. If someone can replace both a file and its accompanying digest, matching the two only shows agreement between those supplied objects. It does not establish who produced the file. This follows directly from the assumed attacker's ability to replace both.

7.1.4 PLAIN — what is really happening inside

1. A program asks the operating system to open a path. The operating system resolves names, applies its access rules and returns a handle through which the program can work with the opened object. Python exposes this through file objects. [S43] [S64]
2. A relative path is interpreted from some starting location. If a script assumes that the current working directory is its own folder, running it from another directory can make it open the wrong file or find no file at all.
3. An absolute path names a location from a filesystem root or other platform-specific anchor. It avoids that particular ambiguity, but it does not make the underlying contents immutable. [S42]
4. Some filesystem identifiers can help recognise an object within their defined scope. They are not universal business identifiers and should not replace order keys. They may also be reused after an object is removed. [S64]
5. Our educational importer avoids a filesystem race by accepting a byte string that the caller has already captured. Every parse, checksum and diagnostic in one call refers to that same byte string.
6. This does not solve safe capture on every operating system. A production capture procedure must also handle a file being modified while it is read, permissions, size limits and the trustworthiness of the source. The exercise starts after that capture boundary.

7.1.5 TECHNICAL — the engineer's version

1. Separate **pathname**, **filesystem object**, **byte representation**, **logical record identity** and **provenance** in the design. Each answers a different question; equality at one layer does not imply equality at all other layers.
2. In Python, `PurePosixPath` and `PureWindowsPath` perform lexical path operations without opening files. `Path` adds filesystem operations for the current platform. A lexical check is not a security decision about a subsequently opened object. [S42]
3. The following example manipulates a synthetic path only. It neither creates directories nor reads a real export.

```
from pathlib import PurePosixPath

p = PurePosixPath("exports/2026-09/orders.csv")
assert p.name == "orders.csv"
assert p.suffix == ".csv"
assert str(p.parent) == "exports/2026-09"
```

4. A content digest is useful for identifying a captured byte sequence with an explicitly chosen algorithm. It should travel with the byte length and capture metadata. Do not call it proof that a sender was authorised or that a sale happened.
5. A byte-oriented fingerprint also changes when insignificant formatting changes. Whitespace or object-member order can change JSON bytes while the application interprets the same fields. Comparing validated domain values is a different operation with its own equality rule. [S07]

6. **Implementation detail:** filesystem case sensitivity, supported link operations and rename behaviour vary. Our exercises do not assume that `Orders.csv` and `orders.csv` are necessarily distinct on every machine. [S42]
7. **Chosen contract:** this lab processes an already captured, at-most-one-mebibyte input in memory. It is not a general directory-walking service, an upload server or a complete defence against malicious filesystem races. Extending that boundary requires a new design and new tests.

7.1.6 WORDS — remember these

1. **Path:** instructions for locating something in a filesystem — a name interpreted under platform-specific pathname-resolution rules.
2. **Relative path:** a location described from a starting point — a pathname whose interpretation depends on a working directory or another supplied base.
3. **File extension:** the familiar suffix on a name — a naming convention that does not independently validate the represented format.
4. **Content digest:** a compact fingerprint of captured bytes — the result of applying a specified hash function to an exact byte sequence.
5. **Capture boundary:** the point at which this exercise obtains its input — the boundary after which one immutable byte string is used for parsing and diagnostics.
6. **Provenance:** where information came from and how it was handled — recorded source, capture and transformation context, not automatic proof of truth.

7.2 Bytes, records and boundaries

7.2.1 PLAIN — in simple words

1. A file normally gives the reader bytes. The reader needs rules to turn those bytes into useful values.
2. When the file is text, the first rule says how bytes represent characters. This is the character encoding introduced in Chapter 3. A second rule says how those characters form records and fields.
3. A third rule says what the fields mean and which values the application accepts. Reading a number successfully does not establish that the number is a valid order quantity.
4. These are separate gates. “The bytes are valid UTF-8”, “the text is valid JSON” and “the record satisfies our order-line contract” are three different statements.
5. A record needs a boundary so the reader knows where it stops. A format might use a delimiter, a fixed size, a length written before the content, or a structured grammar. The reader must use the actual rule, not a guess based on what looks tidy.
6. A displayed line is not necessarily a record. Text wrapping is presentation, and some formats allow a real newline inside a field. Counting screen lines cannot reliably count business records. [S08]

7.2.2 PLAIN — a picture in your head

1. Imagine several parcels arriving in a delivery bag. The bag is the file. Each parcel is a record, and the labelled compartments inside a parcel are fields.

2. The delivery worker needs a way to identify each parcel's boundary. Separate boxes make it obvious. A roll of unmarked paper containing every address one after another does not.
3. One possible rule is "every parcel begins with a label saying how many centimetres of wrapping belong to it". That resembles a length-prefixed representation.
4. Another possible rule is "a certain marker ends the parcel, unless it occurs inside an explicitly protected area". That resembles quoting and delimiters in text formats.
5. **Where the comparison breaks:** a parser has no human intuition about where a torn parcel ought to end. If a damaged length or quotation mark destroys the boundary, confidently skipping to the next visible line can manufacture records that were never present.
6. The safe response may be to stop the whole import, retain the exact input and report a file-level error. A smaller amount of accepted data is not worth a false claim that the remainder was understood.

7.2.3 PLAIN — a worked example

1. Consider the text consisting of the letter A, the rupee sign and a newline. It contains three Unicode code points, but its UTF-8 representation contains five bytes.

```
Displayed meaning      A     rupee sign     newline
Unicode code points    0041  20B9           000A
UTF-8 bytes            41    E2 82 B9       0A
Total                  3 code points; 5 bytes
```

2. The calculation is one byte plus three bytes plus one byte. A byte limit of four cannot hold the complete representation. Cutting after the first two bytes would also cut inside the rupee sign.
3. Now define an original toy format for two ASCII words. Each word starts with a two-byte unsigned length, most significant byte first. The length counts payload bytes, not letters seen on screen.

```
00 03  70 65 6E
length 3; the bytes for "pen"

00 08  6E 6F 74 65 62 6F 6F 6B
length 8; the bytes for "notebook"
```

4. The complete two-record representation uses $2 + 3 + 2 + 8 = 15$ bytes. The format carries four bytes of length information and eleven bytes of word content.
5. If the second length says eight but only six payload bytes remain, the reader has an incomplete record. It must not invent the missing two bytes.
6. Conversely, if the contract permits exactly two records but more bytes follow, the reader must decide according to the contract whether those bytes are an error, another record or a defined extension. Ignoring them accidentally is not a specification.

7.2.4 PLAIN — what is really happening inside

1. A text decoder translates a byte sequence according to an encoding. It can reject an impossible sequence. Replacing damaged bytes with a replacement character may be useful for display, but it loses evidence and can change identifiers. [S43]
2. A parser then follows a grammar. It distinguishes punctuation that structures a record from punctuation that belongs inside a value.
3. The application applies validation after parsing. It can ask whether the expected fields exist, whether the version is supported and whether the quantity is a positive whole number.

4. A streaming reader may receive only part of a record in one read. The amount returned by an input operation is an I/O boundary, not necessarily a record boundary. A parser must retain incomplete state or request additional bytes. [S43]
5. Our lab uses a complete captured input instead of teaching network streaming at the same time. That keeps parsing and record validation visible. The two-word framing example still demonstrates why a known boundary matters.
6. The stages should report errors at the stage where evidence supports them. Invalid UTF-8 is a decoding error. Missing CSV fields are a structural or contract error. An unknown product is a relationship error, not a broken byte encoding.

7.2.5 TECHNICAL — the engineer's version

1. Use an explicit pipeline: **capture → decode → parse → validate fields → check relationships and identity → publish**. The exact partition can vary, but the distinctions must remain visible in results.
2. **Syntactic validity** means conformance to a representation grammar. **Domain validity** means conformance to stated application rules. Neither establishes the truth of a real-world claim; that limitation carries forward from Chapter 2.
3. Count lengths in the unit specified by the format. Bytes, Unicode code points and grapheme clusters are not interchangeable. The rupee example is a fixed illustration, not a universal one-byte-per-character rule. [S14] [S16]
4. For fixed-width fields and length prefixes, specify byte order, widths, signedness, permitted range and the response to truncation. Python's `struct` format prefixes make native versus explicitly specified byte layouts distinguishable. [S46]
5. Do not use a lossy text-decoding mode in an evidence-preserving import unless that loss is an explicit, separately recorded transformation. This lab decodes UTF-8 strictly and retains the original bytes on failure. [S43]
6. **Chosen framing contract:** the two-word example requires a two-byte length followed by exactly that many bytes for each payload. Its format is invented for teaching. It is not Avro, a network protocol or a recommendation to build a production file format.
7. Separate an input's size from the size of objects produced by parsing it. A small encoded input can produce many runtime objects, and nested structures can require substantial processing. This lab is bounded and deliberately small; its limit is not a complete resource-isolation system. [S45]

7.2.6 WORDS — remember these

1. **Decoding:** turning bytes into characters — interpreting a byte sequence under a specified character encoding.
2. **Parsing:** finding the structure in a representation — recognising records, fields and values under a grammar.
3. **Framing:** marking where messages or records begin and end — a boundary convention independent of the meaning of each payload.
4. **Length prefix:** a size written before the content — a field giving the amount of following data in a stated unit.

5. **Truncation:** an input ending before its promised content is complete — insufficient bytes to satisfy a declared structure or length.
6. **Syntactic validity:** being written in the right shape — conformance to a format's grammar, distinct from business rules.

7.3 CSV and quoting

7.3.1 PLAIN — in simple words

1. CSV is a way to write rows of fields as text. The letters stand for comma-separated values. It is useful because many different programs can read and write it.
2. The simple picture is "commas separate fields and line endings separate records". The important correction is that a quoted field can itself contain commas and line endings. A real CSV parser understands that distinction. [S08]
3. A quoted field is not automatically a special kind of business value. Quoting is part of how text is represented; the application still decides whether the resulting value is a product name, quantity or identifier.
4. The column headings do not supply every missing rule. A column called `amount` is still ambiguous until the contract specifies currency, unit, scale and meaning.
5. Different exports can use different delimiters and conventions. A file using semicolons is not repaired by pretending that its semicolons are ordinary content and its commas are separators.
6. For the shop's exercise we choose a precise CSV profile rather than trying to guess every possible export. Inputs that do not match the profile are reported, not silently "cleaned up".

7.3.2 PLAIN — a picture in your head

1. Imagine writing a shopping list in a notebook and using a vertical line to separate each field. That works until a product name contains the same character.
2. You add a rule: text inside a pair of quotation marks belongs to one field even when it contains a separator. The quotation marks describe the boundary; they are not normally part of the resulting field value.
3. What happens when the product name itself includes a quotation mark? In the CSV convention used here, a doubled quotation mark inside a quoted field represents one literal quotation mark. [S08]
4. **Where the comparison breaks:** the reader cannot rely on colour, handwriting or visual alignment. It follows the character sequence and its parsing rules exactly. A missing closing quote can make the boundary ambiguous for the rest of the input.
5. Also, CSV quotation marks are not protection against spreadsheet formulas. A spreadsheet may interpret a field's contents as a formula after the CSV layer has been decoded. Representation safety and spreadsheet behaviour are separate questions. [S63]
6. That is why the lab inspects values as data in Python and does not automatically open unfamiliar exports in a spreadsheet.

7.3.3 PLAIN — a worked example

1. This independent two-column CSV illustration contains three logical records, including the header. The second data field contains a real line ending inside quotation marks.

```
product_id,description
```

```
P-PEN,”Blue, fine-tip pen”
P-NOTE,”Notebook with ””draft”” label
and plain pages”
```

2. The first data record has two fields: `P-PEN` and `Blue, fine-tip pen`. The comma inside the description does not create a third field.
3. The second data record also has two fields. Its description contains quotation marks around the word draft and a newline before the word and. The input has more physical lines than logical data records.
4. In this example, `line.split(',')` gives the wrong answer. Splitting first on every newline and then on commas also gives the wrong answer.
5. The following Python code is a compact, original parsing exercise. The string is complete before it is read.

```
import csv
import io

text = 'product_id,description\r\nP-PEN,”Blue, fine-tip pen”\r\n'
rows = list(csv.reader(io.StringIO(text, newline=””),
                       strict=True))
assert rows[1] == [”P-PEN”, ”Blue, fine-tip pen”]
```

6. The parser returns strings. Turning a field such as `7550` into the integer number of paise is a later step under our explicit import contract. A field such as `007` used as an identifier must not be converted to seven merely because it looks numeric.

■ 7.3.4 PLAIN — what is really happening inside

1. The reader tracks whether it is inside a quoted field. It interprets a separator or newline according to that state rather than simply splitting at every occurrence.
2. A dialect describes representation choices such as the delimiter, quote character and escape behaviour. Python’s CSV module provides explicit reader and writer settings. [S44]
3. A header row maps positions to names. Our profile requires the exact eight distinct headings in the agreed order. Duplicate, missing, unexpected or whitespace-altered headings cause a file-level rejection.
4. Each later row must have eight fields. A valid CSV parser can still return a row with too many or too few fields, so the importer checks width separately.
5. Only after that check does it convert the integer fields and call the existing `validate_line` function from Chapters 1–3. The domain function remains unchanged.
6. A row with an empty quantity is rejected with its raw fields and reason retained. It is not converted to zero, and the importer does not shift the remaining fields left to conceal the problem.
7. If the CSV parser cannot safely finish parsing the input, the lab withholds all candidates from that file. Earlier rows may help diagnose the failure, but they do not become a quietly accepted partial business import.

■ 7.3.5 TECHNICAL — the engineer’s version

1. **Documented format:** RFC 4180 describes a common CSV format and registers `text/csv`; it is an Informational RFC, not a complete universal definition of every file called CSV. Our profile uses its familiar quoting rules with an explicit UTF-8 choice. [S08]

2. Python recommends opening CSV text files with `newline=''`, allowing the CSV module to handle embedded newlines. `strict=True` raises errors for certain malformed inputs, but is not a schema validator or a guarantee that the file obeys every stricter exchange profile. [S44]
3. **Chosen import profile:** header names are `record_type`, `schema_version`, `order_id`, `line_no`, `product_id`, `quantity`, `unit_price_minor`, `currency`. The encoding is UTF-8 without a byte-order mark. Input LF and CRLF record endings are accepted by the Python reader; the supplied writer emits CRLF. This chosen adapter rejects a bare carriage return anywhere, including inside a field; embedded LF or CRLF remains supported.
4. Integer fields use canonical ASCII decimal text: `0` or a nonzero digit followed by digits. Leading signs, outer whitespace, leading zeros on multi-digit numeric fields, fractions and exponent notation are rejected. The narrower text grammar is an adapter decision, not a change to the earlier integer domain model.
5. The separate identifier fields remain text. Their leading zeros, case and punctuation are not normalised automatically. The domain contract continues to reject empty identifiers and outer whitespace.
6. CSV has no universal built-in representation for SQL NULL, types, units or foreign keys. A metadata contract is needed to carry such semantics; W3C's tabular-data model explicitly separates textual cell representations from their semantic values and annotations. [S49]
7. Quoting a spreadsheet-sensitive field is not a general formula-injection defence. Keep archival machine-exchange data separate from a deliberately prepared spreadsheet-view export; the display transformation must be tested against the intended consuming software. The lab implements neither spreadsheet execution nor a universal sanitiser. [S63]
8. The source bytes are the authoritative diagnostic artifact for this exercise. Parsed physical-line ranges and field lists are navigation aids. They do not replace the retained original representation.

7.3.6 WORDS — remember these

1. **CSV:** a text representation of tabular records — comma-separated values with a specified dialect and quoting rules.
2. **Delimiter:** the mark used to separate fields — a structural character interpreted according to the parser's current state.
3. **Quoting:** marking a region as one field — a representation mechanism that allows otherwise structural characters inside a value.
4. **Dialect:** the particular set of parsing conventions — settings governing delimiters, quotation marks, escaping and related behaviour.
5. **Logical record:** one complete parsed record — a structure that may span more than one physical text line.
6. **Import profile:** the exact agreement for an exchange — a bounded specification combining encoding, syntax, fields and application rules.

7.4 JSON and nested data

7.4.1 PLAIN — in simple words

1. JSON gives names and values a written structure. An object groups named members; an array holds an ordered sequence of values. Objects and arrays can contain further objects and arrays. [S07]
2. That makes JSON convenient for an order containing several lines. The order can be one object with an array called `lines` inside it.
3. Convenience does not mean automatic correctness. The JSON parser does not know whether the array should contain one line or five, whether a price is in paise, or whether the order already exists.
4. The value `2` is a number, `"2"` is text, and `true` is a Boolean. A receiving program must not quietly treat them as equivalent merely because all could be described informally as "something positive".
5. A member containing `null` is present with a null value. An absent member is not present at all. The application may assign different meanings to those cases. [S07]
6. Reading JSON successfully means that a representation was recognised. It does not authenticate the sender, authorise an action or verify a transaction in the physical shop.

7.4.2 PLAIN — a picture in your head

1. Picture a folder with labelled pockets. One pocket says order identifier, another says currency, and a third contains an ordered stack of line cards.
2. Each line card has its own labelled spaces for product, quantity and agreed price. This nested structure resembles a JSON object containing an array of objects.
3. An empty pocket is not the same as a missing pocket. Nor is a pocket containing the text "unknown" automatically the same as the application's null value.
4. **Where the comparison breaks:** JSON defines representation kinds, not a full model of your organisation. A folder has physical ownership and access boundaries; a JSON object does not automatically acquire either.
5. The analogy also does not justify relying on object-member order. An array has a defined order, while a JSON object's members are not a suitable substitute for an ordered business sequence. [S07]
6. When order matters, make it explicit with a suitable structure or field. The line number from the earlier chapters remains meaningful even when the surrounding object is printed in a different order.

7.4.3 PLAIN — a worked example

1. Here is the existing canonical first order line in its version-1 JSON representation. It describes two notebooks at 7,550 paise each.

```
{
  "record_type": "agreed_order_line",
  "schema_version": 1,
  "order_id": "O-1042",
  "line_no": 1,
  "product_id": "P-NOTE",
  "quantity": 2,
  "unit_price_minor": 7550,
  "currency": "INR"
}
```

2. The arithmetic is unchanged: 2 × 7,550 = 15,100 paise. JSON did not discover this meaning; the field contract supplied it.
3. Changing the quantity to the text `"2"` produces a JSON document that can still be parsed. Our integer-based line validator rejects it.
4. Changing the quantity to `true` also produces parseable JSON. Python's Boolean type has a relationship to integers, but our domain validator explicitly rejects Booleans for integer fields. [S22]
5. Removing `currency` creates a different contract error: a required field is absent. Setting it to `null` leaves the field present, but still fails the INR-only rule.
6. This separate counterexample is dangerous to interpret casually:

```
{"quantity": 2, "quantity": 9}
```

7. There are two members with the same name. Rather than allowing an implementation's default to choose a winner, our parser rejects the input before any domain decision. The reported conflict is about representation, not evidence that nine items were sold.

7.4.4 PLAIN — what is really happening inside

1. A JSON parser recognises brackets, braces, quoted strings, numbers and the literal values `true`, `false` and `null`, then constructs corresponding program values. [S07]
2. Different languages represent numbers differently. A number that fits comfortably in one runtime's integer type may lose precision in a consumer using a limited-precision floating-point representation. Representation compatibility must be checked across the actual participants.
3. Python's standard JSON decoder exposes hooks for object-member pairs and numeric conversions. The lab uses the pair hook to detect repeated object names before they collapse into an ordinary mapping. [S45]
4. The lab also rejects nonstandard constants such as NaN and Infinity. Python's default JSON behaviour can accept them, so relying on the default would make our stated profile less strict than intended. [S45]
5. After the representation checks, the resulting object goes to the same order-line validator used by the CSV adapter. This keeps the business rules consistent while allowing different formats to reach them.
6. Nested JSON is introduced conceptually here, but the executable version-1 line importer deliberately accepts one line object, not a complete nested order. A nested-order contract would need separate rules for the envelope, its lines and their relationships.
7. Importers should make this boundary visible. "Can parse nested JSON" is not the same capability as "can validate and publish a complete order without partial failure".

7.4.5 TECHNICAL — the engineer's version

1. RFC 8259 recommends unique object-member names and describes divergent behaviour when names repeat. Our profile makes uniqueness mandatory. This is an explicit interoperability choice rather than a claim that the JSON grammar alone resolves every duplicate-member case. [S07]
2. Python's `object_pairs_hook` receives each object as ordered pairs, allowing a duplicate-name check before dictionary construction. `parse_constant` can reject the nonstandard numeric constants accepted by the default decoder. [S45]
3. A compact duplicate-name hook can be understood without hiding its decision:

```
def unique_members(pairs):
```

```
    obj = {}
    for name, value in pairs:
        if name in obj:
            raise ValueError("duplicate JSON member: " + name)
        obj[name] = value
    return obj
```

4. Type validation still follows. In our chosen contract, a parsed floating-point `2.0` is not accepted as an integer quantity even though it describes a mathematically integral value. The exchange rule could be different in another system, but must not vary accidentally between import paths.
5. Version 1 rejects unknown fields instead of discarding them. A future optional field therefore requires a considered compatibility policy, not an assumption that a permissive old reader will preserve information it ignores.
6. JSON is a data-interchange format, not a command language for the importer. Do not evaluate its text as program code. Python's `pickle`, in contrast, can execute code during deserialisation and must not be treated as a safe substitute for untrusted JSON. [S62]
7. The lab checks byte size and catches parsing or recursion failures, but it is not a complete hostile-input sandbox. Production services also need externally enforced CPU, memory, nesting and request-rate limits appropriate to their threat model.
8. **Source of truth:** the schema version describes the meaning and permitted shape of our message. A package version, a Python version and a JSON syntax specification identify different things; no one number replaces the others.

7.4.6 WORDS — remember these

1. **JSON object:** a group of named values — a collection of name/value members within a JSON representation.
2. **JSON array:** a sequence of values — an ordered collection whose elements may themselves be structured.
3. **Duplicate member:** a name repeated within one object — an interoperability hazard rejected by this book's input profile.
4. **Deserialisation:** rebuilding values from a stored representation — decoding and interpreting data into runtime objects under a specific format.
5. **Schema version:** the named edition of a record contract — an identifier for the supported structure and meaning, distinct from a software release.
6. **Unknown field:** a member the contract does not recognise — data requiring an explicit reject, ignore, preserve or extension policy.

7.5 Binary formats and versioning

7.5.1 PLAIN — in simple words

1. Every file is stored as bytes. "Binary format" usually means that its organisation is not simply a human-readable text grammar.
2. A binary format can represent a number directly in a fixed number of bytes instead of writing its decimal digits. To read it, the receiver must know how many bytes belong to the number and how to interpret their order.

3. Saving fewer bytes is not the only reason to choose a format. Efficient scanning, nested structure, compatibility, supported tools and future readability can matter more than a tiny size difference.
4. Some formats carry a description of their structure. That helps a reader decode values, but it still does not tell the reader every business rule or whether the recorded facts are true.
5. A new version should say what changed and which readers can understand it. Writing the number two on a file is not a compatibility strategy by itself.
6. The shop's first requirement is not "use the most advanced format". It is "preserve these particular meanings between these particular writers and readers, with errors that we can explain".

7.5.2 PLAIN — a picture in your head

1. Think of a custom tray containing exactly sized compartments. A four-unit compartment holds a price, another holds a quantity, and a label says which tray design is being used.
2. A reader with the correct tray diagram knows where to find each value. A reader using a different diagram may take part of the price as part of the quantity.
3. A versioned diagram therefore matters as much as the tray. Changing the compartment layout without changing the agreement is how neatly stored bytes acquire the wrong meaning.
4. **Where the comparison breaks:** binary formats need not use fixed compartments. They may use variable-length encodings, compression, repeated structures, dictionaries and metadata. The tray models explicit layout, not every storage technique.
5. Also, a compact tray is not necessarily easier to work with. If Mira's partner cannot read it, the few saved bytes do not compensate for an unusable exchange.
6. The right design is the one that matches a documented workload and an actual compatibility requirement, not the one with the most impressive file extension.

7.5.3 PLAIN — a worked example

1. The decimal price 7,550 can be represented as a four-byte unsigned integer. With the most significant byte first, those bytes are `00 00 1D 7E`.
2. Reversing the byte-order assumption changes the interpretation dramatically. The same four bytes read least-significant-first produce 2,115,829,760, not 7,550.
3. Python makes the convention visible:

```
import struct

encoded = struct.pack(">I", 7550)
assert encoded.hex() == "00001d7e"
assert struct.unpack(">I", encoded)[0] == 7550
assert struct.unpack("<I", encoded)[0] == 2115829760
```

4. The symbols `>` and `<` choose the byte order. The `I` in this standard-size layout represents an unsigned four-byte integer. These are `struct` conventions, not letters stored as part of the integer itself. [S46]
5. Our separate toy envelope begins with four marker bytes, one version byte and a two-byte payload length. The payload is an existing JSON line. It is intentionally simple enough to inspect by hand.

```
4 bytes   marker: KDBF
1 byte    envelope version: 1
2 bytes   payload byte length, big-endian
N bytes   UTF-8 JSON payload, under the line-v1 contract
```

6. The envelope therefore occupies 7 + N bytes. A different marker, unsupported version, short payload or unexpected trailing bytes is rejected. A matching marker only selects a parser; it does not authenticate the file or validate its business meaning.

7.5.4 PLAIN — what is really happening inside

1. The envelope parser checks its fixed header before interpreting the payload. It then compares the declared payload length with the actual remaining bytes.
2. Only a complete, supported envelope reaches the JSON reader. The JSON reader still performs its own checks, and the resulting values still reach the domain validator.
3. These nested checks form layers rather than replacements. A correct envelope does not rescue malformed JSON, and valid JSON does not rescue a negative quantity.
4. A more capable format can organise data for a particular access pattern. Apache Parquet, for example, is column-oriented and designed for efficient bulk storage and retrieval. That design is useful context for the later analytical chapters, not a reason to replace every small file. [S47]
5. Apache Avro specifies schemas and rules for resolving data written under one schema against a reader's schema. That illustrates how compatibility can be defined as concrete rules rather than optimism. [S48]
6. The lab does not implement or benchmark Parquet or Avro. Its invented envelope exists only to expose marker, version, length and payload boundaries. Use mature, appropriate formats for actual interchange rather than promoting the toy to a standard.

7.5.5 TECHNICAL — the engineer's version

1. Specify a format's syntax independently of the domain schema. The toy envelope has version 1, and the enclosed order-line message separately has `schema_version: 1`. Either layer could evolve without the other changing.
2. Python's native `struct` mode can depend on platform byte order, size and alignment. The examples deliberately use explicitly ordered standard-size fields instead of native layout. [S46]
3. Define **backward compatibility** operationally as a new reader handling an older writer's representation, and **forward compatibility** as an older reader handling a newer writer's representation. State the direction because terminology is sometimes used inconsistently in project discussions.
4. Consider a proposed v2 field `discount_minor`. An old reader that silently ignores it could compute a different payable amount. Syntactic tolerance would not establish semantic compatibility. This is an original hypothetical extension; discounts remain outside our actual v1 arithmetic.
5. Under Avro's schema-resolution rules, a reader can use a declared default for a field absent from a writer's record. That is a specified interpretation, not evidence that the writer originally recorded the default value. [S48]
6. Parquet's project documentation cautions that different implementations support different features. Select and test a concrete writer/reader combination; a shared format name is not sufficient evidence of identical feature coverage. [S47]
7. Compression, checksums, encryption and authentication answer different questions. A shorter payload is not thereby secret. A checksum agreement does not, by itself, identify an authorised producer. Our envelope supplies none of those additional security properties.
8. The lab's maximum payload is limited by an unsigned two-byte length, 65,535 bytes. Its other import bounds can be stricter. A bound should be checked before the parser treats a declared size as permission to allocate or read arbitrarily much data.

7.5.6 WORDS — remember these

1. **Binary format:** a byte layout read under explicit rules — a representation not restricted to a plain-text grammar.
2. **Byte order:** which end of a multi-byte number comes first — the endianness used to encode a numeric value.
3. **Magic marker:** an initial clue about the representation — a fixed signature used to select or validate a format boundary, not authenticate a producer.
4. **Envelope:** the wrapper around a payload — metadata and framing interpreted before the enclosed message.
5. **Compatibility:** the ability of specified participants to exchange usable data — a tested relationship between writer, reader and semantic contracts.
6. **Column-oriented format:** a layout that groups values by column within its storage structure — an organisation suited to some selective and analytical reads, not a universal performance guarantee.

7.6 Import validation and quarantine

7.6.1 PLAIN — in simple words

1. An import is not complete merely because a program reached the end of a file. The program must explain what it understood, what it rejected and what it actually made available for use.
2. Rejected material needs a place to wait for review. We call that quarantine. In this chapter it means retained diagnostic data, not a secure storage product.
3. A useful rejection includes the original input, the location of the problem where known and a reason that someone can act on. “Invalid” alone is usually not enough.
4. Repeated rows also need an explicit outcome. Ignoring them without recording that decision makes a row-count reconciliation misleading.
5. A conflict is different from an exact repeat. If two rows claim the same line identity with different quantities, choosing the last one simply because it came later is an invented policy.
6. Our batch importer waits until it has examined the complete parseable input. It withholds all members of a conflicting identity group instead of choosing a winner. This is a new batch policy, distinct from the in-memory event-delivery example in Chapter 5.
7. Finally, acceptance by the file adapter is not automatic publication into a database. The destination still needs its own relationship, permission, conflict and transaction checks.

7.6.2 PLAIN — a picture in your head

1. Picture a receiving desk with three labelled trays: candidates ready for the next check, repeated copies already accounted for, and material needing review.
2. The receiving clerk also retains the delivery bag and a manifest. This lets someone check that nothing was silently thrown away while sorting the contents.
3. If two differently completed forms claim the same order-line number, the clerk places the whole conflicting group in review. Arrival order is not allowed to decide which form is true.

4. **Where the comparison breaks:** storing rejected customer data can create privacy and access responsibilities. "Quarantine" does not mean "keep everything forever in an open folder". Retention and permissions need a separately defined policy.
5. Our laboratory contains only synthetic bytes and reports retained in memory. It neither creates a production quarantine service nor proves compliance with any retention obligation.
6. The transferable idea is accountable handling: a rejected input remains visible as rejected, and a human correction becomes a new input rather than a secret edit to the evidence.

7.6.3 PLAIN — a worked example

1. The supplied untidy CSV fixture contains eight logical data rows after its header. Four rows reproduce the canonical order lines. One repeats the first line exactly. Three are defective new rows.

Row	Claimed line	Relevant feature	Outcome in this fixture
1	O-1042 / 1	Two notebooks at 7,550 paise	Candidate
2	O-1042 / 2	One pen at 2,000 paise	Candidate
3	O-1043 / 1	One notebook at 7,550 paise	Candidate
4	O-1043 / 2	Two pens at 2,000 paise	Candidate
5	O-X1 / 1	Quantity is empty	Rejected
6	O-X2 / 1	Price text is `75.50`, not integer paise	Rejected
7	O-X3 / 1	Currency is USD, not INR	Rejected
8	O-1042 / 1	Exact repeat of row 1	Duplicate

2. The accounting identity is eight data rows = four candidates + three rejected rows + one duplicate. The header is not counted as a data row.
3. The four candidates still describe two orders, four lines and six units. Their total remains 28,650 paise; the duplicate contributes no additional candidate line.
4. The report retains all eight parsed rows, their outcomes and the captured input bytes. Rejection of rows 5–7 is visible rather than concealed in a lower output count.
5. In a separate conflict fixture, two individually valid rows both identify O-1042 / 1 but give quantities two and three. Both are quarantined. There are zero candidates, two rejected rows and no invented "correct" quantity.
6. A third fixture ends inside an open quoted field. The result is a file-level parsing error with zero candidates. The importer retains the raw input and does not pretend that it knows the business identity of every byte after the damaged boundary.

7.6.4 PLAIN — what is really happening inside

1. The importer first checks the byte bound and the declared text profile. It calculates a digest of the captured input and attempts strict decoding.
2. It validates the header, then parses complete records. Each parsed record retains its fields and physical-line range before any numeric conversion.
3. Field conversion and the existing line validator identify individually unacceptable rows. A rejected row's missing quantity is not filled from a neighbouring record.
4. For individually valid rows, the importer groups by `(order_id, line_no)`. Equal validated values form one candidate plus recorded duplicates. Different values under the same key make the complete group conflicting.

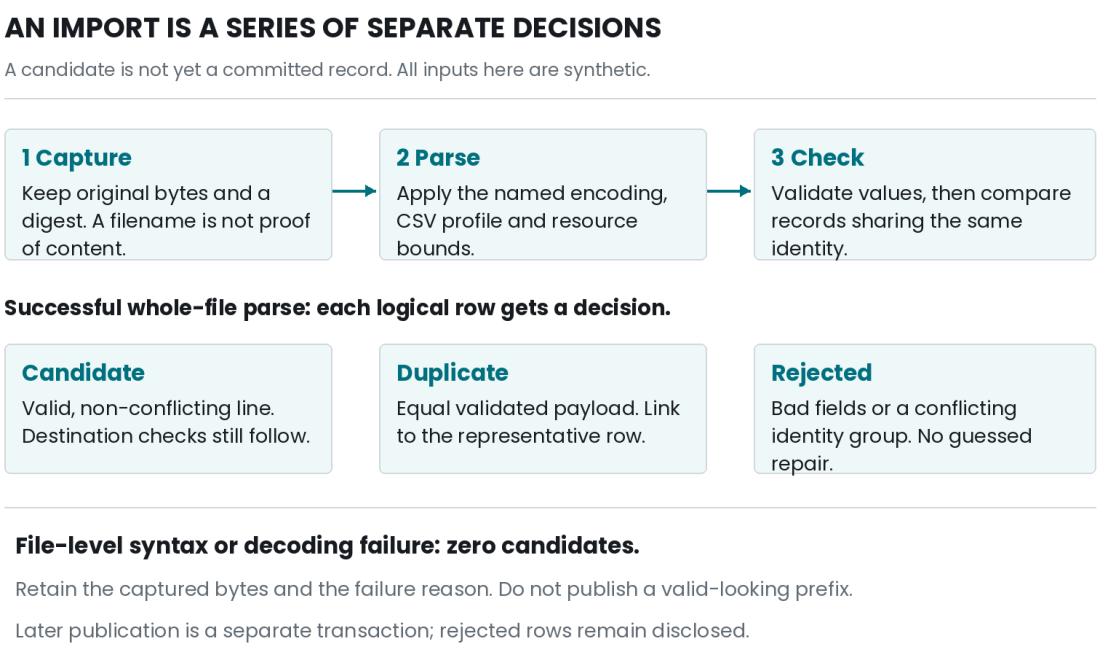

Figure 7.1: Figure 7.1 — An import keeps captured bytes, parses under a named profile, checks values and identity, and separates candidates, duplicate deliveries and rejected material. File-level syntax failure publishes no candidates.

5. Only after the file has been parsed successfully does the function return candidates. Structural file errors, a record-limit breach or an undecodable input withhold the entire candidate set.
6. The next stage may publish an explicitly accepted subset into a destination transaction. It must not claim that the source file was accepted in full when the report contains rejected rows.
7. A corrected resubmission should be separately captured and linked to its predecessor by the surrounding workflow. The lab does not implement that persistent workflow; it leaves the necessary distinction visible.

■ 7.6.5 TECHNICAL — the engineer's version

1. The lab's report separates `file_error` from a successfully parsed file with row-level findings. A **candidate** is a validated, non-conflicting domain object at this stage, not a committed database record.
2. A normal parse has one decision per logical input data row. An exact duplicate retains its relationship to a representative row. A conflicting group retains all its members with an explicit conflict reason.
3. The chosen identity comparison is over validated `OrderLine` values, not raw JSON text or a universal canonicalisation algorithm. Representation differences and application equality must remain separate.
4. File-level failure retains raw bytes and any trustworthy diagnostics collected before the stop. It does not fabricate decisions for records whose boundaries were never established. Thus a file-level failure cannot use the same complete-row reconciliation claim as a fully parsed file.
5. CSV numeric spellings are limited to 18 digits before integer conversion. Each parsed field is limited to 4,096 characters and the complete input to 10,000 logical data records. These are explicit adapter bounds, not universal CSV limits. The one-mebibyte input bound prevents this exercise

from accepting arbitrarily large byte strings, but the caller still has to capture data safely before invoking it.

6. Publication must account for destination state. A candidate whose parent order does not exist can fail a foreign-key check even though its input fields are well formed. Chapter 9 supplies that relational stage. [S59]
7. A transaction should make the chosen publication unit all-or-nothing at the database boundary. It does not make a file-level rejection disappear, and it does not undo an email, payment or physical dispatch outside the transaction. [S01] [S25]
8. The worked report is an original design demonstration. It is not a claim that every organisation should retain rejected bytes indefinitely, use the same conflict policy or accept partial batches. Those are explicit product decisions to be reviewed against the actual workflow.

7.6.6 WORDS — remember these

1. **Quarantine:** retaining rejected material for controlled review — a handling state with separately specified access and retention rules.
2. **Candidate:** a record ready for the next acceptance stage — a validated, non-conflicting input that has not necessarily been published.
3. **Reconciliation:** explaining how the input relates to the output — accounting for candidates, repeats, rejections and file-level limits without silently losing material.
4. **File-level error:** a failure affecting interpretation of the input as a whole — an outcome that withholds publication when trustworthy complete parsing is unavailable.
5. **Conflict group:** records claiming one identity with incompatible content — a set for which this batch policy refuses to choose a winner.
6. **Publication boundary:** the point where accepted work becomes available in the destination — a separately controlled step beyond decoding and input validation.

7.97 Practice and worked answers

1. **Question:** two exports have different names but the same captured bytes. Does that prove two different orders occurred? **Answer:** no. Filenames identify locations or labels, not business events. Inspect the record identities and the intended delivery policy.
2. **Question:** a five-byte UTF-8 input contains A, the rupee sign and a newline. How many code points does it contain? **Answer:** three. The rupee sign needs three of the five bytes. A byte count is not a character count.
3. **Question:** a CSV description contains a comma inside quotes. Should the importer create another column? **Answer:** no. Parse according to the dialect; the comma is part of that field.
4. **Question:** JSON parsing succeeds, but `quantity` is `"2"`. Is the line accepted? **Answer:** not by this contract. The value is text, and the line validator requires an integer that is not a Boolean.
5. **Question:** two valid rows have the same line key but different quantities. Which wins in the batch lab? **Answer:** neither. The entire conflicting identity group is withheld. The event-consumer policy in Chapter 5 is a different, explicitly scoped exercise.
6. **Question:** an eight-row parse returns four candidates, one duplicate and three rejections. What total do the canonical candidates describe? **Answer:** 28,650 paise across four lines and two orders.

The file was not accepted in full, and no candidate is a committed destination row merely because it passed parsing.

7. **Question:** the parser encounters an unfinished quoted field after two apparently good records. May it claim those records were published? **Answer:** not in this lab. File-level parsing failure returns zero candidates and retains the original input. A different partial-publication policy would have to be designed and disclosed separately.
8. **Run the companion exercise:** `python3 data_bridge_lab.py` prints the canonical fixture outcomes, including the untidy-file accounting and relational examples developed in Chapters 8–9. `python3 -m unittest discover -v` runs the supplied test suite. All fixtures are synthetic; read the practice appendix before adapting anything to actual data.

7.98 Common wrong ideas

1. **Wrong:** the filename uniquely identifies the data. **Right:** a name can be reused for changed bytes, and equal bytes can appear under different names.
2. **Wrong:** one newline always ends one CSV record. **Right:** quoted fields can contain line endings; physical lines and logical records differ.
3. **Wrong:** a successful parser has validated the business data. **Right:** decoding, grammar, field rules, relationships and publication are distinct boundaries.
4. **Wrong:** quote every CSV cell and spreadsheet execution becomes harmless. **Right:** quoting handles CSV structure, not all downstream interpretation.
5. **Wrong:** JSON cannot contain ambiguous repeated names because dictionaries have unique keys. **Right:** the representation can repeat a member name; reject or handle it before information is collapsed by a runtime mapping.
6. **Wrong:** a binary format automatically makes data smaller, safer and faster. **Right:** layout, workload, implementation and security properties need separate evaluation.
7. **Wrong:** rejected rows may be discarded because they are useless. **Right:** their diagnostic value and their retention obligations require an explicit policy.
8. **Wrong:** the importer must always recover the next row after an error. **Right:** some errors destroy trustworthy boundaries. Stopping is more accurate than guessing.

7.99 Chapter summary in 20 lines

1. A path names a location; it is not a permanent identity for contents or business records.
2. The same name can lead to changed bytes, and different names can lead to equal bytes.
3. A digest identifies captured bytes under an algorithm; it does not authenticate an unknown sender by itself.
4. Bytes, Unicode code points and business records are different units.
5. Decoding, parsing, domain validation and publication answer different questions.
6. Framing makes record boundaries explicit through a defined rule.
7. A partial input must not be mistaken for a complete record.
8. CSV quoting lets one field contain delimiters and line endings.
9. A CSV header does not supply all missing meaning, units or types.
10. Our CSV adapter uses a named, deliberately narrow input profile.
11. Identifiers remain text instead of being normalised as convenient numbers.
12. JSON objects, arrays, strings, numbers, Booleans and null have distinct representation roles.

13. Parseable JSON can still violate every important rule of an order-line contract.
14. This profile rejects duplicate JSON member names and nonstandard numeric constants.
15. Binary interpretation requires an explicit layout and byte-order agreement.
16. A format marker selects a representation; it does not certify trust or truth.
17. Compatibility must be stated for particular writer and reader contracts.
18. A normal batch accounts for every parsed row as a candidate, duplicate or rejection.
19. Conflicting identities and file-level parsing failures are not silently resolved by arrival order.
20. Candidates still need destination checks and an explicit publication boundary.

8

CHAPTER

Why Databases Exist

8.0 What this chapter gives you

1. You will explain the shared-file problem without claiming that ordinary files are useless.
2. You will distinguish stored data, a database engine and the application that uses them.
3. You will compare an embedded engine with a client-server engine by tracing where the work happens.
4. You will identify the boundary of a transaction and explain what a rollback does not undo.
5. You will separate an input check, a declared database constraint and a business decision.
6. You will describe the responsibilities that remain after choosing a capable database product.
7. You will choose a storage arrangement from explicit requirements rather than a popularity ranking.

Chapter 7 ended with validated import candidates. We now need a destination that can hold related records, answer questions and control changes. The examples in this chapter use isolated synthetic fixtures. In particular, the shared-file counter RACE-STOCK is not the canonical notebook stream from Chapter 6. Nothing here supplies a new explanation for its unresolved physical-stock discrepancy.

8.1 The shared-file problem

8.1.1 PLAIN — in simple words

1. A file can hold useful data perfectly well. Trouble begins when several operations must agree about which changes belong together and which values are current.
2. Imagine that Dev and Mira both open a stock file, each work from the old value and each save a replacement. The later replacement can silently erase the earlier person's contribution.
3. The file has not necessarily become unreadable. It can remain neatly formatted and still tell the wrong combined story.
4. Another problem appears when one operation must update more than one place. Recording an order without recording its lines, or reducing stock without recording why, can leave a half-finished business operation.
5. A third problem appears when a process stops halfway through a write. The file's presence does not prove that it contains a complete new version.
6. These problems do not mean that all files should be abandoned. They mean that coordination, recovery and validation have become requirements. A database engine supplies established mechanisms for many of them, under stated limits. [S53]

7. Adding custom locks, journals and conflict checks around files may be possible. At some point the project is building part of a database system itself, whether or not it uses that name.

8.1.2 PLAIN — a picture in your head

1. Picture a shared notebook on a counter. Two people each photocopy its stock page and walk away to make changes.
2. Dev crosses out five and writes three after recording an issue of two items. Mira crosses out five and writes four after recording an issue of one item.
3. Dev puts his page back. Then Mira puts hers on top. The visible page says four, but the recorded actions together should have reduced five to two.
4. The later page looks complete. Nothing about its handwriting reveals that an earlier contribution disappeared.
5. **Where the comparison breaks:** computers can coordinate access through locks, version checks, transactions and other mechanisms. A shared file is not condemned to behave like uncontrolled photocopies. The failure comes from the specific read-modify-overwrite procedure.
6. Nor does putting the notebook in an expensive cupboard automatically fix the procedure. Moving data to a database while keeping an unsafe application sequence can preserve the same logical error.

8.1.3 PLAIN — a worked example

1. This isolated race fixture starts with the value five. Its identifier is RACE-STOCK, and its two requested deductions are separate from every canonical shop order.

Step	Dev's action	Mira's action	Stored value
0	—	—	5
1	Reads 5	—	5
2	—	Reads 5	5
3	Computes 5 – 2 = 3	—	5
4	—	Computes 5 – 1 = 4	5
5	Writes replacement 3	—	3
6	—	Writes replacement 4	4

2. Applying both deductions in sequence would give 5 – 2 – 1 = 2. The final four therefore does not reflect the complete set of requested changes.
3. If the final write were Dev's instead, the stored value would be three. Changing which person is faster changes which contribution disappears; it does not create a correct coordination rule.
4. Notice what is lost: an earlier update's effect. The bytes can remain valid, and both programs can report that their writes succeeded.
5. A different failure occurs when a program stores a new order header and then stops before storing its lines. There is no race in that example. One operation was split into separately visible pieces.
6. The diagnosis should name the failure: lost update, partial operation, malformed representation or another specific condition. "The file broke" is too vague to guide a repair.

8.1.4 PLAIN — what is really happening inside

1. The unsafe procedure separates reading, deciding and writing. Between those steps, another operation can act on the same old value.

2. Protecting only the final write is not enough if the decision was based on an earlier unprotected read. A lock used for the wrong interval can leave the important race intact.
3. A version check offers another idea: store the new value only if the record still has the version that was read. A failed check tells the caller to reconsider the decision against fresh state.
4. Some changes can instead be expressed directly at the database boundary: "deduct this quantity only when the stored quantity is sufficient". That combines the condition with the update rather than calculating a replacement from a stale value outside the engine.
5. The complete operation may still contain several statements. A transaction groups the related statements, and the chosen isolation behaviour governs interaction with other work. The word transaction alone is not an explanation of every concurrency guarantee. [S01] [S66]
6. SQLite serialises writes to a database; the details of reader interaction depend on its journalling mode. A busy or conflicting operation can require waiting, retrying or failing rather than magically completing alongside another writer. [S54]
7. The lab exercises a controlled two-connection lock case and an explicit rollback. Those are concrete local observations, not proof that every application using SQL is free of races.

