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 C · Chapters 14–20 · First Edition

Finding Answers

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 C

Finding Answers

Follow a query through scans, indexes and plans, then measure the actual work.

This volume contains Chapters 14–20 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 C

Finding Answers

Follow a query through scans, indexes and plans, then measure the actual work.

14

CHAPTER

Scanning, Sorting and the Cost of a Question

14.0 What this chapter gives you

1. A query can return one number after examining millions of records. Another can return hundreds of records after examining only a small region of an index. Output size and work performed are not the same thing.
2. You will follow scans, estimate data volume, understand why a sort may need temporary storage, and separate an optimiser's cost estimate from elapsed time on a clock.
3. The large datasets and machine rates in this chapter are invented arithmetic exercises unless explicitly described as measured. They do not describe KedByte's infrastructure or a benchmark of any database product.

14.1 What a scan does

▪ 14.1.1 PLAIN — in simple words

1. A scan visits records through an access path and checks which ones contribute to the question. Without a more selective route, finding one matching order may require looking through a large collection.
2. Scanning is not inherently bad. To total almost every sale, visiting most of the relevant data may be exactly the right work. An index is useful only if its alternative route saves enough work for this question.
3. A limit on the answer does not necessarily limit the search. "Give me one order above an amount that no order reaches" may still require checking every eligible order.
4. Read a slow query as a sequence of responsibilities: find candidates, test conditions, combine records, calculate results, order them if required, and deliver them. The expensive stage may be nowhere near the final output.

▪ 14.1.2 PLAIN — a picture in your head

1. Mira asks Dev to find a paper order from a cupboard. If folders are arranged by order number but the only clue is a handwritten note inside, Dev may need to open many folders.
2. If Mira asks for the total number of sheets in the cupboard, opening every folder is not a failed search strategy; the question asks about all of them.
3. **Where the comparison breaks:** a database can read pages containing many records, keep pages in memory, process batches and use several workers. A scan is not necessarily one physical disk operation per row or one person walking slowly between folders.

14.1.3 PLAIN — a worked example

1. Imagine 100,000 synthetic orders, with one labelled `TARGET`. A simple linear search could encounter it first, last or somewhere between. A missing target requires examining the entire eligible collection in this elementary algorithm.
2. If the records are already ordered by a searchable key and the structure supports choosing regions, a lookup can skip much of that collection. Chapter 16 explains how a tree makes the skipped regions trustworthy.
3. Now ask for a sum over all 100,000 order amounts. A selective lookup for one key is no longer the answer. Unless an appropriate maintained summary is available and acceptable, the system must account for all contributing amounts.
4. Returning one SUM cell therefore does not establish that one row was read. The output has one aggregate row; its input may contain 100,000 contributions.

14.1.4 PLAIN — what is really happening inside

1. The engine chooses a physical plan that implements the query's logical result. A table scan and an index scan are different paths; each can still examine many entries.
2. A predicate may discard most candidates after they have been read. A narrow result can therefore hide a wide scan. Conversely, an index may narrow candidates before fetching full records.
3. Some engines can avoid reading entire partitions, pages or column blocks using metadata. This is a separate mechanism from discarding rows after reading them. Ask what was skipped, not just how many rows survived.
4. Work also includes checking visibility, interpreting values, evaluating expressions and passing rows between plan stages. A memory-resident scan can consume significant CPU even when it performs no device reads.

14.1.5 TECHNICAL — the engineer's version

1. Separate **access-path work**, **predicate evaluation** and **result cardinality**. SQLite's planning documentation illustrates full scans, rowid lookup and index-based lookup; its EXPLAIN QUERY PLAN distinguishes SCAN from selective SEARCH descriptions. The exact human-readable output is implementation-dependent. [S89] [S90]
2. SQL is declarative about the required result, with explicit ordering when requested. It does not prescribe the physical row-visit algorithm. A logically useful "first filter, then project" explanation is not a claim that an engine always executes those stages in that literal order. [S65]
3. `LIMIT` constrains returned rows after the semantics of the relevant query operations. It is not a universal cap on examined rows, input bytes, sorting work or total runtime. Whether an engine can stop early depends on the plan and the query. [S81]

14.1.6 WORDS — remember these

1. **Scan:** visit data through an access path — sequential or otherwise structured traversal of records or index entries to produce candidates. **Predicate:** a condition used to choose records — an expression whose truth controls qualification in a query operation. **Result cardinality:** how many output rows there are — the row count at a specified stage, distinct from records examined or bytes read.

14.2 Row width and data volume

14.2.1 PLAIN — in simple words

1. A million tiny records and a million records containing large messages create different amounts of work. Counting rows alone ignores how much information each visit carries.
2. Selecting only the fields needed can reduce the amount passed through later stages and sent to the caller. It does not guarantee that an engine avoids reading every other byte on the underlying page.
3. The representation matters. A short integer, a long description and a compressed document have different storage and processing costs. Metadata, alignment, indexes and older row versions can add further space beyond the values shown in a spreadsheet.
4. Start with a rough estimate, label its assumptions, and then measure the actual structure. Arithmetic is useful precisely when its boundaries are visible.

14.2.2 PLAIN — a picture in your head

1. Think of a delivery van carrying boxes. "We have 1,000 boxes" does not say whether one van is enough. The size and packing of the boxes matter.
2. Asking for only a page from each folder reduces what you carry to a meeting, but you may still have to open the same bulky folders to retrieve those pages.
3. **Where the comparison breaks:** databases use pages, compression and indirection. A large value may live outside the main row, and some plans can satisfy a query from a narrower index. The physical route must be inspected rather than inferred from the SELECT list alone.

14.2.3 PLAIN — a worked example

1. Suppose a synthetic table has 1,000,000 rows averaging 120 bytes of relevant stored payload. The payload estimate is `1,000,000 × 120 = 120,000,000` bytes, or 120 decimal MB.
2. At 600 bytes per row it becomes 600 MB, five times as much. This comparison excludes page headers, free space, indexes, transaction information and compression.
3. If a report only needs a 16-byte identifier and an 8-byte amount, its conceptual selected values total 24 MB across the million rows. That is a logical payload estimate, not a guarantee of 24 MB of physical input-output.
4. An assumed sustained stream of 300 MB/s would need at least 0.4 seconds merely to transfer 120 MB and 2 seconds for 600 MB. These are invented lower-bound arithmetic examples. Real execution also includes scheduling, decoding, computation, contention and delivery.

14.2.4 PLAIN — what is really happening inside

1. Row-oriented storage commonly groups a record's attributes physically, while columnar storage groups values by column. The amount a query can avoid reading differs between these layouts; Chapter 43 explains the columnar case.
2. Intermediate rows matter too. A join that carries a long unused description through a sort can consume more memory than one carrying only keys and amounts, even when both eventually return the same concise report.
3. Output encoding can expand values. JSON field names, quotation and text formatting are not free. A server may finish its internal calculation quickly while the application spends longer receiving and decoding the result.

4. A useful measurement distinguishes table size, index size, bytes visited by a plan, intermediate memory, temporary bytes and bytes delivered. One number called "data size" cannot stand in for all six.

14.2.5 TECHNICAL — the engineer's version

1. A plan's estimated row width is an estimate of the values passed by that plan node, not necessarily a physical on-disk tuple length. PostgreSQL EXPLAIN reports estimated rows and width alongside cost; those fields describe different dimensions. [S65]
2. Projection can reduce intermediate width and network payload. A covering access path may additionally avoid some base-table visits, but PostgreSQL index-only access still depends on visibility information and can require heap checks. [S92]
3. State units precisely. This chapter uses decimal MB for rate arithmetic: one MB is 1,000,000 bytes. A reported MiB is 1,048,576 bytes. Do not compare the two as though their numerals were interchangeable.
4. An estimate built from average width can miss heavy tails: a few very large records may dominate particular queries or exceed memory assumptions. Keep a distribution or representative cases when wide values matter.

14.2.6 WORDS — remember these

1. **Row width:** the amount carried by one row representation — estimated or measured bytes per row at a specified physical or logical stage. **Projection:** choose the fields needed — a relational or SQL operation producing selected expressions rather than every available attribute. **Payload:** the useful represented content — bytes under a stated boundary, excluding or including overhead only as explicitly defined.

14.3 Sorting and memory

14.3.1 PLAIN — in simple words

1. Sorting arranges records according to a comparison rule. A list ordered by amount is not necessarily ordered by order number, and ties need a rule when their relative order matters.
2. Sorting generally needs to compare and rearrange many values. A large sort may consume substantial memory even when the final answer contains only a small part of the sorted result.
3. An existing ordered access path can sometimes supply the requested order directly. That can avoid a separate sort, but only when the path's ordering matches the query and the plan chooses it.
4. More memory can help some sorts. It is not automatically the right first change, because many concurrent operations may each use their own allowance.

14.3.2 PLAIN — a picture in your head

1. Dev lays order cards across a table, arranging them from smallest amount to largest. If all cards fit, he can compare and move them without leaving the table.
2. Cards already filed in the requested order can simply be read in sequence. Cards filed by a different field still need work.
3. **Where the comparison breaks:** real sorting algorithms do not repeatedly scan the whole table for the smallest remaining card in the inefficient way a beginner might. The number of comparisons and moves depends on the algorithm, data and implementation.

14.3.3 PLAIN — a worked example

1. Take four invented report rows: `(A, 500)`, `(B, 300)`, `(C, 500)` and `(D, 200)`. Ordering by amount gives D, B, then the two 500 rows in an unspecified relative order unless another rule is supplied.
2. `ORDER BY amount_minor, order_id` supplies a deterministic order for these distinct identifiers: D, B, A, C. It states the business-readable tie-break rather than relying on a current storage accident.
3. A simplified comparison sort on one million entries has work on the scale of `N log2 N`, roughly 20 million comparison steps in an order-of-growth illustration. This is not an exact count for a database sorter.
4. If each temporary entry occupies an assumed 48 bytes, the entries alone need about 48 MB. Pointers, allocator overhead, copies and other plan nodes are outside this simple estimate.

14.3.4 PLAIN — what is really happening inside

1. A sort receives keys and associated values, compares them under the selected ordering, and emits the required sequence. It may retain full rows or references sufficient to retrieve the rest.
2. Sorting text requires a collation rule, not just visual intuition. Null placement and ascending versus descending order also belong to the requested ordering.
3. A top-N plan may keep a bounded set of the best candidates instead of sorting everything completely. It can still need to inspect all candidates when there is no ordered route that proves the remaining entries cannot outrank them.
4. A later join, parallel exchange or other operation can alter ordering guarantees. Only the ORDER BY governing the final result establishes the promised presentation order for that query.

14.3.5 TECHNICAL — the engineer's version

1. Distinguish the logical ORDER BY requirement from a physical Sort node. An appropriately ordered index scan can satisfy the requirement without that node, subject to the actual key order, directions, null handling and other query operations. [S106]
2. PostgreSQL's `work_mem` is an operation-level memory allowance used by operations such as sorts and hashes; multiple nodes and concurrent sessions can multiply total demand. Treating it as a single budget for the entire server is incorrect. [S95]
3. Comparison counts, temporary-entry widths and memory estimates in this section are analytical models. They do not establish observed latency, a cache state or a particular spill threshold for an installed engine.

14.3.6 WORDS — remember these

1. **Sort key:** the values deciding order — the ordered sequence of expressions and comparison rules used to arrange records. **Tie-breaker:** an extra rule for equal first choices — additional ordering attributes that distinguish otherwise tied results. **Top-N:** keep the requested best few — an optimisation that retains a bounded candidate set under an ordering, without necessarily avoiding a full input scan.

14.4 External work and spill

14.4.1 PLAIN — in simple words

1. When an operation cannot keep its working data within its memory allowance, it may move some of that work to temporary storage. This is called spilling.

2. A spill is not automatically a failure. It can be the engine's planned way to finish a large task safely. It may, however, add substantial input-output and compete with other operations for space and bandwidth.
3. Temporary does not mean harmless. Temporary files still need capacity, access controls and cleanup. A query can fail because temporary storage fills even when the final result would have been tiny.
4. Before changing memory settings, inspect whether spilling actually occurred, how much work it added and which concurrent operations share the machine.

■ 14.4.2 PLAIN — a picture in your head

1. The order cards no longer fit on Dev's table. He sorts one table-sized batch, puts it in an ordered stack on the floor, and repeats. Then he merges the ordered stacks by repeatedly taking the next smallest visible card.
2. The floor makes the task possible, but every trip between table and stacks costs effort.
3. **Where the comparison breaks:** database external sorting uses buffers, merge passes and storage-specific algorithms. The floor is not necessarily slower than every possible memory access, and a physical device may itself have caches. The analogy explains the extra movement, not a fixed speed ratio.

■ 14.4.3 PLAIN — a worked example

1. Use an intentionally simplified external-sort model: 1,000 MB of sortable data and room for 100 MB at a time. Initial run generation produces about ten sorted runs, ignoring overhead.
2. A one-pass merge that can read all ten runs concurrently would read the temporary runs and write or stream the final order. If the merge buffers cannot support that fan-in, extra merge passes may be needed.
3. Counting an initial read, a temporary write and a temporary reread already suggests roughly 3,000 MB of traffic for 1,000 MB of original input, before additional output and metadata. This is a model, not an engine measurement.
4. A smaller projection, a more selective predicate or an already suitable ordered path could reduce this work without increasing the per-operation memory allowance.

■ 14.4.4 PLAIN — what is really happening inside

1. The operation builds manageable runs or partitions, writes them to temporary storage, and combines them later. Sorting and hash-based processing use different mechanisms, so identify which operation spilled.
2. A server-wide memory increase can remove one spill while making the system more vulnerable to memory pressure under concurrency. The relevant question is the combined workload, not one isolated query's fastest run.
3. Temp-file sizes and sort-method descriptions can provide evidence. They do not by themselves explain whether the root cause was an oversized result, a poor plan, stale estimates, unexpected skew or an intentionally large analytical job.
4. Preserve a baseline before intervention. Record query text, parameter values or their safe profile, plan, row counts, settings and workload conditions. Change one justified variable, then compare the same defined task.

14.4.5 TECHNICAL — the engineer's version

1. External-memory algorithms explicitly account for movement between a limited fast working set and larger storage. An external merge sort forms sorted runs and merges them with a bounded fan-in. The chapter's traffic estimate deliberately excludes implementation overhead and extra passes.
2. PostgreSQL EXPLAIN ANALYZE can report sort method and memory or disk use; resource settings govern relevant operation allowances. EXPLAIN ANALYZE executes the statement, so use a disposable or explicitly authorised read-only workload rather than treating it as a passive parser. [S65] [S95]
3. A spill observation supports "this execution used temporary storage at this node." It does not establish a universal threshold, an exact device bottleneck or the best production configuration.

14.4.6 WORDS — remember these

1. **Spill:** working data moved out of the operation's memory allowance — temporary external storage used to continue an operation such as sorting or hashing. **Sorted run:** one already ordered batch — an intermediate sequence used by an external merge process. **Merge fan-in:** how many ordered streams are combined at once — the number of input runs a merge step can consume under its buffer and resource limits.

14.5 Selectivity and filtering

14.5.1 PLAIN — in simple words

1. A selective condition keeps a small share of the eligible records. "This exact order number" may be highly selective. "Every completed order this year" may not be.
2. The share matters because a selective access path can avoid visiting much unrelated data. But knowing that few rows survive does not prove the engine found them cheaply.
3. Real values are rarely spread perfectly evenly. One popular product may account for half the sales while thousands of other products share the remainder.
4. The same query shape can therefore require very different work with different parameters. Test common, rare and empty cases rather than choosing only the most flattering example.

14.5.2 PLAIN — a picture in your head

1. Mira asks for orders containing a rare specialist pen. A small product-to-order guide might lead to a handful of folders. Asking for orders containing the shop's best-selling notebook could lead to half the cupboard.
2. The guide still works in both cases, but following thousands of separate pointers may no longer beat reading the relevant cupboard section in order.
3. **Where the comparison breaks:** database cost depends on page locality, caches, correlation and implementation. A popular value does not force one universal plan, and an index need not perform one separate physical read per match.

14.5.3 PLAIN — a worked example

1. In a synthetic population of 100,000 rows, a condition retaining 100 has an observed selectivity of `100 / 100,000 = 0.001`, or 0.1 percent.

2. A second condition retains 60,000 rows, or 60 percent. These are measured counts in the invented population, not estimates of a real shop.
3. Suppose 10 percent of orders are from branch A and 20 percent are evening orders. Multiplying gives a 2-percent joint estimate only under an independence assumption. If branch A operates only in the evening, every branch-A order can be an evening order, giving 10 percent instead.
4. The calculation exposes an assumption rather than proving that a planner always multiplies percentages this way. Chapter 18 explains statistics and correlated attributes.

■ 14.5.4 PLAIN — what is really happening inside

1. The optimiser estimates how many rows each operation will produce. These estimates influence whether an index, scan, join order or aggregation strategy appears economical.
2. A predicate may be evaluated as an index boundary, a partition-pruning condition or a residual filter after reading candidates. These placements can produce the same logical result with different work.
3. Transforming a column inside a predicate can affect whether an ordinary index is usable. A suitable expression index or a query written in an equivalent searchable form may help, but equivalence must include type, collation and boundary semantics.
4. Filtering earlier is useful only when it preserves meaning. Section 13.3 showed a case where moving a predicate around an outer join changes the answer. Correctness comes before reduced row counts.