8.1.5 TECHNICAL — the engineer's version

1. The timeline demonstrates a **lost update** under an application-level read-modify-write sequence. It does not depend on torn bytes, a failed storage device or a parser defect.
2. A correct design states its **invariant**, its **coordination boundary** and the allowed failure outcome. "Expected stock must not become negative" is an example invariant; it is not a universal rule for every inventory domain.
3. An atomic update can avoid one form of lost update by computing from the value at the engine's update boundary. It does not automatically protect a separate precondition checked earlier, a second database or an external action.
4. In PostgreSQL, the chosen transaction-isolation level affects which concurrent behaviours are permitted. Serializable transactions can fail with a serialization error, and the application needs an appropriate whole-transaction retry strategy. [S66]
5. In the supplied SQLite demonstration, connection A holds a write transaction. Connection B has a zero busy timeout and attempts to start another write transaction. It receives a lock-related failure; after A rolls back, B can proceed. [S25] [S54]
6. That demonstration uses a fresh temporary database on the authoring machine. It is not a multi-host shared-filesystem test, an observation of the user's laptop or a simulation of a production incident.
7. **Design rule for the examples:** record a rejected or busy request as such. Do not relabel it a completed sale, and do not blindly rerun external actions while retrying a database operation.
8. Later chapters develop isolation levels, optimistic concurrency and retry protocols in depth. At this stage the required insight is narrower: a correct individual calculation can still produce an incorrect combined history when its coordination boundary is wrong.

8.1.6 WORDS — remember these

1. **Lost update:** one accepted contribution disappearing beneath another write — a concurrency anomaly caused by insufficient coordination of read-modify-write operations.
2. **Read-modify-write:** read a value, compute a replacement, then store it — a multi-step sequence that needs an appropriate concurrency rule.

3. **Invariant:** a condition the system is meant to preserve — a stated property that every accepted state transition must maintain.
4. **Coordination boundary:** the interval or operation within which competing work is controlled — the scope of a lock, version condition or transactional mechanism.
5. **Busy outcome:** the system cannot complete the requested access immediately — a contention result that requires an explicit wait, retry or failure policy.
6. **Partial operation:** only part of a logically related change becoming effective — an incomplete business action whose pieces were not protected by the needed boundary.

8.2 A database and its engine

8.2.1 PLAIN — in simple words

1. A database is an organised body of data. A database engine is software that manages access to that data. An application is the software that uses the engine to perform the shop's work.
2. People often use "database" for all three in casual conversation. That shortcut becomes confusing when deciding which part is responsible for a failure.
3. The engine can store tables, run queries and enforce declared rules. It does not know the shop's intentions unless those intentions have been translated into actual structures, constraints and application logic.
4. A question such as "show the lines of order O-1042" is a query. A request such as "record this new line if its order exists" combines a data change with rules that must be checked.
5. Many engines can choose among several ways to find an answer. The user asks for the result; the engine plans how to obtain it within its supported operations.
6. This chapter focuses on transactional SQL engines. Not every product described as a database provides the same model, query language, transaction features or operational guarantees.
7. The name of a product is therefore not a substitute for understanding its particular promises and testing the parts on which your application relies.

8.2.2 PLAIN — a picture in your head

1. Think of an archive containing labelled files, with a trained clerk at the desk. The files represent stored data. The clerk and the archive procedures represent the engine.
2. A shop application is the person submitting a request: "find this order", "add this line", or "give me the total for these orders".
3. The clerk can refuse a form that violates a declared rule. But if the rule was never supplied, the clerk cannot be expected to infer it from the owner's intentions.
4. An index is like a maintained lookup guide. A query plan is the clerk's chosen sequence of operations for answering one request. It may use the guide, scan a collection or combine results.
5. **Where the comparison breaks:** the engine is not a human interpreter of truth. It executes defined operations. It can enforce a perfectly specified rule over completely fabricated source data.
6. The archive also needs care: backups, access control, maintenance and a recovery procedure. Hiring a clerk does not eliminate responsibility for those tasks; it changes how they are performed.

8.2.3 PLAIN — a worked example

1. Mira asks, "What is the recorded value of each of our two canonical orders?" The expected answer is still O-1042: 17,100 paise and O-1043: 11,550 paise.
2. The application submits a query over agreed line quantities and agreed line prices. The query must not accidentally substitute today's catalogue prices.
3. The engine checks the query's structure, identifies the tables and columns, selects an execution strategy and returns rows containing the calculated values.
4. The application formats paise as rupees for display. If it prints 17,100 paise as INR 17,100.00, the calculation inside the engine could be correct while the presentation is wrong.
5. If the application asks for a sum of order totals repeated on every line, the engine may correctly return 57,300 paise for the intentionally wrong query from Chapter 1. The engine does not automatically know that the application chose the wrong grain.
6. These outcomes identify different responsibilities:

Situation	Question to investigate first
Query refers to a nonexistent column	Does the query match the schema?
Query returns an intentionally double-counted total	Did the application ask the right question?
Output says rupees when the stored unit is paise	Did the presentation preserve units?
A reference names a missing order	Is the relationship declared and enforced?
A query is slow but gives the right result	What work does its execution plan perform?

7. A useful incident report separates those cases instead of saying only that "the database is wrong".

8.2.4 PLAIN — what is really happening inside

1. A SQL engine parses a statement and connects its names to known objects. It can reject syntax it cannot recognise or references that do not resolve.
2. For a query, it forms a plan using available operations such as scans, filters, joins and aggregation. PostgreSQL's `EXPLAIN` displays a representation of such a plan. [S65]
3. During execution, the engine reads or changes data through its storage and transaction machinery. The concrete layout is an implementation detail, developed later in the book.
4. Metadata describes objects such as tables, columns, constraints and indexes. This is data about the managed structure, not an independently verified description of the physical shop.
5. The application receives rows or an error through a programming interface. It must handle both, including cases where a connection fails and the final transaction outcome is uncertain from the client's perspective.
6. An execution plan is useful evidence about the work the engine intends to do. An estimated cost is not automatically a measured duration on the user's machine. [S65]
7. We preserve the distinction between "the engine returned this result for these stored inputs" and "the real-world events were complete and correct". The latter requires evidence beyond query execution.

8.2.5 TECHNICAL — the engineer's version

1. The term **database management system**, or DBMS, refers to the software and facilities managing databases. In our examples, SQLite is the embedded engine and Python is the application runtime. PostgreSQL is used as a separately identified documentation comparison. [S50] [S52]

2. The **schema** is the declared structure and rules visible to the engine. The broader domain contract also includes requirements outside that declaration, such as who may approve a stock adjustment or whether a supplied observation is authentic.
3. SQL separates many logical requests from their physical execution strategy. Two equivalent-looking queries may receive different plans, and changing data volume or available indexes can change the plan. That is why later performance claims must be measured rather than inferred from query appearance. [S65]
4. `EXPLAIN ANALYZE` in PostgreSQL executes the statement to obtain actual execution statistics. It should not be used casually on a modifying statement under the assumption that it is a harmless description-only operation. [S65]
5. A database constraint provides an enforceable rule only within its defined semantics and active configuration. A foreign key that has not been enabled in the SQLite connection is not protecting the relationship merely because its declaration appears in a schema file. [S59]
6. Bound query parameters keep data values separate from SQL text. They do not authorise the caller, validate a business rule or make a dynamically selected table name safe by themselves. The lab uses fixed SQL and bound values. [S20]
7. An ORM, a web framework and a database engine can each participate in one request. A convenient application abstraction does not erase the underlying transaction or query semantics. Diagnose at the relevant boundary instead of treating the abstraction's name as an explanation.
8. The chapter's archive metaphor is original teaching material. The documented implementation claims are specifically attributed; no assertion is made that every DBMS uses the same planner, storage organisation or failure behaviour.

■ 8.2.6 WORDS — remember these

1. **Database engine:** the software managing stored data operations — the component that executes supported queries, changes and integrity mechanisms.
2. **DBMS:** the database-management software — facilities for defining, accessing and administering databases.
3. **Query:** a request for a result from data — an expression evaluated under a data model and its language semantics.
4. **Query plan:** the engine's proposed route to an answer — an execution structure using operations such as scans, filters, joins and aggregation.
5. **Metadata:** information describing other managed information — structural definitions such as columns, constraints and indexes in this context.
6. **Bound parameter:** a value supplied separately from query text — data passed through a driver's parameter interface rather than inserted into SQL by string concatenation.

8.3 Client-server and embedded designs

■ 8.3.1 PLAIN — in simple words

1. An embedded database engine runs inside the application's process. The application calls the engine directly, like calling other library functions.

2. A client-server database engine runs as a separate service. The application sends requests through a connection, and the server process performs database work.
3. SQLite is an example of the embedded arrangement. PostgreSQL uses a client-server arrangement. These describe where the engine runs, not whether one is "real" and the other is "fake". [S50] [S52]
4. An embedded engine can manage durable files. A server engine can run on the same computer as its client. "Embedded" does not mean "temporary", and "server" does not necessarily mean "a faraway cloud machine".
5. Either arrangement can sit behind a website. The important question is which processes open the database and who coordinates access.
6. The right comparison includes deployment, concurrency, failure boundaries, access control and operational responsibility. Counting how many people can visit a website is not enough to choose the database architecture.

8.3.2 PLAIN — a picture in your head

1. In an embedded design, imagine the shop assistant keeping a skilled filing routine at the same desk. A request moves from the application to that routine without travelling to a separate service.
2. In a client-server design, imagine sending a request to a dedicated records office. The office manages its own work and sends a result back.
3. The records office could be in the same building or another city. Its separation is a process and service boundary, not necessarily a long physical distance.
4. **Where the comparison breaks:** a library call and a network request differ in scheduling, memory boundaries, transport errors and authentication, not merely in how far a clerk walks.
5. The metaphor also does not imply that the embedded application is the only possible user of a file. Multiple SQLite connections can coordinate through the engine's mechanisms, subject to SQLite's concurrency limits. [S54]
6. What would be unsafe is to assume that several unrelated machines can freely edit a database file on a shared drive with no architecture-specific concerns. The filesystem and locking assumptions matter. [S51]

8.3.3 PLAIN — a worked example

1. Consider three proposed arrangements for Mira's Corner. They are alternative teaching designs, not descriptions of an existing deployment.

Design	Request path	Who directly opens database storage?
A: one local desktop application	User → application → embedded SQLite	The local application's engine
B: one application server serving browsers	Browser → application service → embedded SQLite	The application service's engine
C: application service and PostgreSQL	Browser → application service → database connection → server engine	The PostgreSQL server

2. Designs A and B use an embedded engine, but B still serves network clients. The browsers do not directly open its SQLite file.
3. Design C adds a separate database-service boundary. The application must handle database connections, service availability and whatever authentication and permissions are configured.

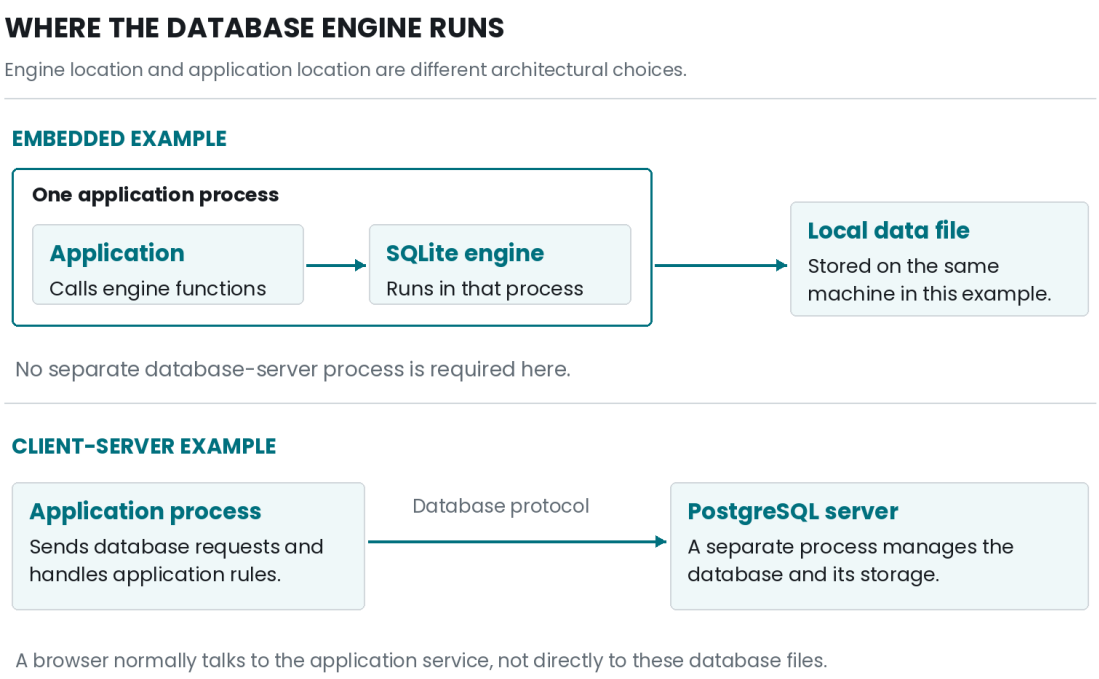

Figure 8.2: Figure 8.1 — Embedded SQLite executes in the application process; a client-server design sends requests to a separate database process. Neither picture determines the business rules or proves a performance claim.

4. None of these drawings proves that the chosen design meets a particular load target. That requires a workload, measurements and failure tests.
5. The classroom lab uses A's local engine arrangement, with in-memory or newly created temporary files. It does not contact a hosted database, expose an internet service or change any user database.
6. To keep experiments distinct, the log records the actual Python and SQLite versions used. The cited PostgreSQL behaviour is not labelled as executed merely because a similar concept appeared in the SQLite lab.

■ 8.3.4 PLAIN — what is really happening inside

1. In an embedded arrangement, calls enter the library in the same process. SQLite then performs storage and coordination through the operating-system interfaces available to it. [S50]
2. In a client-server arrangement, client and server exchange requests and results over a connection. The server manages the database files; clients should not bypass that service by editing those files themselves. [S52]
3. Process separation changes failure handling. If the application process ends, the server process may continue. If the connection fails, the application cannot always conclude that the server did nothing.
4. A connection pool is a set of reusable connections. It can reduce repeated setup work, but it does not make the server's capacity unlimited or safely preserve arbitrary transaction state between unrelated requests.

5. With SQLite, putting the database behind one application service can be different from having many machines directly access a shared database file. The project's usage guidance specifically distinguishes these situations. [S51]
6. The architecture must match the coordination boundary. If several independently deployed services all need to change shared authoritative data, the design needs a clear owner for those writes and a tested concurrency model.
7. Merely changing a connection string or moving a file to a remote folder does not supply that design.

8.3.5 TECHNICAL — the engineer's version

1. **Embedded** describes a library executing in the host application process. **Client-server** describes clients communicating with a separate database-service process. PostgreSQL clients and servers need not be on the same host. [S50] [S52]
2. SQLite's "serverless" terminology means no separate database server process. Do not confuse that meaning with a cloud provider's usage-based function or managed-service product category. [S50]
3. SQLite permits many readers but one writer at a time for a database. Write-ahead logging changes how readers and a writer can overlap; it does not create arbitrarily many simultaneous writers. [S54]
4. For multiple machines directly opening one database file, filesystem latency and locking behaviour become important. SQLite's own guidance recommends a client-server engine when many client programs send SQL to the same database over a network. [S51]
5. Those cautions are not a prohibition on every web application using SQLite. A single application server can receive many network requests while controlling local engine access. Whether that arrangement is sufficient is a workload and reliability question, not a slogan.
6. For the foundations lab, Python connections use `isolation_level=None` and explicit SQL transaction statements. That makes the demonstration's transaction boundaries visible rather than relying on a driver's implicit transaction-opening conventions. [S20]
7. A different driver, engine or transaction mode requires a separate compatibility review. Python's driver API, SQLite's SQL semantics and PostgreSQL's server behaviour must not be conflated.
8. The test suite reports its local environment and whether a test used memory or a temporary file. This disclosure matters more than presenting a diagram with the word production beneath it.

8.3.6 WORDS — remember these

1. **Embedded database:** an engine included in an application process — database functionality supplied through library calls rather than a separate server process.
2. **Client-server database:** an engine operating as a separate service — an arrangement in which clients exchange requests and results with a database server.
3. **Connection:** a session through which an application interacts with an engine — a context carrying database access and potentially transactional state.
4. **Connection pool:** a managed set of reusable connections — a bounded resource-sharing mechanism, not unlimited server capacity.

5. **Process boundary:** a separation between running programs — an isolation and communication boundary that changes failure and resource handling.
6. **Serverless, in SQLite:** no separate database server is required — SQLite's embedded-engine meaning, distinct from some cloud-service uses of the word.

8.4 Transactions and queries

8.4.1 PLAIN — in simple words

1. A transaction groups database work so that a related set of changes can succeed together or be rolled back together.
2. For example, recording an accepted request and deducting its stock should not become two unrelated successes if the application requires both to belong to one operation.
3. Beginning a transaction does not mean it has succeeded. A commit accepts its database changes; a rollback abandons its uncommitted changes. [S01] [S25]
4. A query asks for an answer from the data visible under the applicable rules. It may run inside a transaction or be handled as a statement with its own transaction boundary, depending on the engine and client mode.
5. A transaction cannot invent a missing business rule. If the application records the wrong quantity and that quantity satisfies all declared checks, committing the transaction can reliably preserve the wrong input.
6. A database rollback also cannot pull a sent email out of someone's inbox or bring a physically handed-over notebook back to the shelf.
7. The useful promise is therefore specific: these database changes are grouped under these engine, configuration and failure assumptions. "Everything everywhere is safe" is not a transaction guarantee.

8.4.2 PLAIN — a picture in your head

1. Imagine filling in a two-part shop form behind a counter screen. One part records the request; the other records the corresponding stock deduction.
2. Before the complete form is accepted, it is still a proposed operation. If a required check fails, the proposal can be discarded rather than exposing an incomplete version.
3. That is a starting picture for grouping database changes. It does not imply that all intermediate bytes are physically written at the same instant.
4. **Where the comparison breaks:** engines use concrete logging and recovery mechanisms to make a transaction appear atomic at their boundary. Those mechanisms depend on storage behaviour and configuration. The screen is only an analogy. [S53]
5. The analogy also hides concurrent readers. Different isolation levels can expose different committed snapshots or require retries; a universal "everyone sees the same page forever" story would be wrong. [S66]
6. Finally, the screen does not cover the whole world. A courier leaving while the form is being prepared is an external action, not something the database can undo by discarding the form.

8.4.3 PLAIN — a worked example

1. Use a separate teaching product P-DEMO with five available units. Request Q-01 asks for four units. Request Q-02 later asks for two. These identifiers do not replace any canonical order.

2. The first request is accepted only if sufficient stock is available. Five minus four leaves one.
3. The second request cannot deduct two from one under this fixture's non-negative-stock rule. It is rejected, leaving the stored quantity at one and creating no accepted Q-02 request record.
4. Now return to the initial five in a fresh database. Deduct four, insert Q-01 and deliberately raise an exception before commit. An explicit rollback leaves stock at five and no accepted request record.
5. The operation's intended SQL shape is:

```
BEGIN IMMEDIATE;
UPDATE demo_stock
SET quantity = quantity - ?
WHERE product_id = ? AND quantity >= ?;
-- The application must check that exactly one row changed.
INSERT INTO demo_requests(request_id, product_id, quantity)
VALUES (?, ?, ?);
COMMIT;
```

6. The question marks stand for bound values supplied by Python, not literal SQL to paste unchanged into every database console. An unsuccessful precondition must trigger the application's rollback path before an acceptance record is inserted.
7. The test deliberately raises an ordinary Python exception inside an open SQLite transaction. It is not a power failure. The result demonstrates the application's rollback logic for that controlled failure.
8. The transaction says nothing about whether four goods were physically handed over. The fixture describes accepted database requests only.

8.4.4 PLAIN — what is really happening inside

1. The engine begins a transaction and applies the conditional update. A result indicating zero changed rows tells the application that its acceptance condition was not met for the named product.
2. The application must inspect that result. If it inserts an acceptance record anyway, wrapping the statements in a transaction merely groups an incorrect decision.
3. If the update succeeds, the application inserts the request record. A duplicate request identifier causes a constraint error instead of a second independently identified acceptance.
4. The code catches failures, rolls back an active transaction and re-raises an error for the caller. It does not report success because one earlier SQL statement happened to finish.
5. If all required work succeeds, the code commits. Later chapters explain how durable engines protect committed changes through logging and recovery, and where configuration can weaken the promise. [S04] [S53]
6. Not every statement error automatically rolls back the whole SQLite transaction. Our application therefore uses an explicit rollback path rather than assuming that any exception has already restored all earlier changes. [S25]
7. After a connection failure near commit, the caller may need to establish the outcome before retrying. The local educational labs do not implement a production idempotency or reconciliation protocol for uncertain commits.

8.4.5 TECHNICAL — the engineer's version

1. The familiar acronym **ACID** names atomicity, consistency, isolation and durability. Each term needs a boundary. Atomicity concerns the transaction's database changes, not all external effects of the surrounding request. [S01]

2. Consistency means preserving the relevant declared invariants when correct application logic and engine checks are applied. It is not a claim that the engine verifies the truth of every input or automatically enforces every cross-row business condition.
3. Isolation controls interaction between concurrent transactions. Its exact guarantees depend on engine semantics and the chosen level. A stronger level may reject work that must be retried; "serializable" does not mean "no failures can occur". [S66]
4. Durability is the persistence promise for committed work under specified failures and configuration. An in-memory database supplies no survival promise after that process loses its memory. Our rollback test is not a storage-durability test.
5. With explicit transaction control, the exception path checks `connection.in_transaction` before issuing `ROLLBACK`. The application should not hide the original failure by incorrectly assuming a transaction is still active after every possible error. [S20] [S25]
6. The conditional stock update and request insert run in the same transaction. If the insert fails, the preceding deduction must be rolled back. A unit test specifically checks this path instead of testing only the successful case.
7. Duplicate identifiers are rejected in this lab, not treated as a full exactly-once protocol. A production retry mechanism would compare request identity, payload and previously committed outcome under a durable concurrency-safe design.
8. A transaction that writes one database and sends an email is not automatically atomic across both. One later pattern is to record the intention to send in the database and process it separately with explicit retry semantics; the complete protocol is outside this chapter.
9. **Scope of execution:** SQLite is executed locally. PostgreSQL isolation and asynchronous-commit claims are documentation comparisons, not results from an executed PostgreSQL server. [S04] [S66]

8.4.6 WORDS — remember these

1. **Transaction:** a defined group of database work — a unit with a controlled commit or rollback boundary.
2. **Commit:** accepting the transaction's database changes — successful completion under the engine's stated persistence and visibility semantics.
3. **Rollback:** abandoning uncommitted database work — restoration of the transaction's prior database state, not reversal of unrelated external actions.
4. **Atomicity:** the grouped changes take effect together or not at all — a transaction property at a specified resource boundary.
5. **Isolation:** rules for concurrent work seeing and affecting data — the semantics governing interaction between overlapping transactions.
6. **Durability:** the promised survival of committed work — persistence under defined failures, engine configuration and storage assumptions.

8.5 Operational responsibilities

8.5.1 PLAIN — in simple words

1. Choosing a database does not finish the job. Someone must know how to protect it, monitor it, change it and recover it.
2. A backup is a recoverable copy or representation of data. A file called `backup` is not proof that it can actually restore the required information.
3. A restore test asks a concrete question: can we rebuild a usable system from the retained material and verify the result?
4. Permissions must be deliberate. A program that can change every table is harder to contain when it has a defect or its credentials are misused.
5. Storage can fill, operations can slow and a migration can fail. Monitoring should reveal conditions that someone is prepared to act on, rather than collecting impressive numbers that nobody reads.
6. Managed services change which organisation performs particular tasks. They do not remove the application owner's need to understand its responsibilities and test the promised recovery path.
7. A good operational plan names the owner, evidence and response for each important failure. "The database is reliable" is too general to serve as that plan.

8.5.2 PLAIN — a picture in your head

1. Think of a shop safe. Buying a good safe does not decide who holds its keys, who checks its contents or how the business continues if the building cannot be entered.
2. A spare key is not a copy of the contents. A photograph of the contents is not necessarily enough to reconstruct what was inside.
3. Likewise, a replica, backup, export and screenshot have different properties. The word copy should not conceal their different recovery uses.
4. **Where the comparison breaks:** a database can preserve a transactionally consistent backup while activity continues, using an engine-supported mechanism. A camera pointed at moving paper forms cannot model that mechanism. SQLite provides a backup API for copying database content consistently. [S55]
5. Also, a logical error can be faithfully copied. If the original accepts an unwanted deletion, a current replica may reproduce it. An independent recovery history is a different need from an immediately available copy.
6. The practical question is "which failure can this retained artifact recover from?", followed by a real restoration exercise.

8.5.3 PLAIN — a worked example

1. Mira proposes two classroom recovery targets: lose no more than one hour of accepted work, and restore a usable order view within thirty minutes. These are invented targets for discussion, not measurements or recommendations for a real business.
2. Suppose the last usable backup represents 10:00 and a failure occurs at 12:20. Without additional retained changes, the backup alone leaves a two-hour-twenty-minute gap.
3. The existence of a backup therefore does not establish the one-hour target. The schedule, restore point and any additional recovery material must be compared with the actual target.
4. Suppose a tiny test database restores in one second on an idle machine. That observation does not establish a thirty-minute recovery time for a future system containing many tables, external dependencies and much more data.

5. The supplied lab performs a narrower test: copy a small synthetic SQLite database through its backup API, query the destination and compare the canonical order totals with the source.
6. It checks that the copied destination contains O-1042 at 17,100 paise and O-1043 at 11,550 paise. It does not time a production recovery, restore external documents or demonstrate an off-site backup policy.
7. The distinction between a target and tested evidence should appear in the operational report, not remain an unspoken caveat.

8.5.4 PLAIN — what is really happening inside

1. A useful backup must correspond to a coherent database state. Copying a live engine's files without its documented procedure can miss related state or combine incompatible moments.
2. SQLite's backup interface copies database pages through the engine. PostgreSQL's `pg_dump` creates a consistent logical export of one database while activity can continue. Those are different mechanisms with different scopes. [S55] [S67]
3. A recovery plan may also need application code, schema versions, configuration, required extensions and access to protected supporting material. A data copy alone is not always a complete working service.
4. After restoration, check more than whether the database opens. The exercise checks counts, keys, foreign-key consistency and known totals because each can expose a different class of mistake.
5. Change control matters too. If an application assumes a new column but the restored schema is older, the data may be intact while the service still fails to work.
6. The owner must also decide who may inspect backups and rejected input. These can contain information that the ordinary application hides from most users.
7. This book's practice package contains synthetic material only. Its backup demonstration does not create or validate a real retention, encryption or privacy-control programme.

8.5.5 TECHNICAL — the engineer's version

1. Distinguish the **recovery point objective**, the tolerated age or amount of lost work, from the **recovery time objective**, the tolerated time to resume a specified level of service. In this book they are planning terms, not guarantees achieved by running a tiny lab.
2. Define the recovery unit precisely: one database, several databases, application files, external objects, identity configuration or the complete business service. A test covering only one unit should not be reported as recovery of all of them.
3. A PostgreSQL logical dump concerns a database and its exported objects; cluster-wide objects such as roles and tablespaces need separate attention. The documented backup scope should drive the recovery checklist. [S67]
4. The local SQLite backup test uses the driver's `backup` method, then checks destination values and integrity observations. It does not emulate a failed filesystem or assert that every permitted SQLite deployment has been tested. [S20] [S55]
5. Record execution context: engine version, schema version, source boundary, destination and validation results. Do not use a command's zero exit status as the only evidence that the restored business state is correct.
6. A migration is a controlled change to a schema or related data. Keep its preconditions and rollback or recovery plan explicit; blindly executing an old migration again can be different from retrying a failed transaction.

7. Security controls must apply to ordinary databases, backup artifacts, exports and diagnostic copies according to their purpose. Adding a `noindex` tag to an online preview, for example, is not an access-control mechanism.
8. The operational checklist is an original planning aid. Product-specific backup statements are sourced; the worked recovery objectives are labelled assumptions. No certification or live-service recovery claim follows from them.

8.5.6 WORDS — remember these

1. **Backup:** retained material intended for restoration — a copy or representation with a defined consistency and recovery scope.
2. **Restore test:** trying the recovery procedure and inspecting its result — evidence about a specified backup and environment rather than a promise about all future failures.
3. **Recovery point objective:** how much recent work the plan may lose — a stated acceptable recovery-point gap, not an achieved measurement by itself.
4. **Recovery time objective:** how long recovery may take — a target for restoring a defined service level under a stated scenario.
5. **Migration:** a controlled evolution of structure or data — a versioned change with preconditions and a recovery strategy.
6. **Recovery unit:** the things the restoration claim includes — an explicit scope such as one database or a complete application service.

8.6 When a file is enough

8.6.1 PLAIN — in simple words

1. A small, stable dataset read by one program may not need a running database service. A well-specified file can be the simplest adequate representation.
2. A local application needing queries and transactions may benefit from an embedded engine without adding a separate database server.
3. A shared service with many independent writers may need a client-server architecture, but the decision should follow its actual workload and operating requirements.
4. Do not decide from row count alone. A small table with high contention and strict correctness requirements can be harder to operate than a large read-only file.
5. Do not decide from a fashionable feature list either. A feature is useful only when it addresses a requirement the system actually has.
6. The correct output of an early design discussion is a clear set of requirements and trade-offs. A product name can come afterwards.
7. The goal is the least complicated arrangement that satisfies the necessary behaviour and can be maintained by the responsible team—not the fewest lines in the first demo.

8.6.2 PLAIN — a picture in your head

1. Imagine choosing between a personal notebook, a staffed records desk and a full records office. The office is not automatically the best answer to every note-taking problem.

2. For one person keeping a fixed list of shelf labels, the office adds overhead without solving a pressing coordination issue.
3. For several people accepting overlapping orders while protecting shared stock, the notebook may be too weak unless a substantial coordination process surrounds it.
4. **Where the comparison breaks:** a simple-looking embedded database can provide substantial transactional functionality, while a large service can be badly designed. Physical size and organisational grandeur are poor measures of data-system correctness.
5. Choosing less infrastructure is not the same as ignoring failure. Even the notebook needs a decision about backup, meaning and who may change it.
6. Choosing more infrastructure is not the same as buying automatic truth. The application must still ask the right questions and model the right facts.

8.6.3 PLAIN — a worked example

1. These are three synthetic workloads, described before any product choice.

Workload	Required behaviour	A reasonable design to investigate
A fixed reference list shipped with one application release	Read-only lookup; versioned replacement; simple distribution	A validated, versioned data file
One local shop workstation	Relational queries; grouped changes; local persistence	An embedded transactional engine
Several independently deployed application instances sharing writes	Central coordination; concurrent access; service-level access rules	A client-server engine and an explicit service design

2. "Investigate" matters. The table is a reasoning aid, not a benchmark result or a universal product verdict.
3. For the local workstation, test a representative order, a rejection, a duplicate identifier and restoration of a backup. The happy path alone is not the workload.
4. For the shared service, test simultaneous requests, lock waits, failed connections and maintenance events. Include the team's ability to operate the chosen arrangement.
5. For the fixed file, test unsupported versions, malformed contents and how the application selects the intended release. "Read only" does not mean "no validation needed".
6. When a requirement changes, revisit the decision. Adding a second independent writer can matter more than adding another thousand unchanged rows.

8.6.4 PLAIN — what is really happening inside

1. Every design allocates responsibility somewhere. With a file, the application may perform more validation, searching, coordination and recovery itself.
2. With an engine, some responsibilities move into tested engine mechanisms, but the application still chooses schemas, transactions and business rules.
3. With a separate service, the system gains a clear database-process boundary and also gains connection handling, deployment and service-management work.
4. Operational cost is therefore more than license or hosting cost. It includes time spent understanding failures, restoring data and correcting a poor fit between architecture and workload.
5. Measurements should represent the intended workload. A benchmark consisting only of one fast lookup does not establish behaviour during overlapping writes, backup or schema changes.

6. The database is part of the application's design. It cannot be chosen meaningfully in isolation from who writes, what must stay consistent, what can fail and who will recover the system.

8.6.5 TECHNICAL — the engineer's version

1. Start with a requirement record containing the data model, read/write patterns, contention points, consistency needs, failure scenarios, access boundaries and operational owner.
2. Add concrete quantities only when they have a basis: anticipated record sizes, peak concurrent work, acceptable latency, growth and recovery targets. Label estimates as estimates rather than presenting invented numbers as measurements.
3. SQLite's usage guidance supports local application storage and distinguishes it from situations with many network clients directly issuing database work. Treat that guidance as a starting point, then test the actual proposed architecture. [S51]
4. File formats and database engines are not mutually exclusive categories. A database may use files internally, import CSV, expose JSON and create logical backups. The useful distinction is which component supplies which guarantee.
5. A "single source of truth" is an architectural responsibility, not a magic table property. Identify which representation is authoritative for each decision and how derived copies are updated or reconciled.
6. Avoid premature promises about seamless future migration. Different engines can differ in types, constraints, SQL, isolation and operational tooling. Portability requires tests of the features actually used, not confidence that both products accept `SELECT`.
7. The chosen teaching progression is intentional: first import records without losing their meaning, then use a local transactional engine, then model relationships. More demanding concurrency and distribution are introduced only after those foundations are visible.
8. The chapter does not choose a live storage product for the user's company. It provides the framework and bounded demonstrations needed to make such a decision from evidence.

8.6.6 WORDS — remember these

1. **Workload:** the work the system must actually handle — a defined mixture of reads, writes, sizes, timing and contention.
2. **Contention:** operations competing for a shared resource — overlapping demand that can require waiting, rejection or coordination.
3. **Authoritative representation:** the record a decision is defined to rely on — a source-of-truth role assigned by the architecture.
4. **Portability:** the ability to move a defined workload between environments — compatibility of the used semantics and operations, not merely similar syntax.
5. **Operational owner:** the person or team responsible for running and recovering the system — a named responsibility beyond writing its initial code.
6. **Design trade-off:** a choice that improves some requirements at a cost to others — an explicit comparison of behaviour and responsibility under stated assumptions.

8.97 Practice and worked answers

1. **Question:** Dev reads five and plans to write three; Mira reads five and later writes four. Why is the final four wrong for the combined deductions? **Answer:** the second replacement lost the first deduction's effect. Applying both deductions gives two. This is the isolated RACE-STOCK example, not a new explanation of the canonical stock discrepancy.
2. **Question:** does a syntactically correct SQL query guarantee the right business total? **Answer:** no. It can correctly calculate a total at the wrong grain, use the wrong unit or omit necessary records.
3. **Question:** may a web application use an embedded database? **Answer:** yes as an architectural possibility: browsers can talk to an application service that controls an embedded engine. That differs from many machines directly opening a shared database file. Capacity and reliability still need testing.
4. **Question:** a conditional update changes zero rows. May the application insert an accepted request anyway? **Answer:** not under this exercise's acceptance rule. It must reject and roll back the proposed operation.
5. **Question:** a transaction deducts stock and then its request insertion fails. What should remain? **Answer:** neither change, after the explicit rollback path. The lab tests this failure, not just the successful operation.
6. **Question:** a database rollback occurs after an email was already sent. Has the email been unsent? **Answer:** no. The external action lies outside the database transaction boundary.
7. **Question:** a 10:00 backup is restored after a 12:20 failure. Does it meet a one-hour loss target without further recovery material? **Answer:** no. The uncovered interval is two hours and twenty minutes. The target is a requirement, not a property inferred from having a backup file.
8. **Design exercise:** describe the required behaviour of a two-counter shop before naming a database. Include shared-stock acceptance, order/line consistency, duplicate requests, failure outcomes, access rights and restoration. A good answer can be reviewed even before a product is selected.

8.98 Common wrong ideas

1. **Wrong:** databases exist because files cannot store structured data. **Right:** files can store structure; engines supply additional query, coordination, integrity and recovery mechanisms.
2. **Wrong:** a valid final file proves that every update was preserved. **Right:** the lost-update example ends with valid content and the wrong combined state.
3. **Wrong:** a database automatically understands the business. **Right:** its enforceable rules and the application's decisions must be specified.
4. **Wrong:** embedded means temporary or unsuitable for any website. **Right:** embedded describes where the engine runs, not every workload it can serve.
5. **Wrong:** using a transaction solves every concurrency and external-side-effect problem. **Right:** the boundary, isolation semantics and external actions still matter.
6. **Wrong:** a successful backup command proves complete service recoverability. **Right:** restoration needs a defined scope and actual checks.
7. **Wrong:** the largest product or longest feature list is the safest choice. **Right:** suitability depends on the workload, required behaviour and operating capability.
8. **Wrong:** one successful SQLite lab proves PostgreSQL compatibility and production durability. **Right:** execution evidence belongs to its actual engine, environment and tested failures.

8.99 Chapter summary in 20 lines

1. Ordinary files remain useful; additional coordination requirements motivate database mechanisms.
2. A read-modify-overwrite sequence can lose another operation's update while leaving valid bytes.
3. Related business changes need an explicitly chosen publication boundary.
4. A database is managed information; its engine is software; the application decides how to use it.
5. An engine can execute a wrong business question correctly.
6. Query plans describe execution work, not the truth of the source records.
7. Declared rules protect only what their semantics and configuration actually enforce.
8. Embedded engines run inside the application process.
9. Client-server engines run behind a separate database-service boundary.
10. A networked application can use an embedded engine without browsers directly opening its file.
11. Transactions group database changes under commit and rollback semantics.
12. A successful precondition check must belong to the correct coordination boundary.
13. SQLite statement failure is not a reason to assume every earlier transaction change vanished.
14. Isolation semantics govern concurrent work and can require retries.
15. Durability depends on the relevant storage, engine and configuration assumptions.
16. Rollback does not reverse an external email, payment or physical handover.
17. Backups, replicas and complete service recovery are different claims.
18. Recovery targets must be compared with measured restoration evidence.
19. Choose storage from workload, correctness and operational requirements rather than fashion.
20. Local tests support local observations; they do not certify a future production system.

Tables, Rows and Relationships

9.0 What this chapter gives you

1. You will describe a table as a collection of claims with a defined row meaning, not merely a grid of cells.
2. You will distinguish a column's storage type from the business domain its values are meant to represent.
3. You will model one-to-one, one-to-many and many-to-many relationships without confusing a drawing with an enforced rule.
4. You will explain which side of a relationship a foreign key constrains and which minimum-count rules it does not establish.
5. You will model an optional relationship without inventing a fake customer record for "unknown".
6. You will join related tables while checking whether the result changes the grain or multiplies totals.
7. You will build and test a small relational destination for the canonical shop lines, including deliberately rejected inserts.
8. You will state what the model leaves out before anyone mistakes the teaching schema for a production system.

The canonical arithmetic is unchanged. Order O-1042 contains two notebooks and one pen, totalling 17,100 paise. Order O-1043 contains one notebook and two pens, totalling 11,550 paise. This chapter adds a synthetic relationship annotation for optionality: O-1042 has no linked customer profile, while O-1043 links to the invented profile C-001. Those links are new teaching assumptions, not discoveries about earlier buyers. They do not alter any price, quantity or historical event.

9.1 Tables as sets of claims

9.1.1 PLAIN — in simple words

1. A table groups records with the same intended kind of meaning. Before choosing its columns, finish the sentence "one row in this table means one …".
2. In a product table, one row may describe one product identity. In an order table, one row may describe one identified order. In an order-line table, one row describes one identified line within an order.
3. Those are different grains. An order containing two lines is still one order, and a line with quantity two is still one line.
4. A table's rows have values in named columns. The names help organise the claims, but their full meaning still comes from the contract developed in Part A.

5. A stored row does not have a dependable "position number" unless the model explicitly defines one. A query's displayed order should be requested explicitly rather than inferred from yesterday's screen. [S03]
6. The relational idea treats a relation as a set of tuples. Ordinary SQL tables and query results need an important qualification: duplicate-looking rows can occur unless constraints or query operations prevent them. [S56] [S57]
7. Therefore a useful simple definition is "a collection of same-shaped claims under declared rules", followed by an explicit discussion of keys, duplicates and ordering.

9.1.2 PLAIN — a picture in your head

1. Imagine three trays of cards. One tray contains product cards, another order cards and another order-line cards.
2. Mixing all three kinds of card in one tray would make questions harder. "How many cards?" would no longer tell you how many products or orders existed.
3. The row-grain sentence is the label on the tray: "one card for one order line". It tells the reader how to interpret a count.
4. A key is a reliable way to identify a card under the model's scope. It is not the card's current position in the tray.
5. **Where the comparison breaks:** SQL can produce repeated result rows, and a query can manufacture a new grain by joining or grouping. The result is not necessarily a simple subset of the original cards.
6. The analogy also does not grant the cards truth. A tidy tray can contain incomplete or inaccurate records. The table structure helps state and enforce some rules; it cannot verify the entire world.

9.1.3 PLAIN — a worked example

1. The four canonical lines are shown below. Their identity is the pair `(order_id, line_no)`.

order_id	line_no	product_id	quantity	unit_price_minor	Calculated line total
O-1042	1	P-NOTE	2	7,550	15,100
O-1042	2	P-PEN	1	2,000	2,000
O-1043	1	P-NOTE	1	7,550	7,550
O-1043	2	P-PEN	2	2,000	4,000

2. Count the rows: four order lines. Count distinct order identifiers: two orders. Sum quantities: six units. Sum line totals: 28,650 paise.
3. These are four different questions. A screen showing four rows does not entitle us to label the result "four orders".
4. If a query returns only `product_id`, the visible result includes P-NOTE twice and P-PEN twice. That repetition does not mean the underlying line identities are duplicates.
5. Asking for distinct product identifiers produces two values. It answers "which represented products?" rather than "how many lines?" or "how many units?".
6. Sorting by order identifier and line number makes the display reproducible for this fixture. Without an ordering clause, the application must not depend on incidental row-return order. [S03]

9.1.4 PLAIN — what is really happening inside

1. A table definition describes its columns and constraints. A primary key supplies one chosen way to distinguish its rows by values.
2. The engine may store records in a layout that changes as data is inserted, updated, indexed or maintained. A query result is computed from that managed data, not read as a guaranteed spreadsheet row order. [S56]
3. A projection chooses output columns. It can hide the very key that distinguished two source rows, making them look identical in the result.
4. `DISTINCT` removes duplicate result rows according to the selected values. It does not reconstruct omitted source identities or repair a poorly chosen row grain. [S57]
5. A filter keeps rows satisfying a condition. An aggregate combines rows into a different result, such as one total per order. Each operation should have an explicit meaning for one output row.
6. The application should document that output meaning where it matters. "One row per order" is a checkable requirement; "make a sales report" is too broad to identify many mistakes.

9.1.5 TECHNICAL — the engineer's version

1. In the relational model, a relation has a heading of attributes and a body of tuples. Mathematical set semantics exclude duplicate tuples. SQL's default multiset behaviour and NULL semantics require additional care rather than a claim that every SQL result is literally such a set. [S56] [S57]
2. The implementation uses primary keys on entity tables and the composite key `(order_id, line_no)` on order lines. The key is a uniqueness and presence requirement, not a physical pointer to a screen row.
3. An unordered query can return any order consistent with its semantics. Use `ORDER BY` when sequence is part of the required result. An index or a previously observed order is not a substitute. [S03]
4. A compact query for the canonical order totals is:

```
SELECT order_id,
       SUM(quantity * unit_price_minor) AS total_minor
FROM order_lines
GROUP BY order_id
ORDER BY order_id;
```

5. The result grain is one row per represented order identifier. An order with no lines does not appear in this query, because the input is the line table. Whether such an order should appear with an empty or zero total is a separate reporting decision.
6. SQL aggregation operates on stored values. The toy schema places explicit per-line bounds on quantity and price so individual products fit comfortably in SQLite integer storage. Arbitrarily large accumulated totals still need separate range analysis.
7. Avoid using `DISTINCT` as a general "remove wrong totals" button. Two legitimate lines can have equal amounts. Removing repeated amount values could discard valid contributions rather than eliminate erroneous join multiplicity.
8. The four-line fixture is deliberately small enough to count by hand. That makes the intended meaning independently inspectable before a more complex query obscures it.

9.1.6 WORDS — remember these

1. **Table:** a collection of records with a declared shape — a managed SQL structure whose columns and constraints give rows a defined interpretation.
2. **Row grain:** what one row stands for — the unit of representation against which counts and aggregates must be interpreted.
3. **Tuple:** one same-shaped collection of attribute values — a row-like element of a relation in the relational model.
4. **Projection, in a query:** choosing which values to display — an operation that selects output columns and can hide source identity.
5. **Multiset:** a collection that can contain repeated values — the bag-style semantics relevant to many ordinary SQL results.
6. **DISTINCT:** removing repeated output rows — a SQL operation over selected result values, not automatic repair of a data model.

9.2 Columns and domains

9.2.1 PLAIN — in simple words

1. A column has a storage type, but that is not its complete meaning. An integer can count notebooks, paise, seconds or order-line positions.
2. A domain describes the values that make sense for a particular role. A quantity may need to be a positive whole number. A currency may need to be the literal INR in this particular exercise.
3. The domain can be narrower than the type. The integer type can hold a negative value even when the shop's agreed sale quantity must be positive.
4. It can also be more specific than a range. A product identifier must name an appropriate product, not merely be a nonempty string.
5. A database constraint can enforce some of these rules. Other rules involve application workflow or external evidence and cannot be inferred from a column declaration alone.
6. Naming units in fields such as `unit_price_minor` helps, but a name does not stop a program from supplying rupees where paise were expected. The importer and schema must reinforce the agreement.
7. Clear columns let the simple explanation and the engineering implementation describe the same thing instead of relying on hidden assumptions.

9.2.2 PLAIN — a picture in your head

1. Imagine drawers labelled quantity, price and identifier. All three may contain digits written on cards, but the drawers have different acceptance rules.
2. A price drawer may permit zero for a free item under this toy contract. A sale-quantity drawer does not permit zero because a line here represents a positive number of units.
3. An identifier drawer preserves the text `00017` rather than performing arithmetic on it. Converting it to seventeen would change the agreed label.

4. **Where the comparison breaks:** database engines do not all apply identical type rules. Some convert incoming values when a conversion is possible; some reject values under different conditions. The drawer labels alone do not reveal the engine's coercion semantics.
5. A value can therefore arrive as one runtime type and be stored as another. The application should validate any distinction that would be lost during binding or conversion.
6. The goal is not the strictest imaginable rejection policy. It is a deliberate policy whose actual behaviour matches the contract and is covered by tests.

9.2.3 PLAIN — a worked example

1. Consider the `quantity` column for an agreed line. The following values have different outcomes at the input boundary.