■ 14.5.5 TECHNICAL — the engineer's version

1. Selectivity is a fraction of a specified input population that satisfies a predicate. State whether the fraction is estimated by a planner or calculated from actual observed rows. Estimated and actual cardinalities can diverge under skew or correlated conditions. [S94]
2. An access predicate constrains which entries a chosen path visits; a residual filter rejects candidates after access. Implementation details vary, so examine the plan and observed work rather than assuming every WHERE clause is pushed into an index. [S98]
3. An optimisation must preserve SQL semantics, including NULL handling, outer-join behaviour, comparison rules and overflow or type-conversion boundaries. A faster query that silently changes the population is a different query, not a successful optimisation.

■ 14.5.6 WORDS — remember these

1. **Selectivity:** the fraction that passes a condition — qualifying rows divided by a stated candidate population, either estimated or observed. **Skew:** some values occur much more often than others — a non-uniform distribution that can concentrate work and invalidate uniform assumptions. **Residual filter:** a check made after finding candidates — a predicate evaluated after an access path has already visited candidate entries or rows.

14.6 Cost versus elapsed time

■ 14.6.1 PLAIN — in simple words

1. A planner's cost is a model used to compare candidate plans. It is not necessarily a prediction in seconds. A clock measures one execution under one set of conditions.
2. Two runs of the same plan can take different times because one uses cached data, waits for another operation, competes for CPU or sends results to a slower client.

3. A lower estimated cost is a reason the planner preferred a plan under its assumptions. It is not evidence that you measured a faster end-to-end experience.
4. Keep three questions separate: is the result correct; what work was performed; and how long did the defined operation take? All three are needed for a useful performance claim.

14.6.2 PLAIN — a picture in your head

1. A delivery planner gives each route points for distance, traffic and difficult turns. Those points help choose a route. A driver's stopwatch records what happened on Tuesday.
2. Tuesday's unexpected roadworks can make the chosen route slow without changing the meaning of the planner's points.
3. **Where the comparison breaks:** a database cost model uses its own configurable units and statistics. The analogy does not license treating a cost of 500 as 500 metres, milliseconds or physical reads.

14.6.3 PLAIN — a worked example

1. Consider a purely invented plan comparison. Plan A is assigned cost 100 and plan B cost 160. The planner chooses A. These figures have meaning only inside the stated model.
2. A measured run of A takes 40 ms of server work and waits 120 ms for a contended resource. A second run without the wait takes 42 ms. It would be misleading to announce a fourfold algorithmic improvement between those runs.
3. Now include a client that transfers and decodes results in another 30 ms. A server-only measurement and an application-visible measurement answer different questions even if both are honestly timed.
4. The teaching benchmark will therefore compare equal result sets, record the timing boundary, repeat runs and retain failures instead of presenting only the fastest observation.

14.6.4 PLAIN — what is really happening inside

1. Planning estimates work from statistics and configured assumptions. Execution discovers actual row counts, waits and resource use. Instrumentation can expose some of these observations, with overhead of its own.
2. Cached pages are not the only warm state. Prepared statements, filesystem cache, compiled code, connection pools and storage caches may all change a later run. Closing one connection does not necessarily reset them.
3. Parallel execution can reduce elapsed time while doing more total work. Conversely, less total work can still take longer if it lies on a serial critical path or waits behind other tasks.
4. The useful report records environment and limitations: database and driver versions, dataset construction, indexes, query and parameters, concurrency, warm-up policy, number of samples and what the timer includes.

14.6.5 TECHNICAL — the engineer's version

1. PostgreSQL EXPLAIN costs are expressed in planner cost units, not milliseconds. EXPLAIN ANALYZE adds observed execution information and incurs measurement overhead; actual rows and loops must be read at the relevant node. [S65]
2. Distinguish latency, service time, queueing delay, CPU time and throughput. They are related but not interchangeable. A single duration cannot identify all bottlenecks.

3. A monotonic performance timer is appropriate for elapsed intervals. Python's `perf_counter_ns()` returns integer nanoseconds, but the unit does not guarantee one-nanosecond resolution or accuracy. Record the timer and runtime. [S102]
4. This chapter establishes a vocabulary and analytical models. No fabricated plan output, elapsed-time table or hardware result is presented as a measured experiment. Chapter 20 develops the actual benchmark protocol.

14.6.6 WORDS — remember these

1. **Planner cost:** an estimate for comparing execution choices — a model-specific quantity derived from statistics and configured operation costs. **Elapsed time:** how long the chosen boundary took on a clock — wall-clock duration including waits within that boundary. **Queueing delay:** time waiting before work can proceed — delay caused by contention or scheduling rather than the operation's own active computation.

14.97 Practice and worked answers

1. **Question:** A query returns one COUNT value after checking two million rows. Is it a one-row query for capacity planning? **Answer:** Its result cardinality is one, but the input work may involve two million candidates. Report both the output and examined work.
2. **Question:** Estimate payload for 250,000 rows averaging 200 bytes. **Answer:** 50,000,000 bytes, or 50 MB. This excludes overhead and does not establish physical bytes read.
3. **Question:** A limit of ten returns no matching rows. Can the engine have visited the entire table? **Answer:** Yes. Without a suitable route or metadata proving absence earlier, it may need to reject every candidate.
4. **Question:** Why is `ORDER BY amount_minor` insufficient for repeatable pagination among tied amounts? **Answer:** Equal amounts do not specify a total ordering of distinct rows. Add a stable identifying tie-breaker and consider changes between page requests.
5. **Question:** A sort spills. Should every session immediately receive ten times more working memory? **Answer:** No. First establish the operation, data width, plan and concurrent memory demand. A local improvement can create server-wide pressure.
6. **Question:** A condition selects 3,000 of 120,000 candidates. **Answer:** Observed selectivity is 2.5 percent. State that population rather than saying "the query is 2.5 percent selective" without a denominator.
7. **Question:** Two independent predicates have selectivities 0.1 and 0.2. What is their joint fraction under independence, and what can invalidate it? **Answer:** 0.02. Correlation, different eligible populations or conditional definitions can invalidate the multiplication.
8. **Question:** A plan cost is 80 and measured duration is 40 ms. Is the conversion two cost units per millisecond? **Answer:** No. One observation does not define such a conversion; planner units and measured time are different quantities.

14.98 Common wrong ideas

1. **Wrong:** a scan always means a missing index. **Right:** broad questions can appropriately scan broad inputs.
2. **Wrong:** few returned rows imply little work. **Right:** filtering and aggregation can hide extensive input work.

3. **Wrong:** selected column width equals physical input-output. **Right:** storage layout, visibility and access paths affect physical reads.
4. **Wrong:** a spill proves the engine malfunctioned. **Right:** it can be an expected strategy with measurable costs.
5. **Wrong:** all values have similar frequency. **Right:** skew and correlations can dominate workload behaviour.
6. **Wrong:** a cost is a time prediction in milliseconds. **Right:** cost is a planner-specific comparison model.
7. **Wrong:** closing a connection creates a cold benchmark. **Right:** many other caches and warm states remain.
8. **Wrong:** the fastest observed run is the system's performance. **Right:** workload distributions, failures and measurement boundaries matter.

14.99 Chapter summary in 20 lines

1. A question's answer size does not reveal how much work produced it.
2. Scans visit candidates through a chosen access path.
3. A full scan can be appropriate for a broad calculation.
4. LIMIT does not universally cap input work.
5. Count bytes and intermediate width as well as rows.
6. Projection can reduce carried and delivered data without guaranteeing fewer page reads.
7. Physical layout and cached state affect the route to values.
8. Sorting requires a defined comparison and tie policy.
9. A suitable ordered path can sometimes avoid a separate sort.
10. A top-N operation may still inspect its whole input.
11. Working-memory limits can lead to temporary-storage spill.
12. Spills add movement and capacity requirements, not automatically incorrectness.
13. Per-operation allowances multiply under complex plans and concurrency.
14. Selectivity needs a named input population.
15. Popular values and correlated conditions challenge uniform estimates.
16. Moving a filter must preserve the query's meaning.
17. Planner cost and measured elapsed time are different quantities.
18. Warm state, waits and clients can change observed duration.
19. Record correctness, work and time separately.
20. Use reproducible measurements rather than slogans about scans or indexes.

15

CHAPTER

Indexes: Building a Faster Way to Find Things

15.0 What this chapter gives you

1. An index is an extra structure maintained to support particular ways of finding records. It is not a switch that makes every database operation faster.
2. You will choose candidate indexes from actual questions, trace composite key order, distinguish covering and partial indexes, and explain why a planner can reasonably ignore an available index.
3. The canonical shop remains tiny. It is useful for checking answers, not for proving large-system speed. Larger datasets in the companion exercises are separately generated teaching data, with their construction recorded.

15.1 An extra access path

15.1.1 PLAIN — in simple words

1. An index keeps selected information arranged so that certain questions can reach useful records without inspecting everything. It resembles a guide, but it is part of the database's maintained state.
2. A guide arranged by order number helps with order-number questions. It may do little for a search based on a note buried inside a description.
3. The guide must still lead to the right records. An index key and the rest of a row can live in different structures, so finding an entry is not always the end of the work.
4. Different index designs support different operations. Equality, ordered ranges, text search and geometric proximity are not interchangeable questions.

15.1.2 PLAIN — a picture in your head

1. A book's alphabetical index maps a topic to pages. It can save opening every page to find "transactions." The contents page is another guide, organised by the book's hierarchy rather than by topic.
2. Neither guide replaces the chapters. A useful entry must identify where the explanation actually appears.
3. **Where the comparison breaks:** a printed index is fixed after publication. A database index often changes as records are inserted, updated or removed. It also contains machine comparison keys and record locators rather than human topic descriptions.

15.1.3 PLAIN — a worked example

1. The shop asks, "Which agreed lines refer to P-NOTE?" The original four lines contain two matching line records, representing three units. Keep the distinction between matching rows and units.

```
CREATE INDEX order_lines_product_idx
ON order_lines(product_id);

SELECT order_id, line_no, quantity, unit_price_minor
FROM order_lines
WHERE product_id = ?
ORDER BY order_id, line_no;
```

2. This SQLite example uses a bound parameter for `P-NOTE`. It creates an additional candidate access path in a disposable practice database; the tiny table may still be scanned.
3. The correct result must not change after index creation. Compare complete ordered result rows before discussing the plan or time.
4. An index on `product_id` does not itself promise the final order by order identifier and line number. That ordering remains an explicit requirement of the query.

15.1.4 PLAIN — what is really happening inside

1. An index stores keys in a form suited to a supported operation. Entries also contain, or imply, enough identity to reach the corresponding data according to the engine's layout.
2. A lookup navigates the structure, obtains candidates, applies required checks and retrieves additional fields when needed. The route can be shorter than a broad scan but still involve several steps.
3. Index structure is engine-specific. A SQLite rowid-table secondary index and a PostgreSQL heap index do not use identical record-location and visibility machinery. Chapter 16 separates the common ordered-tree idea from those details.
4. An index can support enforcement as well as access. Unique indexes help enforce uniqueness in specific engines, but the business meaning and NULL rules still come from the chosen schema and product behaviour.

15.1.5 TECHNICAL — the engineer's version

1. Treat an index as a maintained access method with an operator contract. PostgreSQL offers B-tree, hash, GiST, SP-GiST, GIN and BRIN methods for different classes of operations; the mere presence of an index says little without its method and supported operators. [S104]
2. A secondary access path can return candidate record locations rather than a complete final result. Rechecks, visibility checks and additional field retrieval may remain. SQLite and PostgreSQL documentation describe their own layouts; do not transplant one engine's assumptions into another. [S89] [S92]
3. CREATE INDEX is a schema-changing operation. Use disposable databases for the exercises. Building an index on a live table has concurrency, space and operational consequences not established by this four-row demonstration.

15.1.6 WORDS — remember these

1. **Index:** an extra route to selected records — a maintained data structure supporting specified search, ordering or constraint operations. **Access method:** the machinery behind a route — an engine-defined algorithm and operator interface used to organise and search index entries. **Sec-**

ondary index: an additional searchable organisation — an access path distinct from the primary physical or identifying organisation of the table.

15.2 Read benefits and write costs

15.2.1 PLAIN — in simple words

1. A useful index can reduce work for reads. Keeping it correct adds work to changes that affect its entries.
2. Every extra structure also occupies space, participates in backup and maintenance, and competes for memory when used. An unused index can therefore cost something even while no query benefits from it.
3. A write does not always update every index in exactly the same way. Which fields change and how the engine stores versions matter. Avoid the opposite oversimplification that one row update always performs one identical operation per index.
4. Evaluate an index against a workload: how often questions run, what they need, how often data changes, and which delays are acceptable.

15.2.2 PLAIN — a picture in your head

1. Dev maintains three paper guides to the same folders: by order number, customer and product. Looking up a folder can be easier, but a new order now requires the appropriate entries in several guides.
2. A guide nobody consults still takes shelf space and must not be allowed to point to nonexistent folders.
3. **Where the comparison breaks:** database maintenance is coordinated by an engine and can use batching, page splits, logging and versioning. The cost is not simply "three guides equals three times the duration," and an engine may avoid some index work for eligible updates.

15.2.3 PLAIN — a worked example

1. Use a hypothetical workload of 50,000 product lookups and 2,000 line inserts per hour. Suppose measurements in a particular test show the candidate index saves 200 microseconds per lookup but adds an average 20 microseconds to each insert under that test's conditions.
2. The simplified saved read time is `50,000 × 200 µs = 10 seconds` of summed operation durations per hour. The extra insert duration is `2,000 × 20 µs = 0.04 seconds`.
3. This arithmetic is not a production recommendation. The numbers are invented, summing durations does not directly yield wall-clock capacity, and storage, tail latency, contention and index-build cost are omitted.
4. Now reverse the mix: few lookups and millions of writes. The same local trade-off can become unattractive. Requirements, not the word "index," decide what deserves measurement.

15.2.4 PLAIN — what is really happening inside

1. Insertions add index entries. Updates to indexed values may replace or add representations. Deletions and version cleanup remove or retire entries according to the storage engine.
2. Ordered indexes may need page splits when a page has insufficient room. Wider keys and included payload can reduce entries per page, increasing size and affecting cache behaviour.

3. Maintenance includes more than immediate writes. Vacuuming or cleanup, rebuilding when justified, statistics collection and monitoring all have resource costs. The exact operations depend on the engine.
4. Removing an index can also remove an enforcement mechanism or harm a less frequent but important query. Inspect dependencies and workload coverage before treating "rarely observed" as "safe to delete."

15.2.5 TECHNICAL — the engineer's version

1. Read amplification, write amplification and space amplification are separate ratios whose baselines must be specified. An index can improve one while worsening another. Do not report a percentage without stating the counted operations or bytes.
2. PostgreSQL's B-tree implementation includes page management and version-related maintenance details; covering payload increases index size. These observations motivate testing, not a universal write-cost multiplier. [S105] [S92]
3. A uniqueness-supporting structure is not merely a performance accessory. Changing or removing it requires preserving the intended constraint through an explicitly reviewed alternative, not just comparing SELECT timings.
4. Benchmark both the intended benefit and affected write paths, including representative skew and concurrency. A read-only microbenchmark establishes only its read-only result.

15.2.6 WORDS — remember these

1. **Write amplification:** extra work caused by one logical change — a ratio of physical or internal writes to a clearly defined logical write baseline. **Index maintenance:** keeping the extra route correct — updates, cleanup and related work required as the underlying data changes. **Workload mix:** the combination of operations a system serves — their frequencies, parameters, sizes, concurrency and correctness requirements.

15.3 Composite key order

15.3.1 PLAIN — in simple words

1. A composite index organises entries using more than one field. Their order matters because the first field groups entries before the second field orders within each group.
2. An index on `(branch_id, order_date)` is naturally useful for dates within one branch. It is not physically identical to `(order_date, branch_id)`.
3. This does not mean a later column can never help. Engines have additional strategies, and an index may still be useful for other reasons. The safe statement is that leading-key constraints strongly influence the part of an ordinary ordered index that can be narrowed efficiently.
4. Choose an order from concrete questions, not an alphabetic sorting of column names.

15.3.2 PLAIN — a picture in your head

1. A telephone directory sorted by surname and then given name groups everyone named Singh together. Looking for a specific surname gives you a clear region. Looking for everyone named Dev across all surnames is a different task.
2. A directory sorted by given name first would reverse those conveniences.
3. **Where the comparison breaks:** database optimisers may combine indexes, scan narrower index entries, use skip-like strategies in particular versions, or choose other access methods. The di-

rectory analogy explains lexicographic grouping, not a prohibition on all alternative execution techniques.

15.3.3 PLAIN — a worked example

1. Consider invented keys `(A, 1)`, `(A, 3)`, `(A, 8)`, `(B, 2)`, `(B, 7)` in an index ordered by branch and day.
2. The condition `branch = A AND day >= 3` identifies a contiguous suffix within A's region: `(A, 3)` and `(A, 8)`.
3. The condition `day = 7` without a branch does not describe one simple leading-key region in this ordering. The matching key lies inside B's group; more branches would create more groups to consider.
4. For the real four-line fixture, an index on `(product_id, order_id, line_no)` groups by product and orders matching product entries by the remaining key. Whether it removes a sort for a particular query should be checked in that engine's actual plan.

```
CREATE INDEX order_lines_product_order_idx
ON order_lines(product_id, order_id, line_no);
```

5. This is a candidate for the exercise, not an instruction to keep both this index and every earlier overlapping index permanently.

15.3.4 PLAIN — what is really happening inside

1. Ordered tuple comparison examines the first differing field. Equal leading fields allow the search to focus on a narrower set of later-field values.
2. A range condition can widen the region compared with an equality. Additional conditions may still reject entries, but they do not necessarily reduce the initial visited interval in the same way.
3. Sorting requirements add another dimension. A mixture of ascending and descending directions, null placement and collation must match the query's required ordering before an index can be assumed to supply it.
4. Redundant-looking indexes are not automatically redundant in performance: width, uniqueness, predicates and included columns differ. Conversely, several overlapping indexes may create avoidable maintenance. Inspect purpose and measured workload, not names alone.