Input value	What the Python validator sees	Version-1 outcome
`2`	Integer	Accepted
`0`	Integer outside the positive range	Rejected
`-2`	Negative integer	Rejected
`"2"`	Text	Rejected
`2.0`	Floating-point value	Rejected
`true` in JSON	Boolean	Rejected
Missing field	No supplied value	Rejected

2. These are choices in the preserved domain contract. Another application may choose different coercions, but it should make them explicit.
3. The new SQLite destination adds classroom capacity bounds: quantity at most 10,000 and agreed unit price at most 100,000,000 paise. Both canonical orders remain far inside these bounds.
4. Those upper bounds are not retroactive claims about the original business contract. They are additional storage-adapter rules for this executable destination and are reported as such.
5. A line with quantity 10,001 can satisfy the older positive-integer rule but fail this destination's capacity rule. That is a meaningful stage-specific rejection, not evidence that the JSON parser failed.
6. An unknown product identifier can satisfy every local field rule and still fail the relationship check. The next section explains why the database needs that additional rule.

9.2.4 PLAIN — what is really happening inside

1. The importer first validates Python values under the line contract. The database driver binds those values to a statement. The engine then applies its own typing and constraint semantics.
2. SQLite normally has flexible typing. Its `STRICT` table option, available since SQLite 3.37.0, narrows allowed column type names and enforces stricter storage-type behaviour. It can still perform certain lossless conversions. [S21] [S60]
3. Therefore `STRICT` does not mean "the original Python object had exactly the type my business validator wanted". Some source distinctions no longer exist after values cross the driver boundary.
4. A `NOT NULL` declaration rejects a missing database value. A `CHECK` condition can reject values outside an allowed range. In SQLite and PostgreSQL, a CHECK expression yielding NULL does not by itself reject the row, so presence must be declared separately when required. [S02] [S61]
5. A foreign key checks a relationship to another table. It cannot be replaced by a local range check on an identifier.

6. These layers are complementary. Input validation gives useful early diagnostics; database constraints also protect writes made through other code paths within the enforced configuration.
7. The implementation tests both the intended application route and direct constraint violations, without claiming that the engine can reconstruct distinctions already erased by conversion.

9.2.5 TECHNICAL — the engineer's version

1. A conceptual **domain** combines representation, allowed values and meaning. It need not correspond to a database feature named DOMAIN in every engine. SQLite's teaching schema expresses relevant rules with types, `NOT NULL`, `CHECK`, primary keys and foreign keys.
2. The destination's numeric limits are deliberate: `1 <= quantity <= 10000` and `0 <= unit_price_minor <= 100000000`. Each line amount is therefore at most 1,000,000,000,000 paise. These are lab bounds, not price or capacity recommendations.
3. Multiplication for one bounded line fits in a signed 64-bit integer. Summing indefinitely many such lines can still overflow. A per-row bound is not a proof about every future aggregate; Chapter 3's numeric-range reasoning remains necessary.
4. The schema uses `TEXT NOT NULL PRIMARY KEY` for named identifiers. This avoids relying on older SQLite behaviours that permit NULL in some ordinary primary-key declarations. `STRICT` adds further type enforcement, but explicit presence remains readable documentation. [S61] [S60]
5. `CHECK(quantity > 0)` and `NOT NULL` do different work. The CHECK does not establish presence when its expression evaluates to NULL. Declare both when both are required. [S02] [S61]
6. A Python Boolean bound directly into SQLite can become integer 0 or 1. The database then sees the stored value, not a preserved Boolean-origin label. The API's explicit `type(value) is int` rule must therefore run before binding. [S20] [S22]
7. Do not describe the SQL as universally portable. `STRICT`, pragma configuration, autocommit conventions and particular constraint behaviour are SQLite-specific. PostgreSQL is referenced separately to explain concepts and differences, not to claim that this exact script was executed there.
8. Every rejection should identify its layer: input profile, domain value, destination capacity, relational integrity or transaction outcome. This makes the system's explanation useful instead of reducing every failure to "bad data".

9.2.6 WORDS — remember these

1. **Column:** a named role within each row — an attribute position with declared type and associated meaning.
2. **Domain:** the values and meaning permitted for a role — a business-level contract potentially narrower than a storage type.
3. **Coercion:** converting a value into another representation or type — an engine or application behaviour that must be deliberate when it changes interpretation.
4. **NOT NULL:** a requirement that a database value be present — a constraint rejecting SQL NULL for a column.
5. **CHECK constraint:** a declared predicate on acceptable values — an engine-enforced condition whose NULL semantics must also be understood.
6. **Destination capacity rule:** an extra bound imposed by this storage adapter — a limitation distinct from parsing and the earlier domain contract.

9.3 One-to-one and one-to-many

9.3.1 PLAIN — in simple words

1. A relationship describes how records are allowed to refer to one another.
2. One order can contain many lines. Each line in our model belongs to one order. This is a one-to-many relationship from orders to lines.
3. A line therefore carries an `order_id` that refers to an existing order. A foreign key lets the engine check that reference.
4. A one-to-one relationship needs an additional limit. For example, a customer profile may have at most one separate preference record in our optional teaching extension.
5. "At most one" and "exactly one" are different requirements. A unique reference can prevent a second preference record without forcing a preference record to exist for every customer.
6. Likewise, requiring every line to name an existing order does not force every order to have at least one line. The foreign key points from the child to the parent, not the other way around.
7. A diagram should show these minimum and maximum counts clearly enough that someone can test the claims it makes.

9.3.2 PLAIN — a picture in your head

1. Picture one envelope for an order, with several line cards bearing its order number. Many cards can name the same envelope.
2. The line cards are the children and the order envelope is the parent in this particular relationship. These words describe a reference direction, not people or ownership rights.
3. A foreign-key check asks whether the named envelope exists. It does not automatically count how many cards each envelope contains.
4. Now picture a separate preferences card attached to a customer card. A rule permits at most one preferences card for that customer. Uniqueness supplies the upper limit.
5. **Where the comparison breaks:** there are no literal strings attaching stored rows. The engine checks values under constraints. Deleting a parent has a defined database response, not whatever happens when someone tears an envelope.
6. Different relationships can need different deletion rules. The lab chooses rejection for deleting referenced parents rather than silently cascading away order history.

9.3.3 PLAIN — a worked example

1. The two canonical order rows are O-1042 and O-1043. The four canonical lines refer to them in pairs.

Parent order	Referencing lines	Number of lines
O-1042	(O-1042, 1), (O-1042, 2)	2
O-1043	(O-1043, 1), (O-1043, 2)	2

2. Try to insert a line for O-MISSING when no such order exists. With the declared foreign key enabled, the destination rejects the line rather than creating a hidden parent order.
3. Try to insert O-1042 line 1 again. The line's composite primary key rejects that repeated identity. The foreign key alone would not reject it, because O-1042 does exist.

4. Now create the invented customer C-001 and one preference record referring to C-001. The preference record uses `customer_id` as both its primary key and its foreign key.
5. Inserting a second preference record for C-001 fails the primary-key uniqueness rule. Inserting one for C-MISSING fails the foreign-key rule. A customer with no preference row remains permitted.
6. These three outcomes prove different bounded properties: no duplicate preference identity, no dangling reference and optional absence. They do not prove that every real customer has exactly one preference record.

9.3.4 PLAIN — what is really happening inside

1. A primary key identifies a parent row. A foreign key declares that a child value must match an allowed parent key, except where an optional NULL is permitted.
2. In our line table, both parts of the line key are required. The order identifier references the order table; the product identifier separately references the product table.
3. Several child rows can use the same parent key unless another constraint forbids it. This is how the order's two lines can both refer to O-1042.
4. To enforce "at most one preference row per customer", the referenced customer value in the preference table must also be unique. Making it the primary key is one simple way to do that.
5. A parent can exist with no children under these declarations. Enforcing a minimum child count often requires workflow logic or additional mechanisms, not simply another line drawn in a diagram.
6. In SQLite, foreign-key enforcement must be enabled on each connection that uses it and checked before starting the transaction. Attempting to change the setting in the middle of a transaction does not supply the intended protection. [S59]
7. Our connection helper explicitly enables and verifies it. A test then attempts an invalid reference, so the declaration and the observed behaviour are both checked.

9.3.5 TECHNICAL — the engineer's version

1. **Cardinality** here states a minimum and maximum number of related rows. "Zero or one", "exactly one" and "zero or many" are different contracts and should be written separately.
2. The line-to-order declaration is conceptually:

```
order_id TEXT NOT NULL REFERENCES orders(order_id)
```

3. Under an enforced foreign key to a unique parent key, this gives each line exactly one matching order. It does not require every order to have a line. That reverse minimum is not implied by the declaration. [S02] [S59]
4. An optional one-to-one extension can use:

```
CREATE TABLE customer_preferences (
    customer_id TEXT NOT NULL PRIMARY KEY
        REFERENCES customers(customer_id) ON DELETE RESTRICT,
    receipt_preference TEXT NOT NULL
        CHECK(receipt_preference IN ('paper', 'none'))
) STRICT;
```

5. The table permits zero or one preference row per customer, and each preference row must reference a customer. The preference is only an educational delivery preference; it is not a legal consent record or an implemented receipt service.

6. Deletion actions such as RESTRICT, CASCADE and SET NULL express different data changes. They must follow the domain's retention and lifecycle decisions. The lab uses RESTRICT for the important parent references and does not claim that deleting a row constitutes comprehensive erasure. [S59]
7. A foreign key checks the referenced values, not an arbitrary association inferred from similar names. Matching customer display names is not a relational integrity mechanism.
8. Indexing child references can matter for performance, but an index and a foreign-key constraint have different purposes. One concerns access paths; the other concerns permitted relationships. Detailed index design follows in Part C.

9.3.6 WORDS — remember these

1. **Foreign key:** a declared reference to another allowed key — a referential-integrity constraint linking child values to parent values.
2. **Parent row:** the row being referenced — a record whose key is the target of a particular foreign-key relationship.
3. **Child row:** a row carrying a reference — a record constrained to refer to an allowed parent under the declared semantics.
4. **Cardinality:** how many related rows are permitted or required — the minimum and maximum counts on each side of a relationship.
5. **One-to-many:** one parent can have several children — a relationship whose child references need not be unique across the child table.
6. **One-to-one:** no more than one related row at each relevant side — a relationship whose optionality must still be specified separately.

9.4 Many-to-many relationships

9.4.1 PLAIN — in simple words

1. Sometimes many records on one side can relate to many records on another. A product can have several tags, and the same tag can describe several products.
2. A separate linking table can store one row for each permitted association. In our tag example, one row means one product/tag pairing.
3. Order lines also connect orders and products, but their meaning is richer than a bare pairing. A line has a line number, quantity and agreed price.
4. Crucially, the same product may appear on more than one line of the same order. That was already permitted by the line contract. A unique `(order_id, product_id)` pair would accidentally forbid it.
5. Relationships must therefore be modelled from the actual row meaning, not by mechanically turning every many-to-many diagram into the same key.
6. Joining through a linking table can multiply result rows. That can be correct for listing associations and wrong for a total that is supposed to count each order line only once.
7. The cure is to understand the new result grain, not to add `DISTINCT` and hope that the arithmetic repairs itself.

9.4.2 PLAIN — a picture in your head

1. Imagine a wall listing products and another wall listing descriptive tags. Instead of writing a changing comma-separated tag list on every product card, you keep a card for each product/tag connection.
2. A connection card carries two identifiers. It says that this particular product has this particular tag.
3. For order lines, the connection card carries more information: where it belongs within the order, how many units were agreed and at what price.
4. **Where the comparison breaks:** a line is not always just one relationship that may occur once. Repeating a product on different identified lines can be legitimate, and collapsing those lines would lose information.
5. The wall picture also hides query multiplication. When one line connects to two tags, listing line/tag pairs produces two rows from that one line. That is a new grain, not a second sale.
6. Always ask what one output row represents after the connection has been followed.

9.4.3 PLAIN — a worked example

1. Add three synthetic tag associations to the product catalogue. They do not alter any price or quantity.

product_id	tag_id
P-NOTE	paper
P-NOTE	school
P-PEN	writing

2. Joining the four canonical order lines to these associations gives six line/tag rows: each notebook line matches two tags, while each pen line matches one.
3. There are still four order lines and two orders. The joined result contains six representations of line/tag associations.
4. The notebook line amounts are 15,100 and 7,550 paise, totalling 22,650. They appear twice after this join, so their displayed contributions sum to 45,300.
5. The pen line amounts total 6,000 paise and appear once. Summing every joined row therefore gives 45,300 + 6,000 = 51,300 paise, not the correct 28,650.
6. The query did not necessarily malfunction. It summed a different collection of rows than the report intended.
7. In a separate order O-REPEAT, two different lines can both identify P-NOTE at an agreed price of 7,550 paise and quantity one. The key `(order_id, line_no)` preserves both. A key consisting only of order and product would reject the second legitimate line under the existing contract.

9.4.4 PLAIN — what is really happening inside

1. A join combines rows that satisfy its matching condition. If one row has two matches, the joined result normally contains two corresponding combinations. [S58]
2. The tag-link table has a composite primary key `(product_id, tag_id)`. This prevents the same product/tag association from being declared twice under our chosen set-like association rule.
3. That key does not belong automatically on the line table. Its grain is “one identified agreed line”, not “one distinct order/product pair”.

4. An aggregate performed after a join operates on the multiplied rows. It cannot know that the report author intended to preserve each original line's contribution only once.
5. When the question is only whether an association exists, a semantically appropriate existence test can avoid multiplying the measured rows. Another option is to aggregate at the correct grain before combining results, with the remaining relationship carefully checked.
6. Which approach is correct depends on the question. "Sales by each tag" may intentionally allocate or repeat attribution, while "total value of these orders" must not silently count a line twice.
7. Reports should state an allocation rule when one fact contributes to several categories. A label such as "sales by category" does not decide whether overlapping categories should sum to the grand total.

9.4.5 TECHNICAL — the engineer's version

1. A **junction table**, also called an associative table, represents relationships as rows. Its key must match the intended association identity rather than follow a naming convention automatically.
2. The executable tag extension uses:

```
CREATE TABLE product_tags (
    product_id TEXT NOT NULL
        REFERENCES products(product_id) ON DELETE RESTRICT,
    tag_id TEXT NOT NULL
        REFERENCES tags(tag_id) ON DELETE RESTRICT,
    PRIMARY KEY (product_id, tag_id)
) STRICT;
```

3. Both foreign keys are required, and their pair is unique. Each product can have zero or many tags; each tag can describe zero or many products. No rule here requires either side to participate.
4. The order-line relation instead uses `(order_id, line_no)`. `product_id` is a non-unique foreign key. A regression test inserts the same product on two distinct lines in an isolated order and confirms that both survive.
5. The tag-join test calculates the intentional fan-out result, 51,300 paise, and compares it with the base-line total, 28,650. Its purpose is to expose the grain error, not present the larger number as a valid sales total.
6. To ask for orders' lines whose product has the paper tag without changing line grain, one possible form is:

```
SELECT l.order_id, l.line_no, l.quantity, l.unit_price_minor
FROM order_lines AS l
WHERE EXISTS (
    SELECT 1 FROM product_tags AS pt
    WHERE pt.product_id = l.product_id AND pt.tag_id = 'paper'
)
ORDER BY l.order_id, l.line_no;
```

7. The existence test is about whether a matching row is present, not how many matches it has. Joins and existence tests are tools for different result semantics; neither is a universal performance winner. [S58]
8. SQL's join behaviour is documented; the product tags, counts and totals are original synthetic calculations. Their simplicity lets the reader verify the reasoning without relying on an unexplained benchmark or an unseen production dataset.

9.4.6 WORDS — remember these

1. **Many-to-many:** several records on either side can relate — an association that needs an explicit representation and participation rules.
2. **Junction table:** a table of links — an associative relation whose rows represent a defined pairing or richer relationship.
3. **Composite key:** an identity made from several values together — a unique column combination whose scope is not supplied by any one component alone.
4. **Join:** combining rows through a stated condition — an operation whose result can contain multiple matches for one source row.
5. **Fan-out:** one source row producing several matched result rows — multiplicity introduced by a one-to-many match in a query.
6. **Existence test:** asking whether at least one match is present — a condition that need not reproduce every matching row in the output.

9.5 Optional relationships

9.5.1 PLAIN — in simple words

1. A relationship can be optional without the whole row being meaningless. A walk-in order may exist even when the shop has not linked it to a customer profile.
2. Our new teaching annotation makes O-1042 such an order. Its `customer_id` is NULL, meaning "no linked profile in this model". The order's quantities and agreed prices remain known.
3. O-1043 links to the invented customer C-001. That link is a separate recorded association, not proof of a verified real-world identity.
4. A fake customer row called UNKNOWN is not automatically a better model. It can mistakenly group many unrelated people under one identity or suggest that the system has a profile when it does not.
5. NULL does not explain every reason a link is absent. If the distinction between not requested, not supplied and deliberately unlinked matters, the model needs an explicit, carefully defined way to record it.
6. A report must decide whether to keep orders without profiles. An inner join can omit them; a left join can preserve the order side and show absent customer values. [S58]
7. Missing optional relationships should be visible as optionality, not silently converted into lost orders or invented people.

9.5.2 PLAIN — a picture in your head

1. Imagine an order envelope with an optional customer-reference pocket. Some envelopes contain a profile card number; others do not.
2. An empty reference pocket does not mean that the order envelope is empty or that no human bought anything. It means the chosen link is absent.
3. A report listing every order should keep envelopes with empty reference pockets. A report specifically listing linked customer profiles can deliberately omit them.

4. **Where the comparison breaks:** SQL NULL has defined comparison semantics. It is not the text "blank", the number zero or a universal wildcard that matches every customer. [S17] [S24]
5. The picture also does not justify collecting more identifying information merely to fill the pocket. Whether a link is needed belongs to the workflow and data-minimisation decision, not to a desire for visually complete tables.
6. The exercise uses one invented profile because it needs a visible example of optionality, not because every small sale requires a customer identity system.

9.5.3 PLAIN — a worked example

1. These are the newly introduced relationship annotations.

order_id	currency	customer_id
O-1042	INR	NULL
O-1043	INR	C-001

2. The customer table contains one synthetic row: C-001, display label "Study customer". There is no UNKNOWN customer row.
3. An inner join on the customer identifier returns only O-1043. This can be correct for "orders with linked profiles", but is incomplete for "all orders".
4. A left join starting from orders preserves both order rows. O-1042 has NULL in the joined customer columns because no matching profile was required or found.

```
SELECT o.order_id, c.customer_id, c.display_name
FROM orders AS o
LEFT JOIN customers AS c ON c.customer_id = o.customer_id
ORDER BY o.order_id;
```

5. The first output row is O-1042 with absent customer values; the second is O-1043 with C-001 and the study label.
6. Add `WHERE c.customer_id IS NOT NULL` and the result again contains only O-1043. The left join does not override a later condition that intentionally removes the unmatched row.
7. Counts also differ: counting all preserved order rows gives two, while counting the non-null matched customer identifiers gives one. Neither count should be labelled ambiguously as "customers served" without defining what is being counted.

9.5.4 PLAIN — what is really happening inside

1. A nullable foreign-key column can represent no reference. For our single-column SQLite reference, NULL does not require a matching parent row. A supplied non-null key still must match. [S59]
2. During an inner join, only matching row combinations are returned. A left join also keeps unmatched rows from its left input and supplies NULL values for the missing right-side contribution. [S58]
3. Conditions applied afterwards operate on that result. A test requiring a right-side value can remove the very unmatched rows the left join preserved.
4. The query's intended population therefore matters at every stage. "All orders" is a different starting promise from "orders with recorded customer profiles".
5. The database does not treat C-001's display name as the order reference. Renaming the display label need not break the relationship because the stable key is separate.

6. Deleting a referenced customer is rejected by this lab's chosen rule. Automatically removing the linked orders would be a very different lifecycle policy, and this chapter does not implement it.
7. Likewise, setting a link to NULL would remove that association in this table, not prove that every copy or derivative identifier had been erased from the whole system.

9.5.5 TECHNICAL — the engineer's version

1. Specify optionality as a domain statement: each order has zero or one linked profile in the teaching model. A profile can be linked from zero or many orders. The foreign-key column is nullable; the parent key is unique and required.
2. The SQLite foreign-key rule permits a NULL child key. Composite references require additional care because partially null key combinations can affect matching semantics; this introductory customer reference is intentionally single-column. [S59]
3. Equality against NULL does not behave like equality against an ordinary scalar. Use the documented `IS NULL` or `IS NOT NULL` predicates when testing for absence. [S17] [S24]
4. A left join is not a blanket guarantee that all left rows survive every later operation. Subsequent filtering, grouping and projection can change the result's population and grain. [S58]
5. `COUNT(*)` counts result rows, while `COUNT(c.customer_id)` counts non-null expressions. After this left join the values are two and one respectively. This reuses Chapter 3's distinction in a relational setting. [S19]
6. The engine can check that C-001 exists. It does not establish that C-001 corresponds to a verified person, that two profiles describe different people or that a user is allowed to view the profile. Those are separate identity and access concerns.
7. Do not fill NULL with a shared fabricated entity solely to simplify joins. A sentinel entity is justified only when it has a real, explicitly defined meaning in the domain rather than hiding an absence.
8. This model intentionally avoids names, phone numbers and addresses as order keys. Its one display label is invented. The schema is not a customer-identity product or a complete privacy-lifecycle implementation.

9.5.6 WORDS — remember these

1. **Optional relationship:** a permitted link that need not be present — a reference with a minimum participation count of zero.
2. **Nullable foreign key:** a reference column that can contain SQL NULL — an optional association under the engine's stated matching semantics.
3. **Inner join:** returning matching row combinations — a join that omits rows without a match in the joined relation.
4. **Left join:** preserving the left input when no right match exists — an outer join that supplies NULL right-side values for unmatched left rows.
5. **Sentinel entity:** a special record used to stand for a designated case — a modelling choice that must not fabricate identity or conceal unknown meaning.
6. **Result population:** the records or combinations included in an answer — the scope after joins and filters, which must match the report's intended question.

9.6 Modelling the shop

9.6.1 PLAIN — in simple words

1. We now assemble a small destination for the canonical lines. Four core tables are enough for this teaching step: products, customers, orders and order lines.
2. The product table describes catalogue identities. The customer table contains the one invented profile used to teach optionality. The order table identifies orders and their optional profile links. The line table records quantities and agreed prices within orders.
3. Three small extension tables illustrate optional one-to-one preferences and product tags. They are not necessary to compute order totals.
4. The agreed price remains on the line. A later change to a catalogue price must not silently rewrite what an earlier order agreed.
5. The currency is stored once on each order in this destination, because the destination supports only INR orders. The incoming line still carries currency for validation; storing it at the parent is an explicit modelling decision, not an accidental dropped field.
6. The raw input and import report remain separate evidence. Reconstructing a normalised row is not the same as reproducing every original byte, header or formatting choice from the source file.
7. The model is intentionally incomplete. It does not implement payment, tax, permissions, fulfilment, refunds, the Chapter 6 event store or a final order-lifecycle service.

9.6.2 PLAIN — a picture in your head

1. Picture a small records desk with an order envelope, its line cards, a catalogue index and an optional pointer to a customer profile.
2. The order envelope says which order it is and which currency applies. Each line card says which product, how many units and the agreed price for that line.
3. The catalogue index can change its current display price without reaching into every old envelope and changing history.
4. **Where the comparison breaks:** normalisation is not simply "put each fact in one physical place". Repeated values can be meaningful snapshots, and retained source evidence may intentionally duplicate representations for a different purpose.
5. The decision is about which fact is authoritative for which question. "Current catalogue price" and "agreed price on this old line" are different facts even when their numeric values happen to match.
6. This desk is a model of data responsibilities, not a finished shop. The actual business still needs rules governing who can create, confirm, change and retire records.

9.6.3 PLAIN — a worked example

1. Begin with the two product identities P-NOTE and P-PEN, the invented customer C-001, and the two order identities O-1042 and O-1043. Do not create order lines yet.
2. Parse the untidy CSV from Chapter 7. The report yields four candidates, one duplicate and three rejected new rows. The operator-facing explanation still states that the file was only partially acceptable.
3. For this demonstration, explicitly publish the four candidates as a group. The destination checks product and order references, applies its additional numeric bounds and inserts all four lines within one transaction.

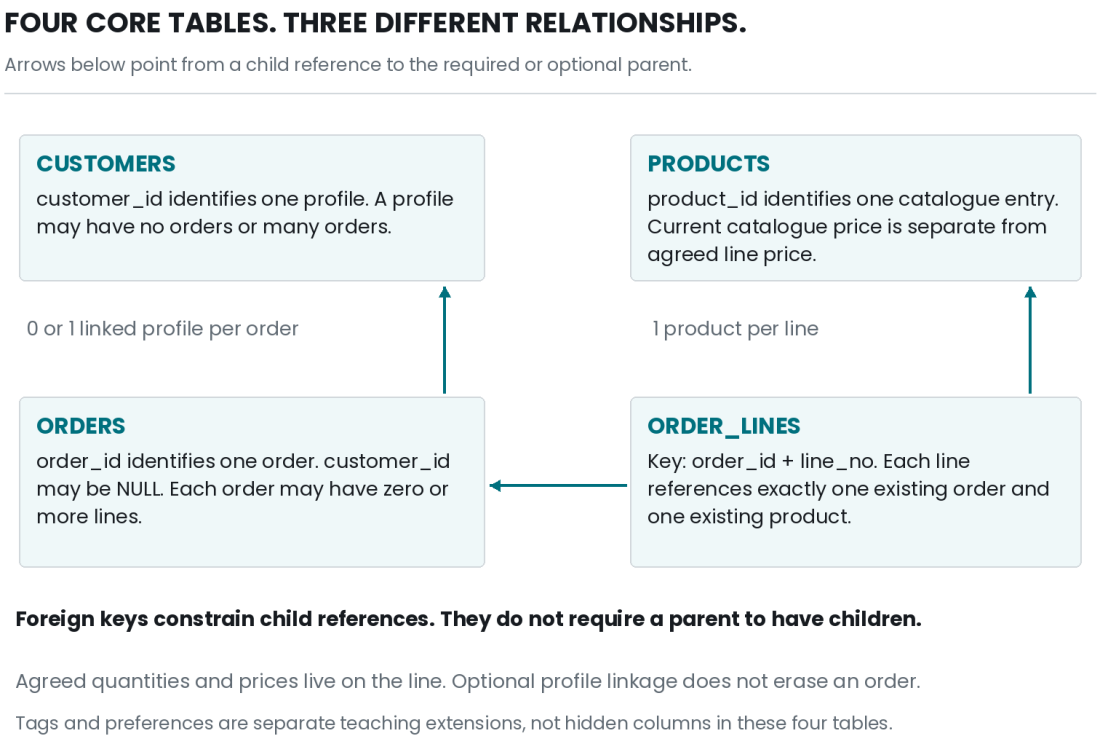

Figure 9.3: Figure 9.1 — Core shop relationships: each line names one existing order and one existing product; an order optionally names one customer. Parent rows may have zero children unless another rule establishes a minimum.

4. Query the destination: O-1042 totals 17,100 paise; O-1043 totals 11,550. The four lines contain six units. These outputs match the earlier hand calculation.
5. Change the current catalogue notebook price from 7,550 to 8,250 paise in a separate catalogue-update exercise. The two historical order totals remain unchanged because they use agreed line prices.
6. Try a separate publication containing one valid line followed by a line referencing P-MISSING. The second line fails its foreign-key check, and the first insertion is rolled back. The destination receives no partial set from that failed publication.
7. Try to publish the same four canonical lines again. Their existing primary keys cause rejection; the lab does not treat that as a new sale or advertise a production retry protocol.
8. These tests connect parsing, relationships and transaction boundaries. None requires invented customer data, a hosted service or a real shop database.

■ 9.6.4 PLAIN — what is really happening inside

1. The database is created in a fresh in-memory connection for the main relational exercise. The connection helper enables foreign keys before any transaction and checks that enforcement is active.
2. Core tables are created with SQLite's STRICT option. Named identifiers are explicitly required, line identity is composite, and quantity and price bounds are declared.
3. The fixture loader inserts known parent rows. The publication function does not silently invent missing products or orders merely to make a foreign-key failure disappear.

4. Before binding, the function checks the source objects and the destination bounds. It starts one explicit transaction for the selected candidate set, inserts each line and commits only after all succeed.
5. A failure rolls back that transaction. The caller receives a failure rather than a count of how many earlier inserts temporarily succeeded.
6. Queries derive totals from the agreed line values. Current product labels or catalogue prices serve different purposes and do not replace those historical values.
7. The extension tables are populated only with the labelled synthetic preferences and tag associations. They demonstrate relationship semantics without being presented as an integrated marketing, receipt or analytics product.
8. The lab closes every connection and removes its own temporary test directory. It neither discovers nor modifies the user's existing files or databases.

9.6.5 TECHNICAL — the engineer's version

1. The core line table used by the lab is:

```
CREATE TABLE order_lines (
    order_id TEXT NOT NULL
        REFERENCES orders(order_id) ON DELETE RESTRICT,
    line_no INTEGER NOT NULL CHECK(line_no > 0),
    product_id TEXT NOT NULL
        REFERENCES products(product_id) ON DELETE RESTRICT,
    quantity INTEGER NOT NULL
        CHECK(quantity BETWEEN 1 AND 10000),
    unit_price_minor INTEGER NOT NULL
        CHECK(unit_price_minor BETWEEN 0 AND 100000000),
    PRIMARY KEY (order_id, line_no)
) STRICT;
```

2. The order table has a required, unique order key, an INR-only currency constraint and a nullable customer reference. The product and customer tables have required unique identifiers. SQL file contents and a full fixture are included in `data_bridge_lab.py` rather than leaving the reader with disconnected pseudocode.
3. The incoming `record_type` and `schema_version` describe the message contract. They are not repeated on every line row in this destination. The adapter version, source bytes and import report are separate concerns; the lab does not claim a persistent audit trail simply because those values exist in memory.
4. The publication function receives an explicitly selected candidate set. Its all-or-nothing guarantee is for that set in one database transaction, not for all eight original source rows, which already have distinct rejection and duplicate outcomes.
5. **Important incompleteness:** the core schema permits an order with no lines. It also does not connect the separate order-state machine from Chapter 6 to a confirmed-order minimum-line rule. Building that integrated workflow requires additional application and database design.
6. Relationship checks, uniqueness checks, source typing and destination bounds are all exercised. A tag fan-out example deliberately demonstrates a wrong aggregate, and an optional-customer query deliberately contrasts inner and left joins.
7. The correct fixture total is calculated independently in Python and in SQL. Agreement provides evidence for these fixture paths, not proof that every future query is correct. Tests must include adversarial and alternative-grain inputs rather than only reproducing one expected number.

8. The schema is SQLite-specific teaching code. A PostgreSQL implementation, durable import ledger, authorization boundary, migration chain and production recovery design are not included or implied.
9. The next chapter, **Designing a Schema That Matches the Work**, develops the workflow questions this small model deliberately leaves open. A diagram and a successful insert are the beginning of a model review, not its conclusion.

9.6.6 WORDS — remember these

1. **Agreed line price:** the price attached to a particular historical agreement — a transaction-specific fact distinct from a product's current catalogue price.
2. **Catalogue price:** the current listed price in the product model — a mutable reference value that must not silently replace historical agreement values.
3. **Referential integrity:** references remaining valid under declared rules — the property enforced by active foreign-key constraints within their semantics.
4. **Publication unit:** the selected set of records accepted together — the transaction scope for this destination, distinct from an entire source file.
5. **Schema boundary:** what the declared database model does and does not enforce — the limit beyond which application workflow or other mechanisms remain necessary.
6. **Regression test:** a repeatable check preserving an intended behaviour — evidence that a known property still holds for the tested inputs after a change.

9.97 Practice and worked answers

1. **Question:** the canonical query returns four order-line rows. How many orders and units are represented? **Answer:** two orders and six units. Four is the line count, not the order or unit count.
2. **Question:** is `(order_id, product_id)` an appropriate unique key for the existing line contract? **Answer:** no. The same product may appear on multiple separately numbered lines. Use the preserved `(order_id, line_no)` identity.
3. **Question:** does a required line-to-order foreign key force every order to contain a line? **Answer:** no. It forces each line to reference an order; it does not establish the reverse minimum count.
4. **Question:** does `CHECK(quantity > 0)` alone establish that quantity is present? **Answer:** no. The relevant engines permit a CHECK whose expression is NULL. Add `NOT NULL` when presence is required.
5. **Question:** why does the canonical line/tag join yield 51,300 paise if every output line amount is summed? **Answer:** the two notebook line amounts, totalling 22,650, each appear twice, while the pen amounts total 6,000 once. The new grain is line/tag association, not order line.
6. **Question:** should `SUM(DISTINCT line_amount)` be used as a general repair? **Answer:** no. Equal amounts can belong to legitimate different lines. The correct repair follows the intended identity and result grain.
7. **Question:** an inner customer join yields only O-1043. Has O-1042 been deleted? **Answer:** no. O-1042 lacks a linked profile under the teaching annotation and was omitted by the join. A left join can preserve it.

8. **Question:** a line passes JSON and field checks but names an absent product. Where should the failure be reported? **Answer:** at relationship validation or the active foreign-key boundary, not as a JSON syntax failure.
9. **Question:** changing the notebook catalogue price to 8,250 changes neither canonical order total. Why? **Answer:** historical calculations use the agreed prices retained on their lines, not the current catalogue values.
10. **Design exercise:** draw the four core tables and label both minimum and maximum participation counts. Then state three rules the diagram does not enforce. Good examples include a confirmed order requiring a line, authorization to change a price and proof that a recorded handover physically occurred.

9.98 Common wrong ideas

1. **Wrong:** a table is a spreadsheet with a guaranteed row order. **Right:** SQL results need explicit ordering when sequence matters.
2. **Wrong:** a column type completely defines its business meaning. **Right:** units, domains, relationships and workflow rules go beyond the type name.
3. **Wrong:** a primary key is a physical row address. **Right:** it is a declared identity by values under the table's model.
4. **Wrong:** a foreign key enforces every count on both sides of a relationship. **Right:** its reference direction and optionality are specific; reverse minimum counts need separate mechanisms.
5. **Wrong:** every many-to-many table should use only its two foreign keys as the key. **Right:** the row's actual identity may include another value, as order-line numbering does here.
6. **Wrong:** a left join guarantees that all left rows remain after every later filter. **Right:** later conditions can remove unmatched rows.
7. **Wrong:** filling every optional link with an UNKNOWN entity improves data quality. **Right:** it can invent identity and conceal meaningful absence.
8. **Wrong:** the current product price can replace the agreed historical price. **Right:** those are different facts with different authority.
9. **Wrong:** passing inserts and a clean diagram make this a complete shop system. **Right:** lifecycle, permissions, durable import history and operational behaviour remain outside the model.

9.99 Chapter summary in 20 lines

1. Define what one row means before choosing its columns.
2. Orders, lines, products and units are different counting grains.
3. SQL tables and query results require care with duplicates and explicit ordering.
4. A storage type is narrower than the complete meaning of a column's domain.
5. Input types can lose distinctions during driver binding or engine coercion.
6. Presence, range and relationship checks supply different protections.
7. Primary keys identify rows by declared values rather than physical positions.
8. Foreign keys constrain references under the engine's active configuration.
9. One-to-many references do not force every parent to have a child.
10. At most one related row is not the same as exactly one required row.
11. Many-to-many relationships need a linking representation with the correct grain.
12. The existing order-line key remains order identifier plus line number.

13. Joining one line to several tags can multiply its contribution to a total.
14. DISTINCT cannot generally repair a misunderstanding of identity or grain.
15. An optional customer link does not make an order absent or incomplete by itself.
16. Left joins preserve unmatched left rows before subsequent filters are applied.
17. Agreed line prices and current catalogue prices are different facts.
18. The canonical totals remain 17,100 and 11,550 paise, together 28,650.
19. The publication transaction groups the selected candidates, not rejected source rows.
20. A useful schema review states both its enforced rules and its unfinished workflow boundaries.

10

CHAPTER

Designing a Schema That Matches the Work

10.0 What this chapter gives you

1. You will turn a description of somebody's work into a small set of records with explicit meanings.
2. You will separate the question "what do people need to do?" from the question "which tables should we create?".
3. You will write a one-row sentence for every table and recognise designs that mix incompatible grains.
4. You will distinguish an editable draft, a current catalogue value and an agreed historical value.
5. You will document units, bounds, missing values, identity scope and source evidence instead of leaving them hidden in names.
6. You will distinguish a business domain from a storage type, and a general schema design from PostgreSQL's particular namespace called a schema.
7. You will review a proposal with examples that should work and counterexamples that should fail.
8. You will identify requirements the teaching database does not yet enforce, without silently claiming that a diagram or test closes them.

Chapters 7–9 gave us formats, an engine and related tables. This chapter asks whether those tables describe the right work. The existing four agreed order lines, their identifiers and their prices stay unchanged. A separate draft workspace introduced below is new fictional teaching material. It is not a replacement for the earlier agreed-line contract and it does not explain the unresolved stock observation.

10.1 Workflows before tables

10.1.1 PLAIN — in simple words

1. A schema is a description of how a database organises information: which kinds of record exist, which values they hold and which rules connect them. Here we use the word for the overall design, not just a drawing of boxes.
2. Start with work that somebody must complete. Mira wants to record an agreed sale, find its lines and answer what was agreed. Dev may also need to prepare an unfinished draft. These activities do not necessarily have the same acceptance rules.
3. "We need an orders table" is already a proposed solution. "We need to distinguish an unfinished quotation from an agreed order" is a requirement the solution must address.

4. A workflow is the sequence of actions and decisions around that requirement. It includes unsuccessful actions: an unknown product, a missing quotation, a repeated submission or a rejected change.
5. A useful design asks what must be known before each action, what is allowed to change, what must remain true afterwards and who is allowed to perform it. The last question is a permission requirement, not something a table name answers.
6. Do not collect a field merely because another application has one. A field creates interpretation, validation, maintenance and access responsibilities. Explain its purpose or leave it out of the first proposal.
7. Equally, do not omit a distinction because it complicates a screen. A convenient screen that combines "proposed price" and "agreed price" can destroy the very information the business needs later.

10.1.2 PLAIN — a picture in your head

1. Imagine designing a kitchen before asking whether it serves tea, full meals or a hundred packed lunches. You might buy impressive equipment and still put the washing area in the wrong place.
2. Understanding the work comes before arranging the cupboards. In a database, cupboards are tables and the movement between workstations resembles the workflow.
3. Follow one request from arrival to completion. Notice where it waits, where someone makes a decision and where another person needs evidence of that decision.
4. Now follow a request that fails. Where does the unfinished work go? Does it remain visibly unfinished, disappear, or accidentally become a completed record?
5. **Where the comparison breaks:** a database can contain several representations of the same activity, and programs can act simultaneously. Physical kitchen order does not model transaction isolation or prove that a computer operation is atomic.
6. The picture is useful for discovering responsibilities, not for choosing a storage engine or deriving its guarantees. Those require their own evidence.

10.1.3 PLAIN — a worked example

1. Write the small shop requirement in sentences before SQL. The following is a teaching proposal, not a report of a real interview with Mira or Dev.

Work somebody needs to do	Information needed	Question the design must settle
Prepare an unfinished line	Product, proposed quantity, possibly a quoted price	Can an unfinished price be absent?
Record an agreed line	Order identity, line identity, product, quantity, agreed price	What evidence and operation establish agreement?
Change today's catalogue price	Product identity and new current price	Which historical records must stay unchanged?
Display an existing order	Its identified lines and their agreed values	Is the screen showing history or repricing it?
Correct an error	Original claim, correction and appropriate authority	Is correction supported here, or only proposed?

2. Our existing agreed-line contract requires a present, nonnegative integer price in paise. An unfinished quotation can be a different kind of record, with an explicitly absent price.

3. Therefore the proposal does not remove the price requirement from `order_lines`. It introduces an isolated `sql_draft_lines` teaching workspace for later exercises.
4. Existing order O-1042 still contains two notebooks at 7,550 paise each and one pen at 2,000 paise. Its total remains 17,100 paise.
5. Draft D-SQL-01 is a different invented activity. Its first line proposes three notebooks at 7,550 paise; its second proposes one pen whose price has not yet been quoted. These are not extra agreed sales.
6. "Move a finished draft into agreed history" remains an unimplemented workflow in the foundations lab. We can describe its requirements without pretending the draft table already performs that transition.

10.1.4 PLAIN — what is really happening inside

1. A user action reaches application code, which interprets it and sends operations to the engine. The engine enforces the rules it knows. It does not automatically know every promise written in a product document.
2. Design work maps each promise to an actual mechanism. A required value may become a not-null rule; a valid product reference may become a foreign key; authority to agree a sale needs a permission and workflow mechanism.
3. Some promises span several records. Publishing an order and its lines as one unit requires more than a collection of individually acceptable inserts. The earlier publication lab uses a transaction for its bounded operation; later chapters develop the larger concurrency problem.
4. Separating responsibilities prevents a common misunderstanding: "the form checks it" is not the same claim as "every writer is prevented from violating it". An import tool or maintenance script may not use that form.
5. The reverse misunderstanding is also costly. A database rule that rejects missing prices everywhere may block a legitimate unfinished draft. The solution is to model the stage honestly, not to fill unknown prices with zero.
6. Keep a requirements-to-mechanisms record. A blank mechanism entry is a visible gap, not permission to assume that some other component must handle it.

10.1.5 TECHNICAL — the engineer's version

1. Separate conceptual, logical and physical decisions. The conceptual model identifies entities, events and relationships in the work. The logical model gives record grain, attributes and constraints. The physical implementation chooses actual tables, types, indexes and engine-specific behaviour.
2. These layers are a design aid, not a claim that a project must finish one permanently before touching the next. A storage limit or awkward query can reveal a conceptual ambiguity and send the design back for clarification.
3. For each operation, record preconditions, postconditions and failure disposition. For the toy publisher, the input must satisfy the agreed-line profile, referenced parents must exist and the owned transaction must either publish the chosen batch or roll back its new lines.
4. Do not confuse those postconditions with facts the publisher cannot establish. A structurally accepted price does not prove a buyer agreed to it. The lab contains no authenticated human approval process.

5. Draw the write paths as well as the read paths. Application edits, imports, scheduled jobs and direct administrative access may have different checks. A rule claimed as universal needs coverage of the relevant writers, not merely the preferred user interface.
6. Use an explicit scope statement. The foundations case models INR-only, whole-unit, positive-quantity agreed lines without discounts, taxes, returns, delivery charges or payment settlement. A future extension must define those meanings rather than smuggle them into negative quantities or overloaded fields.
7. Database definitions specify types and declared constraints; they do not implement an entire business process by implication. PostgreSQL's table-definition reference and SQLite's corresponding reference describe those mechanisms separately from our original shop policy. [S74] [S61]
8. A reviewable proposal can therefore say: "record structure implemented; agreed-price immutability not implemented; draft finalisation not implemented; authorization not implemented". That is a more useful engineering statement than "the order module is complete".

■ 10.1.6 WORDS — remember these

1. **Schema design:** the plan for organising records and their rules — a model of structures, domains, relationships and invariants implemented by a database and its surrounding system.
2. **Workflow:** the steps around completing a piece of work — a sequence of operations, decisions and failure paths with defined state transitions.
3. **Precondition:** something that must hold before an action — a predicate required at an operation's entry boundary.
4. **Postcondition:** something promised after an action — a predicate the completed operation is intended to establish or preserve.
5. **Write path:** a route through which information can change — an application, import, job or administrative mechanism capable of mutating stored state.

10.2 Row grain and boundaries

■ 10.2.1 PLAIN — in simple words

1. Finish this sentence for every table: "one row means one …". Then test it against every column you propose to put there.
2. One order line can contain a quantity of two. It is still one line, not two rows waiting to be unfolded. One order can contain several lines without becoming several orders.
3. A field belongs with the fact it describes. An order's currency applies to the order in this exercise; the agreed unit price belongs to the identified line; today's catalogue price describes a product at the current catalogue state.
4. Choosing a boundary means deciding which things are represented separately. We separate orders from lines because one order can have several independently identified line entries.
5. An identity boundary is equally important. Line number 1 is meaningful within O-1042, not across every order in the shop. The complete identifier is the pair.
6. Do not make a one-row sentence so vague that it covers everything. "One row means some data about a sale" cannot tell you whether repeating an order total on every line is safe to sum.

7. A table can be internally tidy but still describe the wrong grain. The error appears when someone asks a question the model cannot answer without guessing.

10.2.2 PLAIN — a picture in your head

1. Imagine an order envelope containing several numbered slips. The envelope identifies the order; each slip identifies one line within it.
2. The total written on the envelope describes the entire envelope. Copying that total onto each slip does not make it a separate contribution from each slip.
3. Two slips may name the same product and still be different lines. Perhaps they are separately agreed entries. Their difference is recorded by line identity, not inferred from the product name.
4. Product cards live in another tray because products can exist before any order and can appear on many orders. Customer profile cards are another kind of thing again.
5. **Where the comparison breaks:** the computer does not need literal nested storage for an order and its lines. Related SQL rows may live in separate managed structures and be combined by a query.
6. The envelopes also do not settle whether an empty order is allowed. Our teaching schema permits an order without lines; any stronger completion rule must be stated and implemented separately.

10.2.3 PLAIN — a worked example

1. Compare three ways of counting the unchanged fixture.

What one counted thing means	Canonical count	Why
An identified order	2	O-1042 and O-1043
An identified order line	4	Two line identities within each order
One physical unit represented by the quantities	6	2 + 1 + 1 + 2

2. Suppose a report copies 17,100 onto both lines of O-1042 and 11,550 onto both lines of O-1043. Adding that repeated column gives 57,300 paise, twice the correct combined total.
3. Nothing went wrong with addition. The report treated an order-grain value as a line-grain contribution.
4. Now propose a key `(order_id, product_id)`. It would reject a second P-NOTE line on the same order even when the workflow treats it as a legitimate separate entry. Our earlier O-REPEAT exercise deliberately permits that case.
5. The existing key `(order_id, line_no)` identifies the line without inventing a "one product only once per order" policy.
6. Keep the distinction between a record's identity and a display choice. A screen may combine identical products for convenience, but that display should not silently replace the historical line records.
7. A useful review question is: "After this transformation, what does one output row mean?" Ask it after joining, grouping, exporting and flattening, not only when creating the first table.

10.2.4 PLAIN — what is really happening inside

1. A key lets a program address an intended record by values. When an update uses only part of a composite key, it may target several records rather than the intended one.
2. A line update using only `line_no = 1` could touch the first line of both canonical orders. Adding `order_id = 'O-1042'` narrows the target to the full line identity.