15.3.5 TECHNICAL — the engineer's version

1. For the explicitly cited PostgreSQL 17 B-tree behaviour, equality constraints on leading columns and a range condition on the first subsequent non-equality column are central to limiting the scanned portion. Constraints farther right can still be checked; their usefulness is not equivalent to leading equality. [S91]
2. This is version- and method-specific guidance, not a timeless claim that every database can only use a leftmost prefix. SQLite describes its own composite-index and planning rules. [S89] [S98]
3. An index's lexicographic order and the final ORDER BY are separate contracts. Confirm the actual plan and retain the explicit final ordering even when a current execution happens to emit the desired sequence. [S106]

15.3.6 WORDS — remember these

1. **Composite index:** one index built from several fields — an access structure over tuples of expressions in a specified order. **Leading key:** the first part of an ordered index key — the prefix that

groups entries before later components are compared. **Lexicographic order:** compare the first difference — tuple ordering determined by the earliest component on which two tuples differ.

15.4 Covering and partial indexes

15.4.1 PLAIN — in simple words

1. A covering index contains the values a particular query needs, so the engine may be able to answer using the index without retrieving every full table row. Covering is relative to a query, not a permanent label that covers all possible questions.
2. A partial index contains only records satisfying a stated condition. It can be smaller than an index of the whole table and useful for a frequently queried subset.
3. These are different ideas. One concerns which values an entry carries; the other concerns which records receive entries.
4. Both create obligations. Wider entries cost space and maintenance. A partial index helps only when the engine can establish that its contents are sufficient for the query.

15.4.2 PLAIN — a picture in your head

1. A quick-reference card contains an order number and the small amount field a clerk needs. The clerk can answer some questions from the card without fetching the entire folder. That is the covering idea.
2. A separate guide listing only unfinished orders is the partial-index idea. It is useful for a question about unfinished work, not for a complete history of all orders.
3. **Where the comparison breaks:** the database still has transaction-visibility rules. Having the requested values in an index does not always establish that the entry is visible to the current transaction without another check. Nor does an index independently decide what "unfinished" means.

15.4.3 PLAIN — a worked example

1. In a new disposable table for this exercise, define `tasks(task_id, branch_id, status, created_at)`, with status restricted to `open` or `closed`. These are invented work items, not extra canonical sales.

```
CREATE INDEX open_tasks_by_branch
ON tasks(branch_id, created_at, task_id)
WHERE status = 'open';
```

2. The partial index omits closed tasks. A query whose conditions include `status = 'open'` and a branch can be a candidate user of this index. A query for every task in that branch cannot rely on it alone because closed tasks are missing.
3. A query returning only branch, creation time and task identifier may also be covered by these indexed fields in an engine and plan that can use them that way.
4. In PostgreSQL, non-key payload can be expressed with `INCLUDE`, for example `ON tasks(branch_id, created_at) INCLUDE (task_id) WHERE status = 'open'`. This is PostgreSQL syntax, not SQLite syntax, and it is documented here rather than executed in the SQLite lab. [S92]

15.4.4 PLAIN — what is really happening inside

1. Partial-index maintenance evaluates its predicate as relevant records change. A task moving from open to closed leaves the indexed subset; moving back can add it again.

2. To use a partial index safely, the planner needs to know that every qualifying query row belongs to the indexed subset. Superficial similarity between two predicates is not a proof of that implication.
3. A parameterised query may not expose enough information at the relevant planning stage for a particular partial index to be chosen. Behaviour depends on the engine, parameter values and plan strategy. Do not replace safe binding with string concatenation merely to force a desired plan.
4. Covering can reduce full-row retrieval, but carrying a large text payload in an index can enlarge it enough to offset the benefit. Measure the chosen query and affected writes together.

■ 15.4.5 TECHNICAL — the engineer's version

1. SQLite's partial-index documentation frames eligibility as an implication between the query condition and the index predicate, using specific supported proof rules rather than a general theorem prover. Predicate spelling and supported forms can therefore matter operationally. [S93]
2. PostgreSQL index-only scans need both available index values and a visibility route that avoids unnecessary heap checks; visibility-map state affects actual heap visits. An eligible plan is not a guarantee of zero heap access in every execution. [S92]
3. Key columns determine search and ordering properties. Included payload is not automatically an additional search-key component. State this distinction when proposing a covering design.
4. Keep unique constraints separate from a convenient filtered read path. A partial unique index, where supported and appropriate, enforces uniqueness only over its qualifying subset; it does not silently extend that rule to omitted rows.

■ 15.4.6 WORDS — remember these

1. **Covering index:** the guide carries the values this question needs — an index containing sufficient query attributes for an eligible index-only or covering access path. **Partial index:** a guide for a defined subset — an index whose entries include only rows satisfying its predicate. **Included column:** payload carried alongside search keys — a non-key attribute stored in an index by an engine supporting that feature.

15.5 Selectivity and distribution

■ 15.5.1 PLAIN — in simple words

1. An index is often most useful when it leads to a small, useful part of the data. But the shape of that part matters as well as its size.
2. One thousand records close together can be cheaper to retrieve than one thousand scattered records. A narrow index can sometimes be cheaper to scan than a wide table even when it does not eliminate many entries.
3. A field with only two possible values can still support a useful index when one value is rare and important. "Never index a Boolean" is too broad; "every filter needs an index" is equally unreliable.
4. Choose representative parameter values. A plan that excels for a rare product may be poor for the best seller, and one average can hide both behaviours.

■ 15.5.2 PLAIN — a picture in your head

1. A cupboard guide points to 100 folders. If all 100 sit on the same shelf, collection is easy. If each sits in a different room, the same count creates more movement.

2. A two-colour label can also be useful if only ten folders in a million are marked red and the task concerns those ten.
3. **Where the comparison breaks:** an engine can batch accesses, cache pages and reorganise work. Physical clustering, logical key order and device latency interact; the cupboard does not supply a universal cost equation.

15.5.3 PLAIN — a worked example

1. In a synthetic million-row task collection, 999,000 tasks are closed and 1,000 are open. The open subset is 0.1 percent. A partial index on open tasks has a plausible purpose.
2. Six months later in a different hypothetical operating state, 800,000 tasks are open. The same predicate now covers 80 percent. A design justified only by the first distribution needs review.
3. Suppose a report needs every open task's large description. A tiny key-only index may find the tasks but still require many full-row visits. If the report needs only indexed identifiers, the retrieval cost can be quite different.
4. These are workload scenarios, not predicted query times. Record observed counts and payloads when testing them.

15.5.4 PLAIN — what is really happening inside

1. Planner statistics summarise distributions rather than storing every possible answer to every possible condition. Frequent values, distinct counts and correlations inform estimates, with sampling limitations.
2. Data can change faster than those summaries. Bulk loads, unusual seasonal activity or a new dominant value can make prior estimates less representative.
3. Parameter values can select radically different populations. A reusable query shape is not necessarily a single stable cost profile.
4. An index experiment should therefore include rare values, frequent values, absent values and boundary ranges. For composite conditions, include correlated cases rather than assuming independent fields.

15.5.5 TECHNICAL — the engineer's version

1. Distinguish logical selectivity, physical locality and requested projection. Each affects the relative cost of index access versus a scan. A low distinct-value count alone does not determine whether an index is useful. [S89] [S94]
2. Statistics are estimates with a collection policy. PostgreSQL ANALYZE gathers statistics used by the planner; it does not prove that every future parameter combination is modelled exactly. Extended statistics can address selected relationships, not arbitrary unknown business correlations. [S107] [S94]
3. Avoid overfitting an index portfolio to a single benchmark sample. Record the supported workload envelope, including growth and skew assumptions, and define when to reassess it.

15.5.6 WORDS — remember these

1. **Locality:** useful records are close in the accessed representation — spatial or temporal concentration that can reduce repeated movement or improve cache reuse. **Frequent value:** one value appears in many records — a high-frequency member of a distribution relevant to selectivity es-

timation. **Workload envelope:** the conditions a design was evaluated for — declared ranges of size, skew, query mix, concurrency and service requirements.

15.6 When the engine ignores an index

15.6.1 PLAIN — in simple words

1. An index is an available option, not an instruction that every matching-looking query must use it. The engine can choose a scan when the estimated alternative is cheaper or when the index cannot satisfy the required operation.
2. A tiny table is a common case where navigating an extra structure may not help. Our four-row shop is deliberately a correctness fixture, not a test that every index must be selected.
3. A mismatched expression, comparison rule, partial predicate or leading-key condition can also limit usefulness. The right investigation is specific: what query, what values, what index, what plan, what observations?
4. Forcing a plan before understanding the reason can replace one bad assumption with another. First check that the result and workload definition are right.

15.6.2 PLAIN — a picture in your head

1. Dev can see all four order folders on a desk. Looking up their positions in a separate guide would be unnecessary overhead. He reads the desk directly.
2. With a million folders, the guide may become valuable. But not if the requested clue is a field the guide does not organise.
3. **Where the comparison breaks:** the optimiser uses estimates and rules, not a person's judgement. It can be wrong because its information or model is wrong. A chosen scan is neither automatic wisdom nor automatic failure.

15.6.3 PLAIN — a worked example

1. Run the same canonical product query before and after creating the candidate index. Save ordered result rows in both cases. They must agree exactly.
2. In SQLite, run `EXPLAIN QUERY PLAN` with the same bound value and inspect whether the plan describes a table scan or index search. Save the actual output alongside the SQLite version instead of copying a predicted string from this book.
3. Then repeat on a separately generated larger dataset whose product-frequency distribution is recorded. Compare rare and common products. A changed plan is an observation about those inputs, not proof that the index is universally beneficial.
4. Do not assert a speedup from a plan label alone. Time repeated equivalent tasks, include result retrieval, and report the selected boundary and limitations as Chapter 20 specifies.

15.6.4 PLAIN — what is really happening inside

1. The planner considers legal access paths and estimates their costs. Some paths are ineligible because they do not implement the required semantics. Others are legal but not selected.
2. Diagnose those two categories separately. An eligibility problem may involve an expression or predicate mismatch; a cost-estimation problem may involve stale statistics or unexpected distribution.

3. Rewriting a query can expose a useful path, but only a semantics-preserving rewrite is valid. Replacing a substring search with an equality is not an optimisation of the same question.
4. Retain a rollback plan for schema experiments. New indexes consume space and can affect writes; removing them requires confirming that they are not supporting constraints or other workloads.

15.6.5 TECHNICAL — the engineer's version

1. SQLite EXPLAIN QUERY PLAN is an interactive diagnostic interface whose text can change between releases. Tests should not treat one exact human-readable plan string as a universal compatibility contract. [S90]
2. Compare estimates with observed rows and resource evidence where the engine exposes them. PostgreSQL EXPLAIN ANALYZE executes the query; use its output within an authorised experiment, not as a supposedly read-only plan parser for arbitrary statements. [S65]
3. An index recommendation should state target queries, key order, predicates, payload, expected distribution, write impact and measured evidence. "Add an index on every WHERE column" omits almost all of that reasoning.

15.6.6 WORDS — remember these

1. **Eligible access path:** a route that can implement the required semantics — an index or scan option permitted by the query, operators and engine rules. **Plan selection:** choosing among legal routes — the optimiser's decision based on estimated costs and available information. **Plan regression:** a changed execution choice worsens a defined workload — an observed performance deterioration, not merely a different plan diagram.

15.97 Practice and worked answers

1. **Question:** The product index is not used for the four-row canonical table. Is the test failed? **Answer:** Not on that fact alone. Correctness requires unchanged results. Performance and plan selection depend on the tiny input and actual engine; the fixture was not designed to require a search plan.
2. **Question:** Which ordering naturally groups all records for one branch before dates within that branch? **Answer:** `(branch_id, date)`. The reverse order groups by date first and supports a different set of convenient ranges.
3. **Question:** Can an index containing only open tasks answer a query for all tasks by itself? **Answer:** No; closed rows are absent. The query must be restricted to the indexed subset or use another route that supplies missing records.
4. **Question:** A covering index contains the selected values. Does PostgreSQL always avoid all heap visits? **Answer:** No. Transaction visibility can require heap checks; inspect actual execution evidence rather than assuming eligibility guarantees zero visits.
5. **Question:** Why might adding a large description as index payload be counterproductive? **Answer:** It widens entries, increases storage and maintenance, and may reduce cache efficiency. The saved row retrieval must be measured against those costs.
6. **Question:** A Boolean field is true in ten of a million records. Is an index automatically useless? **Answer:** No. The rare subset may justify a selective or partial path. Its usefulness still depends on queries, payload, maintenance and plans.

7. **Question:** A faster query returns fewer rows because it replaces an outer join with an inner join. Has an index problem been solved? **Answer:** Not for the original question. The population changed; correctness must be reconciled before performance comparisons are meaningful.
8. **Question:** What evidence should accompany a proposed production index? **Answer:** Defined target queries and distributions, result-equivalence checks, actual plans, repeated measurements, write and space impact, operational build considerations, and a reviewed rollback or removal procedure.

15.98 Common wrong ideas

1. **Wrong:** an index makes every operation faster. **Right:** it trades maintenance and space for specific access benefits.
2. **Wrong:** the order of composite fields does not matter. **Right:** ordered keys group and compare fields in sequence.
3. **Wrong:** a covering index covers the entire table for every query. **Right:** coverage is relative to requested values and engine behaviour.
4. **Wrong:** a partial index is merely a smaller copy with no semantic consequences. **Right:** omitted rows limit which questions it can answer.
5. **Wrong:** few distinct values always make an index useless. **Right:** skew can make one subset rare and important.
6. **Wrong:** a chosen scan proves the optimiser ignored common sense. **Right:** it may be appropriate, or an estimate may need investigation.
7. **Wrong:** a plan name is a benchmark result. **Right:** plans describe execution choices; measurements establish observed performance.
8. **Wrong:** an unused index can always be dropped. **Right:** constraints and unobserved important workloads may depend on it.

15.99 Chapter summary in 20 lines

1. An index is a maintained additional access path.
2. Its method and operators determine which questions it supports.
3. Correct result rows must remain unchanged after index creation.
4. The four-row shop proves arithmetic and semantics, not large-system speed.
5. Indexes occupy space and add maintenance obligations.
6. Read benefits and write costs depend on the workload mix.
7. Composite-key order determines how entries are grouped.
8. Leading equality and later range conditions have different effects on ordered access.
9. Engine and version details prevent universal left-prefix slogans.
10. Final result order still needs an explicit ORDER BY.
11. Covering concerns the values available for a particular query.
12. Partial indexing concerns the subset of records represented.
13. Visibility checks can remain even when values are covered.
14. Partial-index eligibility requires a justified predicate relationship.
15. Selectivity, locality and projection are different cost dimensions.
16. Popular and rare values can require different plans.

17. Statistics inform estimates without guaranteeing every future workload.
18. An eligible index need not be the selected access path.
19. Diagnose semantics, eligibility, estimates and observed work separately.
20. Recommend indexes with evidence, operational costs and explicit boundaries.

16

CHAPTER

B-Trees: Ordered Pages, Fast Lookups

16.0 What this chapter gives you

1. You will build a small ordered tree by hand, follow a key from its root to a leaf, split a full page, and read a range across neighbouring leaves.
2. The example is an original teaching B+ tree with deliberately tiny capacities. It is not the file format or complete concurrency protocol of SQLite, PostgreSQL or another engine.
3. By the end, "the index found it quickly" will mean something concrete: a sequence of comparisons that eliminates regions while preserving an ordering invariant. You will also know which costs that explanation leaves out.

16.1 Ordered separators

▪ 16.1.1 PLAIN — in simple words

1. An ordered tree stores guides to smaller regions. A guide says which child region can contain the key you want. Each decision removes regions that cannot contain it.
2. The guide works because the keys obey an ordering rule. Without that rule, skipping a region could skip the answer.
3. Our teaching tree keeps actual key entries in leaves. Internal pages contain separator keys and child references. A separator is a boundary, not necessarily another stored record to return.
4. Every path from the root to a leaf has the same number of levels in the balanced tree we use. This prevents one unfortunate branch from becoming a long chain while others remain short.

▪ 16.1.2 PLAIN — a picture in your head

1. A large archive has a sign: numbers below 25 are in the left room; numbers 25 and above are in the right room. Inside a room, another sign may divide the range further.
2. You do not open the wrong room because the signs and filing rules agree. The value 25 belongs to the right side because that is the boundary convention we chose.
3. **Where the comparison breaks:** a separator is encoded data interpreted by an algorithm, not a sign understood by a clerk. Different implementations can use different separator conventions. Equality must follow the specified convention, not an intuitive guess.

▪ 16.1.3 PLAIN — a worked example

1. Here is the smallest two-leaf version of our toy tree. Each leaf can hold at most three keys. The root's separator is the smallest key in its right child.

```
root: [25]
/       \
```

```
          keys < 25      keys >= 25
             /              \
       [10, 20]  ------>  [25, 30]
                   next leaf
```

2. Search for 20: compare with 25, follow the left child, then find 20 in `[10, 20]`.
3. Search for 25: equality follows the right child under our rule, then finds 25 in `[25, 30]`.
4. Search for 27: follow the right child, compare within `[25, 30]`, and report absence. The tree does not return the nearest key unless the operation explicitly asks for a nearest or range result.