3. The engine can enforce uniqueness of the complete pair. It cannot decide whether the chosen pair describes the business's intended thing. That is a modelling decision checked through examples.
4. Relationships describe references between identified records. A foreign key from a line to an order says that a required parent exists; it does not count how many lines every parent must have. [S59]
5. Keeping boundaries explicit also helps changes. Renaming a product's current display name should not require editing every order identifier. A product identity and its current label serve different purposes.
6. Do not split records merely to maximise the number of tables. Split when the meanings, multiplicities, lifecycles or enforcement responsibilities justify it. Chapter 13 develops the formal reasoning behind placing related facts.

10.2.5 TECHNICAL — the engineer's version

1. Write a grain contract with four elements: represented thing, key, allowed multiplicity and temporal interpretation. For `order_lines`: one identified agreed line within one order; key `(order_id , line_no)`; repeated products permitted; price interpreted as the value agreed for that line.
2. The contract separates a functional association from a business rule. Each line key identifies its stored product and quantity. It does not follow that each `(order_id, product_id)` pair identifies at most one line.
3. Document relationship direction. A non-null line reference constrains the child to an existing parent. It does not establish a parent's participation minimum or prove that the order has been completed.
4. Review optionality by state rather than convenience. `orders.customer_id` is nullable in the toy model. Its absence means no linked profile in this record; it does not prove the buyer was anonymous, refused identification or never had a profile.
5. Distinguish transport shape from storage shape. The agreed-line transport repeats `currency` so each incoming record can be validated independently. The relational model places INR currency on the parent order and requires publication to preserve that contract.
6. A flattened export may intentionally repeat parent attributes. Document which fields are dimensions or annotations and which can be added at the export's grain. Otherwise the 57,300-paise error can reappear in somebody else's spreadsheet.
7. The new `schema_snapshot()` helper reports stored definitions using SQLite's metadata interfaces. It can reveal the declared key and references; it cannot infer an unwritten row-grain sentence from business activity. [S75]
8. The review test for a boundary is not "does this look normal?". It is "can this representation distinguish the valid cases we need, reject the invalid cases it promises to reject, and answer the required questions without inventing missing facts?".

10.2.6 WORDS — remember these

1. **Identity boundary:** the scope inside which a label identifies something — the domain over which a key or composite key is interpreted and uniqueness is enforced.

2. **Grain contract:** a precise statement of what one row represents — documentation tying a record's unit, key, multiplicity and temporal meaning together.

3. **Participation minimum:** the least number of required related records — a cardinality condition such as "every completed order has at least one line".

4. **Flattened export:** related information copied into a single rectangular result — a representation that may repeat parent attributes at a finer output grain.
5. **Temporal interpretation:** which point or period a value describes — the time-related meaning that distinguishes a current attribute from a historical observation or agreement.

10.3 Mutable and historical values

▪ 10.3.1 PLAIN — in simple words

1. Two fields can hold the same number today and still mean different things. Today's notebook price and the notebook price agreed on O-1042 both start at 7,550 paise, but they answer different questions.
2. Tomorrow's catalogue can change without changing yesterday's agreement. Replacing an agreed historical price with a current catalogue lookup would answer a new question while appearing to show the old one.
3. A draft is different again. It represents unfinished work whose proposed values may still change. A missing quotation in a draft is not a free item and not an agreed sale.
4. A value is mutable when the intended workflow allows it to change. Calling another value historical describes its intended meaning, but does not automatically prevent a program from overwriting it.
5. Therefore ask two separate questions: "what should this value mean?" and "what mechanism protects that meaning?" A good name answers only the first part.
6. Correction does not mean pretending the earlier record never existed. The required correction process depends on what evidence, timing and authority must remain available. The foundations lab does not implement that full process.
7. Keeping meanings apart makes the application easier to explain: "current price", "proposed price" and "agreed price" can all be shown honestly without forcing one field to stand for three different claims.

▪ 10.3.2 PLAIN — a picture in your head

1. Think of a shop's chalkboard, a pencil-written quotation and a previously issued order document.
2. Erasing the chalkboard changes what the shop is offering now. Editing the pencil quotation changes unfinished work. Neither act should silently rewrite the amount shown on an earlier agreed line.
3. If the old order contains a mistake, the shop needs an appropriate correction process. Borrowing the chalkboard eraser is not a complete policy for historical evidence.
4. **Where the comparison breaks:** digital records are not physically protected by their appearance. A table called `history` can still allow ordinary updates unless the system actually restricts them.
5. Nor does an append-only-looking screen prove that backups, administrators or other write paths cannot change the data. The model must state which protection exists and against which kinds of change.
6. The analogy distinguishes meanings. It does not establish a legal retention rule, an audit guarantee or an implementation of immutability.

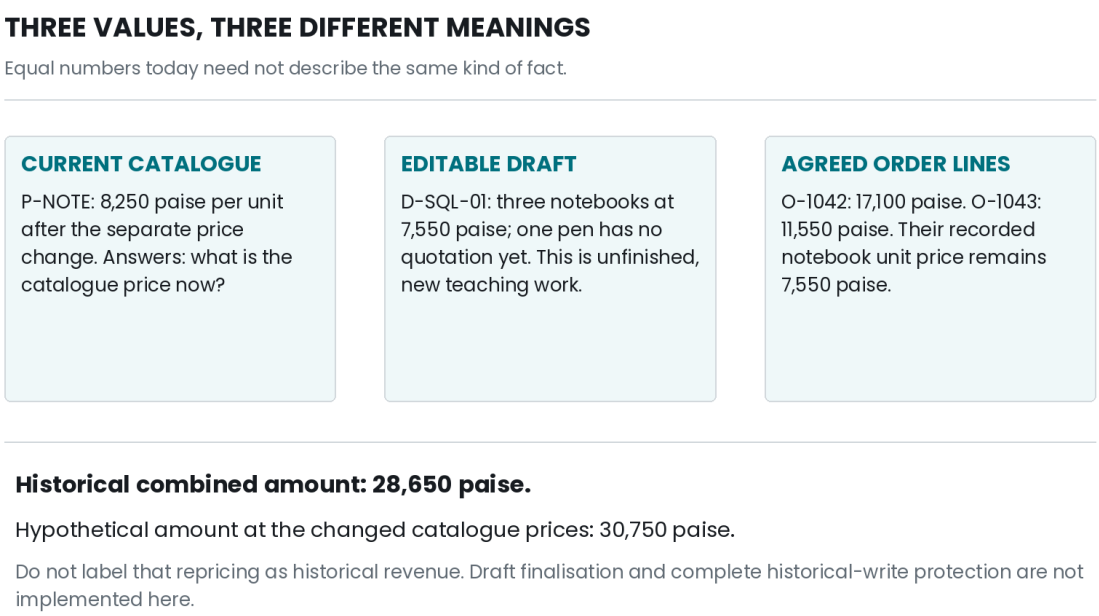

Figure 10.4: Figure 10 — Current catalogue, editable draft and agreed order lines answer different questions. The draft is new teaching data; the historical combined total remains 28,650 paise.

10.3.3 PLAIN — a worked example

1. Reuse the separate current-price experiment from Chapter 9. Change only P-NOTE's current catalogue price from 7,550 to 8,250 paise in a fresh practice database.
2. The three historical notebook units still use 7,550 paise in their agreed lines. The three pen units still use 2,000 paise.
3. Historical total: `3 × 7,550 + 3 × 2,000 = 28,650` paise. O-1042 remains 17,100 and O-1043 remains 11,550.
4. Hypothetical value at today's catalogue prices: `3 × 8,250 + 3 × 2,000 = 30,750` paise. The difference is `3 × 700 = 2,100` paise.
5. Both calculations are internally consistent. They answer different questions. The hypothetical 30,750 is not newly discovered historical revenue.
6. Now inspect the separate draft D-SQL-01. Three proposed notebooks at 7,550 contribute 22,650 paise, but the pen quotation is absent. We cannot claim a complete quoted total by treating that absence as zero.
7. The lab's three meanings are shown in the figure. The draft values and its identifier are explicitly new; the historical fixture is not rewritten.

10.3.4 PLAIN — what is really happening inside

1. The product row and the agreed line store different attributes. Changing the product row does not automatically rewrite the line's copied agreed price because they are separate stored values.
2. A query that multiplies quantities by the product's current price nevertheless produces a repriced result. The historical values can remain intact while the report chooses the wrong source for its question.

3. Protection therefore has at least two concerns: prevent unauthorized or inappropriate writes, and ensure reads use the intended meaning.
4. The present teaching schema checks whether an agreed price is a present integer within its bounds. It does not check whether every later update is an authorized historical correction.
5. A new test deliberately changes one agreed price to 1 paise in an isolated disposable copy. The structure accepts it, and O-1042's computed total becomes 2,002 paise. That is evidence of a missing workflow protection, not a legitimate correction to the book's canonical order.
6. Keeping that negative example visible is useful. It prevents the phrase "we stored a snapshot price" from expanding into the unsupported claim "the system guarantees immutable agreements".

10.3.5 TECHNICAL — the engineer's version

1. A historical snapshot attribute can intentionally duplicate a value that also appears in a current entity record. The duplication is not necessarily redundant in meaning: `catalogue_price_minor` and `unit_price_minor` have different temporal semantics.
2. The snapshot's preservation policy must identify allowed mutations, correction provenance, authorizing actors and read behaviour. This schema implements none of those broader controls merely by using a separate column.
3. The isolated draft table accepts nullable quantity and price under bounded checks, while the agreed-line table remains not-null. This is a separate lifecycle representation, not a backwards-compatible relaxation of the original accepted-record contract.
4. A future finalisation operation would need to validate the complete draft, establish the agreed identifiers and values, coordinate publication, prevent duplicate effects under its chosen retry policy and handle failures. That design is deferred; a successful draft edit is not finalisation.
5. The new lab's `price_meanings_demo()` explicitly returns both agreed totals and a labelled hypothetical current-catalogue total. The arithmetic uses bounded integer paise, not binary floating-point money.
6. The same principle applies outside prices. A customer's current address is not automatically the address used for an earlier delivery. An employee's current role is not automatically the role held when a past approval occurred. Those are design illustrations, not extra implemented shop features.
7. Decide deliberately which historical claims are needed. Snapshotting every attribute forever is not the default answer; each retained field needs a purpose and an appropriate lifecycle. Later chapters discuss retention, access and operations in their own scope.
8. Mark the protection level accurately in documentation: separate meanings stored; selected query results tested; historical update prevention absent; complete evidence and permission workflow absent. This record lets later implementation work close a real gap instead of discovering that an earlier promise was fictional.

10.3.6 WORDS — remember these

1. **Current attribute:** what a record says about the present state — a value whose intended temporal interpretation follows the entity's current representation.

2. **Historical snapshot attribute:** a value copied to describe an earlier event or agreement — a stored attribute with event-specific temporal semantics, not automatic immutability.

3. **Draft record:** unfinished work that remains open to revision — a lifecycle-specific representation with explicit incompleteness rules and no implied finalisation.
4. **Repricing:** calculating using a different price basis — recomputation with current or alternative prices rather than the stored agreed values.
5. **Mutation policy:** the rules for changing a record — a specification of permitted transitions, authorization, evidence and failure handling.

10.4 Domain types

10.4.1 PLAIN — in simple words

1. A storage type says what kind of value an engine can store. A business domain says what those values are allowed to mean in this particular role.
2. An integer might be a count, an amount in paise or an identifier generated by a program. Being an integer does not make it interchangeable with every other integer.
3. Our agreed quantity is a positive whole-unit count. Our agreed price is a nonnegative number of paise. Zero is allowed for the price under this toy contract, but not for the quantity.
4. Missing is another decision. A nullable draft price means the quotation is not present. A not-null agreed price means the accepted line must supply one. Neither decision comes from the word "integer" alone.
5. Bounds are part of the implemented contract. The relational lab limits quantity to 10,000 and unit price to 100,000,000 paise per line. These are teaching adapter limits, not universal laws about shops.
6. A type conversion can lose a distinction before the database sees it. A program may turn a Boolean true into the integer one. If the input contract rejects Booleans as quantities, validate that at the boundary where the distinction still exists.
7. Think of a domain as a complete acceptance description: representation, unit, allowed range, missingness, identity rules and interpretation. A few of those pieces can be enforced in SQL; others need additional application checks or evidence.

10.4.2 PLAIN — a picture in your head

1. Imagine receiving labelled parcels. The outer box size resembles the storage type: it determines what physically fits.
2. A label saying "whole notebooks" adds meaning. The box may physically fit half a broken notebook, but that is not an acceptable value for the agreed count in this exercise.
3. A price parcel labelled "paise" is different from one labelled "rupees", even if both contain the handwritten number 75.
4. The receiving desk must inspect the label and contents together. Checking only that the parcel fits through the doorway misses the unit error.
5. **Where the comparison breaks:** some database engines convert values as they accept them, and a program can adapt values before sending them. There is no universal untouched parcel that every layer inspects identically.
6. The analogy therefore reminds us to test the full path from input representation to stored value. It does not imply that every valid business concept deserves a custom database type.

10.4.3 PLAIN — a worked example

1. Compare the domain decisions below. They deliberately preserve the existing accepted-line profile while allowing a separate incomplete draft.

Field and stage	Meaning	Present?	Accepted range in this lab
Agreed line quantity	Whole units agreed on that line	Required	1–10,000
Agreed unit price	Paise per agreed unit	Required	0–100,000,000
Draft quantity	Proposed whole units	May be absent	If supplied, 1–10,000
Draft unit price	Proposed paise per unit	May be absent	If supplied, 0–100,000,000
Line number	Position label within an identified order or draft	Required	Positive supported integer

2. For O-1042 line 1, quantity 2 and price 7,550 produce 15,100 paise. Quantity `true` is rejected by the original input validator even though a programming language may treat true as numerically related to one.
3. In a separate SQLite STRICT demonstration, a bound Boolean true reaches integer storage as 1. The text value `"2"` can be losslessly converted to integer 2, while the real value 2.5 is rejected for the integer column. These observed cases illustrate the documented conversion boundary. [S60] [S86]
4. Suppose someone sends the integer 75 intending INR75.00, but the field's unit is paise. The stored value 75 is within the allowed range. A range check cannot discover the sender's intention.
5. Our test accepts that plausible-but-wrong-unit value in a separately named example. Its lesson is not to prohibit 75 paise universally; it is to establish units before converting and validate the source contract.
6. The original source validator and destination adapter have different responsibilities. A record can pass a broad source profile and still exceed the destination's chosen bounds. Publication must check that second boundary explicitly.

10.4.4 PLAIN — what is really happening inside

1. The application receives values in some runtime representation, such as a Python string, integer or Boolean. Validation happens against that representation and the incoming record's contract.
2. The database driver then binds supported values to a statement. The engine applies its storage rules and constraints. A later reader receives the stored result, not necessarily the original representation.
3. That sequence explains why a database-only check cannot recover a distinction already erased by the driver or earlier conversion.
4. It also explains why two layers of checking need not be pointless repetition. The input layer can reject an unexpected runtime type; the storage layer can protect a declared range when another writer bypasses the usual input code.
5. Keep errors specific enough to diagnose the boundary. "Unsupported Boolean quantity", "quantity above adapter limit" and "referenced product missing" are different failures with different remedies.
6. Changing a domain can change more than storage. Allowing fractional quantities would affect arithmetic, formatting, validation and the meaning of a unit. Such a change deserves a versioned contract, not an unnoticed switch from integer to real.

10.4.5 TECHNICAL — the engineer's version

1. PostgreSQL has a concrete `CREATE DOMAIN` feature for reusable constraints over a base type. That product feature is one implementation tool; the broader modelling concept of a domain is not limited to it. [S68]
2. The following PostgreSQL 17 fragment is illustrative and was not executed in the SQLite lab. It gives the numeric range a reusable name while leaving presence to the using column.

```
CREATE DOMAIN nonnegative_minor AS bigint
    CHECK (VALUE >= 0 AND VALUE <= 100000000);

CREATE TABLE quoted_amount_example (
    example_id text PRIMARY KEY,
    unit_price_minor nonnegative_minor NOT NULL
);
```

3. The name does not encode the currency by itself, nor establish a buyer's agreement. A complete contract would still state that the amount is in paise for INR under the chosen exercise.
4. Do not teach domain-level not-null as a universal guarantee that no expression can ever produce a null value of that type. PostgreSQL documents important caveats; using column-level presence requirements keeps this introductory example explicit. [S68]
5. SQLite STRICT tables do not provide the same `CREATE DOMAIN` facility. The executed lab uses supported storage type names, row checks, not-null declarations, references and pre-binding validation. [S60]
6. Analyse arithmetic bounds at the operation's grain. A line quantity up to 10,000 multiplied by a price up to 100,000,000 is at most 1,000,000,000,000 paise. That product fits within a signed 64-bit integer, but an unbounded sum over arbitrarily many lines needs its own analysis.
7. Domain documentation should distinguish stored values from derived values. A line total is quantity times its agreed unit price here. A formatted amount string is a display result. Neither should be confused with the original price field.
8. The tested Python boundary uses exact type checks where Booleans must be excluded. This is a deliberate contract choice, not a claim that every application must reject all lossless textual number inputs. Different interfaces can have different documented profiles.

10.4.6 WORDS — remember these

1. **Business domain:** the values that make sense for a particular purpose — a semantic acceptance set including units, bounds, missingness and interpretation.
2. **Storage type:** the kind of value an engine stores in a field — an implementation-level representation with defined conversion and operation rules.
3. **Lossless coercion:** changing representation without losing the represented value — an allowed conversion such as a supported textual integer into integer storage.
4. **Pre-binding validation:** checking input before the driver sends it — validation while original runtime distinctions remain available to application code.
5. **Arithmetic bound:** the largest or smallest result an operation may produce — a range derived from operand limits and the operation's cardinality.

10.5 Naming and documentation

10.5.1 PLAIN — in simple words

1. A name is a short label, not a complete explanation. `amount` leaves too many questions open: amount of what, in which unit, at which time and under which calculation?
2. `unit_price_minor` is better in this model because it distinguishes a unit price from a line total. Its documentation must still state paise, INR, agreement time and allowed values.
3. A data dictionary is a small reference describing tables and fields. It should answer the questions a careful new reader would otherwise have to ask the original developer.
4. Write down missing-value meaning. The nullable customer link says "no linked profile in this record", not a guessed reason such as "customer refused".
5. Write down the grain and key. A future analyst should not have to inspect a screen or guess from the table's plural name before knowing what a row counts.
6. Keep examples next to definitions. One valid value and one misleading-but-plausible value often explain a field better than a long abstract description.
7. Documentation can be wrong or stale. Compare it with the stored definition and tests whenever the model changes. The dictionary is part of the work, not evidence that the work has already been verified.

10.5.2 PLAIN — a picture in your head

1. Think of a building plan with a legend. A symbol is useful because the legend explains its meaning consistently everywhere it appears.
2. Without a legend, one circle might mean a socket, a light or a drain. Guessing from appearances is dangerous when people make real changes.
3. Field names are the symbols. The data dictionary is the legend explaining units, roles and exceptions.
4. **Where the comparison breaks:** the database can change while an old document stays in a folder. Unlike a printed plan issued with one building, software and its documentation need an explicit update process.
5. The dictionary also cannot replace enforcement. Writing "quantity must be positive" in a comment does not make an otherwise unconstrained column reject minus three.
6. A practical review reads three things together: the intended definition, the actual database definition and a small set of observed acceptance tests.

10.5.3 PLAIN — a worked example

1. Here is a compact entry for the existing agreed-line price. It documents the current teaching scope rather than adding a new requirement silently.

Dictionary item	Entry
Table and grain	`order_lines`: one identified agreed line within one order
Field	`unit_price_minor`
Meaning	The unit price recorded as agreed for this line
Unit and currency	Paise; parent order and input contract require INR
Presence and range	Required integer, 0 through 100,000,000 in this adapter
Canonical example	7,550 on O-1042 line 1
Not this	The product's current catalogue price; a total for the whole order

Dictionary item	Entry
Unimplemented protection	Full agreement evidence and authorization; historical update prevention

2. Now document `orders.customer_id`: optional reference to a stored profile. O-1042 has NULL; O-1043 has C-001 under the earlier synthetic annotation.
3. Do not expand C-001 into an invented verified identity. The reference demonstrates linkage and optionality; it does not provide authentication evidence.
4. The new lab exposes a partial `DATA_DICTIONARY` for its core order tables and compares key facts against actual metadata. It is intentionally partial, not a complete enterprise catalogue.
5. Names such as `catalogue_price_minor` and `unit_price_minor` make a difference visible. The definitions explain why the values may diverge after the catalogue price changes.
6. A query returning an alias called `total_minor` should document whether it means one line, one order or the entire selected set. Reusing the same alias at different grains is possible, but consumers need the result contract.

10.5.4 PLAIN — what is really happening inside

1. Engines maintain system information about their own tables and columns. Programs can query that information instead of relying entirely on a separate document.
2. SQLite's metadata interfaces can report column names, not-null settings, primary-key positions and declared foreign keys. They cannot explain why Mira chose INR or what evidence established a particular agreement. [S75]
3. PostgreSQL provides an information schema with a standard-oriented view of many database objects. Product-specific catalogues supply additional details. These are ways to inspect implementation, not automatic business documentation. [S71]
4. Some engines also store comments attached to objects. A comment can explain purpose, but it remains descriptive text rather than an enforced rule. [S70]
5. Keep sensitive values out of examples and comments. This book uses invented records, and the reader package needs no credentials or live customer data.
6. A good documentation update records what changed and why. Merely regenerating a column list cannot reveal that a field quietly changed from paise to rupees while retaining the same integer type.

10.5.5 TECHNICAL — the engineer's version

1. Distinguish three uses of "schema": the overall data design discussed in this chapter, a particular structured-record contract, and PostgreSQL's named namespace containing objects. Context must make the intended use clear.
2. PostgreSQL permits qualified names such as `sales.orders`, where `sales` is a namespace. Name resolution and search-path configuration have operational and security implications; no new PostgreSQL namespace is created by this SQLite exercise. [S69]
3. Prefer a consistent, documented identifier convention. The lab uses lower-case words with underscores and avoids relying on quoted mixed-case identifiers. This is a project convention, not a universal SQL requirement. PostgreSQL's lexical rules explain how quoted and unquoted identifiers are treated in that engine. [S72]
4. The actual SQLite inspection used by the lab includes:

```
PRAGMA table_info(order_lines);
PRAGMA foreign_key_list(order_lines);
PRAGMA foreign_keys;
```

5. `table_info` describes ordinary column metadata, not every possible hidden or generated-column detail; broader inspection can require other metadata interfaces. Read the documented scope before treating a metadata query as a complete schema export. [S75]
6. A field dictionary should include ownership of meaning, expected sources, domain, unit, null policy, derivation if any, constraints, allowed mutation and consumer interpretation. This is our review checklist, not a claim that one industry standard mandates this exact list.
7. Keep a distinction between machine-checkable and human-reviewable statements. A test can compare a declared bound with a constant. A human workflow review is still needed to decide whether the bound or meaning is suitable for the intended work.
8. For publication, generated chapter metadata, glossary and examples must be derived from the same manuscript version. Otherwise a book can teach one field meaning in print and a different one in its browser reference. The same continuity discipline used for a schema applies to its teaching material.

■ 10.5.6 WORDS — remember these

1. **Data dictionary:** the reference explaining what fields mean — maintained metadata describing grain, domains, units, source, lifecycle and interpretation.
2. **Namespace:** a named container used to distinguish object names — in PostgreSQL, a schema that can contain tables and other database objects.
3. **Qualified name:** a name that includes its containing scope — an identifier such as `sales.orders` rather than an unqualified table name.
4. **System metadata:** information the engine stores about its own objects — structural descriptions exposed through catalogues or metadata interfaces.
5. **Result contract:** the meaning promised by a query's output — a specification of columns, units, grain, missingness and ordering where relevant.

10.6 Reviewing the model with users

■ 10.6.1 PLAIN — in simple words

1. A model review asks whether the records match the intended work. It is not a contest to see who knows the most database vocabulary.
2. Show small examples. Ask the person responsible for the workflow what each example means, what action should be allowed and what evidence should remain.
3. Include awkward cases early: the same product twice on one order, no customer link, an unfinished quotation, an order with no lines and a changed current price.
4. Ask about things the database would accept even though the business would reject them. A positive price entered in the wrong unit is a useful example because it passes an ordinary range test.
5. Record decisions as decisions. "The product owner approved this rule" requires actual approval; a developer's proposal or a fictional conversation in a textbook is not a substitute.
6. A counterexample is a specific case that exposes a claim as too broad. Finding one during design is valuable because it points directly to the missing distinction or mechanism.

7. Finish the review with open questions and responsible next steps. A clear list of unresolved items is better than a vague claim that everyone liked the diagram.

10.6.2 PLAIN — a picture in your head

1. Imagine trying a key in the actual locks before making a thousand copies. Looking at the key's neat shape cannot prove it opens the right doors.
2. Valid examples are the doors that should open. Invalid examples are the doors that must stay closed.
3. A model can fail in either direction: reject legitimate work or admit a harmful ambiguity. Testing only that one happy-path order fits examines only one door.
4. **Where the comparison breaks:** an information model has far more possible inputs and sequences than a small ring of locks. A finite test set is evidence about selected cases, not a proof of every future behaviour.
5. Nor can a test establish an unspoken business preference. Someone with authority over the workflow must decide whether a case is valid before the implementation's response can be judged against that requirement.
6. Use the picture to remember both sides of acceptance, while keeping the limits of testing explicit.

10.6.3 PLAIN — a worked example

1. Run this review on the existing model. The outcomes below describe the bounded lab and the stated teaching policy.

Proposed claim	Counterexample or question	Review outcome
A product appears at most once per order	Two separately identified notebook lines	Not our policy; the line-number key permits them
Every order necessarily has a line	Create a parent order with no child lines	Not enforced by the existing foreign key
Every valid integer price uses the right unit	Send 75 while intending 75 rupees	Not discoverable from the stored integer alone
Historical prices cannot change	Update an agreed price within the allowed range	Not prevented by the teaching schema
No customer link means an anonymous buyer	O-1042 has no stored profile link	Reason not established by this record
A valid draft is already an agreed sale	Draft includes a missing quotation	Incorrect; finalisation remains separate and unimplemented

2. The review does not "fix" these by adding constraints blindly. Some entries reveal intended flexibility, some missing mechanisms and some unavailable evidence.
3. A minimum-one-line rule might apply only when an order is finalised. Applying it indiscriminately to every intermediate operation could make a legitimate construction sequence impossible.
4. Historical price protection needs an allowed-correction policy as well as prevention of casual overwrites. "Never update anything" is not automatically a complete business solution.
5. The output is a bounded proposal: keep the canonical agreed-line structure, maintain the optional profile annotation, use a separate draft workspace, and explicitly defer complete finalisation and authorization.
6. This is an author-created design exercise. It records no real shop interview, owner acceptance or production-readiness decision.

10.6.4 PLAIN — what is really happening inside

1. Review turns vague language into predicates and examples. "Correct order" becomes a list of concrete structural, workflow and evidence requirements.
2. Tests then observe the implemented subset. One test inserts a duplicate full line key and expects rejection. Another inserts a different line number for the same product and expects acceptance.
3. A negative demonstration can intentionally show missing protection. The isolated historical-price update is valuable precisely because it succeeds where an overly broad claim would predict failure.
4. Keep those tests separate from an accepted workflow. The demonstration's database is disposable; its changed total must never replace the canonical example or be presented as an authorized correction.
5. When a requirement changes, update the proposal, enforcement and examples together. Otherwise tests can keep passing against an obsolete definition of success.
6. The review ends with a boundary: the current schema and selected examples are described and tested, but future concurrency, permissions, finalisation and operational recovery work remains to be designed and verified.

10.6.5 TECHNICAL — the engineer's version

1. Use a traceability matrix connecting requirement, assumption, representation, enforcement, test and remaining limitation. Each column answers a different question; a source citation or a test name should not be pasted across all columns as if it were universal evidence.
2. Classify failures. A model defect cannot represent a valid case; an enforcement gap permits a case the rule rejects; an evidence gap leaves the system unable to know whether the real-world condition held. These categories can coexist but should not be conflated.
3. The new lab checks the stored seven-table model, canonical totals, current-versus-agreed price separation, accepted repeated-product lines and the deliberately absent protections. It does not execute a schema migration or modify the earlier lab's definitions.
4. Review sample sequences as well as isolated records. "Create draft, edit quantity, leave price absent, cancel draft" differs from "agree order, later change current catalogue, reread agreed total". The representation must keep their meanings distinguishable.
5. Include range boundaries and type boundaries in tests. For the current adapter, quantity 10,000 and price 100,000,000 are accepted limits; values above those limits are rejected. Zero price is valid, zero quantity is not.
6. Document ambiguous failures honestly. A guarded update that finds no matching row may indicate absence or a changed expected value. Do not report one specific explanation unless the available evidence establishes it.
7. A test suite's size is not a coverage measure by itself. The significant question is which classes of records, transitions and failures were exercised and which were omitted.
8. Carry unresolved items into the next milestone. Chapter 11 develops declared constraints and their exact limits; Chapter 12 develops basic queries and bounded mutations; later chapters address larger transactions, concurrency, security and lifecycle operations.

10.6.6 WORDS — remember these

1. **Counterexample:** a concrete case that disproves an overbroad statement — an input or sequence for which a claimed invariant or interpretation does not hold.

2. **Traceability matrix:** a record showing how a requirement is addressed — a mapping among assumptions, representation, enforcement, tests and remaining gaps.
3. **Model defect:** a representation that cannot express the intended work correctly — a structural or semantic mismatch with valid domain cases.
4. **Enforcement gap:** a declared rule without sufficient protection — a condition the implementation can violate despite the intended policy.
5. **Evidence gap:** something the records cannot establish — missing information or verification needed to support a claim about the world.

10.97 Practice and worked answers

1. **Write the grain.** Define one row in `products`, `orders`, `order_lines` and `sql_draft_lines`. Explain why counting four line rows does not mean four orders.
2. **Separate price meanings.** The current notebook price becomes 8,250 paise. Calculate the canonical agreed total and the hypothetical current-catalogue total. State which one belongs on a report of what was agreed.
3. **Review a proposed shortcut.** A developer wants to allow NULL in the existing agreed price column so unfinished quotations fit. Identify the contract change and propose a representation that preserves the earlier agreement meaning.
4. **Find two different gaps.** A structurally acceptable value was entered in rupees rather than paise. Separately, an ordinary SQL update changes a stored agreed price. Explain why those failures need different remedies.
5. **Inspect, do not infer.** Run the new schema lab and examine its metadata snapshot. Which facts can it establish, and which entries of the data dictionary still require human explanation?

Worked answers

1. A product row represents one product identity in the current catalogue. An order row represents one identified order. An order-line row represents one identified agreed line within an order. A draft-line row represents one editable proposed line within a draft. Four canonical lines belong to two orders and represent six units. The record counts differ because the grains differ.
2. Historical notebook amount is `3 × 7,550 = 22,650`; historical pen amount is `3 × 2,000 = 6,000`; total is 28,650 paise. Current-catalogue notebook amount is `3 × 8,250 = 24,750`; adding 6,000 gives 30,750 paise. A report of the existing agreements uses 28,650. A separately labelled repricing illustration can use 30,750.
3. The shortcut changes what an accepted agreed line promises. Instead, the new exercise uses an explicitly separate draft table whose incompleteness rules are stated. It does not claim that creation or editing of a draft implements finalisation. The finalisation operation, authority and evidence remain open design work.
4. A unit error can be invisible after storage because both 75 and 7,500 are valid integer paise amounts. Establish and validate units before conversion, and preserve relevant source evidence. A later unauthorized overwrite needs write-path and mutation-policy protection. A not-null or range check alone solves neither complete problem.
5. Metadata can reveal column declarations, the composite key and foreign-key relationships under the current engine. It cannot discover why a field exists, why the price means an agreement,

whether a buyer really agreed, or who was allowed to approve it. Those meanings must be documented and their implemented protections reviewed separately.

10.98 Common wrong ideas

1. **Wrong:** start by copying the tables from a similar application. **Better:** first identify the actual workflow, row meanings and failure cases; reuse only what fits those requirements.
2. **Wrong:** one visible row always means one business event. **Better:** a row has a declared grain, which a query or export can change.
3. **Wrong:** the same numeric value means two fields are redundant. **Better:** current and historical prices can coincide today while answering different temporal questions.
4. **Wrong:** calling a table historical makes it immutable. **Better:** names describe intent; actual write protections and correction policy determine permitted changes.
5. **Wrong:** STRICT integer storage enforces the original input's complete meaning. **Better:** driver adaptation and lossless conversion can erase distinctions; units and workflow meaning require more than a storage class.
6. **Wrong:** unknown prices should be zero so totals always appear. **Better:** zero is a known amount; an absent draft quotation remains incomplete.
7. **Wrong:** a foreign key guarantees every order contains a line. **Better:** the existing child reference does not enforce that parent minimum.
8. **Wrong:** a data dictionary is automatically authoritative. **Better:** reconcile its intended meanings with stored definitions and actual tests whenever the model changes.
9. **Wrong:** a passing test suite approves the product design. **Better:** selected tests provide evidence about specified cases; they do not replace owner decisions or establish every omitted protection.
10. **Wrong:** a model review must remove every flexibility. **Better:** distinguish legitimate optionality from an enforcement gap before adding a restriction.

10.99 Chapter summary in 20 lines

1. A schema should follow the work rather than force the work to imitate a convenient table.
2. Write the required actions, decisions and failure paths before choosing implementation details.
3. Separate structural acceptance from workflow completion and real-world evidence.
4. Every table needs an explicit one-row sentence.
5. Orders, lines and units are different grains with different counts.
6. The full order-line identity is the pair of order identifier and line number.
7. Repeated products can be legitimate separate lines under the current policy.
8. A required child reference does not enforce a parent's minimum number of children.
9. Current catalogue prices and historical agreed prices have different temporal meanings.
10. The canonical agreed total remains 28,650 paise after a separate catalogue-price change.
11. A draft is unfinished work, not an agreed sale with inconvenient missing fields.
12. Historical naming alone does not prevent ordinary updates.
13. A business domain is narrower and richer than its storage type.
14. Units, missingness and bounds belong in the field contract.
15. Validate distinctions before a driver or conversion erases them.
16. A data dictionary explains meaning that system metadata cannot infer.

17. PostgreSQL's schema namespace is a specific use of a broader word.
18. Review valid cases and invalid counterexamples, including operation sequences.
19. Distinguish model defects, enforcement gaps and evidence gaps.
20. Record unresolved protections explicitly instead of calling a partial model complete.

CHAPTER 11

Constraints: Rules the Database Can Enforce

11.0 What this chapter gives you

1. You will explain a constraint as an enforced acceptance rule, not merely a note beside a field.
2. You will distinguish a required value from a check on values that are supplied.
3. You will test primary keys and uniqueness without assuming that every database treats missing values identically.
4. You will follow a foreign key from a child record to its parent, including a reference made of two components.
5. You will demonstrate why individual row bounds do not necessarily protect a total across several rows.
6. You will distinguish checking at a statement boundary from checking at a transaction boundary.
7. You will inspect an actual failed deferred commit, then compare repairing that transaction with rolling it back.
8. You will distinguish a failed statement, a rolled-back transaction and a structurally successful database check.
9. You will outline a constraint-change review without pretending that the in-memory exercises perform a production migration.

The previous chapter separated intended rules from implemented protection. This chapter examines the protection itself. The executed examples use SQLite in disposable databases; PostgreSQL 17 comparisons are explicitly documented rather than executed. The original shop's four agreed lines remain unchanged. CAP-DEMO, the optional composite receipt links and the parent/child timing tables are isolated teaching examples, not newly discovered business events.

11.1 Not-null and check rules

11.1.1 PLAIN — in simple words

1. A constraint is a rule the engine checks when relevant data is written. If the operation would violate an enforced rule, the engine reports a failure instead of accepting that prohibited result.
2. `NOT NULL` says that a column must contain a supplied value rather than SQL's missing-value marker. It does not say that a text value is nonempty, a number is positive or the information is truthful.
3. `CHECK` tests an expression. For example, a supplied agreed quantity must be between 1 and 10,000 in our storage adapter.

4. These are different jobs. A check such as "value greater than zero" is not a complete way to require a value, because a missing value does not behave like the number zero.
5. In the SQLite examples, a check that evaluates to NULL is not a violation. Use an explicit presence rule when absence is prohibited. [S61]
6. The same field can need several rules at once: required, appropriate storage type and within the permitted range. None of those independently proves the real-world claim behind the record.
7. Treat a successful insert as evidence that the applicable enforced rules accepted it. Do not translate it into "every requirement was checked" unless every relevant requirement actually has a mechanism.

11.1.2 PLAIN — a picture in your head

1. Imagine an entry desk with separate questions. First: "Did you supply the required document?" Second: "Does its stated number fall within the accepted range?"
2. Skipping the first question cannot be repaired merely by looking at the range on documents that arrived. An absent document is not a document showing zero.
3. Similarly, "a value must be present" and "a supplied value must be positive" should be written as separate promises when both matter.
4. A clear label on the desk is helpful, but it is not the same as someone actually checking the condition. A comment in SQL resembles the label; a declared constraint resembles the implemented check.
5. **Where the comparison breaks:** SQL's truth rules are precise, not a human clerk's intuition. A CHECK expression and a WHERE filter treat an unknown result differently for their respective purposes.
6. The entry-desk picture should therefore lead to actual truth cases, not replace them. Test missing, boundary, valid and invalid values directly.

11.1.3 PLAIN — a worked example

1. The new lab creates two isolated tables. They are not modifications to `order_lines`.

```
CREATE TABLE check_only (
    value INTEGER CHECK (value > 0)
) STRICT;

CREATE TABLE required (
    value INTEGER NOT NULL CHECK (value > 0)
) STRICT;
```

2. Try the following inputs. "Accepted" means accepted by these deliberately small definitions, not approved as a complete shop record.

Input	`check_only`	`required`	Explanation
NULL	Accepted	Rejected	Presence is enforced only in the second table
0	Rejected	Rejected	Zero is not greater than zero
1	Accepted	Accepted	Present positive integer
-1	Rejected	Rejected	Negative value fails the check

3. For the original agreed quantity, the stronger declaration uses `INTEGER NOT NULL` together with the chosen positive range. For the separate draft quantity, absence is permitted but a supplied value must satisfy the same range.
4. An empty text value `''` is not SQL NULL. A field needing nonempty text must specify that condition; `NOT NULL` alone does not express it.
5. Nor does `NOT NULL` distinguish an honest zero price from a made-up zero inserted to hide an absent quotation. The schema can enforce the stored representation, but the input workflow must preserve meaning.
6. These tests are small enough to reason through before running them. The surprising NULL case is exactly why rules should be demonstrated rather than memorised from their English names.

11.1.4 PLAIN — what is really happening inside

1. The engine receives a proposed row, applies the relevant type handling and evaluates its declared conditions. It does not infer additional rules from neighbouring prose.
2. An ordinary comparison involving NULL often produces an unknown result rather than true or false. The check-only example therefore does not evaluate `NULL > 0` as false.
3. The purpose of a CHECK constraint is to exclude rows that violate its expression under the engine's constraint semantics. The purpose of a WHERE filter is to retain rows whose predicate is true. These are related uses of expressions with different acceptance decisions.
4. The later SQL chapter demonstrates the distinction: a NULL quantity can pass a nullable positive CHECK but will not be selected by `WHERE quantity > 0`.
5. A check can relate columns of the same row. An optional pair can require both components absent or both present. This remains a row rule even though it mentions two columns.
6. An arbitrary rule about totals across many rows is different. Do not hide a query inside a check and assume that all future changes elsewhere will trigger a correct re-evaluation. The whole-system invariant needs an appropriate mechanism.

11.1.5 TECHNICAL — the engineer's version

1. The lab's row-domain checks distinguish three boundaries: original runtime input validation, SQLite storage acceptance and business interpretation. Python type distinctions lost during binding cannot be reconstructed from a stored integer alone.
2. In SQLite, CHECK failure is based on an expression result interpreted numerically as zero; a NULL result is not a constraint violation. PostgreSQL's Boolean CHECK semantics also allow an unknown result, but expression typing is not identical between the engines. [S61] [S02]
3. Prefer direct, explicit declarations for introductory invariants. `quantity INTEGER NOT NULL CHECK (quantity BETWEEN 1 AND 10000)` states presence and range without relying on a reader to infer either from a column name.
4. Use a separate text predicate when needed. For example, a local nonempty-label exercise could use `CHECK (length(label) > 0)` with `NOT NULL`. That still does not define whitespace policy, identity normalization or permitted characters; those require additional decisions.
5. Boundary testing should include values just below and above the accepted interval. For agreed quantity, 0 and 10,001 fail while 1 and 10,000 pass. For agreed price, 0 is allowed, -1 and 100,000,001 are not.

6. Requiring an integer in SQLite STRICT storage does not prohibit every incoming textual representation. The documented lossless-coercion rules and the lab's observed `"2"` example keep that distinction explicit. [S60]
7. Avoid volatile or external truth hidden behind a row check. PostgreSQL's documentation explains why CHECK conditions should not depend on changing data in other rows. Such a function can make the declaration appear stronger than the maintained invariant actually is. [S02]
8. The tests establish selected acceptance cases on the recorded SQLite runtime. They do not prove that every engine, driver or future schema revision applies identical coercion and error behaviour.

11.1.6 WORDS — remember these

1. **Constraint:** a rule enforced at a defined database boundary — a declared condition whose violation prevents a relevant operation from succeeding under the engine's semantics.
2. **Not-null rule:** a requirement that a value be present — a column condition excluding SQL NULL, distinct from nonempty text or positive numeric value.
3. **Check predicate:** an expression used to reject prohibited row values — a condition evaluated under an engine's CHECK semantics.
4. **Unknown result:** a comparison that cannot be classified as true or false from the supplied values — the third truth outcome commonly produced by SQL expressions involving NULL.
5. **Boundary test:** a test at an acceptance limit — a selected case at, immediately below or immediately above a declared domain boundary.

11.2 Unique and primary keys

11.2.1 PLAIN — in simple words

1. A uniqueness rule prevents repeated values, or repeated combinations of values, under its declared scope and comparison rules.
2. A primary key is the chosen key used to identify rows. In our teaching tables it is required and unique. Some tables use one column; order lines use a pair.
3. A key on `(order_id, line_no)` prevents two records with exactly the same complete line identity. It does not prohibit line number 1 on two different orders.
4. It also does not say that two separate lines cannot name the same product. Adding that restriction would be a new business policy rather than a technical consequence of having a key.
5. "Unique" and "required" should not be treated as interchangeable words. Missing-value behaviour in a uniqueness constraint depends on the database and declaration.
6. A generated identifier is not the whole duplicate-delivery solution. Two retries may receive two different generated identifiers while still representing one attempted business action.
7. Uniqueness protects a declared representation of identity. Deciding what should count as the same thing remains a modelling and workflow responsibility.

11.2.2 PLAIN — a picture in your head

1. Imagine labelled lockers. A rule says that no two assigned lockers may have the same complete label.

2. If a label consists of building and locker number, Building A locker 17 and Building B locker 17 can both exist. The number alone is incomplete.
3. Blank labels raise a separate question. Are blanks prohibited, allowed many times, or allowed only once? The word "unique" alone is not enough to infer a portable answer.
4. **Where the comparison breaks:** database equality can depend on types and text-comparison rules, and generated keys are not physical locker tags. A visually similar spelling is not automatically the same database value under every configuration.
5. The analogy also cannot detect whether somebody created two different labels for one real-world request. That problem needs the application's identity and retry policy.
6. Keep complete-key uniqueness separate from real-world duplicate detection, just as Chapter 5 separated repeated delivery from a genuinely new event.

11.2.3 PLAIN — a worked example

1. Start from the canonical line identity O-1042, line 1. A second insert with that same pair is rejected by the key even if it tries to supply a different quantity.
2. O-1043, line 1 is accepted because the order component differs. O-1042, line 2 is accepted because the line component differs.
3. The earlier repeated-product exercise can contain two P-NOTE lines with different line numbers. The key protects line identity rather than product uniqueness within an order.
4. A separate table demonstrates missing references:

```
CREATE TABLE refs (
    reference TEXT UNIQUE
) STRICT;
```

5. The lab inserts NULL, NULL and the present reference R-1. All three inserts succeed. A second R-1 is rejected. The result contains two missing references, not evidence that those missing references identify the same thing.
6. That observed outcome is SQLite's behaviour for this definition. PostgreSQL 17 provides an explicit `NULLS NOT DISTINCT` option when a different uniqueness treatment is desired; that syntax is not used in the SQLite lab. [S61] [S02]
7. Do not "repair" this by replacing every missing reference with the text `UNKNOWN`. That would invent a present value and then make all unknown cases collide with it.

11.2.4 PLAIN — what is really happening inside

1. The engine compares the constrained values against its maintained key structure. It can therefore reject a competing write that violates the same declared uniqueness rule.
2. A separate application query saying "does this identifier exist?" is not a replacement. Another writer could act after that query and before the insert unless the complete operation has appropriate protection.
3. The key check removes one class of race: accepting duplicate constrained identities. It does not automatically coordinate every other effect associated with a request.
4. A key collision can mean different things. It might be an exact retry, conflicting content under the same identity or an actual identifier-allocation error. The database's rejection does not choose the application's response policy.

5. The earlier file-import and event-consumer exercises already use deliberately different duplicate policies for different boundaries. Do not collapse those into one generic "ignore duplicates" instruction.
6. When a write fails, compare the right evidence under the chosen operation's policy. Automatically overwriting the existing record would change a uniqueness rejection into an unreviewed mutation.

11.2.5 TECHNICAL — the engineer's version

1. Candidate keys describe possible unique identifiers; a primary key designates one for the table. Additional unique constraints can enforce other candidate identifiers where the business contract actually requires them.
2. In this lab, explicit `NOT NULL` declarations accompany the text and composite keys. SQLite has historical primary-key nullability exceptions outside some table forms, so do not generalise from this STRICT schema to every legacy SQLite table. [S61] [S60]
3. The constrained columns define scope. `PRIMARY KEY (order_id, line_no)` is not equivalent to separate uniqueness constraints on each component. The latter would wrongly allow only one line per order and only one use of each line number across the table.
4. PostgreSQL's primary and unique constraints are associated with supporting unique indexes. The broader indexing mechanisms are treated later; at this stage the important distinction is between the declared invariant and an application's preliminary existence query. [S74]
5. Input normalization must be chosen before a uniqueness promise is interpreted. Trimming or case-folding an identifier after acceptance can merge previously distinct labels. Our key examples use their exact stated strings and do not invent a general normalization policy.
6. For retries, a unique request key can be one part of an idempotency design, but the design must also define scope, payload agreement, retained result and transactional effects. None is established merely by adding an auto-generated row identifier.
7. SQLite's `INSERT OR REPLACE` is not a neutral spelling of "update this existing record". For a relevant key conflict, its replacement behaviour can remove the conflicting row. Do not use it as an unexplained repair for failed uniqueness in these exercises. [S76]
8. The new tests retain failed duplicate-key inserts as failures and demonstrate valid alternate line identities separately. That preserves the distinction between a rejected conflict and an accepted new business record.