16.1.4 PLAIN — what is really happening inside

1. Each internal page partitions its permitted key interval between child references. The selected child must contain every key that could match the lookup within that interval.
2. Separators are sorted. With several separators, the search chooses the region between adjacent boundaries or an edge region before the first or after the last.
3. Our equality-goes-right convention corresponds to selecting a child using the insertion position after equal separators. The companion model implements this with `bisect_right` on a small in-memory list. [S109]
4. The sorted keys inside a leaf then decide whether an exact entry exists. The absence conclusion follows from the leaf's ordering and the correctness of the path, not from checking only one arbitrary nearby key.

16.1.5 TECHNICAL — the engineer's version

1. In this pedagogical B+ tree, internal nodes store ordered separators and child references; leaves store searchable entries and a next-leaf link. Search maintains the invariant that the target, if present, lies in the selected subtree.
2. The separator convention is explicit: each separator is the lower bound represented by the first key of the child immediately to its right. Other implementations may encode boundaries differently without changing the high-level ordered-tree purpose.
3. Product documentation often uses "B-tree" for an ordered multiway index with implementation-specific leaf and internal layouts. PostgreSQL's description provides one concrete example, not the exact specification of our toy. [S105]
4. A valid structural invariant is necessary for correct lookup. It does not by itself provide durable writes, concurrent mutation safety, transaction visibility or recovery; those are later mechanisms.

16.1.6 WORDS — remember these

1. **Separator key:** a boundary directing the next step — an internal key dividing ordered child ranges under a defined comparison convention. **Leaf:** the bottom searchable page — a node containing data entries rather than further child pages in this teaching B+ tree. **Root:** the starting guide — the top node from which every lookup begins.

16.2 Pages and fan-out

16.2.1 PLAIN — in simple words

1. A database tree is usually organised around pages rather than one separate allocation for every key. One page can hold many separators and references.

2. A page with many children has high fan-out. Each step can then choose among many regions instead of only two, keeping the number of levels small.
3. Wider keys and extra payload reduce how many entries fit. Partly filled pages also hold fewer entries than a perfectly packed capacity calculation assumes.
4. A shallow tree is helpful, but "three levels" does not automatically mean exactly three slow device reads. Some pages may already be cached, and retrieving the full row may add work beyond the tree.

16.2.2 PLAIN — a picture in your head

1. One archive sign can list hundreds of room ranges. A visitor chooses one room from that sign, then one cabinet from the next sign.
2. If the signs could show only two choices each, the same archive would need many more successive decisions.
3. **Where the comparison breaks:** a physical page has byte limits, headers, reference widths and variable-sized keys. A sign's number of lines is only an analogy for capacity; the real fan-out must be derived from the actual layout and occupancy.

16.2.3 PLAIN — a worked example

1. Use an invented internal-page layout: 8,192 bytes per page, 128 bytes of fixed overhead, 8 bytes per child reference and 16 bytes per separator. With `m` children there are `m - 1` separators.
2. The payload condition is `8m + 16(m - 1) <= 8,192 - 128`, or `24m - 16 <= 8,064`. The largest whole m is 336 under these simplified assumptions.
3. If a leaf entry also occupies an assumed 24 bytes, a leaf holds at most `floor(8,064 / 24) = 336` entries. With one root and one leaf level, full capacity would be `336 × 336 = 112,896` entries.
4. Adding another full internal level gives `336 × 336 × 336 = 37,933,056` leaf entries. This is an upper-capacity illustration, not a guarantee that a real 38-million-row index has exactly three levels.
5. Real page overhead, variable key lengths, occupancy, duplicate representation and record locators change the result. The point is the multiplicative effect of fan-out, not the chosen number 336.

16.2.4 PLAIN — what is really happening inside

1. Each page read brings several possible next decisions together. Searching within the page uses CPU comparisons; moving to a child may require another page access.
2. Root and upper-level pages are frequently reused. Caching them can make many lookups avoid repeated device access to those levels.
3. Index size affects cache residency. A wider covering index may save base-row retrieval while making its own pages less likely to remain in memory. These are competing effects to measure.
4. Page occupancy changes as entries arrive, split, are deleted or are reorganised. A capacity formula using completely full pages does not describe every point in that lifecycle.

16.2.5 TECHNICAL — the engineer's version

1. With effective fan-out B, an ordered balanced tree needs on the order of `log_B N` page levels for N entries under ordinary occupancy assumptions. Comparisons inside each page and fetching associated records are separate costs.

2. The fan-out calculation must include separator representation, child references, slot directories, headers and free-space policy. The arithmetic above deliberately uses a simplified fixed-width model.
3. PostgreSQL's documented B-tree implementation has its own page and tuple constraints, sibling links, splitting behaviour and maintenance details. Do not infer its exact fan-out from our invented layout. [S105]
4. Keep **logical page visits**, **buffer hits**, **device reads** and **bytes transferred** distinct. A statement about one does not establish a count for the others.

16.2.6 WORDS — remember these

1. **Fan-out:** how many next regions one guide can select — the number of child references in an internal tree node. **Occupancy:** how full a page is — the used share of its usable entry capacity under the actual layout. **Tree height:** how many levels a lookup crosses — the root-to-leaf depth under a stated counting convention.

16.3 Search from root to leaf

16.3.1 PLAIN — in simple words

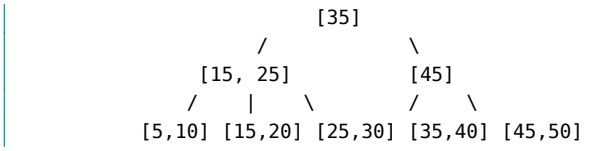
1. Searching is repeated narrowing. At each internal page, compare the target with sorted boundaries and choose exactly the child range that can contain it.
2. At the leaf, compare actual keys and return either the matching entry or an explicit not-found result. A missing key is an ordinary outcome, not a broken tree.
3. The comparison rule must be consistent everywhere. If a parent orders text differently from a leaf, the path can lead to the wrong region.
4. Duplicate values require a representation policy. An index can contain several records with the same product identifier; it must still distinguish the entries or store an associated collection of locators.

16.3.2 PLAIN — a picture in your head

1. Follow road signs to a house: district, street, then house number. At each sign, you keep only the direction that can contain the destination.
2. Arriving on the right street does not prove that house 27 exists. You still check the actual addresses.
3. **Where the comparison breaks:** roads can form loops and offer many routes. Our tree traversal follows a structural hierarchy, and correctness depends on its maintained range invariant. A cached guess is not a substitute for checking the relevant entry.

16.3.3 PLAIN — a worked example

1. Use this larger toy tree, whose construction appears in section 16.4:



2. Search 25: at the root, 25 is below 35, so choose the left internal page. At `[15,25]`, equality to 25 chooses its right-hand region. In leaf `[25,30]`, the key exists.

3. Search 27 follows the same internal path, but the leaf contains no 27. Search 45 takes the root's right side and then equality-goes-right at separator 45.
4. The tree height here is three node levels, counting the root and leaf. The small capacities are chosen to make all steps visible, not to imitate a practical page size.

16.3.4 PLAIN — what is really happening inside

1. With separators `s0, s1, ...`, child zero contains keys below `s0`; child one begins at `s0`; and so on under our convention. Selecting the position after equal separators implements those intervals.
2. A binary search inside a page reduces comparison work compared with a linear search, but a very small page representation may use another strategy. The high-level tree invariant does not mandate one CPU algorithm.
3. For non-unique product keys, a conceptual composite entry `(product_id, row_identity)` supplies a total order among equal products. Another implementation can use a posting list of row locators. Either way, returning the first matching entry alone would not answer "all lines for this product."
4. A real transaction may need to verify that a located row version is visible. Finding an index entry is a structural result; deciding whether its referenced version belongs to this query's snapshot is a separate step.

16.3.5 TECHNICAL — the engineer's version

1. The toy lookup can use `bisect_right(separators, key)` to select an internal child, then `bisect_left(leaf_keys, key)` plus an equality check at the leaf. An insertion position alone is not proof of membership. [S109]
2. Search assumes a stable tree or an implementation that coordinates readers with mutations. Python list operations and the bisect module do not supply a complete concurrent B-tree protocol.
3. Collation and comparator consistency are structural requirements. A changed comparison rule can invalidate ordering assumptions even when stored bytes have not changed; engine-specific upgrade procedures must handle such changes deliberately.

16.3.6 WORDS — remember these

1. **Search invariant:** the condition kept true at every step — the target, if present, remains inside the selected subtree's permitted range. **Record locator:** information leading from an entry to its record — an engine-specific reference or identifying value used to retrieve associated data. **Membership check:** confirm that the target actually exists — equality verification after locating a candidate or insertion position.

16.4 Inserts and splits

16.4.1 PLAIN — in simple words

1. To insert a key, first find the leaf where it belongs. If the leaf has room, place the entry in order. If it is full, the tree must create room without breaking its range rules.
2. Our toy splits an overflowing leaf into two ordered leaves and inserts a new separator into their parent. If the parent overflows, the split can propagate upward.
3. Splitting the root creates a new root and increases the tree's height by one. Every leaf remains at the same depth because the new level is added above the whole tree.
4. This is why a tree can grow while remaining shallow. It is also why an insertion can touch more than the one leaf containing the new key.

16.4.2 PLAIN — a picture in your head

1. A full filing drawer is divided into two drawers. A new sign tells visitors which numbers moved to the second drawer. If the signboard is itself full, it too must be reorganised.
2. The old sign cannot be left pointing to the wrong interval while people rely on it.
3. **Where the comparison breaks:** database engines must coordinate intermediate states, failures and concurrent readers. The toy's neat before-and-after drawings omit the logging and synchronisation that make a real split safe.

16.4.3 PLAIN — a worked example

1. Our toy rules are: at most three keys per leaf; at most four children per internal node; equality follows the right side of a separator. An overflowing four-key leaf splits into two two-key leaves.
2. Insert 10, 20 and 30. The root is still one leaf: `[10,20,30]`. Insert 25: the four sorted keys split into `[10,20]` and `[25,30]`, and a new root has separator `[25]`.
3. Insert 5, then 15. The left leaf overflows as `[5,10,15,20]` and splits into `[5,10]` and `[15,20]`. The root now has separators `[15,25]` and three children.
4. Insert 35, then 40. The right leaf splits into `[25,30]` and `[35,40]`. The root now has separators `[15,25,35]` and four children, still within its capacity.
5. Insert 45, then 50. The last leaf splits into `[35,40]` and `[45,50]`. The root would need five children, so split that internal level: the left internal node has the first three children with separators `[15,25]`; the right has two children with separator `[45]`.
6. A new root `[35]` points to those two internal nodes. This is exactly the tree drawn in section 16.3. All ten inserted keys occur once at the leaves.

16.4.4 PLAIN — what is really happening inside

1. Leaf splitting redistributes entries while preserving their order. The separator inserted into the parent describes the boundary between the resulting child ranges.
2. Internal splitting redistributes child references and their boundaries. A separator moves or is derived upward according to the chosen representation; it is not blindly copied as an extra record into a leaf.
3. Leaf-neighbour links must be updated as well. Range scans would otherwise skip the new leaf or follow an obsolete link even if point lookups through the root appeared correct.
4. A complete implementation must also handle deletion, borrowing, merging, root shrinking, duplicate keys and interrupted operations. This book's insertion trace does not claim those mechanisms have been implemented merely because the drawing remains balanced.

16.4.5 TECHNICAL — the engineer's version

1. In the toy, an internal node with k separators has k+1 children. Splitting an overflowing five-child root into three-child and two-child internal nodes satisfies the chosen minimum of two children for non-root internal nodes.
2. Validate ordering, child-count relationships, equal leaf depth, separator-to-range consistency, leaf-chain order and complete key conservation after every insertion. A successful lookup for the last inserted key alone is weak evidence.
3. Real B-tree implementations use engine-specific split, concurrency and recovery protocols. PostgreSQL documents details including sibling links and page management. These do not follow automatically from the sequential in-memory toy. [S105]

4. A split can increase write and log work. Predicting its latency requires the actual page state, storage and concurrent workload; counting one logical INSERT is not a physical-write count.

16.4.6 WORDS — remember these

1. **Page split:** divide an overfull region — redistribution into two nodes with updated parent boundaries and relevant sibling links. **Split propagation:** a full parent also needs room — upward restructuring triggered when adding a child exceeds an internal node's capacity. **Key conservation:** no entry disappears or is invented — equality between the intended inserted entries and those represented after restructuring.

16.5 Range scans

16.5.1 PLAIN — in simple words

1. A range question asks for several nearby keys rather than one exact key. An ordered tree can first locate the beginning, then continue through neighbouring leaves.
2. The ordering tells the scan when it can stop. Once it reaches a key beyond the upper boundary, later ordered keys cannot belong to the requested range.
3. Boundaries must be explicit. "From 17 to 37" may include or exclude either endpoint. Dates and timestamps make this especially important.
4. A range scan returning many records still performs work proportional to the results and visited pages. A fast start does not make a million-row answer free.

16.5.2 PLAIN — a picture in your head

1. Find the first shelf containing numbers at least 17, then walk along shelves until numbers exceed 37. You use the archive guide once to find the starting region, not again for every neighbouring folder.
2. If the shelf labels and links are correct, no return to the entrance is needed between adjacent folders.
3. **Where the comparison breaks:** physical leaf pages may not be adjacent on a device even when they are neighbours in key order. Following logical links is not a guarantee of perfectly sequential physical input-output.

16.5.3 PLAIN — a worked example

1. Query the toy tree for the closed interval `[17,37]`. The starting lookup reaches leaf `[15,20]`. Skip 15 because it is below 17 and emit 20.
2. Follow the next-leaf link to `[25,30]`; emit 25 and 30. Continue to `[35,40]`; emit 35, then stop at 40 because it exceeds 37.
3. The result is `[20,25,30,35]`. Neither 17 nor 37 needs to exist as a stored key for the range to be meaningful.
4. For time windows, an often useful convention is a half-open interval: `timestamp >= start AND timestamp < end`. Adjacent windows then share a boundary without counting an event at that boundary twice. The convention must match the metric's intended timezone and instant representation.

16.5.4 PLAIN — what is really happening inside

1. The initial search identifies a leaf and an insertion position for the lower boundary. The scan emits qualifying entries from there and follows ordered successor information.
2. Inclusive and exclusive tests decide whether boundary-equal entries belong. A duplicate-key range must include every qualifying entry, not only the first occurrence.
3. The query may still need full-row retrieval and visibility checks for each candidate. If only indexed values are needed and the engine permits an index-only route, some of that work may be avoided.
4. Concurrent changes require a specified visibility contract. A stable ordered structure and a stable transaction snapshot are different properties. The former alone does not explain which newly inserted or deleted rows a long-running reader observes.

16.5.5 TECHNICAL — the engineer's version

1. A simplified B+ tree range cost is an initial tree descent plus traversal of relevant leaf pages and output processing. Writing this as `O(log_B N + K/B)` page-scale work assumes a particular layout and occupancy, with K qualifying entries; record fetching and CPU work are additional.
2. Ordered B-tree methods support equality and range comparisons under their operator families. Whether an SQL query uses that route depends on its expressions, collation, predicates and selected plan. [S104] [S106]
3. Logical leaf order, physical page placement and final result order are not synonyms. Keep the query's ORDER BY explicit, even when an ordered access path currently supplies the desired sequence.

16.5.6 WORDS — remember these

1. **Range scan:** walk entries between boundaries — ordered traversal beginning near a lower bound and stopping when an upper condition fails. **Half-open interval:** include the start but not the end — an interval [a,b) useful for adjoining non-overlapping ranges. **Successor link:** the route to the next ordered region — a reference or traversal mechanism connecting neighbouring entries or leaf nodes.

16.6 Implementation boundaries

16.6.1 PLAIN — in simple words

1. The toy explains ordered navigation and splitting. A production database must solve additional problems: several readers and writers, crashes, damaged pages, space reuse and transaction visibility.
2. A tree that passes a sequential insertion test is not automatically safe to use from several threads. A saved tree file is not automatically durable after sudden power loss.
3. Database products also make different choices about what leaves contain, how duplicates are represented and where the actual row lives.
4. Use the common idea to understand documentation, then read the particular engine's promises before depending on an implementation detail.

16.6.2 PLAIN — a picture in your head

1. A scale model bridge shows how beams connect. It does not certify a full-size bridge's foundations, materials, wind response or inspection schedule.

2. The model is valuable when it teaches the load path and clearly states what it omits.
3. **Where the comparison breaks:** software correctness depends on discrete invariants and failure schedules as well as physical hardware. A database tree cannot be certified merely by making the toy larger or running the same happy path many times.

16.6.3 PLAIN — a worked example

1. Suppose an insertion updates the new leaf but crashes before the parent's separator is saved. A restarted reader following only the old root may miss the inserted region.
2. Alternatively, suppose the parent becomes visible before the new leaf is valid. A reader may follow a reference to incomplete data. These are hypothetical failure schedules, not observations from the toy lab.
3. Logging, ordered persistence and engine-specific recovery protocols address such states. Chapters 28 and 30 explain why "write all the files eventually" does not establish safe acknowledgement.
4. A concurrency test must also control a reader's interleaving with a split. A sequential test that inserts everything and only then reads cannot expose every intermediate-state defect.

16.6.4 PLAIN — what is really happening inside

1. Real implementations protect short structural operations and coordinate transaction-level behaviour. A latch protecting an in-memory page is not automatically the same thing as a transaction lock governing a business record.
2. Page checksums can detect some corruption but do not establish correct business values. Recovery needs a defined relationship between persisted pages and recovery records.
3. Versioned entries can remain after a logical update or deletion until no relevant reader needs them and cleanup is safe. The visible row count may therefore differ from physical entry and page counts.
4. Operational evidence must name the engine, version, access method and observation tool. A generic B-tree diagram cannot establish what a specific production index contains at a particular moment.