11.2.6 WORDS — remember these

1. **Unique constraint:** a prohibition on repeated constrained values — an invariant over one column or a composite value under the engine's comparison and NULL rules.
2. **Candidate key:** a possible way to identify each row uniquely — a minimal identifying attribute set under the model's constraints.
3. **Primary key:** the chosen required identifier for a table — the designated key used as a principal row identity in the current schema.
4. **Key collision:** two attempted records using the same constrained identity — a uniqueness conflict whose business interpretation requires the surrounding policy.
5. **Existence precheck:** asking whether a record is present before attempting a write — a read that does not by itself prevent a concurrent conflicting insert.

11.3 Foreign keys

■ 11.3.1 PLAIN — in simple words

1. A foreign key connects a stored reference to a record that must exist. A line naming P-NOTE should refer to an existing product record rather than an arbitrary spelling that merely looks plausible.
2. The referencing record is often called the child. The record it refers to is the parent. These names describe the direction of the reference, not human family relationships.
3. If the child reference is required, it needs a presence rule as well as a foreign key. An optional reference may be absent without requiring a made-up parent.
4. Removing a referenced parent raises a policy question. Should deletion be blocked, dependent children be deleted, or the link become absent? The right answer depends on what the records mean.
5. The existing shop uses restrictive deletion for the referenced products and orders. It does not silently delete agreed lines because somebody removes a catalogue entry.
6. A foreign key proves a structural relationship under the active enforcement conditions. It does not prove that a person is the rightful customer, that a product is currently sellable or that a payment was authorized.
7. A reference can contain several components. Every component's meaning and missingness must be specified, or a half-filled reference can evade the assumption a reader thought the diagram expressed.

■ 11.3.2 PLAIN — a picture in your head

1. Imagine slips that refer to cards in a catalogue. Before accepting a required slip, a clerk checks that the named catalogue card exists.
2. Before removing a card, the clerk checks what still points to it. Depending on policy, the removal may be refused or coordinated with changes to the slips.
3. An optional blank reference means no card is linked. It should not cause the clerk to create a card named "nobody" and pretend a real link exists.
4. If a reference is "branch plus local number", a slip showing only the branch is incomplete. The desk needs a clear rule for that half-filled case.
5. **Where the comparison breaks:** SQL engines have specified null-matching, deletion-action and timing semantics. A drawing of an arrow contains none of those details unless the actual declaration supplies them.
6. The clerk also does not authenticate the real-world story behind the card. Existence and authority are different claims.

■ 11.3.3 PLAIN — a worked example

1. Reuse Chapter 5's scoped receipt identities as a new relationship demonstration. The parent table contains `(BR-A, 17)` and `(BR-B, 17)`. No amounts or order links are inferred from those labels.
2. An optional child reference uses two columns: branch and local number. The intended policy is "both absent, or both supplied and naming a parent".
3. A plain nullable composite foreign key is not sufficient for that policy in SQLite. A row with branch BR-MISSING and a NULL local number is accepted in the deliberately loose table.
4. The stronger example adds an all-or-none check:

```
CHECK ((branch IS NULL) = (local_no IS NULL)),
FOREIGN KEY (branch, local_no)
    REFERENCES receipts(branch, local_no)
```

5. The check compares two known yes-or-no answers about absence. Both absent gives true; both present gives true; only one absent gives false.

Proposed optional reference	Checked example result	Why
NULL, NULL	Accepted	No linked receipt
BR-A, 17	Accepted	Complete reference with a parent
BR-B, 17	Accepted	Different valid scope
BR-A, NULL	Rejected	Half-specified reference
BR-X, 17	Rejected	Complete reference but missing parent

6. If the relationship were required, both columns would instead need not-null rules as well. Allowing a completely absent optional link is a deliberate policy, not an accident.
7. None of these tests proves that a linked receipt belongs to an authorized actor. The identifiers demonstrate referential structure only.

11.3.4 PLAIN — what is really happening inside

1. The engine uses the child values to look for an eligible parent key. The declaration must reference a suitable identifying set of parent columns, not arbitrary unrelated values.
2. Composite matching is a match on the complete tuple. BR-A's receipt 17 and BR-B's receipt 17 remain distinguishable even though the numeric component is equal.
3. In the nullable SQLite case, any missing child component avoids the ordinary parent-match requirement. That is why the separate all-or-none check is needed for this exercise's intended optional link. [S59]
4. Deletion actions control consequences when an existing parent is removed. Restriction blocks the conflicting removal; cascade propagates a deletion to children; setting a reference to NULL still has to respect the other declared rules.
5. The existing child foreign key does not require every parent to have children. A catalogue product can be unused, and a parent order can be empty in the current model.
6. SQLite foreign-key enforcement is a connection setting that the helper enables and verifies before the exercises. A declaration alone is not a sufficient description of an uninspected connection's behaviour. [S59]
7. The default helper-created practice database keeps enforcement enabled. A deliberately unprotected legacy-data demonstration later uses a different newly created in-memory database and is clearly marked as a counterexample.

11.3.5 TECHNICAL — the engineer's version

1. A foreign key combines a child column list, a parent target and referential actions. It is separate from application authorization, business-state eligibility and parent participation minima.
2. The lab's `order_lines` has non-null `order_id` and `product_id`, referencing `orders` and `products` with `ON DELETE RESTRICT`. The optional `orders.customer_id` has different presence semantics.
3. PostgreSQL supports `MATCH FULL` for an all-or-none nullable composite foreign key. SQLite does not enforce the MATCH clause in that way, so the executed example uses the explicit row CHECK rather than copying unsupported semantics. [S74] [S59]

4. Test a missing parent, an absent optional reference and a half-filled composite reference independently. A successful test of one does not establish the other two.
5. A child-side index is a performance consideration rather than the definition of referential correctness. PostgreSQL does not automatically create every useful child-side index merely because a foreign key exists; later chapters study those access costs. [S74]
6. Avoid using cascade as a default housekeeping shortcut. Deleting a product and automatically deleting historical agreed lines would discard evidence the current model intends to preserve. Whether any cascade is appropriate must follow the represented objects' lifecycle.
7. Check actual connection setup with `PRAGMA foreign_keys`, and inspect existing data with `PRAGMA foreign_key_check` when validating the specific SQLite relationship state. Enabling a setting and checking old data are different operations. [S75]
8. These exercises do not change the owner's database, access a server or disable checks in an existing file. Every deliberately defective reference is confined to a new disposable in-memory database.

■ 11.3.6 WORDS — remember these

1. **Foreign key:** a stored link required to match an eligible parent — a referential constraint on child values with defined missingness, actions and checking time.
2. **Composite reference:** a link made from more than one value — a tuple of child columns identifying a parent within the corresponding scope.
3. **Referential action:** what happens to a relationship when a parent changes — a declared response such as restriction, cascading deletion or setting a reference to NULL.
4. **Half-specified reference:** only part of a multi-component link is supplied — an incomplete child tuple that needs an explicit acceptance policy.
5. **Orphan row:** a child with no required matching parent — an existing record violating the intended referential relationship.

11.4 Cross-row invariants

■ 11.4.1 PLAIN — in simple words

1. An invariant is something that must remain true after the operations covered by a rule. Some invariants describe one row; others describe a collection of rows together.
2. "Every allocation is between one and ten" is a row rule. "The sum of all allocations never exceeds capacity ten" is a collection rule. The first does not imply the second.
3. Every individual entry can look acceptable while their combined result is unacceptable. This is why a collection of green validation marks can still hide an over-allocation.
4. Uniqueness and foreign keys already enforce particular relationships across rows. But they do not express every possible sum, minimum count, time overlap or workflow rule.
5. A complete solution must coordinate the operation that changes the collection. Reading the current total and later inserting a new row without adequate coordination can leave a race between writers.
6. Do not claim that a transaction automatically chooses the right coordination policy. The operation, engine isolation and write conditions must match the invariant being protected.

7. This chapter demonstrates the gap using a small counterexample. It does not claim to implement a production reservation service or explain the earlier shop's unresolved stock discrepancy.

11.4.2 PLAIN — a picture in your head

1. Imagine a room with ten chairs. Each group booking must request between one and ten chairs.
2. A booking for seven passes that individual rule. A booking for six also passes. Together they need thirteen chairs.
3. A clerk checking only each form's range has not checked the room's total. A second clerk may make the same mistake even more quickly.
4. To protect the total, the booking process must make decisions against a shared, appropriately coordinated capacity state rather than accept each form in isolation.
5. **Where the comparison breaks:** different database designs coordinate writes differently, and not every invariant should be represented by a mutable chair counter. The analogy demonstrates arithmetic insufficiency, not one universal implementation.
6. It also does not settle cancellation, expiration, no-shows or authorization. Those are additional workflow requirements, not consequences of the number ten.

11.4.3 PLAIN — a worked example

1. The lab creates a separate capacity record CAP-DEMO with total 10. Its allocation rows each require a present integer quantity between 1 and 10.
2. Allocation A-1 requests 7. Its row check passes. Allocation A-2 requests 6. Its row check also passes.
3. Both references point to the existing capacity record, so the foreign-key checks pass as well.

Check	A-1	A-2	Combined state
Quantity	7	6	13
Individual bound 1–10	Pass	Pass	Not a total constraint
Referenced capacity exists	Pass	Pass	Same CAP-DEMO
Allocated sum no greater than capacity 10	Not established alone	Not established alone	Fails by 3

4. No simultaneous execution is needed to demonstrate this first defect. The two inserts can happen one after the other and still produce a total of 13.
5. That matters diagnostically. Before blaming a complicated race, first check whether the sequential operation even enforces the intended rule.
6. A later concurrency design must also ensure that two operations cannot each make decisions against an inadequately coordinated view. Correctness under one sequential fixture is necessary evidence, not a complete concurrency proof.
7. CAP-DEMO is an isolated arithmetic example. Its excess of three has no connection to the older expected-stock-10, observed-stock-9 case; the cause of that older observation remains unstated.

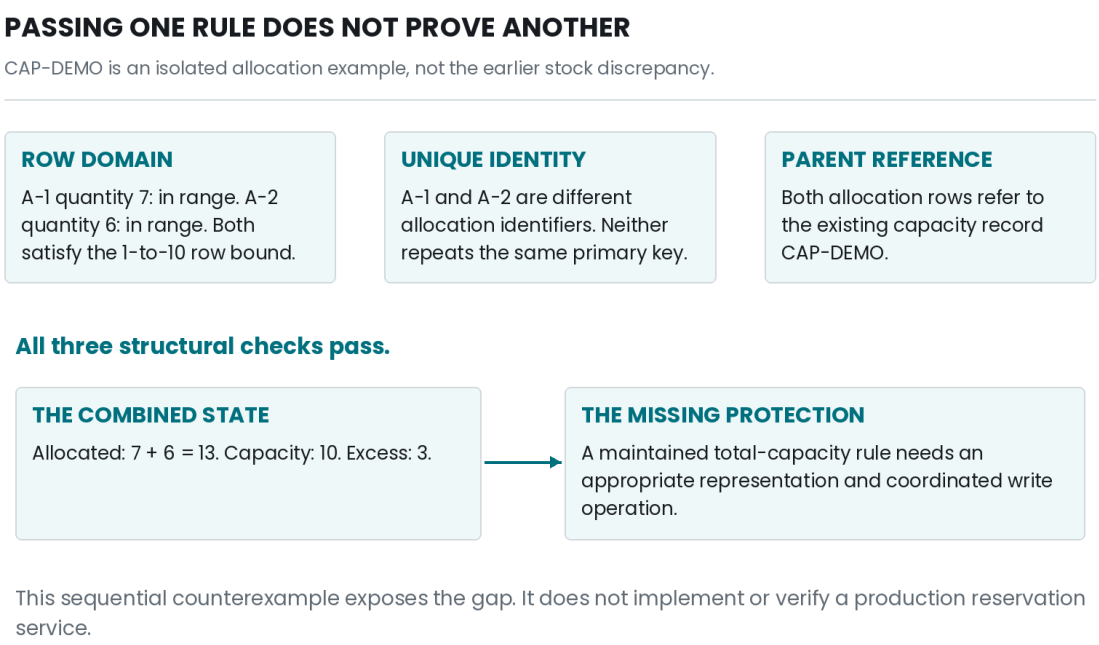

Figure 11.5: Figure 11 — Different rules protect different properties. Row checks, unique identities and parent references can all pass while the combined allocation of 13 exceeds capacity 10.

▪ 11.4.4 PLAIN — what is really happening inside

1. The declared check is evaluated against each proposed allocation row. It sees a permitted quantity, not an automatically maintained sum over every related allocation.
2. The foreign key answers "does CAP-DEMO exist?". It does not interpret the parent's `total` column as an allocation ceiling.
3. A report can calculate the sum and reveal the violation after it exists. Detection is useful, but it is not prevention.
4. An application-only precheck can also calculate a sum before a write. Its usefulness depends on whether the subsequent write is coordinated with every relevant competing change.
5. Possible designs include a guarded update of authoritative capacity state, an appropriately locked transaction or a different representation that makes the desired invariant directly enforceable. The details and trade-offs belong to the later transaction and concurrency chapters.
6. A trigger is code invoked by a database event. Naming a trigger as a solution does not show that its read/write sequence, isolation assumptions and error handling actually protect the invariant.
7. The immediate design task is to state the complete invariant and identify which operations can change it. Only then can the mechanism and tests be judged against the right problem.

▪ 11.4.5 TECHNICAL — the engineer's version

1. Write the two predicates separately. The row predicate is `1 <= q_i <= 10` for each allocation i. The capacity invariant is `sum(q_i for active allocations) <= capacity`. "Active" itself requires a lifecycle definition in a real system; the toy example treats its two rows as the represented allocations.
2. The first predicate allows the assignment `q_1 = 7, q_2 = 6`. The second rejects it when capacity is 10. That explicit assignment disproves the implication from row bounds to total safety.

3. PostgreSQL's CHECK mechanism is not a supported way to query arbitrary other table rows and continuously maintain such a condition. Its constraints documentation directs attention to appropriate mechanisms for cross-row relationships. [S02]
4. A separate stored `allocated_total` would create another invariant: the counter must equal the represented allocations. Protecting the counter's upper bound while allowing it to drift from its underlying records is not a complete solution.
5. For any proposed implementation, enumerate every mutation: add allocation, change quantity, cancel allocation, expire it, reduce capacity and repair historical data. An invariant is only maintained if relevant write paths preserve it under their declared ordering and failure rules.
6. PostgreSQL isolation levels and retry behaviour are product-specific. The book's future concurrent examples must specify those conditions rather than importing SQLite's single-writer observations as a cross-engine guarantee. [S66]
7. The current test intentionally asserts that the row checks pass and the global rule fails. That is a regression test for the explanation's counterexample, not a passing safety test for an allocation service.
8. The same reasoning applies to minimum participation. A child-to-parent foreign key alone does not express "every final order has a line". A finalisation boundary and its mechanism must be designed before testing that stronger claim.

11.4.6 WORDS — remember these

1. **Invariant:** a condition that must keep holding — a predicate preserved across the operations within its specified scope.
2. **Cross-row invariant:** a rule about several records together — a condition involving relationships, aggregates or coordinated state beyond one proposed row.
3. **Detection versus prevention:** finding a violation versus stopping it from being committed — distinct responsibilities requiring different evidence.
4. **Authoritative counter:** a stored total used to make decisions — state whose agreement with the underlying events or allocations needs its own maintained invariant.
5. **Coordination boundary:** the operations that must be ordered or made indivisible together — the scope within which an invariant's read, decision and write are protected.

11.5 Deferred enforcement

11.5.1 PLAIN — in simple words

1. An operation can consist of several statements. Sometimes an intermediate statement temporarily leaves a relationship incomplete, while the completed transaction restores it before becoming accepted work.
2. Immediate checking rejects a violation at the relevant earlier boundary. Deferred checking allows a supported rule to be checked later, commonly at transaction commit.
3. Deferring a rule is not disabling it. The completed transaction must still satisfy the deferred requirement or commit fails.
4. Consider adding a child and its parent in one transaction. A deferred foreign key can allow the child insert first, provided the required parent exists by successful commit.

5. A statement returning successfully is therefore not always evidence that the whole transaction can commit. The remaining deferred check may still reject the batch.
6. Failure handling matters too. In the SQLite demonstration, a failed commit caused by the missing deferred parent leaves the transaction open. The application must decide how to resolve or abandon it.
7. Not every constraint supports deferral, and referential actions can have different timing. Learn the exact engine's rules instead of assuming that one keyword postpones everything.

11.5.2 PLAIN — a picture in your head

1. Imagine assembling an application packet. During assembly, one sheet can refer to another sheet that has not yet been placed in the folder.
2. The submission desk checks the complete folder before accepting it. A temporary gap while assembling is allowed; a gap in the submitted packet is not.
3. Immediate checking resembles checking every sheet as soon as it enters the folder. Deferred checking resembles waiting for the declared submission boundary for the supported relationship.
4. If submission fails, the folder may still be on your desk rather than having vanished. You must either add the missing material under the allowed process or discard the unfinished packet.
5. **Where the comparison breaks:** transactions have engine-defined visibility, locking and error states. A paper folder does not model those details or establish that an application is entitled to repair every failure automatically.
6. The analogy explains timing, not weaker correctness. The final acceptance condition is still required.

11.5.3 PLAIN — a worked example

1. The lab creates a new parent/child pair with a deferred foreign key. These are isolated teaching tables, not the canonical shop tables.

```
CREATE TABLE batch_parent (
    id TEXT NOT NULL PRIMARY KEY
) STRICT;

CREATE TABLE batch_child (
    id TEXT NOT NULL PRIMARY KEY,
    parent_id TEXT NOT NULL REFERENCES batch_parent(id)
        DEFERRABLE INITIALLY DEFERRED
) STRICT;
```

2. Start a transaction. Insert CHILD-1 referencing PARENT-1 while PARENT-1 is absent. The insert returns successfully in this demonstration.
3. Try to commit without adding the parent. Commit fails with a foreign-key constraint error. The lab observes that the connection is still in a transaction.
4. Compare two separately executed outcomes:

Stage	Repair branch	Abandon branch
Insert child with deferred missing parent	Returns successfully	Returns successfully
First COMMIT	Fails	Fails
Transaction after failed COMMIT	Still open	Still open
Chosen next action	Insert PARENT-1, then COMMIT	ROLLBACK

Stage	Repair branch	Abandon branch
Child rows at the end	1	0
Open transaction at the end	No	No

5. The repair branch is allowed because this particular demonstration deliberately supplies the missing parent. It is not a rule to manufacture parent records in a real application merely to silence an error.
6. A second timing test uses `ON DELETE RESTRICT` on a declared deferred reference. Deleting the referenced parent is rejected during the delete operation, before an attempted commit.
7. That second result prevents an overbroad conclusion: "this foreign key is deferred" does not mean every associated referential action is postponed until commit. [S59]

11.5.4 PLAIN — what is really happening inside

1. The engine tracks whether the transaction's supported deferred relationships are satisfied. Intermediate statement success and final transaction acceptance are distinct stages.
2. The child-first insert leaves a pending relationship problem. Inserting the matching parent within the same transaction removes it; leaving the parent absent causes the commit check to fail.
3. In the observed SQLite failure, the transaction remains available for a valid repair or explicit rollback. Ignoring the exception could leave locks or uncommitted work around longer than intended.
4. A rollback abandons the changes within the transaction. It does not mean an earlier committed parent or unrelated database was erased.
5. Restrictive deletion answers a different immediate question: a referenced parent is not to be removed by that operation. Its action timing must be understood separately from the ordinary deferred relationship check.
6. The application needs clear ownership of the transaction. A helper should not silently commit or roll back a transaction started by its caller unless its contract explicitly permits that behaviour.
7. The new guarded-draft helper therefore refuses to start when the caller already has an active transaction. That makes its narrow ownership rule visible rather than guessing what the surrounding application intended.

11.5.5 TECHNICAL — the engineer's version

1. The executed child example declares `DEFERRABLE INITIALLY DEFERRED` on a SQLite foreign key and runs inside an explicit transaction. With no surrounding explicit transaction, a single statement's automatic transaction boundary would not provide the same multi-statement construction window. [S59]
2. The lab records statement return, first commit failure, `Connection.in_transaction`, final child count and final transaction state. Those are distinct observations; a single "error raised" assertion would not establish the complete recovery behaviour.
3. SQLite documents that a failed commit due to deferred foreign-key violations leaves the transaction open. Its RESTRICT action is processed promptly even when the corresponding foreign key is deferred. The two isolated tests exercise those specific documented conditions. [S59]
4. PostgreSQL 17 provides `SET CONSTRAINTS` to change checking mode for eligible deferrable constraints in the current transaction. Switching from deferred to immediate can force outstanding work to be checked then. This is a documented comparison, not an operation run by our SQLite helper. [S73]

5. Do not assume that not-null and CHECK conditions can all be deferred with that command. PostgreSQL documents the eligible classes; the CREATE TABLE declaration must also support the intended timing. [S73] [S74]
6. Deferred enforcement can simplify coordinated insertion of related data, but it changes where an error can surface. Callers must handle failure at commit as well as at execute time.
7. A correct retry or repair policy depends on the failure cause and transaction state. Reissuing one statement blindly can be inappropriate after other failures; read the exact engine/driver documentation and retain the failed operation's context.
8. No power interruption, multi-host transaction or PostgreSQL runtime was tested here. These results establish selected SQLite transaction-boundary behaviour in the disclosed local environment, not a general durability or cross-engine guarantee.

■ 11.5.6 WORDS — remember these

1. **Immediate constraint:** a rule checked at its ordinary early boundary — an invariant enforced without the supported transaction-level postponement.

2. **Deferred constraint:** a rule whose check is postponed to a declared later boundary — an eligible constraint allowed to be temporarily unsatisfied within a transaction but required by successful final checking.

3. **Commit-time failure:** rejection when trying to complete a transaction — an error arising at the commit boundary rather than necessarily during an earlier statement.

4. **Transaction ownership:** responsibility for starting and ending a transaction — a contract defining which component may commit, roll back or leave the connection active.

5. **Repair branch:** a deliberate resolution of incomplete transactional work — an explicitly permitted continuation that restores the failed condition before a later commit.

11.6 Error handling and migration

■ 11.6.1 PLAIN — in simple words

1. An error is information about an operation, not a universal description of everything the database has done. Ask what failed, what remains and whether a transaction is still open.
2. A failed statement can be undone while earlier statements in the same transaction remain pending. A complete rollback is a different event.
3. A key conflict, a missing parent and a locked database have different causes. Showing users the same "something went wrong" message hides useful distinctions; guessing a cause from incomplete evidence is no better.
4. A constraint change also has a before and after. Adding a rule raises two questions: will future operations obey it, and do existing records already satisfy it?
5. Enabling foreign-key enforcement on a connection is not a retroactive cleanup operation. Old orphan rows do not magically gain parents or disappear.
6. A migration is a controlled change to the database structure or representation. It needs a plan for existing data, compatible readers and writers, verification and failure handling.
7. The exercises here illustrate error states and validation gaps in new in-memory databases. They do not run a migration on a real shop database or authorize changes to any existing installation.

11.6.2 PLAIN — a picture in your head

1. Imagine writing three entries in pencil before signing the page. The second entry is rejected and erased, but the first entry may still remain unsigned on the page.
2. "The second entry failed" does not tell you whether the whole page was discarded. You need to inspect the actual procedure.
3. Now imagine introducing a new rule that every old folder must have a matching index card. Announcing the rule today does not manufacture index cards for yesterday's incomplete folders.
4. **Where the comparison breaks:** database engines define precisely which statement effects and transaction effects survive each failure class. The pencil analogy cannot replace checking those rules and actual connection state.
5. It also says nothing about safe restoration after an irreversible external action. A transaction rollback in this lab does not retract an email, refund a payment or reverse a file sent elsewhere.
6. Use the picture to ask the right questions: what is pending, what is committed, what is invalid, and which recovery action is actually allowed?

11.6.3 PLAIN — a worked example

1. The statement-failure demonstration starts an explicit SQLite transaction in a new table with a primary key.
2. Its first insert adds identifier 1. A later single statement tries to insert identifiers 2 and 1. The repeated 1 causes a primary-key failure under the default ABORT behaviour.
3. The lab then inspects the same connection. Identifier 1 from the earlier successful statement is still present inside the transaction; identifier 2 from the failed statement is not.
4. The transaction is still open. An explicit rollback removes the earlier uncommitted identifier 1 as well. The final table is empty.
5. A separate deliberately unprotected database is created only for a legacy-data counterexample. It stores one orphan child while enforcement is off, then enables enforcement before inspection.

Observation in the isolated legacy example	Actual result	What it does not establish
`PRAGMA foreign_keys` after enabling	1	That old rows were retroactively repaired
`PRAGMA integrity_check`	`ok`	That all foreign-key relationships are valid
`PRAGMA foreign_key_check`	One violation	The correct business repair or missing parent identity
Existing child count	1	That the orphan is an approved record

6. The specific relationship check finds the problem that the general integrity check does not cover. SQLite documents their different scopes. [S75]
7. Neither demonstration is an instruction to disable checks in an existing database. Both create their own disposable in-memory structures, and no user file is opened.

11.6.4 PLAIN — what is really happening inside

1. A statement has an execution boundary; a transaction has a larger acceptance boundary. The engine's conflict handling determines which effects are undone for a particular error.

2. In the tested default ABORT case, the failing statement's changes are backed out, while earlier statements in the active transaction remain pending. Explicit rollback abandons that transaction's earlier work too. [S76]
3. The application must not report a successful complete operation merely because the first statement succeeded. It must observe the required final boundary and handle the actual error state.
4. A migration adds another kind of boundary: the database and its consumers move from an old contract to a new contract. Existing rows can conflict with the new rule even when future writes are well validated.
5. Inspecting existing records produces evidence, but an inspection result can go stale while writers continue changing the database. A real migration plan must coordinate validation and enforcement rather than assume a one-time query remains true forever.
6. Any repair needs a meaning-preserving policy. Automatically deleting orphan rows or inventing missing parents may silence structural errors while destroying evidence or creating false claims.
7. For the book's fictional examples, the correct action is to report the deliberate violation and discard the disposable exercise database. There is no production repair decision to infer.

11.6.5 TECHNICAL — the engineer's version

1. The statement-failure test records the SQLite extended error name `SQLITE_CONSTRAINT_PRIMARYKEY`, the remaining rows and transaction state. Programmatic error codes or classes are preferable to treating the exact English wording of one driver message as a permanent interface. [S77] [S86]
2. Do not generalise this one ABORT case to every SQLite error or conflict algorithm. Other failure classes and policies have different rollback effects. The lab intentionally uses ordinary statements and reports the tested scope. [S76]
3. For a planned constraint change, prepare an inventory of old data violations, the proposed resolution policy, the final declaration and the verification query. Keep the evidence before and after the change separate.
4. PostgreSQL's ALTER TABLE facilities include distinct ways of adding and validating supported constraints. Which operations scan existing rows and which locks or compatibility effects arise depends on the exact operation. Those details require a version-specific migration plan rather than a generic "run ALTER TABLE" recipe. [S78]
5. A portable design discussion does not imply portable syntax. SQLite and PostgreSQL expose different table-change capabilities. None of the PostgreSQL migration examples is executed in the companion labs.
6. Test both desired rejections and preserved valid cases after a proposed rule change. A rule that blocks all writes will reject bad data while also destroying the intended workflow; rejection counts alone do not establish correctness.
7. Keep validation queries and the data snapshot they examined identifiable. A report saying "zero violations" without its database, schema version, timing and concurrent-write assumptions is incomplete evidence.
8. The current lab provides no backup orchestration, online migration, multi-client deployment or production rollback plan. Its contribution is to make the error and validation boundaries observable before later operational chapters build on them.

11.6.6 WORDS — remember these

1. **Statement rollback:** undoing effects of one failed statement — a failure response distinct from abandoning all earlier work in the active transaction.
2. **Transaction rollback:** abandoning a transaction's pending changes — an explicit or engine-triggered reversal within that transaction's defined scope.
3. **Constraint validation:** checking records against a declared rule — an operation whose data scope and timing must be distinguished from future enforcement.
4. **Schema migration:** a controlled change to database structure or representation — a transition requiring data compatibility, verification and a defined failure disposition.
5. **Extended error code:** a machine-readable refinement of a failure category — an implementation-specific identifier that distinguishes more precise error causes.

11.97 Practice and worked answers

1. **Presence versus range.** A column is declared `INTEGER CHECK (value > 0)`. Predict the results for NULL, 0 and 3 in the executed SQLite profile. Then state what declaration makes a positive value required.
2. **Scope of uniqueness.** Why can both O-1042 line 1 and O-1043 line 1 exist? Would two separate UNIQUE rules on `order_id` and `line_no` express the same policy as the composite primary key?
3. **Complete optional links.** A receipt reference has branch BR-A but no local number. Explain why the loose nullable composite foreign key accepts it and how the checked example rejects it without forbidding a completely absent optional link.
4. **Find the missing invariant.** CAP-DEMO has capacity 10. Allocations 7 and 6 both pass their row checks. Calculate the violation and explain why neither the row checks nor the parent reference protects the required total.
5. **Trace the failed commit.** A deferred child insert returns successfully, but its missing parent is never inserted. What happens at COMMIT in the lab, and how do the repair and rollback branches differ?
6. **Interpret the checks.** An isolated database reports `foreign_keys = 1` and `integrity_check = ok`, but `foreign_key_check` returns one violation. Are those observations contradictory? What business repair do they establish?
7. **Do not overread an error.** The second statement of a transaction fails with a primary-key conflict under the tested ABORT policy. Does that establish that the first statement's pending change was rolled back too?

Worked answers

1. NULL is accepted because the CHECK's unknown result does not violate that definition. Zero is rejected and three is accepted. `INTEGER NOT NULL CHECK (value > 0)` adds the missing presence rule. The accepted integer still needs an explicit unit and workflow meaning.
2. The pairs differ in their order component. Separate uniqueness on `order_id` would prohibit two lines on one order; separate uniqueness on `line_no` would prohibit line number 1 on another order. Neither is our intended line identity rule.
3. In the loose SQLite definition, a NULL component avoids the ordinary composite parent-match requirement. `CHECK ((branch IS NULL) = (local_no IS NULL))` rejects a half-filled pair, while

both NULL values pass as an intentionally absent reference. The foreign key then rejects complete pairs without a parent.

4. The sum is 13, exceeding 10 by 3. The row conditions restrict each allocation individually, and the foreign key only establishes that CAP-DEMO exists. A coordinated cross-row invariant is missing. This sequential counterexample does not establish a production concurrency solution.
5. The first COMMIT fails and the tested connection remains in a transaction. The repair branch inserts the deliberate matching parent and commits, leaving one child. The abandon branch rolls back, leaving no child. Both finish without an open transaction. A real repair would need authority and truthful parent evidence rather than automatic invention.
6. They are not contradictory. The first reports the connection setting; the second checks its documented structural scope; the third specifically reports relationship violations. None decides whether to remove the child, supply a parent or correct the reference. That decision needs the original meaning and evidence.
7. No. In the observed default ABORT case, the failing multi-row statement is undone, but the earlier successful insert remains pending. Explicit transaction rollback removes that earlier uncommitted work. Other error classes must be interpreted using their own documented behaviour.

11.98 Common wrong ideas

1. **Wrong:** a positive CHECK automatically requires a value. **Better:** combine range and presence explicitly when absence is prohibited.
2. **Wrong:** not-null text cannot be empty or meaningless. **Better:** NULL, empty text and semantic correctness are different questions.
3. **Wrong:** UNIQUE has one portable missing-value policy everywhere. **Better:** inspect the engine and declaration; the SQLite example permits multiple NULL references.
4. **Wrong:** two individually unique columns are the same as a unique pair. **Better:** separate uniqueness is a different and often much stronger restriction.
5. **Wrong:** a foreign key proves a transaction is authorized. **Better:** it checks a structural relationship, not a person's authority or the truth of an agreement.
6. **Wrong:** any composite foreign key requires all components. **Better:** optional and half-filled references need explicit null policy.
7. **Wrong:** safe individual quantities imply a safe total. **Better:** 7 and 6 each fit within 1–10, but their sum exceeds capacity 10.
8. **Wrong:** a deferred constraint is switched off. **Better:** its supported check is postponed, and the required final boundary can still reject the transaction.
9. **Wrong:** a successful execute call proves commit will succeed. **Better:** deferred checks and other failures can surface later.
10. **Wrong:** a failed statement always rolls back the entire transaction. **Better:** observe the specific error policy and actual transaction state.
11. **Wrong:** turning on foreign keys repairs old orphan rows. **Better:** enabling enforcement and validating existing relationships are separate operations.
12. **Wrong:** an integrity check is a complete audit of every business rule. **Better:** every check has a defined scope; unimplemented semantic and workflow rules remain outside it.

11.99 Chapter summary in 20 lines

1. A constraint is an enforced rule with a defined checking boundary.
2. Presence, type, range and real-world meaning are separate requirements.
3. A nullable positive CHECK can accept NULL in the demonstrated engines.
4. NOT NULL excludes missingness, not empty text or false information.
5. A composite key constrains the complete tuple rather than each component separately.
6. Our line key preserves order scope and allows repeated products on separate lines.
7. Missing-value treatment in uniqueness needs an engine-specific declaration.
8. A generated identity is not a complete duplicate-delivery policy.
9. Foreign keys maintain declared child-to-parent relationships.
10. Optional composite links may need an all-or-none presence check.
11. Referential actions must follow record lifecycles, not cleanup convenience.
12. A required child reference does not guarantee every parent has children.
13. Individual allocation bounds do not protect a combined capacity total.
14. State the cross-row invariant before choosing its coordination mechanism.
15. Deferred checking postpones a supported rule rather than removing it.
16. In the SQLite demonstration, a failed deferred commit leaves the transaction open.
17. A restrictive parent deletion can fail before commit even with a deferred reference.
18. Statement rollback and transaction rollback can have different scopes.
19. Existing-data validation is distinct from enabling future enforcement.
20. A migration needs evidence, compatibility and recovery planning beyond these disposable tests.

12

CHAPTER

SQL From Zero: Asking Your First Questions

12.0 What this chapter gives you

1. You will read a small SELECT statement as a question about named records rather than a mysterious command.
2. You will choose explicit output columns and keep track of what one result row represents.
3. You will filter values with comparisons, AND, OR and explicit parentheses.
4. You will explain why a missing customer link needs `IS NULL`, not equality with NULL.
5. You will request a meaningful, deterministic order and resolve ties before applying a row limit.
6. You will calculate line amounts without replacing agreed prices with current catalogue values or losing units.
7. You will use aliases, CASE and COALESCE while distinguishing presentation from stored truth.
8. You will insert, update and delete only inside a disposable draft workspace, inspecting target rows and rollback behaviour.
9. You will bind values as parameters instead of mixing untrusted text into SQL syntax.
10. You will run a guarded draft update and state the limits of its expected-value check, transaction and tests.

All runnable examples in this chapter belong to fresh teaching databases. They do not connect to a live shop, the owner's website or an existing database file. The two canonical agreed orders remain unchanged; editing exercises use the separate fictional draft D-SQL-01. Joins, grouping and normalisation appear in greater depth in subsequent chapters. Here the goal is to ask a few precise questions and observe exactly what the answers establish.

12.1 Reading SELECT aloud

12.1.1 PLAIN — in simple words

1. SQL is a language for describing operations on a database. Its name expands to Structured Query Language. In this chapter we start with SELECT, which describes a result we want to read.
2. Read a simple statement as "show these values from these records". `SELECT` names the output values; `FROM` names the source table.
3. The output is a result table with rows and columns. It can omit source columns, calculate new values and later combine or group records. It is not automatically a full copy of the original table.
4. SQL does not require you to write a loop that visits each record. You describe the required result, and the engine chooses a way to produce it under its supported semantics.
5. Explicit column names make the question easier to review. `SELECT *` is useful for quick exploration, but an application relying on a stable output shape should know which columns it expects.

6. A semicolon commonly marks the end of a SQL statement. Newlines and indentation make a longer statement easier for people to read; they are not instructions to process one row per line of code.
7. The read-only examples below ask about recorded values. They do not establish whether the underlying fictional sale actually occurred; that distinction from Part A still applies.

12.1.2 PLAIN — a picture in your head

1. Imagine asking a filing assistant for a report: "From the order-line cards, copy the order identifier, line number, product, quantity and agreed unit price."
2. You specify the answer's fields, not every movement of the assistant's hands. The assistant can choose an efficient route through the filing system.
3. If you ask only for product names, you may receive repeated names because different line cards mention the same product. The report has not magically become a unique product catalogue.
4. **Where the comparison breaks:** a database follows formal rules and may use indexes, parallel work or other execution strategies. It does not interpret a vague request as a human assistant might.
5. In particular, do not expect it to guess missing ordering, units or business intent. Those belong in the query or its documented result contract.
6. The analogy helps you read the question aloud. The engine's physical execution plan is a separate topic, developed later.

12.1.3 PLAIN — a worked example

1. Open the extracted practice-code folder. The following Python session uses the provided helper to create a fresh in-memory copy of the four agreed lines. No connection string or password is required.

```
from contextlib import closing
from schema_sql_lab import canonical_shop

with closing(canonical_shop()) as db:
    rows = db.execute("""
        SELECT order_id, line_no, product_id,
               quantity, unit_price_minor
        FROM order_lines
        ORDER BY order_id, line_no
    """).fetchall()
    for row in rows:
        print(row)
```

2. Read the SQL aloud: "Show each line's order, line number, product, quantity and agreed unit price, from order lines, ordered by order and then line number."
3. The result is the unchanged canonical fixture:

order_id	line_no	product_id	quantity	unit_price_minor
O-1042	1	P-NOTE	2	7,550
O-1042	2	P-PEN	1	2,000
O-1043	1	P-NOTE	1	7,550
O-1043	2	P-PEN	2	2,000

4. Python prints tuples rather than this typeset table. The underlying values agree; thousands separators in the book are presentation, not literal commas stored inside the integer amounts.
5. Asking only for `product_id` returns four visible product values. Asking for `DISTINCT product_id` returns two distinct represented product identifiers. Neither question asks how many physical units were sold.
6. `fetchall()` is appropriate for this four-row exercise. Fetching an unbounded production result into memory has a different resource cost; later chapters discuss limits and execution plans.

12.1.4 PLAIN — what is really happening inside

1. The driver sends the statement to the engine associated with this connection. The engine parses the SQL, resolves the named table and columns, and prepares a way to produce the result.
2. The select list describes output expressions. A bare column name copies that column's value for the relevant source row; a later arithmetic expression calculates a value from it.
3. The engine returns rows to the driver, and Python exposes those rows through a cursor. `fetchall()` collects the remaining result rows into a list in this interface. [S86]
4. The input table's key need not appear in the result. Omitting the key can make different source records look the same, even when their identities remain distinct in storage.
5. `DISTINCT` operates on the selected output values. It does not decide whether two real-world events were duplicates or whether earlier input was incorrect. [S80]
6. The physical route can change without changing the required answer. Adding a later index should not entitle an application to rely on an unrequested result order.
7. The known fixture gives us a reference answer that can be checked by hand. Start with that small observable case before interpreting the speed or correctness of a large result.

12.1.5 TECHNICAL — the engineer's version

1. SELECT is declarative: its clauses specify result semantics, while the engine chooses an execution strategy. The simplified conceptual sequence used in this chapter is an explanatory model, not a promise about physical operator order. [S79] [S80]
2. In the basic single-table case, identify the source, apply the row predicate, evaluate output expressions, apply requested duplicate elimination, order the result and apply the row limit. Later features introduce additional stages and interactions.
3. An output column can be a source column, a literal or an expression. For example, `quantity * unit_price_minor` is an expression with the current line's agreed quantity and price as operands.
4. Keep identifier and literal syntax distinct. `order_id` names a column; `'O-1042'` is a text value. SQL text literals use single quotes in these examples. Do not teach double-quoted identifiers as a portable replacement for text literal syntax. [S72]
5. The example's Python connection is a new in-process SQLite database. PostgreSQL tutorial syntax is useful comparative documentation, but a PostgreSQL server, roles and connection lifecycle are not exercised here.
6. The Python helper closes the database after the block using `contextlib.closing`. A connection's transaction context-manager behaviour and actually closing that connection are not the same operation; the distinction is explicit in the driver documentation. [S86]
7. Basic read-only SELECT statements here have no intended mutation effects. Do not generalise that every possible SELECT in every engine is incapable of side effects; functions and product-specific features require their own contracts.

8. The result contract is five named fields, one row per canonical order line, integer quantities, integer paise and explicit identity ordering. That is a precise answer to review against the fixture rather than the vague goal "show the sales data".

12.1.6 WORDS — remember these

1. **SELECT statement:** a description of an answer to read — a SQL query specifying output expressions and the operations needed to define a result.
2. **Select list:** the values requested for each output row — the column references or expressions following SELECT.
3. **Text literal:** text written as a value in a statement — a quoted string such as `'O-1042'`, distinct from a column identifier.
4. **Cursor:** the driver's interface to a statement and its results — an object through which rows and relevant execution information are retrieved.
5. **Declarative query:** a request that describes the required result — a statement whose logical meaning does not prescribe one physical execution plan.

12.2 Filtering and NULL

12.2.1 PLAIN — in simple words

1. A filter selects records meeting a condition. In SQL, a WHERE clause expresses that condition.
2. `WHERE quantity = 2` means "keep lines whose recorded quantity is two". It does not mean "keep two rows" or "find orders containing exactly two lines".
3. AND requires both conditions to hold. OR allows either. Parentheses make the intended grouping clear when several conditions are combined.
4. A WHERE filter keeps a row when its condition evaluates to true. False and unknown results are not retained.
5. NULL is the marker used for missing information in these examples. Ordinary equality with NULL does not ask whether a value is missing. Use `IS NULL` for that question.
6. "Not equal to C-001" and "not equal to C-001, including missing links" are different questions. The second needs to mention the missing case explicitly.
7. A filter can answer only a condition defined over available data. No linked profile does not automatically reveal why the link is absent or identify the buyer.

12.2.2 PLAIN — a picture in your head

1. Imagine sorting cards into a tray labelled "quantity is two". A card showing two goes in; a card showing one does not.
2. A card with the quantity field missing does not prove that its quantity is two. It also does not prove that the quantity is one. For this particular positive condition, it is not selected.
3. Now make a different tray labelled "quantity missing". That is a direct question about absence rather than a numeric comparison.
4. **Where the comparison breaks:** SQL does not rely on a person's interpretation of a blank card. Its operators define precise truth outcomes, including how unknown combines with AND and OR.

5. A display may draw NULL as an empty cell, the word NULL or another label. The visual appearance does not change the stored marker's query semantics.
6. Ask the desired question explicitly instead of hoping a comparison will interpret absence in the way a human reader expects.

12.2.3 PLAIN — a worked example

1. Find the canonical lines with quantity two, and calculate their line amounts:

```
SELECT order_id, line_no,
       quantity * unit_price_minor AS line_total_minor
FROM order_lines
WHERE quantity = 2
ORDER BY order_id, line_no;
```

2. The answer contains O-1042 line 1 with 15,100 paise and O-1043 line 2 with 4,000 paise. That is two lines, representing four units and 19,100 paise under the unchanged fixture.
3. Next ask which orders have no linked customer profile:

```
SELECT order_id
FROM orders
WHERE customer_id IS NULL
ORDER BY order_id;
```

4. The answer is O-1042. Replacing `IS NULL` with `= NULL` returns no rows in the lab, not the desired missing-link result.
5. Now compare two other predicates. `customer_id <> 'C-001'` returns no rows: O-1043 equals C-001, while O-1042's comparison is unknown. Adding `OR customer_id IS NULL` returns O-1042.
6. A further deliberately misleading predicate, `customer_id NOT IN ('C-001', NULL)`, also returns no rows for this fixture. The NULL in the list prevents the easy "everything not in this list" intuition from being a safe general explanation. [S24]
7. Each outcome is recorded by `first_questions()`. The test names distinguish the intended missing-value query from the deliberately incorrect equality example.

12.2.4 PLAIN — what is really happening inside

1. An expression can yield true, false or unknown. The missing customer link is not silently replaced with a made-up identifier before evaluation.
2. `customer_id IS NULL` directly tests the marker and produces a known truth result. Ordinary equality compares values, and an unavailable operand cannot generally establish equality or inequality.
3. Combining conditions requires the full truth rules. True AND unknown is unknown; false AND unknown is false. True OR unknown is true; false OR unknown is unknown.
4. Those rules explain the explicit missing branch. On O-1042, `customer_id <> 'C-001'` is unknown but `customer_id IS NULL` is true; unknown OR true is true, so the row is selected.
5. Parentheses determine the grouping you intend. SQL's normal operator precedence does not rescue a predicate whose writer mentally grouped it another way.
6. For example, "quantity is two, or quantity is one and product is notebook" also includes the two-unit pen line. "Quantity is two or one, and product is notebook" requires parentheses around the quantity alternatives and selects the two notebook lines instead.
7. The database evaluates the written predicate, not the surrounding business sentence. Read both aloud and test a row that distinguishes the two interpretations.

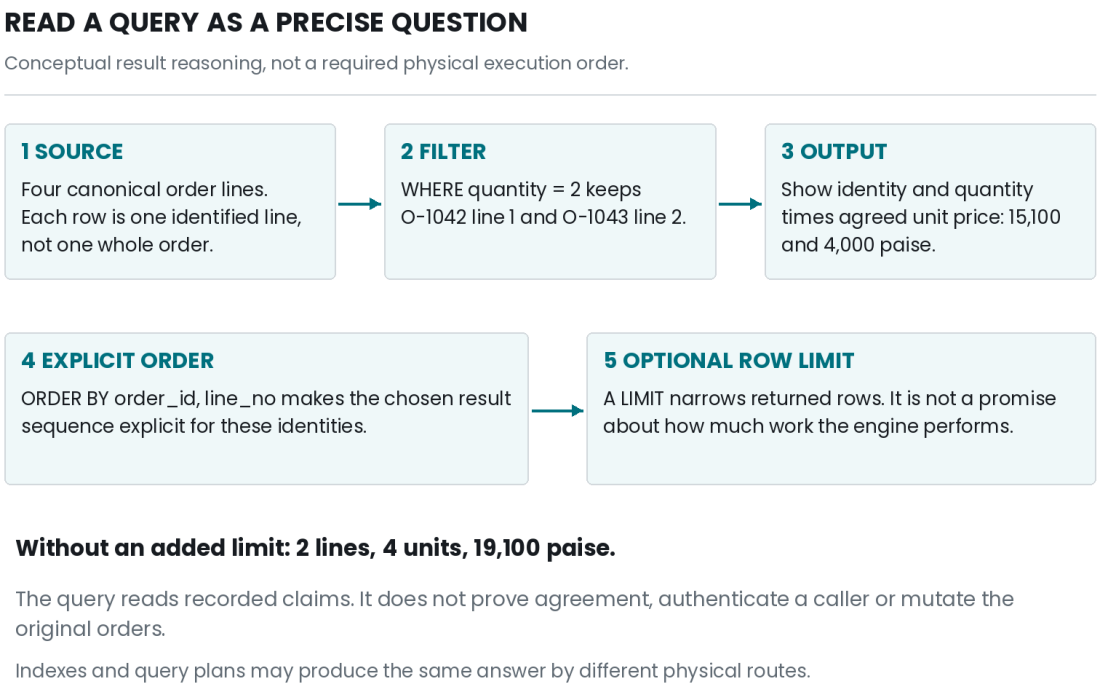

Figure 12.6: Figure 12 — A conceptual path through the quantity-two query: source, filter, output, explicit order and an optional result limit. These are result semantics, not a required physical execution plan.

12.2.5 TECHNICAL — the engineer's version

1. Three-valued logic is easiest to verify with a small table. Here U means unknown and T/F mean true/false. These cases are the logical results relevant to the illustrated predicates. [S24]

Left and right values	AND result	OR result
T and U	U	T
F and U	F	U
U and U	U	U
T and F	F	T

2. `NOT U` remains U. Therefore merely negating an ordinary comparison with a missing operand does not make missingness become a known positive match.
3. `IN` and `NOT IN` need special attention when a list or subquery can contain NULL. The lab's fixed two-item example is a direct demonstration, not a recommendation to clean unknown values by deleting them indiscriminately.
4. Parenthesise mixed AND/OR predicates for intent. Compare `quantity = 2 OR quantity = 1 AND product_id = 'P-NOTE'` with `(quantity = 2 OR quantity = 1) AND product_id = 'P-NOTE'`. The first admits three canonical lines; the second admits two. [S24]
5. `BETWEEN` includes both endpoints in these SQL examples. For the storage range `BETWEEN 1 AND 10 000`, 1 and 10,000 are not off-by-one exceptions. Missingness still needs a separate policy.

6. Keep record identity semantics distinct from convenience filtering. The provided `lines_for_order()` binds the exact identifier rather than trimming or case-folding it. A caller must not silently change an identifier to make a query appear to work.
7. WHERE keeps true rows; it does not report why the others failed. A missing result can mean no matching stored row, a predicate mismatch or an intended exclusion. Diagnose using authorized additional evidence rather than inventing one explanation.
8. For a real application, filtering is not authorization by itself. A user-supplied order identifier that returns a valid record still needs an access decision. This learning database deliberately contains no user authentication or permission system.

12.2.6 WORDS — remember these

1. **WHERE predicate:** the condition deciding which source rows qualify — a Boolean-like SQL expression for which only true rows are retained by the filter.
2. **Three-valued logic:** reasoning with true, false and unknown — SQL expression semantics needed when operands can be NULL.
3. **IS NULL:** a direct test for missingness — a SQL predicate that checks whether an expression yields the NULL marker.
4. **Operator precedence:** the default grouping rules for an expression — syntax rules that determine binding unless parentheses explicitly override the grouping.
5. **Inclusive interval:** a range containing its endpoints — the boundary interpretation used by BETWEEN in the illustrated SQL conditions.

12.3 Explicit ordering

12.3.1 PLAIN — in simple words

1. A query result is not guaranteed to arrive in the order you saw last time unless the query asks for an appropriate order.
2. `ORDER BY order_id, line_no` means sort first by order identifier, then sort lines within equal order identifiers by line number.
3. `ASC` requests ascending order and is the default direction for an ordinary listed sort expression. `DESC` requests descending order.
4. Sorting by a value that can tie does not fully specify the order between tied rows. Add a suitable tie-breaker when the sequence matters.
5. A LIMIT clause restricts how many rows the result includes. Without a defined order, “the first two” is not a reliable description of which two the application will receive.
6. Even a deterministic ordering of one result does not freeze the database for several later page requests. Concurrent changes and snapshot choices are a separate question.
7. Order by the meaning the reader needs, not a guessed physical storage position. The key can be a useful tie-breaker, but it does not automatically mean chronological order.

12.3.2 PLAIN — a picture in your head

1. Imagine sorting exam cards by score. Two students can have the same score, so their order is still undecided.

2. Adding an agreed second rule, such as a unique registration identifier, makes the ordering explicit for ties.
3. Taking the first two cards before sorting answers a different question from sorting by score and then taking the first two.
4. **Where the comparison breaks:** database execution need not physically sort every row before applying a limit. An index or another plan may produce the same required answer more directly.
5. The picture explains the logical result, not performance. Nor does the registration identifier prove which student took the exam earlier.
6. A repeatable order also does not prevent a card's score changing between separate requests. That requires a defined snapshot or another consistency strategy.

12.3.3 PLAIN — a worked example

1. Ask for the first two canonical line identities when higher agreed unit prices come first:

```
SELECT order_id, line_no
FROM order_lines
ORDER BY unit_price_minor DESC, order_id, line_no
LIMIT 2;
```

2. The two notebook lines both have an agreed unit price of 7,550 paise. The order identifier and line number break the tie. The result is O-1042 line 1, followed by O-1043 line 1.
3. Removing the tie-breakers leaves the order between those equal-price lines unspecified by that reduced ordering request. The fact that a tiny database happens to return them the same way twice is not a stronger contract.
4. Notice the question uses unit price, not line total. O-1042's two notebooks have a line total of 15,100, while O-1043's one notebook totals 7,550. They tie by unit price but not by line total.
5. The helper supports three fixed sort choices: identity, quantity and price. Quantity and price sorts both add the full line identity as tie-breakers.
6. An invalid sort choice is rejected; it is not copied into the statement. Parameter binding and the distinction between values and SQL structure are developed in section 12.6.
7. The number two limits returned rows. It does not promise that only two source rows were inspected or that the query took constant time. [S81]

12.3.4 PLAIN — what is really happening inside

1. The engine must produce an answer consistent with the requested ordering. It may sort intermediate results or use a suitable existing access path.
2. If no ordering is requested, a different plan, index, maintenance operation or dataset can change the observed sequence without violating the query's contract. [S03]
3. A multi-column order compares the first expression, then subsequent expressions for ties. A complete unique key after the business sort gives the illustrated fixture a total order over its returned line identities.
4. The choice of text comparison matters when text is sorted. The book's fixed ASCII-like identifiers keep the examples small; language-aware sorting of names needs explicit comparison rules rather than a universal "alphabetical" assumption.
5. NULL placement also needs an engine-aware decision when sorting optional values. This chapter's principal sorts use the non-null canonical line fields, avoiding an unstated missing-value ordering policy.

6. Pagination adds another boundary. A page shown now and a page fetched later may refer to different database states. An explicit sort is necessary for meaningful selection but not sufficient to define the whole multi-request experience.
7. Later chapters connect ordering to indexes, execution plans and keyset pagination. At this stage, do not confuse "I asked for the first two" with "the engine found the only possible first two without any other assumptions".

12.3.5 TECHNICAL — the engineer's version

1. Distinguish partial ordering by a non-unique sort key from a complete deterministic sequence under chosen comparison semantics. A unique tie-breaker completes the illustrated order, provided the query does not introduce duplicate output identities through later transformations.
2. `ORDER BY unit_price_minor DESC, order_id, line_no` orders by recorded agreed unit price and then full source identity. It does not imply ranking by quantity, contribution, recency or product popularity.
3. ORDER BY belongs to the query whose final output order is required. Do not rely on an internal subquery's incidental sequence as a substitute for a declared final ordering. [S03] [S80]
4. LIMIT is an output-cardinality constraint rather than a resource budget. A correct plan might inspect many records to identify the requested top results. Execution-plan analysis is needed before making a performance claim. [S81] [S65]
5. OFFSET pagination and concurrent changes can produce user-visible shifts between requests. Stable identity tie-breakers do not themselves supply a shared snapshot across separate transactions. The later concurrency chapters make the required isolation assumptions explicit.
6. Dynamic ordering requires trusted SQL structure. This lab maps a validated choice to one of three complete fixed statements. It does not try to bind a column name into `ORDER BY ?` and assume the placeholder becomes an identifier.
7. Test tie cases deliberately. A fixture in which every price differs can conceal missing tie-breakers. The two equal notebook unit prices make the issue observable in the canonical dataset.
8. Keep claims at their scope: the tests verify chosen result sequences for the known records and sort choices. They do not benchmark sorting or certify multilingual collation, stable web pagination or concurrent reading behaviour.

12.3.6 WORDS — remember these

1. **ORDER BY:** the requested ordering of a query result — a clause specifying sort expressions, directions and applicable ordering semantics.
2. **Tie-breaker:** an additional rule for equal primary sort values — a subsequent sort expression used to distinguish otherwise tied rows.
3. **Deterministic result order:** a sequence fixed by the stated query and data assumptions — an ordering whose relevant ties are resolved under chosen comparison rules.
4. **LIMIT:** a cap on rows returned by a query — an output selection clause that does not by itself bound all execution work.
5. **Pagination:** splitting a result into separately retrieved pages — a reading pattern needing ordering and cross-request consistency decisions.

12.4 Expressions and aliases

12.4.1 PLAIN — in simple words

1. An expression calculates a value. `quantity * unit_price_minor` multiplies two values from a line to obtain that line's amount in paise.
2. An alias gives an output expression a readable name. `AS line_total_minor` labels the result; it does not add a stored column to the original table.
3. Calculating a new output value does not necessarily change the row grain. One calculated amount per selected line is still one row per selected line.
4. Units still matter. A quantity in units multiplied by paise per unit gives paise. Multiplying an entire order total by each line's quantity would be a different and usually incorrect calculation for this question.
5. CASE chooses among explicitly stated output alternatives. For example, it can display "multi-unit line" when quantity is at least two and "single-unit line" otherwise in the present canonical fixture.
6. COALESCE chooses the first non-null expression from its arguments. It can provide a display label for a missing profile link without changing the stored link.
7. A fallback label is not recovered information. Showing "[no linked profile]" does not establish the absent person's identity or the reason no link was recorded.

12.4.2 PLAIN — a picture in your head

1. Think of writing a calculation beside a card rather than erasing the card. The original quantity and price stay intact while you display their product.
2. The alias is the heading above your calculated column. Naming it "line total" tells the reader what the new number means.
3. A missing-value display label resembles placing a clearly marked note beside an empty field. It should not pretend to be the original field's missing content.
4. **Where the comparison breaks:** expression types and arithmetic rules are defined by the engine, not by handwritten arithmetic alone. Integer division and floating-point arithmetic can surprise a reader who assumes every slash produces an exact decimal fraction.
5. SQL also has rules about where output aliases can be referenced. A convenient label is not automatically a new input column available in every clause.
6. Keep representation, calculation and presentation separate so the simple explanation remains correct when we add the engineering details.

12.4.3 PLAIN — a worked example

1. Label every canonical line and calculate its amount:

```
SELECT order_id, line_no,
       quantity * unit_price_minor AS line_total_minor,
       CASE WHEN quantity >= 2 THEN 'multi-unit line'
            ELSE 'single-unit line'
       END AS quantity_label
FROM order_lines
ORDER BY order_id, line_no;
```

2. The resulting amounts are 15,100, 2,000, 7,550 and 4,000 paise in identity order. The first and fourth lines receive the multi-unit label.

3. The label does not group the rows. There are still four line results. No new stored label or total has been inserted into `order_lines`.
4. For customer-profile display, use:

```
SELECT order_id,
       COALESCE(customer_id, '[no linked profile]') AS profile_label
FROM orders
ORDER BY order_id;
```

5. O-1042 displays the bracketed label; O-1043 displays C-001. The original NULL remains NULL in the stored order row.
6. A separate expression test asks SQLite for `7550 / 100` and `7550 / 100.0`. The results are 75 and 75.5 respectively in this runtime: the first uses integer arithmetic, while the second includes a real operand.
7. Do not use that second result as an excuse to move monetary calculations into binary floating-point. Keep exact integer paise for the bounded accounting arithmetic; format the number deliberately for display.
8. Formatting 7,550 paise as INR75.50 should preserve both decimal places. A display string and an exact numeric amount are different outputs with different purposes.

12.4.4 PLAIN — what is really happening inside

1. For each selected source row in this simple query, the engine evaluates the requested expressions and produces the corresponding output values.
2. The alias changes the result's label, not the source definition. A later SELECT from the original table does not discover a new stored `quantity_label` column.
3. CASE evaluates the stated conditions to select an output alternative. Our example's ELSE branch means single-unit only because the canonical quantity domain is present and positive and the predicate distinguishes quantities of at least two.
4. If we reused that CASE on a nullable draft quantity, an unknown quantity could fall through to ELSE. Labelling it single-unit would be false. A separate missing branch would be needed for that different domain.
5. COALESCE works on missingness, not on every kind of undesirable value. An empty string is present, so it is not skipped merely because it looks blank on screen. [S87]
6. The arithmetic's operand types affect the result. The unit of the value remains a semantic responsibility even when the expression executes without error.
7. The right habit is to inspect a result using known inputs, including missing and boundary cases, before embedding the expression in a larger report.

12.4.5 TECHNICAL — the engineer's version

1. Aliases improve the result contract, but do not assume uniform alias visibility in WHERE across engines. A portable introductory pattern writes the input expression in WHERE or uses a deliberately structured outer query once subqueries have been taught. [S57]
2. For example, filter a line amount with `WHERE quantity * unit_price_minor >= 10000` rather than relying on the select-list alias as though it were an original table column. Only O-1042 line 1 meets that threshold in the canonical fixture.
3. SQLite's arithmetic behaviour depends on operand types, and its expression documentation distinguishes integer from floating-point operations. The `7550/100` demonstration is an observed type-semantic example, not a recommendation for monetary rounding. [S24]

4. SQL NULL commonly propagates through ordinary arithmetic. A draft line with an absent price does not yield a known zero amount when multiplied by a supplied quantity. A report must choose how to represent incomplete quotations, not erase them through unlabelled substitution.
5. CASE and COALESCE are tools for expressing explicit decisions. PostgreSQL's conditional-expression documentation provides the corresponding product-specific details, while the executed SQLite expressions follow its expression and scalar-function references. [S82] [S24] [S87]
6. A nullable draft quantity needs a three-way label: absent, one, or more than one under its valid range. Reusing the canonical two-way CASE without adapting the domain would silently misclassify absence.
7. Arithmetic overflow analysis remains separate from data type names. Per-line bounds make the demonstrated integer product safe; a future large aggregation must also account for the number of contributing rows and engine-specific overflow behaviour.
8. The present tests check selected calculated outputs, aliases as result labels, original stored values and the canonical totals after draft-only work. They do not establish every possible expression's cross-engine portability or all monetary rounding policies.

12.4.6 WORDS — remember these

1. **SQL expression:** a rule for calculating an output value — a combination of column references, literals, operators or functions evaluated under engine semantics.
2. **Alias:** a name attached to an output expression — a result-column label that does not itself alter the source table's schema.
3. **CASE expression:** an explicit choice among result alternatives — a SQL conditional expression selecting an output according to its stated conditions.
4. **COALESCE:** choosing the first supplied value — an expression returning the first non-null argument, not a recovery of missing truth.
5. **Integer division:** division evaluated using integer operands and rules — an operation that can discard the fractional component rather than produce a decimal quotient.

12.5 Inserting and updating safely

12.5.1 PLAIN — in simple words

1. Reading a result and changing stored information are different responsibilities. INSERT adds rows, UPDATE changes matching rows and DELETE removes matching rows.
2. Before a change, identify its exact target and intended meaning. "Change the first line" is incomplete unless we also know the order or draft it belongs to.
3. The exercises use only a new mutable draft workspace. They do not edit the agreed canonical orders, perform finalisation or imitate an approved historical correction.
4. A WHERE clause controls the target of an UPDATE or DELETE. Leaving it out can apply the operation to every row in the table.
5. Even a present WHERE clause can be too broad. `WHERE line_no = 1` can match the first line of many drafts. Use the complete identity when the intention is one identified line.

6. A transaction lets the exercise inspect pending changes and then either commit or roll them back. It is not a licence to experiment against a live database with real consequences.
7. A successful write also needs result checking. Updating zero rows is not automatically an engine error, and changing several rows is not proof that several changes were intended.

12.5.2 PLAIN — a picture in your head

1. Imagine editing a pencil draft on a separate desk, away from the already agreed documents. You can try a change, inspect it and erase the experiment without altering the accepted records.
2. Writing a new slip resembles INSERT. Changing a field on a selected slip resembles UPDATE. Removing a selected draft slip resembles DELETE.
3. A precise target label prevents you from applying "change quantity to nine" to every slip on the desk.
4. **Where the comparison breaks:** database changes can be observed and coordinated under different transaction rules, and external actions are not automatically undone by a database rollback.
5. Nor does a separate table create a complete authorization boundary. The lab is isolated by its new in-memory database and controlled code, not by an implemented multi-user permission system.
6. The picture is a reason to practise on disposable material and to inspect scope, not a promise that all real-world changes are reversible.

12.5.3 PLAIN — a worked example

1. `draft_workspace()` creates a new canonical shop copy and a separate `sql_draft_lines` table. Its initial draft is:

draft_id	line_no	product_id	quantity	unit_price_minor
D-SQL-01	1	P-NOTE	3	7,550
D-SQL-01	2	P-PEN	1	NULL

2. In a newly created workspace, the following SQL illustrates the write syntax. The entire sequence is intentionally rolled back rather than becoming an agreed sale.

```
BEGIN;
INSERT INTO sql_draft_lines
    (draft_id, line_no, product_id, quantity, unit_price_minor)
VALUES ('D-SQL-01', 3, 'P-PEN', 1, 2000);

UPDATE sql_draft_lines
SET quantity = 2
WHERE draft_id = 'D-SQL-01' AND line_no = 3;

SELECT line_no, quantity, unit_price_minor
FROM sql_draft_lines
WHERE draft_id = 'D-SQL-01'
ORDER BY line_no;

DELETE FROM sql_draft_lines
WHERE draft_id = 'D-SQL-01' AND line_no = 3;
ROLLBACK;
```

3. The inserted line 3 is an additional temporary teaching record. During inspection it has quantity 2 and price 2,000; after the sequence and rollback, the workspace returns to its original two draft lines.

4. A different deliberate mistake omits WHERE: `UPDATE sql_draft_lines SET quantity = 9`. In its own fresh transaction it matches both original draft rows. Their quantities become 9 and 9 until the demonstration rolls back to 3 and 1.
5. Both nine values pass the row range. This is not a constraint failure; it is a target-selection mistake. The engine followed a valid statement whose scope was wrong for a hypothetical one-line intention.
6. The canonical agreed totals remain O-1042 = 17,100 and O-1043 = 11,550 throughout these separate draft-only exercises.
7. The teaching SQL uses explicit literal fixture values so its meaning is visible. Application code receiving values from outside should use bound parameters, as section 12.6 demonstrates.

■ 12.5.4 PLAIN — what is really happening inside

1. INSERT matches the supplied values to the named columns, then the engine applies the relevant type and constraint rules. Explicit columns avoid depending on an unexplained physical column order. [S83]
2. UPDATE identifies matching rows using its predicate and assigns the specified values. Omitting WHERE means the statement does not narrow the table to a selected subset. [S84]
3. DELETE removes matching rows under the active constraints and transaction. It does not automatically erase every copy of the information in backups, logs, exports or other systems. [S85]
4. A transaction groups the demonstrated statements, but the caller still has to decide the appropriate target, parameters and final action.
5. The wrong-scope update demonstrates an important limit: constraints protect allowed states, not every intended choice among allowed states. Setting a quantity to 9 may be structurally valid while being the wrong business action.
6. An inspection before a write helps a person understand the target, but a separate preliminary SELECT does not by itself prevent another writer changing the state between inspection and mutation.
7. The following section puts an expected quantity into the UPDATE predicate itself. That narrows one race window, but it is deliberately not presented as a complete versioning or permission protocol.

■ 12.5.5 TECHNICAL — the engineer's version

1. Treat write operations as contracts: target identity, expected prior state if relevant, proposed new values, allowed actor, transaction ownership, result cardinality and failure disposition. The toy helper implements only a disclosed subset.
2. Its draft quantity is nullable at the table level because unfinished work can be incomplete. The specific guarded-edit function requires both expected and new quantities to be supplied positive integers in range; it does not implement an operation for filling an initially absent quantity.
3. The SQL syntax exercise includes a temporary third draft line solely inside its disposable transaction. It neither publishes an agreed line nor changes the historical fixture.
4. Inspect row counts where the driver defines them for the operation. Python's SQLite `rowcount` behaviour is not a universal "business events completed" counter and differs from how rows are fetched for SELECT. [S86]

5. Do not assume a matched row's new value proves a meaningful transition occurred. Setting a field to the value it already holds can still be an accepted write under the engine's counting behaviour. A business transition needs its own definition.
6. Explicit rollback in a memory-only exercise is useful evidence of transaction handling, not evidence of power-loss recovery. None of these tests interrupts a process at a filesystem boundary or validates an off-site restore.
7. Avoid convenient conflict replacement as an unexplained INSERT policy. Rejecting a duplicate draft key keeps the collision visible; a silent replacement could discard information or trigger different referential consequences. [S76]
8. The quoted DML is deliberately bounded and non-production. A real deployment needs user authorization, durable error reporting, concurrency testing, safe operational controls and the particular correction/finalisation policy before similar operations are exposed to users.

12.5.6 WORDS — remember these

1. **INSERT:** adding records — a SQL write operation that proposes new rows under the target table's active rules.
2. **UPDATE:** changing values on selected records — a SQL write operation applying assignments to rows matching its predicate.
3. **DELETE:** removing selected stored rows — a SQL operation whose effects are constrained by referential rules and transaction boundaries.
4. **Target cardinality:** how many records an operation is intended to affect — a result-scope requirement distinct from whether each resulting row is structurally valid.
5. **Wrong-scope mutation:** a valid change applied to the wrong set of records — an operation whose predicate does not express the intended target boundary.

12.6 Parameters and transactions

12.6.1 PLAIN — in simple words

1. A SQL statement has structure and values. The structure says which operation, table and condition to use. A value might be an order identifier supplied by a caller.
2. A parameter is a place where the driver supplies a value without treating that value as new SQL instructions. In these Python/SQLite examples the placeholder is `?`.
3. Do not build a statement by pasting an outside value between SQL quotes. A name containing a quote should remain a name, not change the meaning of the command.
4. Parameters solve that value-versus-syntax problem; they do not decide whether the caller is authorized or whether the amount is in the right unit.
5. Table names and sort instructions are SQL structure, not ordinary values. This lab validates a small sort-choice label and selects a complete fixed statement rather than pasting the label into the query.
6. A transaction groups the operation's database effects under a stated boundary. The helper starts its own transaction, checks the result and either commits or rolls back.
7. An expected-value condition can reject an edit when the currently stored value differs from the one the caller expected. It does not prove that no other field or earlier sequence changed.

8. The final habit is to report only what the operation establishes: exact target matched, chosen value updated and transaction completed in this test—not a claim of complete business correctness or exactly-once delivery.

12.6.2 PLAIN — a picture in your head

1. Imagine a printed instruction form with fixed boxes. The form says "find the order whose identifier equals [value]". The caller fills the value box; they do not rewrite the instructions printed around it.
2. A bound parameter keeps that separation. Even text that looks like a command remains the supplied value for the comparison.
3. Choosing the form itself is a separate decision. A caller may choose "sort by price" from an approved set, but does not get to replace the printed form with arbitrary instructions.
4. An expected-value edit resembles saying "change the draft quantity from three to two, but only while it still says three". If it now says four, the condition does not match.
5. **Where the comparison breaks:** a value can change from three to four and back to three. The final visible three does not reveal the intervening edit. The simple expected-value check cannot detect that history.
6. Transaction and parameter mechanisms therefore protect specific boundaries. They are useful precisely when their limits are kept clear.

12.6.3 PLAIN — a worked example

1. The provided lookup binds an order identifier as a value:

```
rows = db.execute(
    "SELECT order_id, line_no, product_id, quantity, unit_price_minor "
    "FROM order_lines WHERE order_id = ? ORDER BY line_no",
    ("O-1042",),
).fetchall()
```

2. The comma in the one-item tuple matters in Python. It makes a sequence containing one bound value rather than just a parenthesised string.
3. The lab also supplies the hostile-looking text `O-1042' OR 1=1 --` as the parameter. It returns no rows because there is no order with that literal identifier. It does not turn the comparison into "return every order".
4. Now use a fresh draft workspace and ask for a guarded change from three notebooks to two:

```
from contextlib import closing
from schema_sql_lab import draft_workspace, update_draft_quantity

with closing(draft_workspace()) as db:
    update_draft_quantity(db, "D-SQL-01", 1, 3, 2)
    print(db.execute(
        "SELECT line_no, quantity FROM sql_draft_lines "
        "WHERE draft_id = ? ORDER BY line_no",
        ("D-SQL-01",),
    ).fetchall())
```

5. The result is `[(1, 2), (2, 1)]`. The second line's absent quotation remains absent, and the agreed order totals remain unchanged.
6. Repeat the edit while still expecting quantity three: its predicate matches no row because the first draft quantity is now two. The helper reports "absent or expected quantity no longer matches" and rolls back its owned transaction.

7. A separate injected Python exception after the UPDATE but before COMMIT causes rollback. The fresh test workspace returns to the earlier draft quantity three rather than retaining the pending two.
8. Another test changes quantity three to four and back to three before running the expected-three update. The update succeeds. This deliberately demonstrates the check's inability to detect intervening history that restores the same value.

12.6.4 PLAIN — what is really happening inside

1. The driver binds values to the prepared statement's placeholders. Their content is treated as data at those positions rather than reparsed as additional SQL structure. [S86]
2. The guarded helper validates its identifiers and numeric inputs before starting. It rejects Boolean quantities, out-of-range values, a disabled foreign-key setting and a connection already inside a caller-owned transaction.
3. It starts `BEGIN IMMEDIATE`, then executes a single UPDATE whose predicate includes the draft identifier, line number and expected quantity. The same statement both tests the expected field and proposes the new value.
4. If the affected-row count is not exactly one, it does not claim a successful edit. It raises an error and rolls back the transaction it owns.
5. The message deliberately combines two possibilities: the line may be absent, or its quantity may differ. The failed predicate alone does not establish which explanation is true.
6. If the statement succeeds and no error interrupts the operation, the helper commits. If an exception occurs while its transaction is active, it rolls back and re-raises the failure rather than silently returning success.
7. This is not a full version protocol. It checks one expected field, not every field, an authenticated actor or the entire sequence of earlier edits.
8. It also performs no external effects. No email, payment, shipment or website update is coordinated by this transaction, so it cannot demonstrate exactly-once completion of any of those activities.

12.6.5 TECHNICAL — the engineer's version

1. Python's SQLite adapter supports bound parameters for values; placeholder conventions vary between database drivers. Do not copy `?` blindly into every driver's API or interpolate values manually to imitate another parameter style. [S86]
2. The lookup uses exact identifier equality and a bounded input length. Binding prevents syntax injection at the value position, but does not impose business authorization, normalize identifiers or automatically make wildcard-pattern queries literal.
3. Dynamic structure is allowlisted. `SORT_QUERIES` contains complete fixed statements for identity, quantity and price. The externally supplied choice must match one of those keys, and no untrusted identifier or direction text is concatenated into the SQL.
4. The core guarded mutation is:

```
UPDATE sql_draft_lines
SET quantity = ?
WHERE draft_id = ?
  AND line_no = ?
  AND quantity = ?;
```

5. The binding order is new quantity, draft identifier, line number, expected quantity. Mixing that order can produce a syntactically valid statement with incorrect semantics, so the code and tests keep it explicit.
6. `BEGIN IMMEDIATE` selects a SQLite-specific transaction start with write intent. The earlier local writer-contention exercise and SQLite documentation explain the single-writer behaviour; this new memory-only helper is not a benchmark or proof of multi-host concurrency. [S54]
7. The expected-field predicate is a narrow compare-and-change condition. It cannot detect the ABA pattern: quantity 3, then 4, then 3. A comprehensive optimistic concurrency design needs an appropriately maintained version or other state predicate and a defined conflict policy; later chapters develop those choices.
8. The helper's `rowcount == 1` check verifies the intended result cardinality for this statement and driver. It is not an authorization check, a complete immutable event record or proof that a non-trivial business transition occurred.
9. The connection setup used by the retained labs uses explicit SQL transaction boundaries. Python's `autocommit`, legacy transaction-control behaviour and `isolation_level` must be understood for the exact configuration; this book records the tested runtime rather than assuming all defaults stay unchanged. [S86]
10. Failure evidence distinguishes an injected application exception, a constraint error and a stale-or-absent predicate. The code does not implement network retry deduplication, full conflict resolution, production logging, finalisation or durable crash testing.

12.6.6 WORDS — remember these

1. **Bound parameter:** a value supplied separately from SQL structure — driver-mediated binding to a placeholder without interpreting the supplied value as new statement syntax.
2. **Placeholder:** a marked position for a bound value — syntax such as `?` in the demonstrated Python/SQLite interface.
3. **Allowlisted query choice:** selecting among explicitly permitted statement structures — a validated mapping from a caller's option to trusted fixed SQL.
4. **Guarded update:** a mutation conditional on identity and expected state — a write whose predicate tests the specific prior condition required by the operation.
5. **ABA pattern:** a value changes away and later returns to its earlier value — an intervening history that a comparison of only the final value cannot detect.
6. **Optimistic concurrency check:** accepting a change only when relevant expected state still matches — a conflict-detection technique whose adequacy depends on the chosen predicate and version-maintenance policy.

12.97 Practice and worked answers

1. **Read the question.** Write a query showing the canonical notebook line identities and quantities in identity order. How many lines and units does it represent?
2. **Do the arithmetic.** Select canonical lines whose calculated amount is at least 10,000 paise. State the expected result before running the query.
3. **Find missing links.** Write the predicate for orders without a linked customer profile. Explain why `customer_id = NULL` is not an equivalent spelling.

4. **Resolve a tie.** Return the first two line identities ordered by descending agreed unit price. Include enough ordering terms to make the canonical tie explicit.
5. **Separate missing from one.** Write a CASE expression for nullable draft quantity with labels "quantity absent", "single-unit proposal" and "multi-unit proposal". State the valid-domain assumption behind the last branch.
6. **Inspect mutation scope.** In a fresh draft workspace, predict how many rows `UPDATE sql_draft_lines SET quantity = 9` changes. Roll the demonstration back. Which agreed totals should remain unchanged?
7. **Check the guard honestly.** Quantity was three, changed to four, then changed back to three. Will the helper's expected-three update detect that intervening sequence? What stronger claim must not be made?
8. **Explain parameters.** Why is binding an order identifier different from concatenating it into the statement? Does parameter binding decide whether a caller may view that order?

Worked answers

1. Use the following query. It returns O-1042 line 1 with quantity two and O-1043 line 1 with quantity one: two lines representing three notebook units.

```
SELECT order_id, line_no, quantity
FROM order_lines
WHERE product_id = 'P-NOTE'
ORDER BY order_id, line_no;
```

2. The predicate is `quantity * unit_price_minor >= 10000`. Only O-1042 line 1 qualifies, with 15,100 paise. Keep the calculated amount's grain as one line; do not substitute the whole order's total.
3. Use `customer_id IS NULL`. It selects O-1042. Equality with NULL yields an unknown comparison rather than a known missingness test, so that wrong predicate selects neither canonical order.
4. Use `ORDER BY unit_price_minor DESC, order_id, line_no LIMIT 2`. Both notebook unit prices tie at 7,550, and the identity terms order O-1042 line 1 before O-1043 line 1. This ordering does not claim which sale happened first.
5. A suitable expression is shown below. The final ELSE is meaningful because the draft's supplied quantities are constrained to positive whole numbers; without that domain, zero or negative values would need their own treatment.

```
CASE WHEN quantity IS NULL THEN 'quantity absent'
     WHEN quantity = 1 THEN 'single-unit proposal'
     ELSE 'multi-unit proposal'
END AS quantity_label
```

6. Both original draft rows match the unqualified update, temporarily becoming nine and nine. Rollback restores three and one. O-1042's agreed total remains 17,100 and O-1043's remains 11,550 because these operations concern a separate draft table in a new practice database.
7. The expected-three update succeeds when it sees three again. It has no field recording the intervening four. Do not claim it detects every intervening modification or implements a complete version protocol. The lab includes this counterexample deliberately.
8. A bound value stays at its data position rather than becoming new SQL syntax. A quote or command-looking fragment in the identifier is compared as part of that literal value. Authorization is a separate decision; binding alone does not grant or restrict the caller's right to see the matched record.

12.98 Common wrong ideas

1. **Wrong:** SELECT automatically means "give me the original table exactly". **Better:** the select list and other clauses define a result that can omit identity or calculate new values.
2. **Wrong:** quantity two means return two rows. **Better:** the predicate tests each row's quantity; result count is a separate question.
3. **Wrong:** a missing value equals zero or empty text. **Better:** NULL has its own expression semantics and should not be replaced without an explicit meaning.
4. **Wrong:** NOT makes every unknown comparison become true. **Better:** negating unknown still leaves it unknown.
5. **Wrong:** the order seen yesterday is the query's guaranteed order. **Better:** request ORDER BY and resolve relevant ties.
6. **Wrong:** LIMIT two proves the query only examined two rows. **Better:** it limits the result, not necessarily the work needed to find it.
7. **Wrong:** an alias adds a stored column. **Better:** it names a result expression unless a separate schema change stores something.
8. **Wrong:** COALESCE recovers missing facts. **Better:** it chooses a non-null alternative, which must be labelled according to its actual meaning.
9. **Wrong:** a valid UPDATE predicate is necessarily the intended target. **Better:** scope errors can produce structurally valid but inappropriate changes.
10. **Wrong:** transaction rollback undoes every external consequence. **Better:** it concerns the database transaction's scope; this lab sends no external effects.
11. **Wrong:** bound parameters can stand for arbitrary SQL grammar. **Better:** bind values and select trusted statement structure separately.
12. **Wrong:** matching an expected quantity proves no intervening edit occurred. **Better:** the ABA counterexample shows the limit of comparing one field's current value.

12.99 Chapter summary in 20 lines

1. SQL describes database operations; SELECT starts with a precise result question.
2. Name the source and output columns, then state the result's grain and units.
3. Omitting a key can make distinct source rows look identical in the output.
4. WHERE retains rows for which its predicate is true.
5. Missing values require explicit reasoning rather than substitution with zero.
6. IS NULL tests absence; ordinary equality with NULL does not.
7. Parentheses make mixed AND and OR conditions match the intended question.
8. ORDER BY is the declared result order, not a hint about yesterday's storage layout.
9. Add a suitable tie-breaker when equal sort values must have a stable sequence.
10. LIMIT controls returned row count rather than guaranteeing a fixed execution cost.
11. Expressions can calculate per-line amounts without changing the source records.
12. Aliases name output values rather than creating stored columns.
13. CASE must cover the actual domain, including missingness where it is permitted.
14. COALESCE can supply a display label but cannot recover absent truth.
15. Exact integer paise and deliberate formatting remain separate responsibilities.
16. INSERT, UPDATE and DELETE need explicit target and transaction boundaries.

17. The editing exercises use separate disposable drafts and preserve agreed orders.
18. Bind outside values as parameters and keep SQL structure trusted.
19. The guarded update checks one expected field and cannot detect every intervening history.
20. State what the query or mutation established, and keep authorization, finalisation and production guarantees separate.

13

CHAPTER

Joins and Normalisation: Putting Facts in the Right Place

13.0 What this chapter gives you

1. You can now create tables and ask questions of one table. This chapter explains how to divide information between tables without losing its meaning, and how to bring it together without inventing extra sales.
2. You will distinguish useful repetition from avoidable duplication, identify the fact a key determines, read inner and outer joins, and recognise a report whose rows have quietly changed meaning.
3. Mira's Corner remains our fictional shop. Its four agreed order lines still contain six units and total 28,650 paise. A change to a catalogue, a tag or a customer's description does not rewrite those agreements.
4. Normalisation is a way to reason about the placement of facts. It is not a rule that every database must contain the largest possible number of tables. Our goal is a model that preserves meaning and a query whose answer matches its question.

13.1 Why facts repeat

13.1.1 PLAIN — in simple words

1. Seeing the same value twice does not automatically reveal a mistake. Two people may buy the same product. Two order lines may legitimately have the same price. A product identifier must appear in several places if those records refer to that product.
2. The more useful question is: are these separate facts, or several copies of one fact that should change together? Today's product description and the description printed on an old agreement may look identical while carrying different responsibilities.
3. Suppose Mira corrects a spelling mistake in the current product catalogue. Updating one catalogue record is straightforward. Updating a thousand copied descriptions is harder: some copies may be missed, and some may be historical descriptions that should not change at all.
4. Decide which meaning you need before trying to remove repetition. Less repetition is not an improvement when it destroys the evidence of what was agreed.

13.1.2 PLAIN — a picture in your head

1. Imagine a product card on a shelf and a folder for each completed order. The card says what the shop offers now. An order folder says what a particular customer agreed to buy at that time.

2. Several folders may point to the same product card. That repeated reference is useful. The price inside a signed folder is a different fact from the current price written on the card.
3. **Where the comparison breaks:** a database does not know that a folder is signed or that a card is current. Those distinctions require explicit fields, keys, rules and controlled write paths. A table name such as `agreed_orders` is documentation, not immutability enforcement.

13.1.3 PLAIN — a worked example

Order	Line	Product	Quantity	Agreed unit price, paise
O-1042	1	P-NOTE	2	7,550
O-1042	2	P-PEN	1	2,000
O-1043	1	P-NOTE	1	7,550
O-1043	2	P-PEN	2	2,000

1. These are the established four lines. Their amount is `2 × 7,550 + 2,000 + 7,550 + 2 × 2,000 = 28,650` paise.
2. The later catalogue illustration changes the notebook's current price to 8,250. Replacing agreed notebook prices with this current price would add `3 × 700 = 2,100`, producing 30,750. That is a hypothetical current-price calculation, not a corrected historical total.
3. It is appropriate to keep one current catalogue price per product in this simple model. It is also appropriate to retain the agreed price on each line. The two fields answer different questions.

13.1.4 PLAIN — what is really happening inside

1. Repeated copies of one mutable fact create several opportunities for disagreement. An update can change only some copies. An insertion can require irrelevant information merely to create the first copy. A deletion can remove the last copy of a fact we still need.
2. Separating a product's current details from its order lines removes those particular dependencies. The product can exist before its first sale, and deleting an isolated teaching order need not delete the catalogue entry.
3. This separation introduces a new responsibility: references must point to an existing product when the chosen model requires that. The foreign key from Chapter 11 addresses this structural condition; it does not establish that the human selected the right product.
4. Historical snapshots need an explicit policy. A preserved description may be deliberately redundant relative to the current catalogue, yet necessary for an old document. Record why it is copied and when, if ever, it may be corrected.

13.1.5 TECHNICAL — the engineer's version

1. **Update, insertion and deletion anomalies** motivate decomposition: represent facts in relations whose keys match the dependencies of those facts. Normalisation reasons about allowed states and functional dependencies, not just the strings visible in today's sample. [S88]
2. A product reference on a line is not a redundant copy of every product attribute. It records a relationship. Conversely, an agreed line price is not functionally determined by the product identifier alone when different agreements can use different prices.
3. Separate authoritative write models from derived read models. A report or materialised projection may intentionally repeat descriptions for a workload. It then needs a refresh, correction and provenance policy rather than the unsupported label "single source of truth."

4. This chapter's schema reasoning does not add an immutable-agreement mechanism to the earlier lab. The intentionally demonstrated missing protection remains missing until a separately specified implementation enforces it.

13.1.6 WORDS — remember these

1. **Redundancy:** several representations of a fact — duplication whose consistency depends on maintaining a specified relationship between representations. **Update anomaly:** one correction leaves conflicting copies — a state inconsistency caused by storing a dependent fact in multiple independently editable rows. **Historical snapshot:** a saved description of an earlier state — data retained under an explicit capture time, purpose and correction policy.

13.2 Joining by meaning

13.2.1 PLAIN — in simple words

1. A join combines records that satisfy a condition. The condition must represent the relationship you actually mean. "These two cells happen to contain 17" is not enough.
2. If receipt numbers restart at each branch, receipt 17 at branch BR-A and receipt 17 at branch BR-B are different receipts. Matching only the number can attach one branch's details to the other branch's receipt.
3. A join can return several rows for one input row. That is often correct: one order has several lines. Trouble begins when a later calculation assumes those output rows still represent whole orders exactly once.
4. Write the intended meaning of one output row beside a query. This short sentence often catches an error earlier than a large, plausible-looking total.

13.2.2 PLAIN — a picture in your head

1. Think of matching luggage tags at two airport desks. A sequence number may only be unique within one flight. You need the flight identifier and bag number together.
2. Matching only the bag number creates several possible pairings. The machinery has not confused the bags; the instruction supplied to it was incomplete.
3. **Where the comparison breaks:** SQL does not exercise human judgement when several matches appear. It produces the rows required by the written condition. Choosing one arbitrary match with a limit can hide the defect rather than repair the relationship.

13.2.3 PLAIN — a worked example

1. Take two invented receipt headers: `(BR-A, 17)` and `(BR-B, 17)`. Take two detail records with exactly the same respective keys. This is a separate key-scope exercise, not another sale in the canonical four-line dataset.
2. Joining on `receipt_no` alone allows four pairs: A-to-A, A-to-B, B-to-A and B-to-B. Joining on both `branch_id` and `receipt_no` allows only the two intended pairs.

```
SELECT h.branch_id, h.receipt_no, d.description
FROM receipt_headers AS h
JOIN receipt_details AS d
  ON d.branch_id = h.branch_id
 AND d.receipt_no = h.receipt_no;
```

3. These table names specify the isolated exercise, not a migration to the existing lab. Before running it, create the two small practice tables with a composite header key and a matching foreign key.
4. More generally, if one key has two left records and three right records, an equality join can produce six matching pairs for that key. Two times three is the consequence of the relationship expressed, not a database arithmetic bug.

13.2.4 PLAIN — what is really happening inside

1. Logically, a join considers pairs and keeps those for which the condition is true. An engine can use indexes or other strategies to avoid physically checking every pair; those strategies must preserve the specified result.
2. Key constraints supply useful facts about possible matches. A join to a genuinely unique parent key has at most one matching parent per child. Joining to a description that is not unique supplies no such assurance.
3. A missing value introduces another boundary. Ordinary equality involving SQL NULL is not true, so it does not make a successful match. Treating every missing customer as one shared fake identifier would invent a relationship the input did not establish.
4. A text comparison also uses a comparison rule. Case handling and collation belong to the contract. Two spellings that appear similar to a human are not automatically the same identifier, and a permissive matching rule may merge distinct records.

13.2.5 TECHNICAL — the engineer's version

1. An equijoin uses equality predicates. Composite relationships need the complete tuple of joining attributes. A declared foreign key and a unique referenced key help constrain cardinality, but nullable child keys and extra filters still affect the result. [S58] [S59]
2. SQL normally preserves duplicate result rows unless an operation explicitly removes them. The same projected values can come from different underlying matches. `DISTINCT` removes identical projected rows, not an incorrectly specified relationship. [S57]
3. Prefer explicit `ON` conditions, or an intentional `USING` list, over an unreviewed `NATURAL JOIN`. A natural join derives its matching columns from shared names; a later same-named column can alter behaviour without an obvious change to the query. [S58]
4. A correct join is not an authorisation check. Matching tenant identifiers in a query is useful only within a wider policy that establishes which tenant the caller may access and prevents alternative unscoped paths. Chapter 47 develops that separate responsibility.

13.2.6 WORDS — remember these

1. **Join predicate:** the rule for pairing records — a Boolean condition controlling which row combinations contribute to a joined relation. **Composite key:** several fields identify one thing together — an identifying attribute set whose combined values are unique in its declared scope. **Cardinality:** how many records or matches there are — a count or relationship multiplicity whose context must be stated.

13.3 Inner and outer joins

13.3.1 PLAIN — in simple words

1. An inner join returns successful matches. A left outer join also keeps each left record that has no successful match, filling the missing right-side fields with NULL.

2. These operations answer different questions. “Orders with a recorded customer” excludes anonymous orders. “Every order, with customer details where available” must keep them.
3. Keeping an unmatched order does not create a customer. The empty right-hand fields say that this join found no matching record. They do not explain whether the order was intentionally anonymous, its identifier was missing, or a relationship was filtered out.
4. A condition applied after a left join can remove the unmatched rows again. The word `LEFT` alone does not guarantee that the final report includes every left record.

13.3.2 PLAIN — a picture in your head

1. Imagine an attendance sheet and a box of submitted assignments. An inner join lists pupils for whom a matching assignment was found. A left join starts with every pupil and attaches an assignment where one exists.
2. If someone subsequently discards every row whose assignment mark is not above 50, pupils without assignments disappear as well. The original complete attendance sheet does not prevent the later filter.
3. **Where the comparison breaks:** a NULL mark might mean either an unmatched assignment or an existing assignment whose mark is not yet recorded. Inspect a non-null assignment key to distinguish these cases; the mark alone is insufficient.

13.3.3 PLAIN — a worked example

1. In our established annotation exercise, O-1042 has no customer identifier and O-1043 points to C-001. C-001’s display name is “Study customer.” These annotations do not change the two order totals.

```
SELECT o.order_id, c.customer_id, c.display_name
FROM orders AS o
LEFT JOIN customers AS c ON c.customer_id = o.customer_id
ORDER BY o.order_id;
```

2. The left join keeps both orders. An inner join on the same relationship keeps only O-1043. These are counts of order rows, not counts of lines or physical units.
3. Adding `WHERE c.display_name = 'Study customer'` after the left join also retains only O-1043. The NULL comparison for the unmatched order is not true.
4. Putting that condition inside `ON` instead asks for every order and, where available, a customer matching that name. O-1042 survives with empty customer fields. O-1043 still matches. The difference becomes even clearer with a separately labelled customer whose name does not pass the filter. [S58]

13.3.4 PLAIN — what is really happening inside

1. Conceptually, the join first forms matches according to `ON`. For a left join, a left row with no match contributes one null-extended output row. A later `WHERE` condition decides which of those resulting rows remain.
2. An existing right row can itself contain nullable attributes. Therefore, do not count “customers found” by checking an optional nickname. Use the customer’s non-null key when the model guarantees one.
3. `COUNT(*)` counts the rows produced. `COUNT(c.customer_id)` counts rows with a non-null customer identifier. In this two-order example the results are two and one respectively. Neither expression

counts distinct real people unless the intended grouping and identity policy establish that meaning. [S19]

4. To find missing relationships, a left join followed by `WHERE c.customer_id IS NULL` can be useful when this is the right-side non-null key. A `NOT EXISTS` query can express the same absence question more directly, without projecting right-hand fields.