16.6.5 TECHNICAL — the engineer's version

1. The chapter's toy specification covers unique integer keys, sequential insertions, point membership, range traversal and structural validation. It intentionally omits concurrent mutation, deletion rebalancing, crash recovery, disk formats and transaction isolation.
2. SQLite's query-planning description and PostgreSQL's B-tree implementation documentation illustrate related but different storage organisations. Treat their details as product contracts, not interchangeable definitions. [S89] [S105]
3. Structural validation and model-based tests are useful: compare every operation with a simple sorted-set oracle and verify all invariants after each mutation. They establish the tested toy behaviour, not a proof for untested mechanisms.
4. This distinction will recur throughout the book. An explanatory model can be correct within its boundary without constituting a deployable system.

16.6.6 WORDS — remember these

1. **Latch:** short-lived protection of an internal structure — a synchronisation mechanism distinct from a transaction's logical locking policy. **Oracle:** an independent expected-result mechanism —

a simpler model used to compare the behaviour of a tested implementation. **Structural validation:** check that the representation obeys its rules — inspection of ordering, ranges, links, occupancy and depth invariants.

16.97 Practice and worked answers

1. **Question:** In the two-leaf tree with separator 25, which child receives a search for 25? **Answer:** The right child, because this toy defines separators as the smallest key in the right-hand region. Another convention would require its own consistent rule.
2. **Question:** Can an insertion position prove membership? **Answer:** No. Searching for 27 in `[25,30]` identifies a position, but the key there is 30. An equality check is still required.
3. **Question:** Recalculate maximum children for the invented 8,192-byte page. **Answer:** `8m + 16(m-1) <= 8,064`, so `m <= 336.666...`; the maximum integer is 336. This excludes layout features not included in the model.
4. **Question:** Insert the ten keys from section 16.4 in order. What is the final root? **Answer:** `[35]`, with left internal separators `[15,25]` and right internal separator `[45]`. The five leaves are `[5,10]`, `[15,20]`, `[25,30]`, `[35,40]`, `[45,50]`.
5. **Question:** What does `[17,37]` return? **Answer:** 20, 25, 30 and 35. Start in `[15,20]`, follow two successor links, and stop before 40.
6. **Question:** Does a three-level tree guarantee three device reads? **Answer:** No. Cached pages can avoid device reads, while associated-row retrieval can add work. The level count is a logical structural fact.
7. **Question:** Why validate the leaf chain as well as root-based lookup? **Answer:** Range traversal uses the chain. A broken successor link can omit a leaf even when individual root-to-leaf lookups work.
8. **Question:** What is still missing after every sequential insertion test passes? **Answer:** Among other things, concurrent mutation safety, deletion behaviour, durable logging, crash recovery and transaction visibility. Those mechanisms require their own specifications and evidence.

16.98 Common wrong ideas

1. **Wrong:** a B-tree has only two children per node. **Right:** it is a multiway structure whose fan-out is central to its shallow depth.
2. **Wrong:** a separator is always a separate record to return. **Right:** internal boundaries and leaf entries have different roles in this toy.
3. **Wrong:** a missing key means a search failed incorrectly. **Right:** not-found is a valid result established by the ordered path and leaf check.
4. **Wrong:** one insertion changes only one page. **Right:** splits can update neighbours, parents and a new root.
5. **Wrong:** adjacent keys guarantee adjacent disk blocks. **Right:** logical order and physical placement are different.
6. **Wrong:** height equals device-read count. **Right:** caching and full-row retrieval change physical work.
7. **Wrong:** a sequential toy is safe for concurrent use. **Right:** synchronisation and visibility require additional mechanisms.

8. **Wrong:** a diagram proves durability. **Right:** persistence ordering and crash recovery require explicit protocols and tests.

16.99 Chapter summary in 20 lines

1. Ordered trees skip regions using maintained range boundaries.
2. The toy B+ tree stores entries in leaves and separators internally.
3. Equality follows the right side of a separator in this specification.
4. Every root-to-leaf path has the same node depth.
5. A page can contain many child references and separator keys.
6. High fan-out keeps a large index relatively shallow.
7. Key width, overhead and occupancy affect real capacity.
8. Logical page visits are not identical to physical device reads.
9. Search narrows a permitted interval at each level.
10. A leaf membership check distinguishes an exact key from an insertion position.
11. Duplicate values require a representation that preserves every qualifying entry.
12. Insertion can split an overflowing leaf.
13. A new separator can propagate a split into its parent.
14. Splitting the root adds one level above the whole tree.
15. Range scans locate a start and traverse ordered successors.
16. Inclusive and exclusive boundaries must be stated.
17. Leaf order does not guarantee physical contiguity.
18. Structural invariants should be checked after every toy mutation.
19. Concurrency, visibility and recovery are separate from the sequential tree model.
20. Use the model to understand an engine, not to invent guarantees it never supplied.

17

CHAPTER

Hashing, Search Keys and Collisions

17.0 What this chapter gives you

1. Hashing maps an input to a value that can help organise or check data. The same word appears in hash tables, file digests, partition routing and security protocols, but those uses do not demand the same properties.
2. You will handle collisions correctly, distinguish equality lookup from ordered search, calculate a simple collision estimate, and explain why an unkeyed digest is not proof of who supplied a file.
3. The small hash functions below are deliberately weak teaching examples. They are not proposed security designs, production partition functions or password-storage schemes.

17.1 A deterministic mapping

17.1.1 PLAIN — in simple words

1. A hash function follows a rule to turn an input into an output value. Deterministic means that the same input under the same rule produces the same output.
2. This can help choose a storage bucket or produce a compact fingerprint of bytes. The function does not understand whether the input represents an order, a photograph or a lie.
3. Different inputs can produce the same output when the output space is smaller than the possible input space. This is a collision, not necessarily a malfunction.
4. The meaning of "same input" must be explicit. Text with different encodings, newline conventions or invisible characters can have different bytes even when a screen looks similar.

17.1.2 PLAIN — a picture in your head

1. Mira assigns folders to four trays using the remainder of the folder number divided by four. Folder 10 goes to tray 2; folder 14 also goes to tray 2.
2. The tray number is a routing clue, not the folder's full identity.
3. **Where the comparison breaks:** practical hash functions mix input bits in more sophisticated ways, and cryptographic hashes have additional security goals. The remainder rule demonstrates determinism and collisions only.

17.1.3 PLAIN — a worked example

1. Define the toy function `h(k) = k mod 4` for non-negative integer keys. Then `h(2)=2`, `h(6)=2`, `h(10)=2`, while `h(3)=3`.
2. The outputs tell us which tray to inspect. They cannot tell us whether keys 2 and 6 are equal, because those different inputs deliberately share the same result.
3. For byte fingerprints, Python's standard SHA-256 interface can compute a digest:

```
import hashlib
fingerprint = hashlib.sha256(b"abc").hexdigest()
print(fingerprint)
```

4. The expected hexadecimal digest is `ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`. The code hashes the three bytes for `abc`, not a file containing those bytes plus a newline. [S99] [S100]

17.1.4 PLAIN — what is really happening inside

1. A hash algorithm consumes a precisely represented input and computes an output according to fixed operations. Changing one input byte gives a different input to that algorithm, even if the application considers two records equivalent.
2. Encoding and framing therefore belong before hashing. Concatenating fields without boundaries can lose distinctions: `ab` followed by `c` and `a` followed by `bc` both produce `abc`.
3. A digest cannot restore distinctions erased before the hash function received the bytes. Length prefixes, explicit field names or a reviewed canonical encoding can preserve the intended boundaries.
4. A semantic equality policy may deliberately ignore some differences, such as harmless formatting. That policy must be defined separately; it is not supplied automatically by using a cryptographic function.

17.1.5 TECHNICAL — the engineer's version

1. A hash is a mapping from an input domain to an output space. Fixed-length digests cannot be injective over an unbounded input domain; the pigeonhole principle guarantees that some distinct inputs share an output.
2. SHA-256 produces a 256-bit digest under the Secure Hash Standard. Its security goals are different from the convenience goals of an ordinary in-memory bucket function. [S100]
3. Specify the algorithm identifier, input encoding, framing and any canonicalisation version when storing a digest intended for later verification. "Hash of the order" is underspecified when one implementation hashes pretty-printed JSON and another hashes a different byte representation.
4. Python's `hash()` is not a portable persisted content digest. In particular, string and byte hashes use process-level randomisation, and the data model defines equality/hash relationships rather than a universal cross-process wire value. [S108]

17.1.6 WORDS — remember these

1. **Hash function:** a rule making an organising or checking value — a deterministic mapping under specified parameters and input representation. **Digest:** a compact fingerprint of bytes — a fixed-length output of a hash algorithm, whose meaning depends on the exact input bytes and algorithm. **Framing:** make field boundaries unambiguous — an encoding convention that preserves where components begin, end and differ in type or length.

17.2 Buckets and collisions

17.2.1 PLAIN — in simple words

1. A bucket groups entries whose hash-based route leads to the same place. Several different keys may belong there.

2. Correct collision handling keeps enough information to distinguish those keys. A lookup checks the original key, not just the bucket number or shortened hash.
3. A collision is not the same as a duplicate key. Keys 2 and 6 collide under our toy function but remain separate keys. Inserting key 2 twice raises a different question: replace its value, reject the insertion or store several values?
4. Choose that duplicate-key policy explicitly. A hash table's storage mechanics do not decide the business meaning of repeated requests.

17.2.2 PLAIN — a picture in your head

1. Tray 2 contains folders numbered 2, 6 and 10. Dev searches inside the tray until he finds the exact folder number.
2. Putting every folder that shares a tray into one combined folder would destroy information. Throwing away later folders would be equally wrong.
3. **Where the comparison breaks:** implementations may use linked chains, arrays, probing or other layouts instead of a literal tray. Some do not store a separate container object for each bucket. The invariant is correct discrimination of keys, not a particular picture.

17.2.3 PLAIN — a worked example

1. Insert `(2, 'red')`, `(6, 'blue')` and `(10, 'green')` into a toy table with four buckets. All three entries route to bucket 2.
2. Lookup 6 checks bucket 2 and compares keys: 2 is not 6; 6 is 6, so return `blue`. Lookup 14 visits the same bucket and returns not found after checking all three different keys.
3. If the implementation returned the first entry merely because the bucket matched, lookup 6 would incorrectly return `red`. This is an exact counterexample to treating hash equality as key equality.
4. A second insertion of key 6 with value `navy` needs a chosen policy. In our dictionary-style toy it replaces the value for key 6; it must not alter entries for keys 2 or 10.

17.2.4 PLAIN — what is really happening inside

1. With separate chaining, a bucket points to a collection of entries that share the route. With open addressing, collisions lead to a defined sequence of alternative slots in one array.
2. Open-addressed deletion needs care. Turning a removed slot into an ordinary never-used slot can prematurely stop a later search for an entry placed farther along the same probe sequence.
3. Resizing changes routing when the number of buckets changes. Entries must be re-established under the new arrangement; copying only the bucket labels is not enough.
4. Equality comparison and hash computation must agree on the key contract. If two keys compare equal but route inconsistently, lookup can fail to find an existing equal key.

17.2.5 TECHNICAL — the engineer's version

1. Chaining and open addressing are collision-resolution strategies, not different definitions of equality. Expected constant-time hash-table operations require assumptions about distribution, load and the cost of hashing and comparing keys.
2. The load factor is entry count divided by bucket or slot capacity under the implementation's convention. Higher load can increase chain lengths or probe lengths; acceptable thresholds depend on the design.

3. Python requires equal objects used as hashable keys to have equal hashes. Mutable equality-relevant state is incompatible with a stable key unless the type carefully preserves the required contract. [S108]
4. The chapter's toy explicitly compares original keys. This establishes collision correctness for the exercised representation; it does not establish resistance to adversarial inputs or bounded latency for arbitrarily large keys.

17.2.6 WORDS — remember these

1. **Collision:** different inputs share one hash result — equality of hash outputs without equality of original inputs. **Load factor:** how full the hash structure is — entries divided by the relevant bucket or slot count under a stated convention. **Probe sequence:** alternative places checked after a collision — a defined traversal of slots in an open-addressed hash table.

17.3 Hash tables and equality

17.3.1 PLAIN — in simple words

1. Hash tables are naturally suited to exact-key questions: "Is this order identifier present?" or "What value belongs to this key?"
2. A normal hash route does not preserve the original order of keys. Nearby order numbers can land in unrelated buckets. Asking for all keys between two numbers therefore needs another strategy or a broad examination.
3. The key's equality rule matters. Are uppercase and lowercase identifiers different? Are two differently formatted timestamps the same instant? Decide before building the route.
4. A fast lookup can still answer the wrong question when the application chose the wrong key or ignored part of its scope.

17.3.2 PLAIN — a picture in your head

1. The four trays group folders by a remainder, not by consecutive number ranges. Folder 3 and folder 4 go to different trays; folder 2 and folder 10 share one.
2. To list every folder numbered 5 through 12, Dev cannot simply start at one tray and stop at the next as he could with sorted shelves.
3. **Where the comparison breaks:** some programming-language dictionaries preserve insertion order for iteration. That is not sorted-key order and does not turn their hash lookup mechanism into a B-tree range index.

17.3.3 PLAIN — a worked example

1. Suppose a new exercise tracks receipts by `(branch_id, receipt_no)`. `(BR-A,17)` and `(BR-B,17)` must remain distinct keys even though the receipt number is equal.
2. Hashing only the number and comparing only that number would conflate them. Hashing the number alone but retaining and comparing the complete tuple could remain logically correct, yet might create unnecessary collisions.
3. The safest specification states both the complete key and its equality rule, then chooses a suitable hash representation of that key.
4. For a range such as receipts 10–20 within BR-A, an ordered composite structure may better match the question. A hash table can still answer by examining entries, but ordinary exact-key routing does not directly provide the ordered range.

17.3.4 PLAIN — what is really happening inside

1. Hash lookup narrows the candidate region using a derived value, then performs equality checks. Ordered lookup narrows regions using comparison boundaries. These are different information structures.
2. If equality normalises case or Unicode representations, the hash path must respect that equality. Normalising only during final comparison while routing raw unequal representations can violate the equal-key contract.
3. Normalisation can also be wrong for the domain. Lowercasing an opaque case-sensitive token destroys distinctions. "Make matching easier" is not sufficient authority to merge identities.
4. Keep identifiers immutable when used as keys, or explicitly remove and reinsert entries under a changed key. A key that silently changes after insertion can become unreachable through the expected route.

17.3.5 TECHNICAL — the engineer's version

1. Hash access primarily supports equality semantics; ordinary hashing does not preserve total order or neighbourhoods. PostgreSQL's hash index is equality-oriented and uses a lossy stored hash representation requiring rechecking of candidates. This differs from an in-memory language dictionary. [S101]
2. A hash function and an equality comparator form a joint contract. Equal keys must have compatible routing; unequal keys are allowed to collide and must remain distinguishable by the equality check. [S108]
3. Expected O(1) lookup is not an unconditional worst-case guarantee. It excludes neither long collision chains nor the cost of computing a hash over a long input. State the assumptions when discussing complexity.

17.3.6 WORDS — remember these

1. **Equality contract:** the rule for deciding that two keys are the same — a domain-specific equivalence relation that hashing and comparison must respect. **Expected complexity:** cost under specified distribution assumptions — an average or probabilistic bound, not a guarantee for every input. **Lossy index entry:** a representation that does not retain all distinguishing information — an entry requiring rechecking against fuller data to establish a match.

17.4 Cryptographic and non-cryptographic purposes

17.4.1 PLAIN — in simple words

1. A fast bucket function tries to spread ordinary keys usefully. A cryptographic hash additionally aims to make certain deliberate manipulations computationally infeasible under a defined security model.
2. A cryptographic digest can help detect that bytes differ from a trusted expected digest. It does not tell you that the bytes are truthful, safe to execute or written by a particular person.
3. If an attacker can replace both a file and the untrusted digest displayed beside it, comparing the two does not establish authenticity. The expected value needs a trustworthy origin.
4. Encryption, message authentication and password storage are different tasks. Calling all of them "hashing" hides important requirements.

17.4.2 PLAIN — a picture in your head

1. A parcel's recorded weight can reveal that it changed, but matching weight does not prove the contents or sender. A cryptographic digest is a far stronger byte fingerprint than weight, yet the question "who supplied the expected fingerprint?" remains.
2. A signed receipt or a message-authentication mechanism addresses a different relationship: who could have authorised or produced the evidence under the relevant key system.
3. **Where the comparison breaks:** cryptographic security is not based on physical weight or a tamper-evident sticker. It depends on precise algorithms, key handling, threat assumptions and computational limits. The analogy must not be used to design a new security protocol.

17.4.3 PLAIN — a worked example

1. Mira stores a backup file and a SHA-256 digest in a separately controlled manifest. Later, recomputing the digest can detect a byte mismatch relative to that manifest.
2. A match supports "these bytes have this digest, consistent with the trusted recorded value," subject to the algorithm's security assumptions. It does not establish that the backup includes every required table or that restoration succeeds.
3. Copying the file and digest into the same publicly writable folder weakens the origin claim: someone able to replace both can create a new matching pair.
4. The recovery exercise in Chapter 31 therefore combines byte checks with a real isolated restore and business reconciliation. Each check answers a different question.

17.4.4 PLAIN — what is really happening inside

1. Cryptographic hash goals include difficulty finding an input for a chosen digest, a different input matching a given input's digest, or any pair of distinct inputs with the same digest. These are related but distinct attack goals.
2. An unkeyed digest has no secret. Anyone can calculate one for any bytes. Authenticity requires a trusted channel, a signature, a properly used message-authentication construction or another explicit source of trust.
3. A keyed message-authentication code verifies possession of a key under its protocol assumptions. It is not a public signature from a uniquely identified person when several parties share that key.
4. Ordinary fast hashes are also not complete password-storage systems. Password verification needs a purpose-designed, reviewed construction and appropriate operational controls; storing a bare general-purpose digest does not supply them.

17.4.5 TECHNICAL — the engineer's version

1. Distinguish preimage resistance, second-preimage resistance and collision resistance. A 256-bit output length is not a promise that every attack requires 2^256 work; generic collision reasoning has a different scale. [S100]
2. Verification must bind the algorithm, exact bytes and trusted expected value. A checksum can detect accidental corruption while offering no meaningful adversarial collision resistance. A cryptographic digest still requires provenance when used for authenticity.
3. Do not improvise a MAC by concatenating a secret and message and calling a hash function. Use a standard reviewed authentication construction and protocol. This book does not introduce a home-made cryptographic scheme.

4. A digest's byte-level equality evidence does not replace semantic validation, malware analysis, access control, retention review or restore testing.

17.4.6 WORDS — remember these

1. **Preimage resistance:** hard to work backwards from a digest — computational difficulty of finding an input producing a specified hash output. **Collision resistance:** hard to deliberately find two matching fingerprints — computational difficulty of finding distinct inputs with equal digests under the algorithm's security assumptions. **Message authentication code:** a keyed check on a message — an integrity and authenticity mechanism for parties sharing a key, not automatically a public signature.

17.5 Distribution and adversarial inputs

17.5.1 PLAIN — in simple words

1. A hash table works well when keys spread across its available locations under the workload it actually receives. Poor distribution creates crowded regions and longer searches.
2. Uniform-looking ordinary data is not a guarantee against deliberately chosen inputs. A service accepting untrusted keys must consider resource limits and the hash design's threat model.
3. Random-looking digests also do not eliminate collisions mathematically. A finite output space eventually forces repetition; the useful question is how likely or difficult a collision is at the relevant scale.
4. Probabilistic estimates need assumptions. State whether outputs are being modelled as independent and uniformly distributed, and do not present the model as a proof about every input source.

17.5.2 PLAIN — a picture in your head

1. Four trays receive folder numbers ending in a pattern that always gives remainder two. The trays exist, but three remain empty while one becomes crowded.
2. Adding more trays without changing the relationship between inputs and routing may not solve the underlying concentration.
3. **Where the comparison breaks:** real hash designs can randomise or key their mixing, and resize policies change the structure. A simple remainder function is intentionally a counterexample, not a model of every language runtime's protection.

17.5.3 PLAIN — a worked example

1. For n independently uniform outputs in a space of size M, there are `n(n-1)/2` distinct pairs. Each pair has collision probability `1/M`. The expected number of colliding pairs is therefore `n(n-1)/(2M)`.
2. This expectation is not itself always the probability of at least one collision. When the expectation is small, it approximates that probability; a common birthday approximation is `1 - exp(-n(n-1)/(2M))`.
3. For 100,000 outputs in a 32-bit space, the expected colliding-pair count is about 1.164, and the birthday approximation gives an at-least-one-collision probability around 68.8 percent.
4. For one billion outputs in a 256-bit space, the small-probability pair estimate is about `4.32 × 10^-60`. This is conditional mathematical reasoning under the uniform model, not a guarantee about a flawed implementation or compromised trust chain.

5. The lesson is not "a long digest makes every system safe." A wrong encoding, missing key comparison or replaced manifest can break the application without requiring a cryptographic collision at all.

17.5.4 PLAIN — what is really happening inside

1. Collision frequency and lookup cost depend on both the hash output and how the table maps that output into its present capacity. Bucket reduction, resizing and equality checks all matter.
2. Resource attacks can also use huge keys or huge numbers of distinct keys. A collision-resistant digest does not bound memory consumption or the cost of hashing gigabytes of input.
3. Bound accepted input size, operation counts and memory growth at the application boundary. These are operational protections, not replacements for a correct key contract.
4. Test distribution with representative skew and deliberately concentrated toy inputs. Keep security claims separate from ordinary benchmark results; a fast sample does not establish adversarial robustness.

17.5.5 TECHNICAL — the engineer's version

1. The birthday calculations are original derivations from a uniform finite-space model. Expected colliding pairs follow linearity of expectation; the exponential probability expression is an approximation, not an exact identity for arbitrary n and M.
2. Hash-table worst cases and cryptographic collision attacks are different analyses. A runtime may use keyed or randomised hashing to make predictable collision patterns harder, while still enforcing application-level resource limits. Python documents process randomisation for string and byte hashes. [S108]
3. A persisted partition or identity mapping needs stable, versioned behaviour. A deliberately process-randomised runtime hash is not suitable as an undocumented cross-process routing contract.

17.5.6 WORDS — remember these

1. **Birthday effect:** collisions become plausible before the space is full — pair growth causes collision probability to rise around the square root of a uniform output-space size. **Adversarial input:** data chosen to exploit assumptions — inputs deliberately selected to cause incorrectness, excessive resource use or other unwanted behaviour. **Uniform model:** assume every output is equally likely — a mathematical distribution assumption whose applicability must be justified separately.

17.6 Hashing in storage designs

17.6.1 PLAIN — in simple words

1. Storage systems use hashes for several jobs: locating a bucket, comparing content, choosing a partition and building summaries that rule out some unnecessary searches.
2. Each job needs a separate contract. A partition hash should route the same key consistently under the active mapping. A content digest should identify exact bytes under a named algorithm. Neither automatically proves a business record's meaning.
3. Some structures deliberately permit false positives while avoiding false negatives under their own assumptions. They can tell the system "definitely not here" or "possibly here," with a later exact check required.

4. These mechanisms save work by narrowing possibilities. They do not justify discarding the original identity or skipping required verification.

17.6.2 PLAIN — a picture in your head

1. A warehouse guide can route a box to a building, a checklist can suggest whether a product might be inside, and a sealed manifest can identify an exact file of records. All are guides, but they answer different questions.
2. Treating "possibly present" as "definitely the right record" confuses a filter with an answer.
3. **Where the comparison breaks:** probabilistic filters and distributed routing have precise algorithms and update rules. The warehouse picture does not prove their error bounds, deletion support or behaviour during reconfiguration.

17.6.3 PLAIN — a worked example

1. A toy membership summary has eight bits initially zero. For integer key k, set bit `k mod 8`. Inserting keys 1 and 9 sets the same bit, number 1.
2. Querying 3 finds bit 3 unset and can conclude that 3 was not inserted into this correctly maintained toy summary. Querying 17 finds bit 1 set and can only say "possibly present." It must check the actual stored keys to reject the false positive.
3. Clearing bit 1 to remove key 1 would also erase the evidence for key 9. This simple bitset therefore does not support arbitrary deletion safely by just clearing the bit.
4. This is a deliberately simplified filter illustration, not a full Bloom-filter implementation or a claim about a production filter's false-positive rate.

17.6.4 PLAIN — what is really happening inside

1. Content-addressed storage names an object using a digest or related content identifier. The system must still define how objects are encoded, verified, authorised and retained.
2. Hash partitioning chooses a location from a key and a mapping. When the partition arrangement changes, existing and new readers need an explicit transition protocol; independently recomputing a different modulus can make records appear missing.
3. Hash joins build a hash structure on one input and probe it with another, still checking join keys and preserving the required multiplicity. They do not normalise away valid duplicates.
4. Sorted storage systems may use probabilistic membership filters to avoid reading runs that cannot contain a key. A positive result usually requires further search; a negative result is safe only under the filter's maintained assumptions and versioned encoding.

17.6.5 TECHNICAL — the engineer's version

1. Separate hash-based **routing**, **candidate generation**, **integrity checking** and **authentication**. Their correctness and security requirements are not interchangeable.
2. A lossy database hash index rechecks candidate values; PostgreSQL documents this explicitly for its hash method. A hash join likewise requires equality verification and correct duplicate handling. [S101] [S65]
3. The toy bitset's no-false-negative claim holds only for insert-only maintenance with the exact stated mapping and uncorrupted state. It does not survive arbitrary clearing, changed encodings or lost updates.

4. Chapters 29 and 35 return to filters and partitioning inside larger designs. Preserve the boundary learned here: a compact derived value helps navigate evidence but does not replace the evidence's identity and meaning.

17.6.6 WORDS — remember these

1. **False positive:** a possible match that is not actually present — a positive candidate result rejected by a later exact check. **Content addressing:** name bytes by a content-derived identifier — object identification based on a specified representation and digest scheme. **Hash partitioning:** choose a data region from a key's hash — routing through a versioned mapping from derived values to partitions.

17.97 Practice and worked answers

1. **Question:** Under `h(k)=k mod 4`, do keys 2 and 6 represent duplicates? **Answer:** No. They are different keys with the same hash. Compare original keys within the collision-resolution structure.
2. **Question:** Why can `ab` plus `c` and `a` plus `bc` have the same digest without any cryptographic weakness? **Answer:** Naive concatenation produced identical input bytes before hashing. Preserve field boundaries with an explicit encoding.
3. **Question:** Can Python's built-in hash of a string be an undocumented stable file identifier across processes? **Answer:** No. Its contract includes process randomisation, not portable content-digest stability.
4. **Question:** Does matching a backup's trusted digest prove that it restores every required business record? **Answer:** No. It checks byte consistency against that digest. Restore and reconciliation tests address recoverability and semantic completeness.
5. **Question:** What is the expected colliding-pair count for n=100 and M=1,000? **Answer:** `100×99/(2×1,000)=4.95`. This is not a 495-percent probability; expectation and probability are different quantities.
6. **Question:** Why is a normal hash table not a direct ordered-range index? **Answer:** Hash routing does not preserve key order. Range retrieval needs another structure or examination of the relevant entries.
7. **Question:** The toy eight-bit filter has bit 1 set. Does key 17 exist? **Answer:** Not necessarily. It is a possible match and needs exact verification.
8. **Question:** Can clearing bit 1 safely delete key 1 when key 9 was also inserted? **Answer:** No. The bit represents shared evidence. Clearing it creates a false negative for 9 in this toy.

17.98 Common wrong ideas

1. **Wrong:** equal hashes prove equal records. **Right:** collisions exist and candidate equality must be checked at the appropriate level.
2. **Wrong:** a collision is the same as duplicate delivery. **Right:** one is an output-space event; the other concerns repeated business-message identity.
3. **Wrong:** hashing automatically preserves field boundaries. **Right:** encoding must preserve them before hashing.
4. **Wrong:** a digest authenticates its own source. **Right:** the expected digest needs a trusted origin or a suitable authentication mechanism.
5. **Wrong:** a 256-bit output guarantees 2^256 work for every attack. **Right:** different attack goals have different analyses.

6. **Wrong:** expected constant time is a worst-case promise. **Right:** distribution, load and key-processing cost matter.
7. **Wrong:** a positive membership filter result proves presence. **Right:** false positives require a later exact check.
8. **Wrong:** changing a partition modulus automatically moves old data. **Right:** routing changes need an explicit transition and migration protocol.

17.99 Chapter summary in 20 lines

1. Hashing maps a precisely represented input to a derived value.
2. Determinism applies to the same rule, parameters and input bytes.
3. Finite output spaces permit collisions.
4. A hash result is not the original key's complete identity.
5. Collision handling retains and compares the necessary key information.
6. Duplicate-key policy is separate from collision resolution.
7. Framing prevents distinct field sequences becoming identical byte strings.
8. Hash and equality rules must agree.
9. Ordinary hash routing does not preserve ordered ranges.
10. Runtime hash values are not automatically portable persisted digests.
11. Cryptographic and ordinary bucket hashes serve different purposes.
12. Preimage, second-preimage and collision resistance are distinct goals.
13. An unkeyed digest does not establish who supplied the bytes.
14. Trusted provenance and restore testing answer questions a digest does not.
15. Birthday estimates require a uniform-output model and a stated scale.
16. Expected collision counts are not probabilities above one.
17. Untrusted inputs also require size and resource limits.
18. Probabilistic filters can produce candidates rather than confirmed matches.
19. Hash-based routing needs a stable, versioned mapping.
20. Keep compact navigation evidence separate from exact identity and business meaning.

18

CHAPTER

Query Plans and the Optimiser

18.0 What this chapter gives you

1. A query states the answer required; a plan describes a way to obtain it. You will read that plan as a flow of records, estimates and operations rather than a collection of impressive node names.
2. You will compare nested-loop, hash and merge joins, identify the first important estimate error, and design a bounded experiment before changing settings.
3. PostgreSQL examples in this chapter explain its version-17 documentation. The executable companion work uses the recorded SQLite runtime. A PostgreSQL-looking illustration is never presented as output from a PostgreSQL server that was not run.

18.1 From SQL to a plan

18.1.1 PLAIN — in simple words

1. The database first needs to understand the query's structure and names. It then chooses operations that can produce the requested result and executes those operations.
2. There can be several correct routes. An order can be found by scanning a small table or navigating an index. A join can begin with different inputs if the transformation preserves the required meaning.
3. The optimiser chooses using available information and a cost model. "Optimiser" does not mean it knows every future wait or proves the globally fastest possible execution.

18.1.2 PLAIN — a picture in your head

1. A recipe says what dish to produce, while a kitchen work plan decides which preparation steps can be combined, reordered or shared.
2. A cook may prepare a small sauce first to avoid carrying a large pot through every later step.
3. **Where the comparison breaks:** a database transformation must preserve formal query semantics. It cannot replace an expensive ingredient with a different one simply because the result seems similar. Outer joins, NULL and duplicate contributions limit legal reorderings.

18.1.3 PLAIN — a worked example

1. Ask for each order's agreed total, retaining only totals above 12,000 paise. The canonical answer is O-1042 with 17,100; O-1043's 11,550 does not qualify.

```
SELECT order_id, SUM(quantity * unit_price_minor) AS total_minor
FROM order_lines
GROUP BY order_id
HAVING SUM(quantity * unit_price_minor) > 12000
ORDER BY order_id;
```

2. A conceptual flow is: obtain lines, group them by order, calculate each sum, test the group total, then produce the requested ordering. This describes logical responsibilities, not literal measured node output.
3. Filtering individual lines above 12,000 instead would answer a different question. It would discard the pen contribution and fail to produce the correct order total.

18.1.4 PLAIN — what is really happening inside

1. Parsing identifies syntax; name and type resolution connect expressions to objects and operations. Rewriting can expand views or apply other supported transformations.
2. Planning considers access paths, join arrangements and methods, estimated row counts, ordering and intermediate work. Execution follows the resulting plan while interacting with storage and transaction visibility.
3. A named common table expression can make a query easier to read, but does not universally force a physical temporary table. Whether it is folded or materialised depends on engine rules and query properties. [S116]

18.1.5 TECHNICAL — the engineer's version

1. PostgreSQL describes parsing, rewriting, planning and execution as distinct stages. A query tree and an execution plan are different representations. Views can be expanded through the rewrite system. [S111]
2. Plan search is bounded by computational feasibility and engine heuristics. Statistics and configured costs describe an estimated environment, not complete knowledge of runtime conditions. PostgreSQL explicitly documents cases where exhaustive search is impractical. [S112]
3. Legal rewrites preserve the result contract, including multiplicity and NULL behaviour. Optimisation evidence begins with result equivalence, not a shorter SQL string.

18.1.6 WORDS — remember these

1. **Execution plan:** the chosen route to the answer — a structured set of access, join, calculation and ordering operations used by the executor. **Query rewrite:** a semantics-preserving change in representation — transformation of a query before or during planning under the engine's rules. **Materialisation:** save an intermediate result for reuse — creation of a stored working representation rather than recomputing or streaming every use.

18.2 Statistics and estimates

18.2.1 PLAIN — in simple words

1. The optimiser needs an idea of how much data each stage will produce. It usually relies on summaries and samples rather than executing every possible route first.
2. Those summaries can be incomplete or outdated. A rare value can become common, or two fields can be related in a way that separate summaries do not capture.
3. A wrong early estimate can affect later choices. Planning for ten candidate rows when there are 100,000 can lead to a route whose repeated work becomes expensive.

18.2.2 PLAIN — a picture in your head

1. Mira hires one temporary clerk after estimating ten deliveries. A promotion unexpectedly creates ten thousand. The staffing choice was based on the estimate, not on knowledge of the actual workload.
2. Knowing only the average number of deliveries per branch can also miss one unusually busy branch.
3. **Where the comparison breaks:** planner statistics are structured numerical summaries and engine-specific estimators. The example illustrates information limits, not a claim that a database literally allocates one worker for ten rows.

18.2.3 PLAIN — a worked example

1. Imagine a synthetic 100,000-row table where 90,000 rows have status `closed` and 10,000 have status `open`. A uniform two-value assumption would estimate 50,000 for either status and misrepresent both.
2. Now suppose every open row belongs to branch A. Estimating `status='open' AND branch='A'` by multiplying independent fractions can substantially undercount the actual open-A population.
3. Inspect the earliest stage where an estimate departs from observed rows. A later sort receiving too many rows may be a consequence rather than the origin of the mistake.
4. These counts are invented to expose assumptions. They are not copied planner outputs or a benchmark of a statistics feature.

18.2.4 PLAIN — what is really happening inside

1. Statistics can include approximate distinct counts, frequent values, histograms and selected correlations. No compact summary describes every possible predicate combination exactly.
2. Collection has costs and policies. A newly loaded table may need updated statistics before its distribution is represented usefully.
3. Even with current statistics, complex expressions, parameter uncertainty or correlations outside the supported summaries can produce errors. Re-running statistics collection is not a universal cure.