13.3.5 TECHNICAL — the engineer's version

1. Predicate placement matters around outer joins because null extension and filtering do not generally commute. Moving a predicate between `ON` and `WHERE` requires a semantic argument, not a formatting preference. [S58]
2. For a left row with `m` matching right rows, the left join contributes `m` rows when `m > 0` and one null-extended row when `m = 0`. This still permits multiplication when the right side is one-to-many.
3. Do not use outer joining as silent data repair. A nullable annotation and a required reference with a missing parent are different conditions. Enforced constraints, import diagnostics and query output serve different purposes.
4. Query results are observations under a database's visibility rules. Combining independently fetched customer and order lists in application memory may not reproduce the same snapshot as one query. Chapters 21–25 explain the conditions behind that distinction.

13.3.6 WORDS — remember these

1. **Inner join:** keep successful pairings — a join that contributes only row combinations satisfying the join condition. **Left outer join:** keep every left record, even without a match — a join adding a null-extended row for each unmatched left input row. **Null extension:** empty right-hand fields for an unmatched record — synthetic NULL-valued attributes introduced by an outer join, not a stored right-hand record.

13.4 Functional dependencies

13.4.1 PLAIN — in simple words

1. Some fields determine other fields under a rule. In this model, a product identifier selects one current product record. A complete order-line key selects one agreed line, including its quantity and agreed price.
2. "Determines" means that two legal records with the same determining fields cannot disagree on the dependent fields. It does not mean the first field physically causes the second one.
3. A pattern in four sample rows is not enough to establish such a rule. Our two notebook lines happen to have the same price. A future promotion could legitimately create a different agreed price for the same product.
4. Write dependencies from the meaning of the work, then use examples to challenge them. Do not infer permanent business rules solely from a small export.

13.4.2 PLAIN — a picture in your head

1. Think of a numbered locker assigned to one student during one school term. Within that term, the locker number determines the assigned student under the school's one-assignment rule.
2. Across several terms, the locker number alone no longer determines the student. The term becomes part of the identifying information.

3. **Where the comparison breaks:** the arrow in a dependency is not an arrow of time or cause. A student may have several lockers, or the school may allow shared lockers. The dependency follows the stated policy, not the object called a locker.

13.4.3 PLAIN — a worked example

1. Use a deliberately simplified combined relation with attributes `order_id, line_no, product_id, current_description, quantity, agreed_price`.
2. Its agreed line rule gives `(order_id, line_no) → product_id, quantity, agreed_price`. The current catalogue rule gives `product_id → current_description`.
3. Starting with an order-line key, we can obtain its product identifier, then that product's current description. This chain explains why copying the current description into every line creates repeated dependent information.
4. Split the relation into `OrderLine(order_id, line_no, product_id, quantity, agreed_price)` and `Product(product_id, current_description)`. Provided each product identifier has exactly one permitted current description and references are represented consistently, joining on `product_id` reconstructs the intended combined rows.
5. Do not move `agreed_price` into the current product relation. The proposed dependency `product_id → agreed_price` is not a rule of our agreements. Equal numbers in the current fixture cannot make it one.

13.4.4 PLAIN — what is really happening inside

1. A dependency lets us calculate an attribute closure: start with known attributes, repeatedly apply dependencies whose left sides are known, and add the attributes they determine.
2. When the closure contains every attribute of the relation, the starting set is a superkey. A candidate key is a superkey with no unnecessary attribute under the stated dependencies.
3. Decomposition must preserve information. Splitting a table and joining the pieces can produce combinations that were never in the original unless the split has the right conditions.
4. Consider three invented assignments: `(Mira, blue, morning)`, `(Mira, red, evening)` and `(Dev, blue, evening)`. Splitting these into person-colour and colour-shift pairs, then joining only on colour, invents `(Mira, blue, evening)` and `(Dev, blue, morning)`. The pieces lost which person-colour pair belonged to which shift. This is a separate relational exercise, not shop staffing history.

13.4.5 TECHNICAL — the engineer's version

1. A functional dependency `X → Y` holds on a relation schema under its intended constraints when every legal relation instance has equal Y values for any two tuples equal on X. Sample agreement can falsify neither all counterexamples nor future policy changes. [S88]
2. For a binary decomposition of R into R1 and R2, a standard sufficient-and-necessary lossless condition under functional dependencies is that the shared attributes functionally determine R1 or R2 under the implied dependencies. This is a schema theorem, distinct from an accidental successful join on one sample.
3. **Losslessness** concerns reconstructing relation instances without spurious tuples. **Dependency preservation** concerns checking the original dependencies through the decomposed relations without needing a join. These are different properties; preserving one does not automatically preserve the other. [S88]

4. Classical relational dependency reasoning assumes well-defined attribute domains and relation semantics. SQL NULL, duplicate rows, collation, incomplete constraints and temporal rules require additional care when translating a theoretical decomposition into a concrete database.

13.4.6 WORDS — remember these

1. **Functional dependency:** one set of fields fixes another under a rule — X → Y means equality on X implies equality on Y in every legal relation instance. **Attribute closure:** everything the starting fields let you determine — the set of attributes implied by repeatedly applying the specified functional dependencies. **Lossless decomposition:** splitting without inventing or losing combinations on reconstruction — decomposition whose natural join recovers every legal original relation instance under the stated dependencies.

13.5 Normal forms and trade-offs

13.5.1 PLAIN — in simple words

1. A normal form is a named set of conditions on how a relation stores dependencies. The names help reviewers ask precise questions instead of saying that a table "looks tidy."
2. Start by deciding what one field means and what identifies a row. Then ask whether some other fact depends on only part of that identifier, or on another non-identifying fact copied into the row.
3. Splitting such facts can make corrections safer. It also means that some questions need joins. Whether to maintain additional read-friendly copies is a separate workload decision.
4. More tables do not prove a better design. A decomposition that loses information, makes an important rule unenforceable without extra coordination, or confuses historical and current facts can be worse despite impressive terminology.

13.5.2 PLAIN — a picture in your head

1. Think of organising a workshop. One drawer holds the current specification for each component; a separate folder holds each job's agreed requirements. You stop copying the supplier's current telephone number onto every screw's usage record.
2. You may still pin a printed job summary beside a machine because it is convenient to read. The pinned summary is a controlled copy, not a new authority for the underlying job.
3. **Where the comparison breaks:** normal forms are mathematical conditions on relations and dependencies, not a general rule that physical objects must be stored in separate drawers. A neat filing system can still contain the wrong dependencies.

13.5.3 PLAIN — a worked example

1. Consider an isolated order-header relation `(order_id, customer_id, current_customer_name)`. Assume each order has one required customer for this exercise, and each customer identifier has one current name. These assumptions differ from the canonical model's optional customer annotation and are stated only to illustrate the dependency.
2. `order_id` determines the customer identifier; `customer_id` determines the current name. Copying the current name into every order repeats a fact whose change is about the customer, not each order.
3. A decomposition into `OrderHeader(order_id, customer_id)` and `Customer(customer_id, current_name)` isolates that current fact. A foreign key can enforce that each header references an existing customer under this exercise's required relationship.

4. Now change the meaning: `name_printed_on_agreement` is what was printed on that particular order. This field can belong to the agreement record because later customer renaming must not silently rewrite the printed document. A superficially similar layout now has a different dependency contract.
5. Normalisation follows the contract. It cannot decide which of these two fields the business intended.

13.5.4 PLAIN — what is really happening inside

1. Review every proposed split with three questions: can the original facts be reconstructed; can required dependencies still be enforced; and what operations must now read or write several relations together?
2. A report can join normalised tables as needed. A stored report, cache or materialised view can reduce repeated work, but then readers need to know the copy's freshness and how corrections reach it.
3. Do not combine two independent many-valued facts into every possible pair without a reason. If a product has two tags and three suppliers, a six-row tag-supplier cross-product repeats both relationships. Separate product-tag and product-supplier relations represent independence more directly.
4. But do not split a genuinely three-way fact. "Supplier S can supply product P under condition C" may be a specific permitted combination, not three independent pairwise facts. The spurious-tuple exercise in section 13.4 explains the danger.

13.5.5 TECHNICAL — the engineer's version

1. In the usual relational treatment, first normal form uses relation attributes with values from their declared domains, rather than repeating groups masquerading as columns. "Atomic" is relative to the domain and operations: it does not mean that text may never contain several characters or that SQL must forbid every structured type.
2. Second normal form removes partial dependencies of non-prime attributes on proper subsets of candidate keys. Third normal form permits a nontrivial dependency $X \rightarrow A$ only when X is a superkey or A is prime, meaning it belongs to some candidate key. Boyce–Codd normal form requires the determinant of every nontrivial functional dependency to be a superkey. [S88]
3. The common phrase "depends on the key, the whole key and nothing but the key" is a mnemonic, not a substitute for checking all candidate keys and dependencies. In particular, third normal form and BCNF are not identical.
4. Some BCNF decompositions do not preserve all original dependencies locally. A dependency-preserving third-normal-form design can be a deliberate choice, with trade-offs documented. Independent multivalued relationships motivate fourth normal form; more general join dependencies lead to fifth normal form. This book uses those ideas to ask about independent facts rather than claiming a full database-theory proof course.
5. Denormalisation is the deliberate maintenance of redundant representations for specified operations. It needs ownership, refresh rules and reconciliation. It does not mean "remove constraints until the benchmark improves."

13.5.6 WORDS — remember these

1. **Prime attribute:** a field belonging to at least one minimal key — an attribute contained in some candidate key, not necessarily the selected primary key. **Normal form:** a named condition for

placing relational facts — a dependency-based schema property such as 3NF or BCNF. **Denormalisation:** deliberately keeping an extra representation — controlled redundancy introduced for stated requirements with explicit maintenance obligations.

13.6 Avoiding accidental multiplication

13.6.1 PLAIN — in simple words

1. A correct relationship can still be the wrong input to a sum. Joining each line to its product tags produces line-tag pairs. A notebook line with two tags now appears twice.
2. Summing the line's amount over those pairs counts the notebook twice. The join is answering "which tags belong to each line's product?" while the sum pretends it is seeing each line only once.
3. Fix the question, not the symptom. To total order lines, use one contribution per line. To ask whether a qualifying tag exists, ask an existence question. To combine several different totals, calculate each at its intended level before joining them.
4. Removing duplicate amounts is not the same as removing duplicate line contributions. Two separate lines can legitimately have equal amounts.

13.6.2 PLAIN — a picture in your head

1. Imagine putting a photocopy of each receipt in every relevant product-category folder. A receipt mentioning two categories appears in two folders.
2. Adding every photocopy's amount overstates sales. Throwing away every receipt with a repeated amount also fails: two different customers may have spent the same amount.
3. **Where the comparison breaks:** a SQL join need not make physical photocopies. The duplication is in the logical result rows. The error persists even when the engine streams those rows without storing a copied table.

13.6.3 PLAIN — a worked example

1. The established tag example gives P-NOTE two tags, `paper` and `school`, and P-PEN one, `writing`. The notebook contributions total 22,650 paise; pen contributions total 6,000.
2. The line-tag join therefore yields `2 × 22,650 + 6,000 = 51,300` paise when line amounts are wrongly summed. The correct line total remains 28,650.

```
SELECT order_id, SUM(quantity * unit_price_minor) AS total_minor
FROM order_lines
GROUP BY order_id
ORDER BY order_id;
```

3. This query stays at line grain until it deliberately groups to order grain. Its results are O-1042: 17,100 and O-1043: 11,550.
4. To select lines whose product has a `paper` tag, use a correlated existence condition rather than adding a tag row to the output for every match. To combine order totals and payment totals in a later exercise, aggregate the lines and payments separately by order, then join those one-row-per-order results.
5. The separate O-REPEAT counterexample contains two genuine notebook lines of 7,550 each. Their sum is 15,100. `SUM(DISTINCT line_amount)` returns 7,550 and is therefore not a general repair for fan-out.

13.6.4 PLAIN — what is really happening inside

1. Track grain through the query: order line, then line-tag pair, then group. Once a line can contribute more than once, every aggregate that uses its values needs a reason why that multiplicity is intended.
2. A useful diagnostic query lists original keys and their contribution counts. A line key appearing twice is evidence to investigate. It is not automatically wrong: perhaps the metric intentionally counts category memberships rather than sales.
3. Test empty parents, several children, equal amounts from different children, repeated descriptions and optional relationships. One parent with one child in every table hides most multiplication errors.
4. Reconcile both totals and row identities. Two wrong calculations can have the same grand total if one omitted contribution happens to cancel one duplicated contribution. Agreement on one number is evidence, not a complete proof of matching populations.

13.6.5 TECHNICAL — the engineer's version

1. Join cardinality and aggregation grain are separate design choices. Joining two independent one-to-many child relations through their parent generally forms combinations of children before aggregation. Pre-aggregation can restore one row per parent on each side when those aggregates answer the intended questions. [S58] [S19]
2. `EXISTS` expresses a semi-join-like membership question: whether at least one qualifying row exists. It avoids introducing one output contribution per qualifying child. An optimiser may implement that question with different physical strategies while preserving its semantics.
3. A distinct aggregate operates on its argument values within a group. It does not deduplicate by an unmentioned business key. Preserve the contribution identity explicitly when deciding what is counted once. [S19] [S80]
4. The companion exercises verify result rows and arithmetic in disposable databases. They are not evidence that every report, migration or production workload has been reconciled. Chapter 19 extends these ideas to averages and windows; Chapter 46 examines the population definitions behind a metric.

13.6.6 WORDS — remember these

1. **Fan-out:** one record becomes several joined contributions — multiplication of rows through a one-to-many or many-to-many relationship. **Pre-aggregation:** calculate a total before joining it to other detail — reducing a relation to the required grouping grain before a later join. **Reconciliation:** compare defined representations of the same work — an evidence-based check of populations, keys, amounts and explained differences.

13.97 Practice and worked answers

1. **Question:** Two headers share receipt number 17 but belong to different branches. Each has two detail lines. What does a number-only join do? **Answer:** Each of two headers matches all four detail lines, producing eight header-detail pairs rather than four. The fix is the complete branch-and-receipt relationship, not a limit of four rows.
2. **Question:** Why can a left join followed by a right-side name filter remove anonymous orders? **Answer:** The unmatched row has NULL right-side values. Ordinary equality to the selected name

is not true. A WHERE clause retains only true conditions. Put a condition in ON only when the intended question is "keep all orders, attaching only qualifying customer details."

3. **Question:** Does a product identifier determine an agreed price because both notebook lines currently use 7,550? **Answer:** No. That is an observation about the fixture. A new separately identified agreement at a different price is permitted by the meaning of agreed price, so the proposed dependency is not a rule.
4. **Question:** A tag join inflates a sum. Why not apply DISTINCT to the price? **Answer:** The price is not the line's identity. Distinct real lines can have equal prices and equal amounts. Deduplicate contribution keys only under a justified duplicate policy, or avoid introducing the unnecessary multiplicity.
5. **Question:** In the combined relation of section 13.4, what does `(order_id, line_no)` determine? **Answer:** Product, quantity and agreed price directly under the line rule; current description through the product dependency. The closure contains all six attributes under these assumptions.
6. **Question:** Is a dependency-preserving split automatically lossless? **Answer:** No. Local rule checking and faithful reconstruction are separate properties. Inspect both. The person-colour-shift counterexample shows why retaining plausible pairs can still invent triples.
7. **Question:** The right-side customer key is non-null in every stored customer. Can `COUNT(c.customer_id)` count matched order rows after a left join? **Answer:** Yes, at the output grain in that query. It still is not a distinct-customer count when several orders refer to one customer.
8. **Question:** Should an old agreement's printed customer name always be replaced from today's customer record? **Answer:** Not without a policy that says this is the intended meaning. Current identity data and a historical document snapshot have different purposes. A correction may require a new version and retained explanation rather than an overwrite.

13.98 Common wrong ideas

1. **Wrong:** repeated values prove bad design. **Right:** repetition may express multiple valid events, references or intentionally preserved snapshots.
2. **Wrong:** equal column names prove a relationship. **Right:** meaning, scope and declared keys establish the relationship.
3. **Wrong:** a left join always preserves every left record in the final result. **Right:** later filters can remove the null-extended rows.
4. **Wrong:** normalisation deletes all historical copies. **Right:** history can be a separate fact with its own key and correction policy.
5. **Wrong:** a dependency observed in a sample is a permanent rule. **Right:** a functional dependency constrains every permitted instance.
6. **Wrong:** more normalised always means faster. **Right:** logical dependency properties and physical workload performance are different questions.
7. **Wrong:** DISTINCT repairs every duplicate-looking report. **Right:** it removes equal projected values, which may represent separate valid contributions.
8. **Wrong:** matching grand totals proves matching records. **Right:** opposite errors can cancel; inspect identities and populations as well.

13.99 Chapter summary in 20 lines

1. Repeated values can represent different facts or several copies of one fact.

2. Current catalogue details and historical agreement details have different meanings.
3. Preserve the agreed 28,650-paise total when the current catalogue changes.
4. A join pairs records under a specified condition, not human intuition.
5. Use the complete key, including the scope in which an identifier is unique.
6. One-to-many matches change the meaning of an output row.
7. Inner joins keep successful matches.
8. Left joins also introduce null-extended rows for unmatched left records.
9. A later filter can remove those unmatched rows.
10. Count a non-null key when the question is whether a right-side record matched.
11. Functional dependencies describe rules over every legal relation instance.
12. A small sample can suggest a dependency but cannot establish the whole policy.
13. Attribute closure traces what a set of fields determines.
14. Lossless decomposition avoids spurious combinations during reconstruction.
15. Dependency preservation concerns where the original rules can be checked.
16. Third normal form and BCNF impose different dependency conditions.
17. Deliberate read-friendly copies need maintenance and reconciliation rules.
18. A line-tag result is not a one-row-per-line input to a sales total.
19. Equal amounts can belong to distinct valid contributions.
20. State grain, population, keys and exclusions before trusting a report.

Companion Labs

The supplied practice package uses Python's standard library and fresh synthetic data. It does not connect to your website, payment provider, production database or cloud account. Read this guide before running code.

Start in an isolated folder

1. Extract `practice-code.zip` into a new folder. Keep its Python files together because the later suites import the earlier teaching modules.
2. Use Python 3.11 or newer with SQLite 3.37.0 or newer. This minimum covers the syntax and STRICT tables used here; it is not a recommendation to operate an old runtime in production. The accompanying `test-results.json` identifies the actual authoring environment.
3. Open a terminal in that extracted folder and run the test command below. It discovers the six test modules. Some cases deliberately confirm a missing safeguard or a failing design; their success means the stated counterexample was observed.
4. The examples include in-memory databases and one earlier bounded temporary-file locking demonstration. Temporary resources are cleaned up by their tests. No personal database or user records are required.

```
python3 -m unittest discover -s . -p 'test_*.py' -v
python3 advanced_lab.py
python3 capstone_lab.py
```

The last two commands print a small local performance observation and a capstone transaction/replay/restore trace. Timing results vary by machine and workload. They do not promise that a particular index always wins.

What is implemented

Module	Chapters and exercises	Boundary
`foundations_lab.py`	1–3: canonical orders, number/text/time representations and NULL behavior	Pure functions and a fresh SQLite fixture.
`history_lab.py`	4–6: measurements, scoped identity, duplicates and reconstructed history	Small explicit state machines; not a production event store.
`data_bridge_lab.py`	7–9 and 13: CSV/JSON import, the shop tables, relationships, totals, backup and a lock example	Bounded input profile, synthetic fixtures and local SQLite.
`schema_sql_lab.py`	10–12: schema rules, SQL, parameters, rollback and deliberately missing protections	Product-specific SQLite observations; PostgreSQL is documented, not executed.

Module	Chapters and exercises	Boundary
`advanced_lab.py`	14–46: query measurements, a leaf split, hashes, wait graphs, revisions, LRU, recovery prefixes, tombstones, replicas, fencing, majorities, protocol states, caches, windows, manifests, encodings, search, vectors and uncertainty	Local models and isolated observations, not full storage engines or network protocols.
`capstone_lab.py`	47–52: permission fixtures, tenant-scoped orders, outbox/inbox tasks, retries, local restore, retention status, resumable backfill and capacity/cost arithmetic	Trusted Principal objects, in-memory SQLite and bounded functions; not authentication, a public API or a deployed service.

The advanced models

The B-tree exercise implements ordered separator selection and a single overflowing-leaf split. It does not implement a complete balanced persistent B-tree. The log example replays committed toy transactions from a supplied durable prefix; it does not parse an engine's actual WAL or simulate torn storage sectors. The compaction example can discard tombstones only under its stated complete-history, latest-only assumptions.

The replica model tracks contiguous applied positions. The fencing model has the resource check a monotonically increasing authority token. The consensus exercises enumerate majorities, compare last-term/last-index pairs and test the current-term direct-commit condition. These are selected rules, not a full Raft implementation or proof of progress under a real network.

The transaction participant and compensation classes show explicit state transitions and repeated-decision handling. The cache and manifest classes model generation/version checks in one process; they do not implement shared-memory atomicity or a distributed metadata service.

The stream examples distinguish fixed windows, sliding windows, session overlap and late corrections. The columnar exercise packs signed 64-bit integers and uses zlib to compare representations. It is not a Parquet writer or a disk-I/O benchmark. Text search uses a tiny tokenized corpus, with an additional SQLite FTS5 check when that feature is present. Vector examples calculate exact distances and filter by tenant before selecting results; they do not train an embedding model or implement a production approximate-nearest-neighbor index.

The Wilson interval, missing-status bounds and nearest-rank percentiles check arithmetic under explicit assumptions. They do not establish representative sampling, independence or causal identification in real business records.

The capstone boundary

The capstone accepts a canonicalized cart, reads current catalogue prices during acceptance, conditionally reduces stock and stores the agreed order, request identity and outbox event in one owned transaction. An identical scoped replay returns the committed result without repricing or reducing stock again. A conflicting payload using the same scoped request identity is rejected.

The consumer stores one local task and its inbox identity atomically. An injected lost acknowledgement occurs after that local consumer commit; redelivery does not create another task. This is not proof that an external courier, email provider or payment processor performs its work exactly once.

The local restore uses SQLite's backup API, enables foreign-key checking on the target, checks database integrity and references, compares logical records and continues a synthetic operation. It does not test power loss, cloud loss, independent failure domains or an off-site recovery objective.

The Principal objects are trusted fixtures constructed by the test. They are not login tokens. Holding the raw SQLite connection bypasses the helper authorization boundary; a test explicitly demonstrates this limit. The model does not enforce immutable agreements against an administrator or implement a complete checkout lifecycle.

Reading the test record

The release test record gives the exact test count, failures, errors, skips, runtime versions and hashes of the executable files. A skipped feature check is not a pass for that feature. Test names and assertions are the evidence; the number of tests is only a count.

Exercises elsewhere in the book can ask you to draw a schedule, inspect a plan, choose a workload or design a real restore procedure. Not every such task is a ready-made automated implementation. PostgreSQL deployments, actual network faults, device flush behavior, complete tenant security and live schema migrations remain tasks for an appropriately isolated environment with explicit authorization.

A–Z Glossary

Definitions are retained with their teaching context. Some familiar terms have several related meanings. Chapter and section identifiers are stable across editions.

A

ABA pattern

a value changes away and later returns to its earlier value — an intervening history that a comparison of only the final value cannot detect.

Read in context: 12.6.6

Agreed line price

the price attached to a particular historical agreement — a transaction-specific fact distinct from a product's current catalogue price.

Read in context: 9.6.6

Alias

a name attached to an output expression — a result-column label that does not itself alter the source table's schema.

Read in context: 12.4.6

Allowlisted query choice

selecting among explicitly permitted statement structures — a validated mapping from a caller's option to trusted fixed SQL.

Read in context: 12.6.6

Arithmetic bound

the largest or smallest result an operation may produce — a range derived from operand limits and the operation's cardinality.

Read in context: 10.4.6

Atomicity

the grouped changes take effect together or not at all — a transaction property at a specified resource boundary.

the selected changes take effect together — all-or-nothing transactional acceptance of the covered database work.

Read in context: 8.4.6

Attribute closure

everything the starting fields let you determine — the set of attributes implied by repeatedly applying the specified functional dependencies.

Read in context: 13.4.6

Authoritative counter

a stored total used to make decisions — state whose agreement with the underlying events or allocations needs its own maintained invariant.

Read in context: 11.4.6

Authoritative representation

the record a decision is defined to rely on — a source-of-truth role assigned by the architecture.

the selected source of meaning — the version whose value governs while multiple copies coexist.

Read in context: 8.6.6

B

Backup

retained material intended for restoration — a copy or representation with a defined consistency and recovery scope.

a copy retained for a defined recovery purpose — data and metadata acquired under a procedure that supports a stated restoration objective.

Read in context: 8.5.6

Binary format

a byte layout read under explicit rules — a representation not restricted to a plain-text grammar.

Read in context: 7.5.6

Bound parameter

a value supplied separately from query text — data passed through a driver's parameter interface rather than inserted into SQL by string concatenation.

a value supplied separately from SQL structure — driver-mediated binding to a placeholder without interpreting the supplied value as new statement syntax.

Read in context: 8.2.6, 12.6.6

Boundary test

a test at an acceptance limit — a selected case at, immediately below or immediately above a declared domain boundary.

Read in context: 11.1.6

Business domain

the values that make sense for a particular purpose — a semantic acceptance set including units, bounds, missingness and interpretation.

Read in context: 10.4.6

Busy outcome

the system cannot complete the requested access immediately — a contention result that requires an explicit wait, retry or failure policy.

Read in context: 8.1.6

Byte order

which end of a multi-byte number comes first — the endianness used to encode a numeric value.

Read in context: 7.5.6

C

Candidate

a record ready for the next acceptance stage — a validated, non-conflicting input that has not necessarily been published.

Read in context: 7.6.6

Candidate key

a possible way to identify each row uniquely — a minimal identifying attribute set under the model's constraints.

Read in context: 11.2.6

Capture boundary

the point at which this exercise obtains its input — the boundary after which one immutable byte string is used for parsing and diagnostics.

Read in context: 7.1.6

Cardinality

how many; here, the number of rows in a dataset or intermediate result.

how many related rows are permitted or required — the minimum and maximum counts on each side of a relationship.

how many records or matches there are — a count or relationship multiplicity whose context must be stated.

how many distinct labelled combinations exist — a property that can drive metric storage and query cost.

Read in context: 9.3.6, 13.2.6

CASE expression

an explicit choice among result alternatives — a SQL conditional expression selecting an output according to its stated conditions.

Read in context: 12.4.6

Catalogue price

the current listed price in the product model — a mutable reference value that must not silently replace historical agreement values.

Read in context: 9.6.6

CHECK constraint

a declared predicate on acceptable values — an engine-enforced condition whose NULL semantics must also be understood.

Read in context: 9.2.6

Check predicate

an expression used to reject prohibited row values — a condition evaluated under an engine's CHECK semantics.

Read in context: 11.1.6

Child row

a row carrying a reference — a record constrained to refer to an allowed parent under the declared semantics.

Read in context: 9.3.6

Client-server database

an engine operating as a separate service — an arrangement in which clients exchange requests and results with a database server.

Read in context: 8.3.6

COALESCE

choosing the first supplied value — an expression returning the first non-null argument, not a recovery of missing truth.

Read in context: 12.4.6

Coercion

automatic conversion; technically, a transformation between types that may preserve or change intended meaning.

converting a value into another representation or type — an engine or application behaviour that must be deliberate when it changes interpretation.

Read in context: 9.2.6

Column

a named role within each row — an attribute position with declared type and associated meaning.

Read in context: 9.2.6

Column-oriented format

a layout that groups values by column within its storage structure — an organisation suited to some selective and analytical reads, not a universal performance guarantee.

Read in context: 7.5.6

Commit

accepting the transaction; technically, the transaction-completion operation under the engine's documented configuration.

accepting the transaction's database changes — successful completion under the engine's stated persistence and visibility semantics.

accept a transaction's database changes — the completion operation whose acknowledgement has engine- and configuration-specific guarantees.

Read in context: 8.4.6

Commit-time failure

rejection when trying to complete a transaction — an error arising at the commit boundary rather than necessarily during an earlier statement.

Read in context: 11.5.6

Compatibility

the ability of specified participants to exchange usable data — a tested relationship between writer, reader and semantic contracts.

Read in context: 7.5.6

Composite key

an identifier made from several parts; technically, a key consisting of more than one attribute.

several fields identifying a row together; technically, a key consisting of more than one attribute.

an identity made from several values together — a unique column combination whose scope is not supplied by any one component alone.

several fields identify one thing together — an identifying attribute set whose combined values are unique in its declared scope.

Read in context: 9.4.6, 13.2.6

Composite reference

a link made from more than one value — a tuple of child columns identifying a parent within the corresponding scope.

Read in context: 11.3.6

Conflict group

records claiming one identity with incompatible content — a set for which this batch policy refuses to choose a winner.

Read in context: 7.6.6

Connection

a session through which an application interacts with an engine — a context carrying database access and potentially transactional state.

Read in context: 8.3.6

Connection pool

a managed set of reusable connections — a bounded resource-sharing mechanism, not unlimited server capacity.

Read in context: 8.3.6

Constraint

a rule enforced at a defined database boundary — a declared condition whose violation prevents a relevant operation from succeeding under the engine's semantics.

Read in context: 11.1.6

Constraint validation

checking records against a declared rule — an operation whose data scope and timing must be distinguished from future enforcement.

Read in context: 11.6.6

Content digest

a compact fingerprint of captured bytes — the result of applying a specified hash function to an exact byte sequence.

Read in context: 7.1.6

Contention

operations competing for a shared resource — overlapping demand that can require waiting, rejection or coordination.

Read in context: 8.6.6

Coordination boundary

the interval or operation within which competing work is controlled — the scope of a lock, version condition or transactional mechanism.

the operations that must be ordered or made indivisible together — the scope within which an invariant's read, decision and write are protected.

Read in context: 8.1.6, 11.4.6

Counterexample

a concrete case that disproves an overbroad statement — an input or sequence for which a claimed invariant or interpretation does not hold.

Read in context: 10.6.6

Cross-row invariant

a rule about several records together — a condition involving relationships, aggregates or coordinated state beyond one proposed row.

a rule about several records together — a correctness condition involving a set of rows rather than one row's fields.

Read in context: 11.4.6

CSV

a text representation of tabular records — comma-separated values with a specified dialect and quoting rules.

Read in context: 7.3.6

Current attribute

what a record says about the present state — a value whose intended temporal interpretation follows the entity's current representation.

Read in context: 10.3.6

Cursor

the driver's interface to a statement and its results — an object through which rows and relevant execution information are retrieved.

Read in context: 12.1.6

D

Data dictionary

the reference explaining what fields mean — maintained metadata describing grain, domains, units, source, lifecycle and interpretation.

Read in context: 10.5.6

Database engine

the software managing stored data operations — the component that executes supported queries, changes and integrity mechanisms.

Read in context: 8.2.6

DBMS

the record-management machinery; technically, the engine providing database operations and its documented guarantees.

the database-management software — facilities for defining, accessing and administering databases.

Read in context: 8.2.6

Declarative query

a request that describes the required result — a statement whose logical meaning does not prescribe one physical execution plan.

Read in context: 12.1.6

Decoding

turning bytes into characters — interpreting a byte sequence under a specified character encoding.

Read in context: 7.2.6

Deferred constraint

a rule whose check is postponed to a declared later boundary — an eligible constraint allowed to be temporarily unsatisfied within a transaction but required by successful final checking.

Read in context: 11.5.6

DELETE

removing selected stored rows — a SQL operation whose effects are constrained by referential rules and transaction boundaries.

Read in context: 12.5.6

Delimiter

the mark used to separate fields — a structural character interpreted according to the parser's current state.

Read in context: 7.3.6

Denormalisation

deliberately keeping an extra representation — controlled redundancy introduced for stated requirements with explicit maintenance obligations.

Read in context: 13.5.6

Deserialisation

rebuilding values from a stored representation — decoding and interpreting data into runtime objects under a specific format.

Read in context: 7.4.6

Design trade-off

a choice that improves some requirements at a cost to others — an explicit comparison of behaviour and responsibility under stated assumptions.

Read in context: 8.6.6

Destination capacity rule

an extra bound imposed by this storage adapter — a limitation distinct from parsing and the earlier domain contract.

Read in context: 9.2.6

Detection versus prevention

finding a violation versus stopping it from being committed — distinct responsibilities requiring different evidence.

Read in context: 11.4.6

Deterministic result order

a sequence fixed by the stated query and data assumptions — an ordering whose relevant ties are resolved under chosen comparison rules.

Read in context: 12.3.6

Dialect

the particular set of parsing conventions — settings governing delimiters, quotation marks, escaping and related behaviour.

Read in context: 7.3.6

DISTINCT

removing repeated output rows — a SQL operation over selected result values, not automatic repair of a data model.

Read in context: 9.1.6

Domain

the allowed kind of value; technically, the set of permitted values and associated meaning for an attribute.

the values and meaning permitted for a role — a business-level contract potentially narrower than a storage type.

Read in context: 9.2.6

Draft record

unfinished work that remains open to revision — a lifecycle-specific representation with explicit incompleteness rules and no implied finalisation.

Read in context: 10.3.6

Duplicate member

a name repeated within one object — an interoperability hazard rejected by this book's input profile.

Read in context: 7.4.6

Durability

surviving specified failures; technically, the persistence guarantee for accepted changes under stated assumptions.

the promised survival of committed work — persistence under defined failures, engine configuration and storage assumptions.

acknowledged work survives specified failures — persistence under a named configuration, recovery model and failure boundary.

Read in context: 8.4.6

E

Embedded database

an engine included in an application process — database functionality supplied through library calls rather than a separate server process.

Read in context: 8.3.6

Enforcement gap

a declared rule without sufficient protection — a condition the implementation can violate despite the intended policy.

Read in context: 10.6.6

Envelope

the wrapper around a payload — metadata and framing interpreted before the enclosed message.

Read in context: 7.5.6

Evidence gap

something the records cannot establish — missing information or verification needed to support a claim about the world.

Read in context: 10.6.6

Existence precheck

asking whether a record is present before attempting a write — a read that does not by itself prevent a concurrent conflicting insert.

Read in context: 11.2.6

Existence test

asking whether at least one match is present — a condition that need not reproduce every matching row in the output.

Read in context: 9.4.6

Extended error code

a machine-readable refinement of a failure category — an implementation-specific identifier that distinguishes more precise error causes.

Read in context: 11.6.6

F

Fan-out

one source row producing several matched result rows — multiplicity introduced by a one-to-many match in a query.

one record becomes several joined contributions — multiplication of rows through a one-to-many or many-to-many relationship.

how many next regions one guide can select — the number of child references in an internal tree node.

Read in context: 9.4.6, 13.6.6

File extension

the familiar suffix on a name — a naming convention that does not independently validate the represented format.

Read in context: 7.1.6

File-level error

a failure affecting interpretation of the input as a whole — an outcome that withholds publication when trustworthy complete parsing is unavailable.

Read in context: 7.6.6

Flattened export

related information copied into a single rectangular result — a representation that may repeat parent attributes at a finer output grain.

Read in context: 10.2.6

Foreign key

a checked reference to another stored key; technically, a constraint requiring referencing values to correspond to an eligible referenced key, subject to the database's rules.

a declared reference to another allowed key — a referential-integrity constraint linking child values to parent values.

a stored link required to match an eligible parent — a referential constraint on child values with defined missingness, actions and checking time.

Read in context: 9.3.6, 11.3.6

Framing

marking where messages or records begin and end — a boundary convention independent of the meaning of each payload.

make field boundaries unambiguous — an encoding convention that preserves where components begin, end and differ in type or length.

Read in context: 7.2.6

Functional dependency

one part determines another; technically, a constraint under which equal determining attributes imply equal dependent attributes.

one set of fields fixes another under a rule — $X \to Y$ means equality on X implies equality on Y in every legal relation instance.

Read in context: 13.4.6

G

Grain contract

a precise statement of what one row represents — documentation tying a record's unit, key, multiplicity and temporal meaning together.

Read in context: 10.2.6

Guarded update

a mutation conditional on identity and expected state — a write whose predicate tests the specific prior condition required by the operation.

Read in context: 12.6.6

H

Half-specified reference

only part of a multi-component link is supplied — an incomplete child tuple that needs an explicit acceptance policy.

Read in context: 11.3.6

Historical snapshot

a saved description of an earlier state — data retained under an explicit capture time, purpose and correction policy.

Read in context: 13.1.6

Historical snapshot attribute

a value copied to describe an earlier event or agreement — a stored attribute with event-specific temporal semantics, not automatic immutability.

Read in context: 10.3.6

I

Identity boundary

the scope inside which a label identifies something — the domain over which a key or composite key is interpreted and uniqueness is enforced.

Read in context: 10.2.6

Immediate constraint

a rule checked at its ordinary early boundary — an invariant enforced without the supported transaction-level postponement.

Read in context: 11.5.6

Import profile

the exact agreement for an exchange — a bounded specification combining encoding, syntax, fields and application rules.

Read in context: 7.3.6

Inclusive interval

a range containing its endpoints — the boundary interpretation used by BETWEEN in the illustrated SQL conditions.

Read in context: 12.2.6

Inner join

returning matching row combinations — a join that omits rows without a match in the joined relation.

keep successful pairings — a join that contributes only row combinations satisfying the join condition.

Read in context: 9.5.6, 13.3.6

INSERT

adding records — a SQL write operation that proposes new rows under the target table's active rules.

Read in context: 12.5.6

Integer division

division evaluated using integer operands and rules — an operation that can discard the fractional component rather than produce a decimal quotient.

Read in context: 12.4.6

Invariant

a rule that must keep holding; technically, a predicate required of accepted states or permitted transitions.

a rule that must remain true — a property the chosen model preserves across accepted operations.

a condition the system is meant to preserve — a stated property that every accepted state transition must maintain.

a condition that must keep holding — a predicate preserved across the operations within its specified scope.

a rule every accepted state must preserve — a condition maintained by constraints and operation protocols across permitted transitions.

a rule accepted operations must preserve — a predicate over the relevant system state that the protocol is designed to maintain.

Read in context: 8.1.6, 11.4.6

IS NULL

a direct test for missingness — a SQL predicate that checks whether an expression yields the NULL marker.

Read in context: 12.2.6

Isolation

rules for concurrent work seeing and affecting data — the semantics governing interaction between overlapping transactions.

rules for overlapping work — guarantees governing visibility and interaction among concurrent transactions.

Read in context: 8.4.6

J

Join

combining rows through a stated condition — an operation whose result can contain multiple matches for one source row.

Read in context: 9.4.6

Join predicate

the rule for pairing records — a Boolean condition controlling which row combinations contribute to a joined relation.

Read in context: 13.2.6

JSON array

a sequence of values — an ordered collection whose elements may themselves be structured.

Read in context: 7.4.6

JSON object

a group of named values — a collection of name/value members within a JSON representation.

Read in context: 7.4.6

Junction table

a table of links — an associative relation whose rows represent a defined pairing or richer relationship.

Read in context: 9.4.6

K

Key collision

two attempted records using the same constrained identity — a uniqueness conflict whose business interpretation requires the surrounding policy.

Read in context: 11.2.6

L

Left join

preserving the left input when no right match exists — an outer join that supplies NULL right-side values for unmatched left rows.

Read in context: 9.5.6

Left outer join

keep every left record, even without a match — a join adding a null-extended row for each unmatched left input row.

Read in context: 13.3.6

Length prefix

a size written before the content — a field giving the amount of following data in a stated unit.

Read in context: 7.2.6

LIMIT

a cap on rows returned by a query — an output selection clause that does not by itself bound all execution work.

Read in context: 12.3.6

Logical record

one complete parsed record — a structure that may span more than one physical text line.

Read in context: 7.3.6

Lossless coercion

changing representation without losing the represented value — an allowed conversion such as a supported textual integer into integer storage.

Read in context: 10.4.6

Lossless decomposition

splitting without inventing or losing combinations on reconstruction — decomposition whose natural join recovers every legal original relation instance under the stated dependencies.

Read in context: 13.4.6

Lost update

one accepted contribution disappearing beneath another write — a concurrency anomaly caused by insufficient coordination of read-modify-write operations.

one accepted change is overwritten by another — a write anomaly in which a later write fails to preserve an effect that should remain represented.

Read in context: 8.1.6

M

Magic marker

an initial clue about the representation — a fixed signature used to select or validate a format boundary, not authenticate a producer.

Read in context: 7.5.6

Many-to-many

several records on either side can relate — an association that needs an explicit representation and participation rules.

Read in context: 9.4.6

Metadata

information describing other managed information — structural definitions such as columns, constraints and indexes in this context.

Read in context: 8.2.6

Migration

a controlled evolution of structure or data — a versioned change with preconditions and a recovery strategy.

Read in context: 8.5.6

Model defect

a representation that cannot express the intended work correctly — a structural or semantic mismatch with valid domain cases.

Read in context: 10.6.6

Multiset

a collection that can contain repeated values — the bag-style semantics relevant to many ordinary SQL results.

Read in context: 9.1.6

Mutation policy

the rules for changing a record — a specification of permitted transitions, authorization, evidence and failure handling.

Read in context: 10.3.6

N

Namespace

where a name has its meaning; technically, the scope within which identifiers are interpreted and uniqueness is defined.

the context in which an identifier has meaning; technically, an issuing or interpretive scope within which names are assigned and resolved.

a named container used to distinguish object names — in PostgreSQL, a schema that can contain tables and other database objects.

Read in context: 10.5.6

Normal form

a named condition for placing relational facts — a dependency-based schema property such as 3NF or BCNF.

Read in context: 13.5.6

NOT NULL

a requirement that a database value be present — a constraint rejecting SQL NULL for a column.

Read in context: 9.2.6

Not-null rule

a requirement that a value be present — a column condition excluding SQL NULL, distinct from nonempty text or positive numeric value.

Read in context: 11.1.6

Null extension

empty right-hand fields for an unmatched record — synthetic NULL-valued attributes introduced by an outer join, not a stored right-hand record.

Read in context: 13.3.6

Nullable foreign key

a reference column that can contain SQL NULL — an optional association under the engine's stated matching semantics.

Read in context: 9.5.6

O

One-to-many

one parent can have several children — a relationship whose child references need not be unique across the child table.

Read in context: 9.3.6

One-to-one

no more than one related row at each relevant side — a relationship whose optionality must still be specified separately.

Read in context: 9.3.6

Operational owner

the person or team responsible for running and recovering the system — a named responsibility beyond writing its initial code.

the accountable responder — the person or team responsible for a signal, policy or unresolved task.

Read in context: 8.6.6

Operator precedence

the default grouping rules for an expression — syntax rules that determine binding unless parentheses explicitly override the grouping.

Read in context: 12.2.6

Optimistic concurrency check

accepting a change only when relevant expected state still matches — a conflict-detection technique whose adequacy depends on the chosen predicate and version-maintenance policy.

Read in context: 12.6.6

Optional relationship

a permitted link that need not be present — a reference with a minimum participation count of zero.

Read in context: 9.5.6

ORDER BY

the requested ordering of a query result — a clause specifying sort expressions, directions and applicable ordering semantics.

Read in context: 12.3.6

Orphan row

a child with no required matching parent — an existing record violating the intended referential relationship.

Read in context: 11.3.6

P

Pagination

splitting a result into separately retrieved pages — a reading pattern needing ordering and cross-request consistency decisions.

Read in context: 12.3.6

Parent row

the row being referenced — a record whose key is the target of a particular foreign-key relationship.

Read in context: 9.3.6

Parsing

finding the structure in a representation — recognising records, fields and values under a grammar.

Read in context: 7.2.6

Partial operation

only part of a logically related change becoming effective — an incomplete business action whose pieces were not protected by the needed boundary.

Read in context: 8.1.6

Participation minimum

the least number of required related records — a cardinality condition such as “every completed order has at least one line”.

Read in context: 10.2.6

Path

instructions for locating something in a filesystem — a name interpreted under platform-specific pathname-resolution rules.

Read in context: 7.1.6

Placeholder

a marked position for a bound value — syntax such as `?` in the demonstrated Python/SQLite interface.

Read in context: 12.6.6

Portability

the ability to move a defined workload between environments — compatibility of the used semantics and operations, not merely similar syntax.

Read in context: 8.6.6

Postcondition

something promised after an action — a predicate the completed operation is intended to establish or preserve.

Read in context: 10.1.6

Pre-aggregation

calculate a total before joining it to other detail — reducing a relation to the required grouping grain before a later join.

Read in context: 13.6.6

Pre-binding validation

checking input before the driver sends it — validation while original runtime distinctions remain available to application code.

Read in context: 10.4.6

Precondition

something that must hold before an action — a predicate required at an operation's entry boundary.

what must be true before a change — the checked state and authority on which safe execution depends.

Read in context: 10.1.6

Primary key

the table's chosen identifying key; technically, a selected set of columns whose combined values uniquely identify each row and cannot be null under the relevant database rules.

the chosen required identifier for a table — the designated key used as a principal row identity in the current schema.

Read in context: 11.2.6

Prime attribute

a field belonging to at least one minimal key — an attribute contained in some candidate key, not necessarily the selected primary key.

Read in context: 13.5.6

Process boundary

a separation between running programs — an isolation and communication boundary that changes failure and resource handling.

Read in context: 8.3.6

Projection, in a query

choosing which values to display — an operation that selects output columns and can hide source identity.

Read in context: 9.1.6

Provenance

the history behind a record; technically, information about entities, activities, agents and derivations involved in producing it.

where information came from and how it was handled — recorded source, capture and transformation context, not automatic proof of truth.

the recorded origin and processing history — information linking source entities, transformations and resulting data.

Read in context: 7.1.6

Publication boundary

the point where accepted work becomes available in the destination — a separately controlled step beyond decoding and input validation.

Read in context: 7.6.6

Publication unit

the selected set of records accepted together — the transaction scope for this destination, distinct from an entire source file.

Read in context: 9.6.6

Q

Qualified name

a name that includes its containing scope — an identifier such as `sales.orders` rather than an unqualified table name.

Read in context: 10.5.6

Quarantine

retaining rejected material for controlled review — a handling state with separately specified access and retention rules.

Read in context: 7.6.6

Query

a precise question; technically, an operation expressing a requested result over data.

a request for a result from data — an expression evaluated under a data model and its language semantics.

Read in context: 8.2.6

Query plan

the engine's proposed route to an answer — an execution structure using operations such as scans, filters, joins and aggregation.

Read in context: 8.2.6

Quoting

marking a region as one field — a representation mechanism that allows otherwise structural characters inside a value.