18.2.5 TECHNICAL — the engineer's version

1. Distinguish relation cardinality, predicate selectivity and join cardinality estimates. PostgreSQL's row-estimation examples show how statistics feed specific estimators; they are not guarantees for arbitrary business distributions. [S115]
2. Compare estimated and actual rows at corresponding nodes, accounting for loops and the output convention. A large ratio at an early node is a clue to investigate, not proof that one particular setting is wrong.
3. Extended statistics can represent selected multicolumn relationships. Their existence does not imply that every join or expression correlation is modelled. Keep the exact engine version and supported statistic type in the diagnosis. [S94]

18.2.6 WORDS — remember these

1. **Histogram:** a summary of how values are distributed — a bucketed representation used for approximate frequency or range reasoning. **Cardinality estimate:** a predicted row count — the planner's expected output size for an operation under its available statistics and model. **Correla-**

tion: values vary together — a relationship that can invalidate an assumption that conditions are independent.

18.3 Join strategies

18.3.1 PLAIN — in simple words

1. The same matching rule can be implemented in different ways. A nested loop checks matching work for each row of an outer input. A hash join builds a keyed working structure on one input and probes it with the other. A merge join advances through suitably ordered inputs.
2. None is always best. A small outer input with a useful lookup index can make nested loops effective. Large equality joins can favour hashing. Already ordered inputs can make merging attractive.
3. Every strategy must preserve the matches and multiplicities required by the logical join. A faster algorithm is not allowed to drop repeated valid rows.

18.3.2 PLAIN — a picture in your head

1. To match two lists, you could search the second list separately for every name on the first; build a name-to-record guide for one list; or sort both lists and walk them together.
2. The work depends on list sizes, available ordering and memory.
3. **Where the comparison breaks:** database joins can use indexes, batches, parallelism, spilling and complex conditions. The three pictures explain families of strategies, not exact runtime behaviour for every node.

18.3.3 PLAIN — a worked example

1. Use invented key lists A=`[1,2,2]` and B=`[2,2,3]`. An equality join contains four key-2 pairs because two A entries match two B entries.
2. A naive nested loop examines nine pairs and retains four. A toy hash join stores both B entries for key 2, then emits two matches for each of A's two key-2 entries.
3. A merge-style walk groups the equal keys on both ordered sides and emits the same two-by-two combinations. Advancing both cursors once and emitting only one pair would be incorrect.
4. This small oracle checks semantics independently of any database's chosen strategy. Larger inputs are needed to compare performance meaningfully.

18.3.4 PLAIN — what is really happening inside

1. Nested-loop cost depends strongly on the inner operation. Repeating a full scan is different from repeating a selective index lookup, and repeated keys may enable reuse in some engines.
2. A hash join needs memory for its build side and must resolve collisions with actual equality checks. If working data exceeds available memory, partitioning and spill may add work.
3. A merge join needs compatible ordering and comparison rules. Obtaining that order can require sorts, so count preparation costs rather than praising only the final merge step.

18.3.5 TECHNICAL — the engineer's version

1. PostgreSQL documents nested-loop, merge and hash join strategies. The planner considers their prerequisites and costs; the logical join type, predicate and available paths constrain legal choices. [S112]

2. In a simplified naive nested loop, comparison work is O(NM). An indexed inner lookup changes the model. A well-distributed in-memory hash join can approach O(N+M+K), where K is output size, but its assumptions and output multiplicity must be stated.
3. The output term matters: a many-to-many equal-key group can produce a large result even with an efficient algorithm. No implementation can return K distinct output contributions without accounting for them somewhere.

18.3.6 WORDS — remember these

1. **Nested-loop join:** repeat matching work for each outer record — a join strategy whose inner access can range from a full scan to a selective lookup. **Hash join:** match through a temporary hashed key structure — a build-and-probe strategy that still checks equality and preserves required duplicates. **Merge join:** match while advancing through ordered inputs — a strategy using compatible sorted join keys and correct handling of equal-key groups.

18.4 Access-path choices

18.4.1 PLAIN — in simple words

1. A plan is more than a choice between "index" and "no index." It combines ways to obtain rows, match them, reduce them and order the result.
2. An index can be useful for selecting a range, supplying order or carrying needed values. A scan can be useful when much of the input is required.
3. The cheapest first step is not always part of the cheapest whole plan. A slightly more expensive ordered input might avoid a much larger sort later.

18.4.2 PLAIN — a picture in your head

1. A train ticket with a slightly longer first leg can make the overall journey shorter by avoiding a lengthy transfer.
2. Comparing only the first station's travel time misses the rest of the route.
3. **Where the comparison breaks:** plan costs are not simply a sum of independent journey times. Pipelining, blocking operations, parallelism and shared work can change the relationship between local work and elapsed time.

18.4.3 PLAIN — a worked example

1. Suppose a synthetic report filters one branch and orders by creation time. Candidate A scans the table, filters, then sorts. Candidate B uses an ordered `(branch_id, created_at)` index and retrieves matching records.
2. If the branch contains very few narrow results, B may save work. If it contains most of a wide table, A may be competitive. If B covers all selected fields, the comparison changes again.
3. Do not attach invented millisecond numbers to these possibilities. Capture actual plans and measurements for a defined dataset.
4. A result-equivalence check must include ties, NULL handling and the final ORDER BY. Equal totals alone do not establish equal ordered records.

18.4.4 PLAIN — what is really happening inside

1. The planner tracks useful properties of intermediate paths, such as ordering and estimated size, while considering later operations.

2. Some nodes can emit rows as they arrive; others need to accumulate input before producing their answer. A full sort and a final aggregate can create startup work even when the eventual result is small.
3. A query returning only the first few ordered rows may value startup cost differently from a full export. Measure the use case actually required rather than assuming all consumers need the entire result immediately.

18.4.5 TECHNICAL — the engineer's version

1. PostgreSQL's EXPLAIN distinguishes startup and total estimated cost. These support different consumption patterns, including limited results; they are not measured wall-clock durations. [S113]
2. Path properties can make an apparently more expensive local access useful globally. Ordering supplied by an index can affect sort and merge opportunities. [S106]
3. Planner method settings are diagnostic levers, not universal recommendations. Disabling one strategy can expose an alternative for investigation but does not establish that the alternative is correct for all parameter values or concurrent workloads. [S114]

18.4.6 WORDS — remember these

1. **Startup cost:** estimated work before the first result — a planner quantity relevant to operations that must prepare or consume input before emitting rows. **Blocking operation:** input must accumulate before output can proceed — a stage such as a full sort under its particular execution strategy. **Path property:** useful information about an intermediate route — characteristics such as ordering, parameter dependence and estimated size used in planning.

18.5 Reading EXPLAIN

18.5.1 PLAIN — in simple words

1. Read a plan from the data sources through the operations that use them. Ask what each node receives, what it returns and how often it runs.
2. Keep estimates separate from measurements. A plan without execution observations tells you what the engine intends and predicts, not what actually happened.
3. Diagnostic commands differ across engines. In PostgreSQL, adding ANALYZE executes the statement. SQLite's EXPLAIN QUERY PLAN provides a different, more compact description and does not supply PostgreSQL's runtime fields.

18.5.2 PLAIN — a picture in your head

1. A work order describes the tasks expected on a factory line. A completed production log adds what was actually processed and how long it took.
2. Repeating a task 1,000 times matters even when the time per repetition is small.
3. **Where the comparison breaks:** plan-node timings may include descendant work and use per-loop conventions. Adding every displayed duration can double-count. Read the documented units and nesting rather than treating the display as independent stopwatch rows.

18.5.3 PLAIN — a worked example

1. In the SQLite practice database, inspect the product query using bound values:

```
sql = """SELECT order_id, line_no, quantity
         FROM order_lines WHERE product_id = ?
         ORDER BY order_id, line_no"""
plan_rows = db.execute("EXPLAIN QUERY PLAN " + sql, ("P-NOTE",)).fetchall()
result_rows = db.execute(sql, ("P-NOTE",)).fetchall()
```

2. Save the actual plan rows, result rows and SQLite version. The concatenation here adds a fixed diagnostic prefix to a fixed authored query; it does not interpolate untrusted values into SQL.
3. For PostgreSQL, a documented diagnostic form is `EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON) SELECT ...`. This is not executed in the SQLite lab, and the ellipsis is explanatory notation, not runnable SQL.
4. A hypothetical node reporting 4 output rows per loop over 100 loops represents approximately 400 output-row events under that convention. It does not imply 400 distinct business records.

18.5.4 PLAIN — what is really happening inside

1. Planning-only output is an estimate. Execution instrumentation can add row counts, loops, memory and buffer observations, with overhead of its own.
2. Buffer hits, reads, dirtied pages and temporary work have different meanings. A buffer read does not necessarily mean the physical device was uncached at every lower layer.
3. EXPLAIN ANALYZE does not reproduce a normal client's full result-delivery experience. PostgreSQL's documented options can include serialization work, but network transmission still needs a separate application-level measurement. [S113]

18.5.5 TECHNICAL — the engineer's version

1. Read node estimates and actual values using the engine's documented per-loop and inclusive conventions. Do not sum nested times or buffer totals without accounting for included child work. [S65]
2. PostgreSQL ANALYZE execution can cause the statement's ordinary side effects. A database ROLLBACK does not universally undo external effects of invoked functions or restore consumed sequence values. Use an isolated authorised experiment, not a casual production probe. [S113]
3. SQLite's EXPLAIN QUERY PLAN text is intended for interactive diagnosis and may change across releases. Retain raw output as evidence without making a library depend on one exact wording. [S90]

18.5.6 WORDS — remember these

1. **Instrumentation:** measurements added to execution — observation code collecting timings, counts or resource evidence, potentially with its own overhead. **Loop count:** how often a node is invoked — an execution multiplicity that must be considered alongside per-loop measurements. **Buffer hit:** a needed page was already in the relevant cache — an engine-level observation, not a statement that all possible lower-level work vanished.

18.6 Bad estimates and safe intervention

18.6.1 PLAIN — in simple words

1. Start with a reproducible slow case and its expected answer. Preserve the query, data profile, settings and baseline observations before changing anything.

2. Find the earliest important mismatch between expected and observed work. Check data distribution, statistics, predicate meaning and available paths before reaching for a setting that forces a favourite node.
3. Change one justified factor in an isolated environment. Recheck both answers and performance across the workload envelope, then document costs and remaining uncertainty.

18.6.2 PLAIN — a picture in your head

1. When a delivery route is slow, first determine whether the delay is at the warehouse, on the road or at the customer's door. Replacing the van does not fix a two-hour queue at dispatch.
2. A useful experiment changes a suspected cause while keeping enough of the rest comparable.
3. **Where the comparison breaks:** some database changes interact strongly, and a perfectly isolated variable is not always possible. Record those interactions instead of pretending the comparison is controlled when it is not.

18.6.3 PLAIN — a worked example

1. A hypothetical query is estimated to find 20 open tasks but actually finds 80,000 after a bulk import. First verify that the open-task population is correct and that the import did not misclassify statuses.
2. Inspect statistics freshness and the actual parameter values. Updating statistics in a test environment may change the plan; it may also leave the relevant error unchanged if the estimator still lacks a needed relationship.
3. A candidate partial index or narrower projection can then be tested against the same result contract. Include the common closed-task case and a missing-value case to avoid fixing one parameter while degrading another.
4. The intervention record should show before/after query results, plans, sample distributions, durations, write impact and the exact change. A screenshot of a lower number alone is not sufficient.

18.6.4 PLAIN — what is really happening inside

1. A plan regression can come from changed data, schema, statistics, configuration, software version or workload contention. More than one may change together.
2. Reproducibility requires saving the relevant state, not just SQL text. A query with different bindings is a different test input.
3. A successful experiment supports a bounded conclusion: this change improved this tested workload under these conditions. Production rollout adds monitoring, rollback and compatibility responsibilities.

18.6.5 TECHNICAL — the engineer's version

1. Use a hypothesis-driven sequence: establish correctness; identify the dominant discrepancy; collect bounded evidence; choose a reversible intervention; validate representative cases; and record operational consequences.
2. PostgreSQL exposes planner-method and cost parameters, but its documentation treats these as parts of a model. Global changes can affect unrelated queries, so local experimental evidence is not blanket tuning authority. [S114]
3. The query-plan lab supplies educational observations and equivalence checks. It does not provide a production advisor, automatic index deletion policy or proof of the globally optimal plan.

18.6.6 WORDS — remember these

1. **Baseline:** the saved comparison point — a defined workload, configuration and observed result before an intervention. **Hypothesis-driven tuning:** change what evidence suggests — performance investigation linking an explicit suspected cause to a controlled test. **Regression envelope:** cases checked for unintended deterioration — representative workloads beyond the one case an intervention aims to improve.

18.97 Practice and worked answers

1. **Question:** Can HAVING on an order total be replaced by the same threshold on individual line amounts? **Answer:** Not generally. Group totals and individual contributions are different quantities; the replacement changes the question.
2. **Question:** A hash join receives two left and three right rows with the same key. How many matching pairs are required? **Answer:** Six. The algorithm must preserve the logical two-by-three multiplicity.
3. **Question:** A plan's final sort receives 100,000 rows instead of an estimated ten. Where should diagnosis begin? **Answer:** Trace backward to the earliest important estimate divergence and verify the population. The sort can be a downstream symptom.
4. **Question:** Is a named WITH clause always a physical temporary table? **Answer:** No. Folding and materialisation depend on engine rules and query properties; inspect the relevant plan.
5. **Question:** Why not add every node's displayed time to get total latency? **Answer:** Nodes can include descendant work and use per-loop conventions. Summing without understanding them can double-count.
6. **Question:** Does EXPLAIN ANALYZE safely avoid executing a modifying statement? **Answer:** No. It executes it. Use isolated, authorised inputs and account for side effects beyond transactional row changes.
7. **Question:** A forced plan is faster for one rare key. Is that a global tuning result? **Answer:** No. Test frequent, absent and correlated cases, concurrent effects and affected writes before drawing a broader conclusion.
8. **Question:** What distinguishes a useful intervention record from a before/after screenshot? **Answer:** Exact inputs, correct outputs, environment, plans, repeated observations, the change made, resource effects and stated limitations.

18.98 Common wrong ideas

1. **Wrong:** the optimiser proves the fastest possible execution. **Right:** search and estimates are bounded and can be wrong.
2. **Wrong:** SQL text order is physical execution order. **Right:** legal transformations and plans determine execution.
3. **Wrong:** one join strategy is always superior. **Right:** inputs, ordering, memory, indexes and output size matter.
4. **Wrong:** a low startup cost means low total export cost. **Right:** consumption patterns differ.
5. **Wrong:** estimates are measured row counts. **Right:** execution evidence must be collected separately.
6. **Wrong:** a buffer hit means no work. **Right:** CPU, visibility, comparison and output processing remain.

7. **Wrong:** a query-plan command has identical semantics across engines. **Right:** syntax and execution behaviour are product-specific.
8. **Wrong:** forcing a prettier plan is a successful optimisation. **Right:** correct results and measured workload improvement are required.

18.99 Chapter summary in 20 lines

1. A query specifies a result; a plan chooses an execution route.
2. Parsing, rewriting, planning and execution serve different roles.
3. Legal transformations preserve multiplicity and NULL semantics.
4. Optimiser search is constrained by information and computational cost.
5. Statistics summarise data rather than predicting every question exactly.
6. Skew and correlation can create large estimate errors.
7. Find the earliest significant mismatch, not only the last expensive node.
8. Nested loops repeat an inner operation for outer rows.
9. Hash joins build and probe while checking real keys.
10. Merge joins require compatible ordered inputs and correct duplicate groups.
11. Output multiplicity can dominate even an efficient join.
12. Useful path properties affect the whole plan, not only one step.
13. Startup and total cost represent different estimated boundaries.
14. Planning output and execution observations must be kept separate.
15. Loops and inclusive node measurements require careful interpretation.
16. EXPLAIN ANALYZE executes the supplied statement.
17. Server instrumentation does not replace end-to-end client timing.
18. Preserve a baseline before changing schema or settings.
19. Test representative regressions as well as the desired improvement.
20. A tuning conclusion is only as broad as its evidence.

19

CHAPTER

Aggregation, Windows and Useful Reports

19.0 What this chapter gives you

1. A report is a calculation over a defined population. Its title, denominator and treatment of missing data matter as much as its SQL syntax.
2. You will move deliberately between line, order and reporting-period grain; calculate weighted averages; use windows without collapsing detail; and reconcile totals back to identified contributions.
3. The canonical four-line amounts remain unchanged. Additional numerical examples are explicitly separate teaching populations, not new transactions silently added to Mira's Corner.

19.1 Groups and their grain

19.1.1 PLAIN — in simple words

1. Grouping collects rows that share selected values so an aggregate can describe each collection. A line table can become one total per order, or one quantity per product, depending on the chosen grouping.
2. The output row now means something different. A row that previously represented one line may now represent an entire order's calculated total.
3. Say which input records qualify before grouping. A report for accepted orders, all drafts or only delivered items uses different populations even if the same amount expression appears in each query.

19.1.2 PLAIN — a picture in your head

1. Mira sorts slips into envelopes by order number, then adds the amounts in each envelope. Sorting the same slips by product instead creates different envelopes and different useful answers.
2. A summary envelope is not another original sale. It is a derived description of the slips selected for it.
3. **Where the comparison breaks:** SQL grouping follows comparison and NULL rules, not a clerk's judgement. It can group missing values together without proving that the underlying unknown real-world values are the same.

19.1.3 PLAIN — a worked example

```
SELECT order_id,
       COUNT(*) AS line_count,
       SUM(quantity) AS unit_count,
       SUM(quantity * unit_price_minor) AS total_minor
FROM order_lines
GROUP BY order_id
```

```
ORDER BY order_id;
```

1. O-1042 produces two lines, three units and 17,100 paise. O-1043 produces two lines, three units and 11,550 paise.
2. Grouping instead by `product_id` gives P-NOTE three units and 22,650 paise; P-PEN three units and 6,000 paise.
3. Both sets reconcile to six units and 28,650 paise. They answer different questions while accounting for the same four line contributions.