Read in context: 7.3.6

R

Read-modify-write

read a value, compute a replacement, then store it — a multi-step sequence that needs an appropriate concurrency rule.

Read in context: 8.1.6

Reconciliation

comparing related accounts of the same work; technically, identifying and resolving or explicitly retaining discrepancies under a defined procedure.

checking whether related records agree as expected — a comparison of identities, relationships and totals rather than a single balancing sum.

explaining how the input relates to the output — accounting for candidates, repeats, rejections and file-level limits without silently losing material.

compare defined representations of the same work — an evidence-based check of populations, keys, amounts and explained differences.

compare related records to resolve disagreement — a procedure that checks quantities, identities and outcomes against stated invariants and evidence.

comparing expected and observed information — a set of coverage, identity, value and invariant checks across representations.

comparing meaningfully corresponding records — checking identities, values and declared exclusions rather than one headline total.

Read in context: 7.6.6, 13.6.6

Recovery point objective

how much recent work the plan may lose — a stated acceptable recovery-point gap, not an achieved measurement by itself.

Read in context: 8.5.6

Recovery time objective

how long recovery may take — a target for restoring a defined service level under a stated scenario.

Read in context: 8.5.6

Recovery unit

the things the restoration claim includes — an explicit scope such as one database or a complete application service.

Read in context: 8.5.6

Redundancy

several representations of a fact — duplication whose consistency depends on maintaining a specified relationship between representations.

Read in context: 13.1.6

Referential action

what happens to a relationship when a parent changes — a declared response such as restriction, cascading deletion or setting a reference to NULL.

Read in context: 11.3.6

Referential integrity

references remaining valid under declared rules — the property enforced by active foreign-key constraints within their semantics.

Read in context: 9.6.6

Regression test

a repeatable check preserving an intended behaviour — evidence that a known property still holds for the tested inputs after a change.

Read in context: 9.6.6

Relative path

a location described from a starting point — a pathname whose interpretation depends on a working directory or another supplied base.

Read in context: 7.1.6

Repair branch

a deliberate resolution of incomplete transactional work — an explicitly permitted continuation that restores the failed condition before a later commit.

Read in context: 11.5.6

Repricing

calculating using a different price basis — recomputation with current or alternative prices rather than the stored agreed values.

Read in context: 10.3.6

Restore test

trying the recovery procedure and inspecting its result — evidence about a specified backup and environment rather than a promise about all future failures.

Read in context: 8.5.6

Result contract

the meaning promised by a query's output — a specification of columns, units, grain, missingness and ordering where relevant.

Read in context: 10.5.6

Result population

the records or combinations included in an answer — the scope after joins and filters, which must match the report's intended question.

Read in context: 9.5.6

Rollback

cancelling uncommitted work; technically, discarding the transaction changes within the requested rollback scope.

abandoning uncommitted database work — restoration of the transaction's prior database state, not reversal of unrelated external actions.

abandon uncommitted transactional work — restoration of the transaction's database effects under its supported semantics.

returning within a defined reversible boundary — undoing a transaction or deployment, with scope that must be stated.

Read in context: 8.4.6

Row grain

what one row stands for — the unit of representation against which counts and aggregates must be interpreted.

what one row asserts — the identity and meaning that determine valid joins and constraints.

Read in context: 9.1.6

S

Schema boundary

what the declared database model does and does not enforce — the limit beyond which application workflow or other mechanisms remain necessary.

Read in context: 9.6.6

Schema design

the plan for organising records and their rules — a model of structures, domains, relationships and invariants implemented by a database and its surrounding system.

Read in context: 10.1.6

Schema migration

a controlled change to database structure or representation — a transition requiring data compatibility, verification and a defined failure disposition.

Read in context: 11.6.6

Schema version

which contract applies; technically, an identifier for a defined revision of the schema and its interpretation.

the named edition of a record contract — an identifier for the supported structure and meaning, distinct from a software release.

Read in context: 7.4.6

Select list

the values requested for each output row — the column references or expressions following SELECT.

Read in context: 12.1.6

SELECT statement

a description of an answer to read — a SQL query specifying output expressions and the operations needed to define a result.

Read in context: 12.1.6

Sentinel entity

a special record used to stand for a designated case — a modelling choice that must not fabricate identity or conceal unknown meaning.

Read in context: 9.5.6

Serverless, in SQLite

no separate database server is required — SQLite's embedded-engine meaning, distinct from some cloud-service uses of the word.

Read in context: 8.3.6

SQL expression

a rule for calculating an output value — a combination of column references, literals, operators or functions evaluated under engine semantics.

Read in context: 12.4.6

Statement rollback

undoing effects of one failed statement — a failure response distinct from abandoning all earlier work in the active transaction.

Read in context: 11.6.6

Storage type

the engine's representation category; technically, a supported type or storage class with documented operations and limits.

the kind of value an engine stores in a field — an implementation-level representation with defined conversion and operation rules.

Read in context: 10.4.6

Syntactic validity

being written in the right shape — conformance to a format's grammar, distinct from business rules.

Read in context: 7.2.6

System metadata

information the engine stores about its own objects — structural descriptions exposed through catalogues or metadata interfaces.

Read in context: 10.5.6

T

Table

a collection of records with a declared shape — a managed SQL structure whose columns and constraints give rows a defined interpretation.

Read in context: 9.1.6

Target cardinality

how many records an operation is intended to affect — a result-scope requirement distinct from whether each resulting row is structurally valid.

Read in context: 12.5.6

Temporal interpretation

which point or period a value describes — the time-related meaning that distinguishes a current attribute from a historical observation or agreement.

Read in context: 10.2.6

Text literal

text written as a value in a statement — a quoted string such as `'0-1042'`, distinct from a column identifier.

Read in context: 12.1.6

Three-valued logic

reasoning with true, false and unknown — SQL expression semantics needed when operands can be NULL.

Read in context: 12.2.6

Tie-breaker

an additional rule for equal primary sort values — a subsequent sort expression used to distinguish otherwise tied rows.

an extra rule for equal first choices — additional ordering attributes that distinguish otherwise tied results.

an additional ordering key — a rule making otherwise tied results deterministic where required.

Read in context: 12.3.6

Traceability matrix

a record showing how a requirement is addressed — a mapping among assumptions, representation, enforcement, tests and remaining gaps.

Read in context: 10.6.6

Transaction

a controlled unit of database work; technically, operations managed together under an engine's atomicity and related semantics.

a defined group of database work — a unit with a controlled commit or rollback boundary.

Read in context: 8.4.6

Transaction ownership

responsibility for starting and ending a transaction — a contract defining which component may commit, roll back or leave the connection active.

responsibility for completion — the component entitled to commit or roll back the particular unit of work.

Read in context: 11.5.6

Transaction rollback

abandoning a transaction's pending changes — an explicit or engine-triggered reversal within that transaction's defined scope.

Read in context: 11.6.6

Truncation

an input ending before its promised content is complete — insufficient bytes to satisfy a declared structure or length.

Read in context: 7.2.6

Tuple

one same-shaped collection of attribute values — a row-like element of a relation in the relational model.

Read in context: 9.1.6

U

Unique constraint

a rule against repeated key values; technically, database enforcement that the relevant value combination cannot appear in conflicting rows under its null and comparison rules.

a prohibition on repeated constrained values — an invariant over one column or a composite value under the engine's comparison and NULL rules.

Read in context: 11.2.6

Unknown field

a member the contract does not recognise — data requiring an explicit reject, ignore, preserve or extension policy.

Read in context: 7.4.6

Unknown result

a comparison that cannot be classified as true or false from the supplied values — the third truth outcome commonly produced by SQL expressions involving NULL.

Read in context: 11.1.6

UPDATE

changing values on selected records — a SQL write operation applying assignments to rows matching its predicate.

Read in context: 12.5.6

Update anomaly

one correction leaves conflicting copies — a state inconsistency caused by storing a dependent fact in multiple independently editable rows.

Read in context: 13.1.6

W

WHERE predicate

the condition deciding which source rows qualify — a Boolean-like SQL expression for which only true rows are retained by the filter.

Read in context: 12.2.6

Workflow

the steps around completing a piece of work — a sequence of operations, decisions and failure paths with defined state transitions.

Read in context: 10.1.6

Workload

the work the system must actually handle — a defined mixture of reads, writes, sizes, timing and contention.

Read in context: 8.6.6

Write path

a route through which information can change — an application, import, job or administrative mechanism capable of mutating stored state.

Read in context: 10.1.6

Wrong-scope mutation

a valid change applied to the wrong set of records — an operation whose predicate does not express the intended target boundary.

Read in context: 12.5.6

Sources and Version Register

These primary sources support adjacent technical claims. The synthetic shop, arithmetic fixtures and teaching models are original examples. Versioned references are not automatically descriptions of a newer release.

[S01] PostgreSQL 17 documentation: Transactions

PostgreSQL Global Development Group

Atomic changes, commit and rollback; product-specific tutorial.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/tutorial-transactions.html

[S02] PostgreSQL 17 documentation: Constraints

PostgreSQL Global Development Group

CHECK, NOT NULL, primary-key and foreign-key behaviour.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/ddl-constraints.html

[S03] PostgreSQL 17 documentation: Sorting Rows (ORDER BY)

PostgreSQL Global Development Group

No guaranteed result order without explicit ordering.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/queries-order.html

[S04] PostgreSQL 17 documentation: Asynchronous Commit

PostgreSQL Global Development Group

Acknowledgement and durability depend on configuration.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/wal-async-commit.html

[S05] PROV-DM: The PROV Data Model (2013)

W3C; editors Luc Moreau and Paolo Missier

Entities, activities, agents and derivation in provenance.

Consulted: 23 September 2026

https://www.w3.org/TR/prov-dm/

[S06] Data on the Web Best Practices (2017)

W3C

Metadata, provenance, quality, versioning and persistent identifiers.

Consulted: 23 September 2026

https://www.w3.org/TR/dwbp/

[S07] RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format (2017)

T. Bray, editor; IETF

JSON value types, interoperable numbers, names and encoding.

Consulted: 23 September 2026

https://www.rfc-editor.org/info/rfc8259/

[S08] RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files (2005)

Y. Shafranovich; IETF

Informational RFC describing common CSV conventions; not a universal spreadsheet schema.

Consulted: 23 September 2026

https://www.rfc-editor.org/info/rfc4180/

[S09] RFC 3339: Date and Time on the Internet: Timestamps (2002)

G. Klyne and C. Newman; IETF

Timestamp syntax and numeric UTC offsets.

Consulted: 23 September 2026

https://www.rfc-editor.org/info/rfc3339/

[S10] PostgreSQL 17 documentation: Numeric Types

PostgreSQL Global Development Group

Integer ranges, exact numeric types and floating-point types.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/datatype-numeric.html

[S11] PostgreSQL 17 documentation: Date/Time Types

PostgreSQL Global Development Group

Date and timestamp semantics; time-zone handling.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/datatype-datetime.html

[S12] Python 3.12 tutorial: Floating-Point Arithmetic, Issues and Limitations

Python Software Foundation

Binary fractions and displayed floating-point results.

Consulted: 23 September 2026

https://docs.python.org/3.12/tutorial/floatingpoint.html

[S13] Python 3.12 library reference: decimal

Python Software Foundation

Decimal construction, precision, quantisation and rounding contexts.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/decimal.html

[S14] FAQ: UTF-8, UTF-16, UTF-32 and BOM

Unicode Consortium

Unicode encoding forms and UTF-8 byte lengths.

Consulted: 23 September 2026

https://www.unicode.org/faq/utf_bom.html

[S15] Unicode Standard Annex #15: Unicode Normalization Forms

Unicode Consortium

Canonical equivalence, NFC and the limits of normalisation.

Consulted: 23 September 2026

https://www.unicode.org/reports/tr15/

[S16] Unicode Standard Annex #29: Unicode Text Segmentation

Unicode Consortium

Grapheme clusters and user-perceived character boundaries.

Consulted: 23 September 2026

https://www.unicode.org/reports/tr29/

[S17] PostgreSQL 17 documentation: Comparison Functions and Operators

PostgreSQL Global Development Group

NULL comparisons and IS NULL.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/functions-comparison.html

[S18] PostgreSQL 17 documentation: Logical Operators

PostgreSQL Global Development Group

Three-valued Boolean logic.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/functions-logical.html

[S19] PostgreSQL 17 documentation: Aggregate Functions

PostgreSQL Global Development Group

COUNT and AVG treatment of non-null inputs.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/functions-aggregate.html

[S20] Python 3.12 library reference: sqlite3

Python Software Foundation

Embedded SQLite connections and bound parameters.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/sqlite3.html

[S21] Datatypes in SQLite

SQLite project

Dynamic typing, storage classes and differences from other engines.

Consulted: 23 September 2026

https://www.sqlite.org/datatype3.html

[S22] Python 3.12 library reference: Built-in Types

Python Software Foundation

Integer precision and the bool subclass relationship.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/stdtypes.html

[S23] Python 3.12 library reference: datetime

Python Software Foundation

Aware datetimes, offsets and conversion to UTC.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/datetime.html

[S24] SQL Language Expressions

SQLite project

NULL and logical operations in the supplied SQLite lab.

Consulted: 23 September 2026

https://www.sqlite.org/lang_expr.html

[S25] Transaction

SQLite project

Explicit BEGIN, COMMIT and ROLLBACK in the supplied lab.

Consulted: 23 September 2026

https://www.sqlite.org/lang_transaction.html

[S26] VIM3, 2.3: Measurand

JCGM / BIPM

Defining the quantity intended to be measured.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.3.html

[S27] SI prefixes

BIPM

Decimal factors and prefix symbols; conversions in the text are original calculations.

Consulted: 23 September 2026

https://www.bipm.org/en/measurement-units/si-prefixes

[S28] VIM3, 2.13: Measurement accuracy

JCGM / BIPM

Accuracy is distinguished from precision.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.13.html

[S29] VIM3, 2.15: Measurement precision

JCGM / BIPM

Agreement among repeated indications under specified conditions.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.15.html

[S30] VIM3, 2.39: Calibration

JCGM / BIPM

Calibration is not the same operation as adjustment or verification.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.39.html

[S31] Combining uncertainty components

NIST

Propagation of uncertainty, sensitivities and covariances.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/combination.html

[S32] Type A evaluation of standard uncertainty

NIST

Standard uncertainty of a mean for independent observations.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/typea.html

[S33] VIM3, 4.14: Resolution

JCGM / BIPM

A perceptible response to a change in the measured quantity.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/4.14.html

[S34] Type B evaluation of standard uncertainty

NIST

Uncertainty evaluated from other information and assumed distributions.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/typeb.html

[S35] VIM3, 2.26: Measurement uncertainty

JCGM / BIPM

Dispersion of values attributed to a measurand using available information.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.26.html

[S36] Expanded uncertainty and coverage factors

NIST

U = k times combined standard uncertainty; coverage depends on assumptions.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/coverage.html

[S37] PostgreSQL 17 documentation: Identity Columns

PostgreSQL Global Development Group

Generated values do not by themselves enforce uniqueness.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/ddl-identity-columns.html

[S38] RFC 9562: Universally Unique IDentifiers (UUIDs), 2024

IETF; K. Davis, B. Peabody and P. Leach

UUID version 4 has 122 random bits; collision estimate is an original conditional calculation.

Consulted: 23 September 2026

https://www.rfc-editor.org/rfc/rfc9562.html

[S39] CloudEvents specification, version 1.0.2

Cloud Native Computing Foundation; CloudEvents project

Event source and id uniqueness; the wire specversion value is 1.0.

Consulted: 23 September 2026

https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/spec.md

[S40] Event Sourcing pattern

Microsoft Azure Architecture Center

Architecture context: event history, projections, snapshots, replay and trade-offs. The shop state machines are original teaching designs, not a Microsoft reference implementation.

Consulted: 23 September 2026

https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

[S41] Time, Clocks, and the Ordering of Events in a Distributed System (1978)

Leslie Lamport

Communications of the ACM 21(7), 558–565; happened-before and logical clocks. Counter exercises are original illustrations.

Consulted: 23 September 2026

https://lamport.azurewebsites.net/pubs/time-clocks.pdf

[S42] Python 3.13 library reference: pathlib

Python Software Foundation

Pure paths versus filesystem operations; path components and lexical interpretation.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/pathlib.html

[S43] Python 3.13 library reference: io

Python Software Foundation

Text/byte streams, encoding, newline handling and input boundaries.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/io.html

[S44] Python 3.13 library reference: csv

Python Software Foundation

CSV dialects, reader and writer behaviour; strict mode is not domain validation.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/csv.html

[S45] Python 3.13 library reference: json

Python Software Foundation

Object pairs hooks, numeric constants, supported types and parser resource cautions.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/json.html

[S46] Python 3.13 library reference: struct

Python Software Foundation

Explicit byte order, widths and binary packing; the KDBF envelope is original teaching material.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/struct.html

[S47] Apache Parquet documentation: Overview

Apache Software Foundation

Column-oriented file-format purpose; no Parquet performance measurements are claimed.

Consulted: 24 September 2026

https://parquet.apache.org/docs/overview/

[S48] Apache Avro 1.12.0 specification

Apache Software Foundation

Schema-based representation and writer/reader schema resolution; not implemented by the toy envelope.

Consulted: 24 September 2026

https://avro.apache.org/docs/1.12.0/specification/

[S49] Model for Tabular Data and Metadata on the Web

W3C

Representations, semantic values, cells and annotations in tabular data.

Consulted: 24 September 2026

https://www.w3.org/TR/tabular-data-model/

[S50] SQLite Is Serverless

SQLite project

Embedded, in-process engine versus a separately running database server.

Consulted: 24 September 2026

https://www.sqlite.org/serverless.html

[S51] Appropriate Uses For SQLite

SQLite project

Embedded-use scope, local access and situations favouring client-server databases.

Consulted: 24 September 2026

https://www.sqlite.org/whentouse.html

[S52] PostgreSQL 17 documentation: Architectural Fundamentals

PostgreSQL Global Development Group

Database client/server architecture and separation from application code.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/tutorial-arch.html

[S53] Atomic Commit In SQLite

SQLite project

Journaling and filesystem/device assumptions. This lab does not inject power failures.

Consulted: 24 September 2026

https://www.sqlite.org/atomiccommit.html

[S54] Isolation In SQLite

SQLite project

Writer serialization and transaction visibility; local lock demonstration is separately bounded.

Consulted: 24 September 2026

https://www.sqlite.org/isolation.html

[S55] SQLite Online Backup API

SQLite project

Engine-supported copying into a destination database; no production recovery-time claim.

Consulted: 24 September 2026

https://www.sqlite.org/backup.html

[S56] PostgreSQL 17 documentation: Table Basics

PostgreSQL Global Development Group

Tables, columns and declared data types.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/ddl-basics.html

[S57] PostgreSQL 17 documentation: Select Lists

PostgreSQL Global Development Group

Projection, output columns and duplicate elimination.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/queries-select-lists.html

[S58] PostgreSQL 17 documentation: Table Expressions

PostgreSQL Global Development Group

Joins, outer joins, qualification and the effect of later filtering.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/queries-table-expressions.html

[S59] SQLite Foreign Key Support

SQLite project

Per-connection enforcement, parent/child references and nullable child keys.

Consulted: 24 September 2026

https://www.sqlite.org/foreignkeys.html

[S60] STRICT Tables

SQLite project

SQLite 3.37.0 minimum, strict storage types and permitted lossless coercion.

Consulted: 24 September 2026

https://www.sqlite.org/stricttables.html

[S61] CREATE TABLE

SQLite project

CHECK, NOT NULL, uniqueness and primary-key implementation details.

Consulted: 24 September 2026

https://www.sqlite.org/lang_createtable.html

[S62] Python 3.13 library reference: pickle

Python Software Foundation

Do not unpickle untrusted data; serialization is not automatically safe execution.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/pickle.html

[S63] CSV Injection

OWASP Foundation community

Spreadsheet formula interpretation is separate from CSV quotation; no universal sanitizer is claimed.

Consulted: 24 September 2026

https://community.owasp.org/attacks/CSV_Injection

[S64] Python 3.13 library reference: os

Python Software Foundation

Filesystem operations, replacement semantics and system-dependent boundaries.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/os.html

[S65] PostgreSQL 17 documentation: Using EXPLAIN

PostgreSQL Global Development Group

Plan costs and actual execution with EXPLAIN ANALYZE.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/using-explain.html

[S66] PostgreSQL 17 documentation: Transaction Isolation

PostgreSQL Global Development Group

Engine-specific isolation guarantees and transaction retry requirements.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/transaction-iso.html

[S67] PostgreSQL 17 documentation: SQL Dump

PostgreSQL Global Development Group

Logical backup and restoration; cited context, not executed in the SQLite lab.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/backup-dump.html

[S68] PostgreSQL 17 documentation: CREATE DOMAIN

PostgreSQL Global Development Group

Reusable constrained base types and domain-nullability caveats; the SQL illustration is not executed in the SQLite lab.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-createdomain.html

[S69] PostgreSQL 17 documentation: Schemas

PostgreSQL Global Development Group

Schema namespaces, qualified names and name-resolution context; distinct from the general modelling meaning of schema.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-schemas.html

[S70] PostgreSQL 17 documentation: COMMENT

PostgreSQL Global Development Group

Object comments as descriptive metadata, not enforced business rules or a place for secrets.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-comment.html

[S71] PostgreSQL 17 documentation: The Information Schema

PostgreSQL Global Development Group

Standard-oriented metadata inspection and its distinction from business meaning.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/information-schema.html

[S72] PostgreSQL 17 documentation: Lexical Structure

PostgreSQL Global Development Group

Identifiers, text literals and quoting; naming conventions in the book are editorial choices.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-syntax-lexical.html

[S73] PostgreSQL 17 documentation: SET CONSTRAINTS

PostgreSQL Global Development Group

Eligible constraint classes and checking-mode changes; PostgreSQL is documented, not executed.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-set-constraints.html

[S74] PostgreSQL 17 documentation: CREATE TABLE

PostgreSQL Global Development Group

Keys, references, match semantics and deferrability; product-specific rather than a cross-engine promise.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-createtable.html

[S75] PRAGMA Statements

SQLite project

Metadata interfaces and separate scopes of foreign_keys, integrity_check and foreign_key_check.

Consulted: 25 September 2026

https://www.sqlite.org/pragma.html

[S76] The ON CONFLICT Clause

SQLite project

Default ABORT statement rollback and the distinction between replacement and an ordinary update.

Consulted: 25 September 2026

https://www.sqlite.org/lang_conflict.html

[S77] Result and Error Codes

SQLite project

Machine-readable error categories; selected extended constraint codes observed by the lab.

Consulted: 25 September 2026

https://www.sqlite.org/rescode.html

[S78] PostgreSQL 17 documentation: Modifying Tables

PostgreSQL Global Development Group

Constraint-change and existing-data considerations; no production or PostgreSQL migration is run.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-alter.html

[S79] PostgreSQL 17 documentation: Querying a Table

PostgreSQL Global Development Group

SELECT fundamentals, source columns, expressions and result questions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/tutorial-select.html

[S80] SELECT

SQLite project

Logical result semantics, output expressions, filtering, duplicate elimination and ordering.

Consulted: 25 September 2026

https://www.sqlite.org/lang_select.html

[S81] PostgreSQL 17 documentation: LIMIT and OFFSET

PostgreSQL Global Development Group

Output limits and the need for predictable ordering; not a performance bound.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/queries-limit.html

[S82] PostgreSQL 17 documentation: Conditional Expressions

PostgreSQL Global Development Group

CASE and COALESCE as explicit conditional choices; the narrative fixtures are original.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/functions-conditional.html

[S83] INSERT

SQLite project

Explicit target columns and value insertion in isolated draft examples.

Consulted: 25 September 2026

https://www.sqlite.org/lang_insert.html

[S84] UPDATE

SQLite project

Assignment and predicate scope; unqualified update demonstrated only in a disposable transaction.

Consulted: 25 September 2026

https://www.sqlite.org/lang_update.html

[S85] DELETE

SQLite project

Selected-row deletion semantics; not a claim of erasure across backups or external copies.

Consulted: 25 September 2026

https://www.sqlite.org/lang_delete.html

[S86] Python 3.13 library reference: sqlite3

Python Software Foundation

Driver binding, cursors, result counts, connection closing and explicit transaction configuration.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/sqlite3.html

[S87] Built-In Scalar SQL Functions

SQLite project

COALESCE behaviour, including the difference between NULL and empty text.

Consulted: 25 September 2026

https://www.sqlite.org/lang_corefunc.html

[S88] Database System Concepts, seventh edition: Chapter 7 author slides, Normalization

Abraham Silberschatz, Henry F. Korth and S. Sudarshan

Functional dependencies, lossless decomposition, dependency preservation, BCNF and third normal form. The shop exercises are original.

Consulted: 25 September 2026

https://www.db-book.com/slides-dir/PDF-dir/ch7.pdf

[S89] Query Planning

SQLite project

Scans, ordered lookup, compound and covering indexes, and sorting choices.

Consulted: 25 September 2026

https://www.sqlite.org/queryplanner.html

[S90] EXPLAIN QUERY PLAN

SQLite project

Human-readable SCAN, SEARCH and temporary-tree descriptions; output format is not a stable application interface.

Consulted: 25 September 2026

https://www.sqlite.org/eqp.html

[S91] PostgreSQL 17: Multicolumn Indexes

PostgreSQL Global Development Group

Column order and constraints on efficient access in the explicitly cited version.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-multicolumn.html

[S92] PostgreSQL 17: Index-Only Scans and Covering Indexes

PostgreSQL Global Development Group

INCLUDE payloads and visibility checks; covering does not guarantee zero heap visits.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-index-only-scans.html

[S93] Partial Indexes

SQLite project

Predicate-restricted indexes and eligibility based on implication.

Consulted: 25 September 2026

https://www.sqlite.org/partialindex.html

[S94] PostgreSQL 17: Statistics Used by the Planner

PostgreSQL Global Development Group

Sampling, distribution estimates and extended statistics.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/planner-stats.html

[S95] PostgreSQL 17: Resource Consumption

PostgreSQL Global Development Group

Operation-level memory allowances and spill; not a benchmark of this manuscript.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/runtime-config-resource.html

[S96] PostgreSQL 17 tutorial: Window Functions

PostgreSQL Global Development Group

Window partitions, ordering and preservation of detail rows.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/tutorial-window.html

[S97] Window Functions

SQLite project

Window frames, peers and aggregate-window behaviour in the educational SQL exercises.

Consulted: 25 September 2026

https://www.sqlite.org/windowfunctions.html

[S98] The SQLite Query Optimizer Overview

SQLite project

Access-path transformations, joins and implementation-specific planning decisions.

Consulted: 25 September 2026

https://www.sqlite.org/optoverview.html

[S99] Python 3.13: hashlib

Python Software Foundation

Digest interfaces; the collision and comparison exercises are original.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/hashlib.html

[S100] FIPS 180-4: Secure Hash Standard, August 2015

NIST

Standardized SHA-2 digests, distinguished from authentication and encryption.

Consulted: 25 September 2026

https://csrc.nist.gov/pubs/fips/180-4/upd1/final

[S101] PostgreSQL 17: Hash Indexes

PostgreSQL Global Development Group

Equality-oriented, lossy hash access and collision rechecking.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/hash-index.html

[S102] Python 3.13: time

Python Software Foundation

Monotonic performance timers and their units, not an assertion of nanosecond accuracy.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/time.html

[S103] Monitoring Distributed Systems

Rob Ewaschuk; Google SRE

Latency, traffic, errors and saturation as monitoring context; workload examples are invented.

Consulted: 25 September 2026

https://sre.google/sre-book/monitoring-distributed-systems/

[S104] PostgreSQL 17: Index Types

PostgreSQL Global Development Group

Index methods support different operators; a method name is not a universal performance promise.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-types.html

[S105] PostgreSQL 17: B-Tree Indexes

PostgreSQL Global Development Group

B-tree structure and implementation notes. The hand-worked B+ tree is deliberately not a PostgreSQL implementation.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/btree.html

[S106] PostgreSQL 17: Indexes and ORDER BY

PostgreSQL Global Development Group

Ordered access and explicit query ordering.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-ordering.html

[S107] PostgreSQL 17: ANALYZE

PostgreSQL Global Development Group

Updating sampled planner statistics and operational scope.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-analyze.html

[S108] Python 3.13: Data Model

Python Software Foundation

Hash/equality contracts and process-randomized string and byte hashes.

Consulted: 25 September 2026

https://docs.python.org/3.13/reference/datamodel.html

[S109] Python 3.13: bisect

Python Software Foundation

Ordered-list insertion positions; insertion cost and concurrent-mutation limitations.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/bisect.html

[S110] PostgreSQL 17: Window Functions

PostgreSQL Global Development Group

Ranking, offsets, frames and frame-sensitive last_value behaviour.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/functions-window.html

[S111] PostgreSQL 17: The Path of a Query

PostgreSQL Global Development Group

Parsing, rewriting, planning and execution stages.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/query-path.html

[S112] PostgreSQL 17: Planner/Optimizer

PostgreSQL Global Development Group

Plan search, nested-loop, merge and hash joins; optimality is bounded by search and estimation.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/planner-optimizer.html

[S113] PostgreSQL 17: EXPLAIN

PostgreSQL Global Development Group

Diagnostic options, actual execution with ANALYZE, instrumentation boundaries and non-text formats.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-explain.html

[S114] PostgreSQL 17: Query Planning

PostgreSQL Global Development Group

Planner cost and method settings; experimentation is distinct from production tuning.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/runtime-config-query.html

[S115] PostgreSQL 17: Row Estimation Examples

PostgreSQL Global Development Group

Selectivity estimation and its assumptions; the shop arithmetic is original.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/row-estimation-examples.html

[S116] PostgreSQL 17: WITH Queries (Common Table Expressions)

PostgreSQL Global Development Group

Named query stages and materialisation/folding behaviour in the explicitly cited version.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/queries-with.html

[S117] PostgreSQL 17 tutorial: Transactions

PostgreSQL Global Development Group

Transaction blocks, commit, rollback and the scope of database changes.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/tutorial-transactions.html

[S118] PostgreSQL 17: Explicit Locking

PostgreSQL Global Development Group

Lock modes, row/table distinctions, deadlocks and advisory-lock lifetimes.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/explicit-locking.html

[S119] PostgreSQL 17: Concurrency Control Introduction

PostgreSQL Global Development Group

Multiversion visibility and distinction from application history.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/mvcc-intro.html

[S120] PostgreSQL 17: Serialization Failure Handling

PostgreSQL Global Development Group

Whole-transaction retries, SQLSTATE 40001 and careful classification of other failures.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/mvcc-serialization-failure-handling.html

[S121] PostgreSQL 17: SAVEPOINT

PostgreSQL Global Development Group

Savepoint scope within a transaction.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-savepoint.html

[S122] PostgreSQL 17: Sequence Manipulation Functions

PostgreSQL Global Development Group

Sequence allocation and non-rollback behaviour; gaps are not missing-business-event proof.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/functions-sequence.html

[S123] PostgreSQL 17: Routine Vacuuming

PostgreSQL Global Development Group

Version cleanup, visibility maps, space reuse and transaction-identifier maintenance.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/routine-vacuuming.html

[S124] PostgreSQL 17: System Columns

PostgreSQL Global Development Group

Version-related metadata and the limits of physical tuple identifiers.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-system-columns.html

[S125] Transactional outbox pattern

Amazon Web Services

Atomic recording of business changes and delivery intent; duplicate delivery still requires handling.

Consulted: 25 September 2026

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html

[S126] Making retries safe with idempotent APIs

Amazon Web Services Builders Library

Caller request identity, repeated intent and semantic-equivalence considerations.

Consulted: 25 September 2026

https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

[S127] Savepoints

SQLite project

ROLLBACK TO and RELEASE, including the difference between inner release and outer commit.

Consulted: 25 September 2026

https://www.sqlite.org/lang_savepoint.html

[S128] PostgreSQL 17 documentation: Database Page Layout

PostgreSQL Global Development Group

Page headers, item identifiers, tuple layout and implementation boundaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/storage-page-layout.html

[S129] PostgreSQL 17 documentation: Write-Ahead Logging

PostgreSQL Global Development Group

WAL-before-data ordering, redo and grouped synchronization.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-intro.html

[S130] PostgreSQL 17 documentation: Reliability

PostgreSQL Global Development Group

Storage-cache assumptions, partial page writes and reliability limits.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-reliability.html

[S131] PostgreSQL 17 documentation: Continuous Archiving and Point-in-Time Recovery

PostgreSQL Global Development Group

Base backups, continuous WAL coverage, recovery targets and timelines.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/continuous-archiving.html

[S132] Write-Ahead Logging

SQLite project

WAL frames, checkpointing, concurrency limits and the disclosed 2026 WAL-reset bug; not an endorsement of the historical lab runtime.

Consulted: 25 September 2026

https://www.sqlite.org/wal.html

[S133] Database File Format

SQLite project

Page sizes, header fields, overflow and journal/WAL representation.

Consulted: 25 September 2026

https://www.sqlite.org/fileformat.html

[S134] Compaction

RocksDB project

Sorted runs, leveled/tiered strategies and read/write/space trade-offs.

Consulted: 25 September 2026

https://github.com/facebook/rocksdb/wiki/Compaction

[S135] PostgreSQL 17 documentation: WAL Configuration

PostgreSQL Global Development Group

Checkpoints, redo positions and resource trade-offs.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-configuration.html

[S136] PostgreSQL 17 documentation: Asynchronous Commit

PostgreSQL Global Development Group

Acknowledgement before WAL flush and the distinction from disabling fsync.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-async-commit.html

[S137] PostgreSQL 17 documentation: Data Checksums

PostgreSQL Global Development Group

Detection scope and limitations of page checksums.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/checksums.html

[S138] PostgreSQL 17 documentation: pg_verifybackup

PostgreSQL Global Development Group

Manifest verification and the explicit need for test restores.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/app-pgverifybackup.html

[S139] How To Corrupt An SQLite Database File

SQLite project

Operational corruption hazards, copying and journal-handling cautions.

Consulted: 25 September 2026

https://www.sqlite.org/howtocorrupt.html

[S140] RocksDB Overview

RocksDB project

Memtables, log files, sorted tables and the library boundary.

Consulted: 25 September 2026

https://github.com/facebook/rocksdb/wiki/RocksDB-Overview

[S141] fsync(2)

Linux man-pages project

File synchronization, directory metadata and error reporting; Linux-specific.

Consulted: 25 September 2026

https://man7.org/linux/man-pages/man2/fsync.2.html

[S142] write(2)

Linux man-pages project

Partial writes and the difference between write success and persistence.

Consulted: 25 September 2026

https://man7.org/linux/man-pages/man2/write.2.html

[S143] PostgreSQL 17 documentation: TOAST

PostgreSQL Global Development Group

Large-value compression/out-of-line storage and physical row boundaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/storage-toast.html

[S144] PostgreSQL 17 documentation: The Cumulative Statistics System

PostgreSQL Global Development Group

I/O statistics, cache boundaries, update timing and monitoring scope.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/monitoring-stats.html

[S145] VACUUM

SQLite project

File rebuilding, space reclamation and operational constraints.

Consulted: 25 September 2026

https://www.sqlite.org/lang_vacuum.html

[S146] PostgreSQL 17 documentation: amcheck

PostgreSQL Global Development Group

Table/index structural verification and limitations.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/amcheck.html

[S147] PostgreSQL 17 documentation: Log-Shipping Standby Servers

PostgreSQL Global Development Group

Physical streaming, lag, slots and synchronous standby acknowledgement boundaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/warm-standby.html

[S148] PostgreSQL 17 documentation: Logical Replication

PostgreSQL Global Development Group

Publication/subscription, initial synchronization, identities and conflicts.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/logical-replication.html

[S149] PostgreSQL 17 documentation: Write Ahead Log settings

PostgreSQL Global Development Group

synchronous_commit modes, local/remote flush and replay distinctions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/runtime-config-wal.html

[S150] PostgreSQL 17 documentation: Failover

PostgreSQL Global Development Group

Promotion, external failure detection and preventing two active primaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/warm-standby-failover.html

[S151] PostgreSQL 17 documentation: pg_rewind

PostgreSQL Global Development Group

Divergent histories, prerequisites and recovery/reconfiguration cautions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/app-pgrewind.html

[S152] PostgreSQL 17 documentation: Table Partitioning

PostgreSQL Global Development Group

Range/list/hash table partitions, pruning and implementation restrictions; not automatic multi-host sharding.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-partitioning.html

[S153] Perspectives on the CAP Theorem

Seth Gilbert and Nancy A. Lynch

Authors' retrospective explaining atomic read/write consistency, availability, unreliable communication and the proof sketch.

Consulted: 25 September 2026

https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf

[S154] In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout

Raft terms, durable voting, log matching, current-term commitment, membership and client-read safeguards.

Consulted: 25 September 2026

https://raft.github.io/raft.pdf

[S155] PostgreSQL 17 documentation: Two-Phase Transactions

PostgreSQL Global Development Group

Prepared-transaction state and the external transaction-manager boundary.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/two-phase.html

[S156] PostgreSQL 17 documentation: PREPARE TRANSACTION

PostgreSQL Global Development Group

Prepared state, retained locks, completion responsibilities and operational cautions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-prepare-transaction.html

[S157] Sagas (1987)

Hector Garcia-Molina and Kenneth Salem

Original paper on long-lived transactions, interleaving and application-defined compensation.

Consulted: 25 September 2026

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

[S158] Consumer Acknowledgements and Publisher Confirms

RabbitMQ project

Separate confirmation boundaries, delivery tags, redelivery and bounded unacknowledged work; unversioned documentation.

Consulted: 25 September 2026

https://www.rabbitmq.com/docs/confirms

[S159] Queues

RabbitMQ project

Ordering scope, multiple consumers, redelivery and queue properties; unversioned documentation.

Consulted: 25 September 2026

https://www.rabbitmq.com/docs/queues

[S160] Client-side caching reference

Redis

Invalidation races, connection loss and cache-resource bounds; unversioned documentation.

Consulted: 25 September 2026

https://redis.io/docs/latest/develop/reference/client-side-caching/

[S161] Cache-Aside pattern

Microsoft

Loading/invalidation trade-offs and consistency limits of derived caches.

Consulted: 25 September 2026

https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

[S162] Dynamo: Amazon's Highly Available Key-value Store (2007)

Giuseppe DeCandia and co-authors

Original Dynamo design: consistent hashing, versions, sloppy quorums and reconciliation; not a statement about current DynamoDB.

Consulted: 25 September 2026

https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[S163] The Chubby lock service for loosely-coupled distributed systems (2006)

Mike Burrows

Lock generations and recipient-validated sequencers for rejecting obsolete operations.

Consulted: 25 September 2026

https://storage.googleapis.com/gweb-research2023-media/pubtools/4444.pdf

[S164] PostgreSQL 17 documentation: Logical Replication Restrictions

PostgreSQL Global Development Group

DDL, sequence and object coverage boundaries and subscriber compatibility.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/logical-replication-restrictions.html

[S165] PostgreSQL connector documentation, stable page showing version 3.6 when consulted

Debezium project

Initial snapshot/change-stream boundary, offsets and connector failure behaviour; no connector deployment is performed.

Consulted: 25 September 2026

https://debezium.io/documentation/reference/stable/connectors/postgresql.html

[S166] PostgreSQL 17 documentation: Logical Decoding Concepts

PostgreSQL Global Development Group

Slots, retained log positions and possible change redelivery following a crash.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/logicaldecoding-explanation.html

[S167] Apache Beam Programming Guide: windows, watermarks, triggers and state

Apache Software Foundation

Event-time membership, late-data policy, triggers and accumulation modes; the small window model is original, not a Beam runtime.

Consulted: 25 September 2026

https://beam.apache.org/documentation/programming-guide/

[S168] Apache Iceberg table specification

Apache Software Foundation

Field IDs, snapshots, manifests, atomic metadata replacement and snapshot retention. Discussion is limited to these concepts; not an implementation of the entire evolving specification.

Consulted: 25 September 2026

https://iceberg.apache.org/spec/

[S169] PostgreSQL 17 documentation: Materialized Views

PostgreSQL Global Development Group

Stored query results and refresh/freshness trade-offs.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/rules-materializedviews.html

[S170] Apache Parquet: File Format

Apache Software Foundation

File metadata, row groups and column chunks; companion byte-layout model does not produce Parquet files.

Consulted: 25 September 2026

https://parquet.apache.org/docs/file-format/

[S171] Apache Parquet: Encodings

Apache Software Foundation

Plain, dictionary, run-length and delta encoding concepts; actual Parquet bitstream rules are distinct from the toy codec.

Consulted: 25 September 2026

https://parquet.apache.org/docs/file-format/data-pages/encodings/

[S172] Apache Parquet: Page Index

Apache Software Foundation

Optional statistics and offsets for conservative page skipping.

Consulted: 25 September 2026

https://parquet.apache.org/docs/file-format/pageindex/

[S173] SQLite FTS5 Extension

SQLite project

Tokenizers, phrase queries, external-content responsibilities and negative BM25 scores. Only basic available FTS5 facilities are exercised.

Consulted: 25 September 2026

https://www.sqlite.org/fts5.html

[S174] PostgreSQL 17 documentation: Full Text Search Introduction

PostgreSQL Global Development Group

Documents, lexemes and full-text query representation.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/textsearch-intro.html

[S175] PostgreSQL 17 documentation: Controlling Text Search

PostgreSQL Global Development Group

Parsing, normalization, query construction, ranking and safe display limitations.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/textsearch-controls.html

[S176] MetricType and distances

Faiss project

Squared Euclidean distance, inner product, cosine and normalization. The lab uses direct Python arithmetic, not Faiss.

Consulted: 25 September 2026

https://github.com/facebookresearch/faiss/wiki/MetricType-and-distances

[S177] Faiss indexes

Faiss project

Exhaustive versus approximate indexes, vector-memory estimates and index trade-offs.

Consulted: 25 September 2026

https://github.com/facebookresearch/faiss/wiki/Faiss-indexes

[S178] Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

Yu. A. Malkov and D. A. Yashunin

Original HNSW research, first submitted 2016; graph navigation concept, not a claim about every present product or workload.

Consulted: 25 September 2026

https://arxiv.org/abs/1603.09320

[S179] e-Handbook of Statistical Methods: Autocorrelation

NIST / SEMATECH

Dependence between observations across lags; repeated observations are not automatically independent.

Consulted: 25 September 2026

https://www.itl.nist.gov/div898/handbook/eda/section3/eda35c.htm

[S180] e-Handbook of Statistical Methods: Scatter Plot

NIST / SEMATECH

Association can be inspected graphically but does not establish cause. All business examples in the chapter are synthetic.

Consulted: 25 September 2026

https://www.itl.nist.gov/div898/handbook/eda/section3/scatterp.htm

[S181] Introduction to Information Retrieval: Evaluation of unranked retrieval sets

Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze

Authors-hosted textbook definitions of precision and recall; all local query judgements are synthetic.

Consulted: 25 September 2026

https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-unranked-retrieval-sets-1.html

[S182] Introduction to Information Retrieval: Okapi BM25, a non-binary model

Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze

Term-frequency saturation, document-length normalization and development-set tuning.

Consulted: 25 September 2026

https://nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html

[S183] e-Handbook of Statistical Methods: Confidence intervals for a proportion

NIST / SEMATECH

Wilson interval formula, binomial assumptions and small-sample cautions. Numerical examples in the book are original.

Consulted: 25 September 2026

https://www.itl.nist.gov/div898/handbook/prc/section2/prc241.htm

[S184] PostgreSQL 17 documentation: Preferred Index Types for Text Search

PostgreSQL Global Development Group

GIN inverted indexes and GiST signatures/rechecking; PostgreSQL examples are documented, not executed.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/textsearch-indexes.html

[S185] Authorization Cheat Sheet

OWASP Foundation

Default-deny authorization, permission checks and tests; not authentication implementation.

Consulted: 25 September 2026

https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

[S186] PostgreSQL 17: Privileges

PostgreSQL Global Development Group

Role privileges and object ownership.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-priv.html

[S187] PostgreSQL 17: Row Security Policies

PostgreSQL Global Development Group

Row-policy semantics, default denial and privileged bypass; not executed in the SQLite labs.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-rowsecurity.html

[S188] Multi-Tenant Security Cheat Sheet

OWASP Foundation

Tenant-aware authorization and isolation across storage and caches.

Consulted: 25 September 2026

https://cheatsheetseries.owasp.org/cheatsheets/Multi_Tenant_Security_Cheat_Sheet.html

[S189] Secrets Management Cheat Sheet

OWASP Foundation

Secret inventory, lifecycle, rotation and avoiding leakage.

Consulted: 25 September 2026

https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

[S190] PostgreSQL 17: Routine Vacuuming

PostgreSQL Global Development Group

Dead tuples, reuse of storage and cleanup; not proof of media sanitization.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/routine-vacuuming.html

[S191] PRAGMA secure_delete

SQLite project

Secure-delete configuration and limitations, including virtual tables and separate copies.

Consulted: 25 September 2026

https://www.sqlite.org/pragma.html#pragma_secure_delete

[S192] SP 800-88 Revision 2: Guidelines for Media Sanitization (September 2025)

NIST

Publication landing record and sanitization scope. Not a claim that a device was sanitized or the full publication independently audited.

Consulted: 25 September 2026

https://csrc.nist.gov/pubs/sp/800/88/r2/final

[S193] Object Model

OpenLineage project

Datasets, jobs, runs and metadata used to describe lineage.

Consulted: 25 September 2026

https://openlineage.io/docs/spec/object-model/

[S194] PostgreSQL 17: ALTER TABLE

PostgreSQL Global Development Group

Schema changes, constraint validation and lock behavior in the cited version.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-altertable.html

[S195] PostgreSQL 17: CREATE INDEX

PostgreSQL Global Development Group

Concurrent index creation restrictions and operational caveats.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-createindex.html

[S196] ALTER TABLE

SQLite project

Supported alterations and table-rebuild procedure; version-sensitive documentation.

Consulted: 25 September 2026

https://www.sqlite.org/lang_altertable.html

[S197] Service Level Objectives

Google SRE

Indicators, objectives and measurement boundaries.

Consulted: 25 September 2026

https://sre.google/sre-book/service-level-objectives/

[S198] Signals

OpenTelemetry project

Roles of traces, metrics, logs and related telemetry signals.

Consulted: 25 September 2026

https://opentelemetry.io/docs/concepts/signals/

[S199] Handling Overload

Google SRE

Load, saturation and overload-control considerations.

Consulted: 25 September 2026

https://sre.google/sre-book/handling-overload/

[S200] PostgreSQL 17: The Cumulative Statistics System

PostgreSQL Global Development Group

Engine-specific statistics and observation context.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/monitoring-stats.html

[S201] Python 3.13: unittest

Python Software Foundation

Test discovery, assertions, subtests and skipped-test semantics.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/unittest.html