19.1.4 PLAIN — what is really happening inside

1. The grouping keys determine which rows share an aggregate state. SUM accumulates values; COUNT increments under its defined condition; other aggregates maintain different working state.
2. A WHERE condition selects input rows. HAVING selects groups after aggregation according to group-level conditions. Moving a threshold between them can change the answer.
3. Selecting an arbitrary ungrouped description beside a grouped total can make a report ambiguous or engine-dependent. Join a clearly identified description at the correct grain or include the appropriate grouping key.

19.1.5 TECHNICAL — the engineer's version

1. Define input grain, qualification and output grain in the report contract. Grouping and aggregation do not establish that the input has no accidental join fan-out; that must be checked separately. [S58] [S19]
2. SQL grouping treats NULL values as belonging to the same grouping category for this purpose. That grouping convention does not assert equality of unknown business identities.
3. The physical engine may aggregate with sorting, hashing or another supported strategy. The query's grouping semantics are distinct from the selected implementation.

19.1.6 WORDS — remember these

1. **Grouping key:** the fields defining each collection — expressions whose equal grouping values identify one aggregate group. **Report grain:** what one result row describes — the precise unit represented after selection, joining and aggregation. **Aggregate state:** information accumulated for a group — working values sufficient to compute an aggregate such as a sum and count.

19.2 Counts, sums and averages

19.2.1 PLAIN — in simple words

1. A count needs a noun: rows, known prices, distinct customers or physical units. Those quantities are not interchangeable.
2. An average needs a denominator. Average order value divides order amount by orders; average price per unit divides amount by units. Averaging line prices gives each line equal weight, not each unit.
3. Missing values must not silently become zero. A zero price can be a known free item; an unknown proposed price is a different state.

19.2.2 PLAIN — a picture in your head

1. Two boxes contain one pen and nine notebooks. Averaging the two box labels gives each box equal influence. Averaging the ten individual items gives the larger box nine times the influence.
2. Neither calculation is mysterious once you say what is being averaged.
3. **Where the comparison breaks:** a SQL aggregate's NULL behaviour is a defined rule, not a human decision to exclude an incomplete observation. The report author must explain whether that exclusion matches the intended metric.

19.2.3 PLAIN — a worked example

1. The canonical mean order amount is `28,650 / 2 = 14,325` paise. The mean line amount is `28,650 / 4 = 7,162.5` paise. The quantity-weighted mean unit price is `28,650 / 6 = 4,775` paise.
2. These are three different metrics. Their fractional results need a display-rounding policy; calculating an average does not create a half-paise payable obligation.
3. In a separate two-line example, buy one pen at 2,000 and nine notebooks at 7,550. The unweighted average of line unit prices is 4,775, but the unit-weighted mean is `(2,000 + 9×7,550)/10 = 6,995` paise.
4. In the draft D-SQL-01, one proposed price is NULL. A SUM over known line amounts does not become a complete quote merely because the database returns a number. Report the missing-price count alongside any known subtotal.

19.2.4 PLAIN — what is really happening inside

1. COUNT(*) counts input rows. COUNT(expression) counts non-null expression results. SUM and AVG generally ignore NULL arguments in these documented SQL implementations, so missingness can change a denominator. [S19]
2. An empty aggregate input needs explicit interpretation. COUNT is zero; SUM can be NULL. COALESCE can present a chosen zero, but the application must decide whether "no observations" and "observed total zero" are equivalent for this report.
3. Combining subgroup averages requires their weights. A branch with one order and a branch with 999 orders must not receive equal weight when calculating the average order value across all 1,000 orders.

19.2.5 TECHNICAL — the engineer's version

1. A weighted mean is `Σ(w_i x_i) / Σw_i` for the defined included observations and valid weights. Guard the zero-denominator case and state whether zero, NULL or a labelled "not available" result is intended.
2. Preserve sufficient statistics for recombination. Sum and count can combine to recover an overall mean; an unweighted average of subgroup means generally cannot.
3. Integer division and numeric result types differ across engines and expressions. Use an explicit numeric conversion where fractional results are intended, and distinguish exact arithmetic from floating-point approximation. [S21] [S80]

19.2.6 WORDS — remember these

1. **Denominator:** what the total is divided by — the explicitly defined count or weight underlying a ratio. **Weighted mean:** larger contributions receive proportionate influence — the sum of value-

times-weight divided by total valid weight. **Known subtotal:** a total over available values only — a partial calculation that must not be labelled complete when required observations are missing.

19.3 Windows versus groups

■ 19.3.1 PLAIN — in simple words

1. Grouping normally collapses several detail rows into one result per group. A window function can calculate across related rows while keeping each detail row visible.
2. This is useful for running totals, rankings and comparisons with a previous observation. The calculation needs a partition, an ordering where relevant, and sometimes an explicit frame describing which neighbouring rows contribute.
3. The order used inside a window calculation does not automatically promise the final display order. Keep the final ORDER BY as well.

■ 19.3.2 PLAIN — a picture in your head

1. Instead of putting slips into one summary envelope, write a running subtotal beside each slip while leaving every slip on the page.
2. A fresh running total can begin for each order, just as a new column of arithmetic starts on a new receipt.
3. **Where the comparison breaks:** a window frame can include peers with equal ordering values or a selected range of rows. It is not always simply "everything printed above this line."

■ 19.3.3 PLAIN — a worked example

```
SELECT order_id, line_no,
       quantity * unit_price_minor AS line_minor,
       SUM(quantity * unit_price_minor) OVER (
         PARTITION BY order_id ORDER BY line_no
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS running_minor
FROM order_lines
ORDER BY order_id, line_no;
```

1. For O-1042, the running amounts are 15,100 and 17,100. For O-1043, they are 7,550 and 11,550. Four detail rows remain.
2. In a separate ordered population with values 100, 100 and 250, a row-by-row SUM with a unique tie-breaker gives 100, 200 and 450. A peer-inclusive range frame ordered only by value can give 200, 200 and 450.
3. The different answers follow different frames. Neither should be called an arithmetic failure when the intended frame was not specified.

■ 19.3.4 PLAIN — what is really happening inside

1. PARTITION BY separates independent window calculations. ORDER BY inside OVER arranges rows for order-sensitive functions. The frame then identifies the subset used by frame-sensitive calculations.
2. ROW_NUMBER assigns positions; RANK gives tied peers the same rank and leaves gaps; DENSE_RANK gives tied peers the same rank without those gaps. On 100,100,250 in ascending order, ranks are 1,1,3 and dense ranks 1,1,2.

3. LAST_VALUE can surprise readers because its default frame may end at the current peer group rather than the partition's last row. Request the whole-partition frame when that is actually the intended question. [S110]

19.3.5 TECHNICAL — the engineer's version

1. Window partition, ordering and frame are distinct clauses. ROWS, RANGE and supported GROUPS frames count or interpret boundaries differently; peers are rows equal under the window ordering. [S97]
2. A stable row-by-row running total needs an ordering that distinguishes ties, such as `(event_time, event_id)` when that identifier supplies the intended deterministic order. Determinism does not prove that this tie-break is causal order.
3. Windows operate over the query's relevant result stage, not automatically over raw base rows. A common table expression can make a prior grouping stage explicit before applying a window to grouped totals. [S96] [S116]

19.3.6 WORDS — remember these

1. **Window function:** calculate across related rows without collapsing each one — an SQL function evaluated over a partition and, where applicable, an ordered frame. **Peer group:** rows tied under a window's ordering — entries equal on all ordering expressions used to define peers. **Window frame:** the contributing region for one current row — a bounded subset of its partition under ROWS, RANGE or another supported frame mode.

19.4 Time buckets

19.4.1 PLAIN — in simple words

1. A daily report needs a definition of "day." The event's time, the system's recording time and the business's reporting timezone can produce different groupings.
2. A late-arriving record can describe yesterday's event. Decide whether to revise yesterday's report, show an adjustment today or use another explicit policy.
3. Adjacent periods need boundaries that do not double-count events exactly at midnight. A half-open interval includes its start and excludes its end.

19.4.2 PLAIN — a picture in your head

1. Mira places slips into dated trays according to when the sale happened, not necessarily when Dev carried the slip to the desk.
2. A tray labelled "Tuesday" is ambiguous unless everyone agrees which clock and calendar define Tuesday.
3. **Where the comparison breaks:** timezones can have changing offsets and daylight-saving transitions. A local calendar day need not equal exactly 24 hours of elapsed time. Calendar grouping and duration measurement are different tasks.

19.4.3 PLAIN — a worked example

1. In a separate UTC exercise, event E1 occurs at `2026-09-01T23:59:59Z`; E2 occurs at `2026-09-02T00:00:00Z`. The interval from September 1 midnight inclusive to September 2 midnight exclusive includes E1 but not E2.
2. The next daily interval includes E2. The shared boundary creates no duplicate contribution.

3. If E1 is recorded after E2, an event-time report still places it on September 1. A recording-time report can place it on September 2. Both need labels that make the distinction visible.
4. These invented timestamps are not dates assigned to the canonical orders, whose earlier fixture did not define a complete business-time reporting policy.

19.4.4 PLAIN — what is really happening inside

1. Parse and validate timestamp representations before grouping. Text prefixes only behave as reliable date keys when the representation and timezone policy make that interpretation valid.
2. Preserve the original instant and relevant source context when needed. A display conversion should not silently replace the stored event time with a server-local assumption.
3. Time buckets also need an inclusion policy for missing or invalid times. Dropping those records without a count can make a daily dashboard appear more complete than its evidence.

19.4.5 TECHNICAL — the engineer's version

1. Specify event-time versus processing-time semantics, timezone or UTC convention, interval boundaries and late-correction policy. Chapter 41 develops these requirements for streaming systems.
2. Prefer explicit `>= start AND < end` boundaries for adjacent instant ranges when that matches the metric. A local calendar boundary should be converted with the appropriate timezone rules rather than assuming every day is a fixed number of seconds.
3. Retain report version or as-of metadata when later records can change an earlier period. A reproducible historical report needs its source snapshot and transformation version, not just a date label.

19.4.6 WORDS — remember these

1. **Time bucket:** the period a record contributes to — a reporting interval defined by clock semantics, timezone and boundary rules. **As-of time:** when the available information was evaluated — the observation or snapshot boundary attached to a report. **Late arrival:** a record received after its event period — an observation whose processing time occurs later than the event-time window it describes.

19.5 Distinctness and duplicates

19.5.1 PLAIN — in simple words

1. A distinct count needs an identity rule. Counting distinct display names does not necessarily count people, and counting distinct prices does not count sale lines.
2. Repeated delivery of one event and two separate events with equal values must be treated differently. The difference comes from event identity and policy, not visual similarity.
3. A deduplicated report should say what was considered the same and what happened to conflicts. Silent removal can hide a data-quality problem.

19.5.2 PLAIN — a picture in your head

1. Two photocopies of one receipt should not create two sales. Two genuine receipts with the same amount should still count twice.
2. The receipt's identity and provenance distinguish those cases; the amount alone does not.

3. **Where the comparison breaks:** a real system can receive conflicting payloads under one identifier. Choosing the first or last without a defined rule can hide corruption or policy disagreement. Deduplication is not merely deleting identical-looking paper.

19.5.3 PLAIN — a worked example

1. The separate O-REPEAT fixture has two different line numbers with the same notebook amount, 7,550 each. Counting distinct amounts gives one value; counting lines gives two; summing amounts gives 15,100.
2. A redelivered copy of `(O-REPEAT,1)` with identical payload is different from line 2. Under a declared input policy, the repeated line key can be recognised as a replay while line 2 remains a separate contribution.
3. If the replay changes quantity, it is a conflicting same-key observation, not an ordinary duplicate. The earlier import and event-consumer policies must be respected rather than replaced by a new "last one wins" rule.

19.5.4 PLAIN — what is really happening inside

1. SQL DISTINCT applies to the selected expressions. Removing duplicate projected values can collapse records that differ on unselected identity fields.
2. Distinct aggregation also has NULL rules and resource costs. Exact distinct counting may need to retain or sort many keys. Approximate counting structures answer a different question with an error model that must be disclosed.
3. Combining distinct counts across groups by addition can double-count identities appearing in more than one group. Preserve the union or a suitable reviewed mergeable representation when the metric is overall distinct identity.

19.5.5 TECHNICAL — the engineer's version

1. Define the deduplication key, scope, payload-equivalence rule, conflict disposition and retention horizon. A SQL DISTINCT operation is only one possible implementation step, not the full policy. [S57]
2. `COUNT(DISTINCT expression)` counts distinct non-null expression values in the documented aggregate semantics. It is not a universal distinct-person estimator. [S19]
3. Exact totals, exact distinct counts and approximate sketches have different composability properties. Never merge subgroup metrics without verifying the algebra required by that metric.

19.5.6 WORDS — remember these

1. **Distinct count:** count different values under a rule — cardinality of the included value set after the specified equality and NULL treatment. **Deduplication key:** what identifies one contribution — the scoped fields used to recognise a replay rather than a separate event. **Mergeable summary:** subgroup evidence can be combined correctly — a summary with a defined merge operation preserving the intended metric or error model.

19.6 Reconciliation totals

19.6.1 PLAIN — in simple words

1. Reconciliation checks that a report accounts for the intended source records and explains differences. It is more than comparing one grand total.

2. Count input records, accepted contributions, exclusions, duplicates and conflicts. Compare identities and amounts at useful intermediate levels.
3. A total can match by accident when one missing amount cancels one duplicated amount. The report needs enough traceability to detect that cancellation.

19.6.2 PLAIN — a picture in your head

1. Mira checks both the total money and the list of receipts placed in the envelope. The right amount with the wrong receipts is not a completed reconciliation.
2. An exclusion note belongs beside the envelope, not in an undocumented memory of why some slips disappeared.
3. **Where the comparison breaks:** a database report can involve snapshots, transformations and late corrections across several stores. A matching receipt list at one stage does not establish the whole pipeline's completeness.

19.6.3 PLAIN — a worked example

1. For the canonical report, expected line identities are `(O-1042,1)`, `(O-1042,2)`, `(O-1043,1)` and `(O-1043,2)`. Expected counts are four lines, two orders and six units.
2. Expected order totals are 17,100 and 11,550; expected product totals are 22,650 and 6,000. Both routes sum to 28,650.
3. Add an explicitly separate empty-order fixture. A report sourced only from order lines still has no row for that order. A report sourced from all orders with a left join can include it, with a chosen zero-line interpretation. The denominator must say which report is intended.
4. A useful report header therefore includes population, period, currency, missing-value treatment, source snapshot and transformation version before presenting a large number.

19.6.4 PLAIN — what is really happening inside

1. Reconciliation compares a defined expected set with the observed contributions, then classifies differences. Missing, extra, duplicated and changed contributions require different explanations.
2. Checks at several grains help locate a defect: overall amount, order totals, product quantities and original line identities. Agreement at one grain cannot substitute for all the others.
3. Save the evidence needed to reproduce the result without exposing unnecessary personal data. A diagnostic report should not become an uncontrolled copy of every sensitive field.

19.6.5 TECHNICAL — the engineer's version

1. A reconciliation contract specifies source snapshot, contribution identity, inclusion predicate, units, arithmetic, grouping and tolerances where appropriate. Exact integer-paise examples use exact equality; uncertain measurements may require a separately justified tolerance.
2. Treat report generation as a versioned transformation. Reproducing a number requires both the relevant source state and the transformation's semantics, not merely a saved SQL filename.
3. These educational checks validate the stated fixture and exercises. They do not certify live financial accounts, legal reporting obligations or an organisation's complete data lineage.

19.6.6 WORDS — remember these

1. **Contribution identity:** which source fact a result uses — a stable scoped identifier for a counted or summed input contribution. **Exclusion register:** explain what did not contribute — recorded

counts and reasons for records outside the report's accepted population. **Transformation version:** identify the calculation rules used — a versioned definition of parsing, filtering, joining and aggregation semantics.

19.97 Practice and worked answers

1. **Question:** Calculate mean order amount, mean line amount and weighted unit price for the canonical fixture. **Answer:** 14,325; 7,162.5; and 4,775 paise respectively. The denominators are two orders, four lines and six units.
2. **Question:** Why does averaging two branch averages usually fail? **Answer:** Branches may have different numbers of contributing observations. Combine sums and counts, or apply the correct weights.
3. **Question:** What running totals appear within O-1042 under line-number ordering? **Answer:** 15,100 followed by 17,100. The detail rows remain present.
4. **Question:** Why can a RANGE running sum for 100,100,250 start at 200? **Answer:** The first two rows are peers under value ordering, and the frame includes that peer group. Use an explicit ROWS frame and unique tie-breaker for a specific row-by-row interpretation.
5. **Question:** An event occurs exactly at a period's exclusive end. Where does it go? **Answer:** Into the next adjoining half-open period, assuming the same time basis and no gap.
6. **Question:** Does one distinct amount mean one sale? **Answer:** No. Separate identified sales can have equal amounts.
7. **Question:** A SUM ignores one unknown draft price. Can it be labelled the complete quote? **Answer:** No. Report the known subtotal and missing contribution, or withhold a complete total under the chosen policy.
8. **Question:** Which canonical controls supplement the 28,650-paise grand total? **Answer:** Four line identities, two orders, six units, the two order totals and the two product totals.

19.98 Common wrong ideas

1. **Wrong:** every count measures the same thing. **Right:** rows, known values, distinct identities and units differ.
2. **Wrong:** all averages can be averaged again. **Right:** weights and sufficient statistics are required.
3. **Wrong:** NULL is a free item. **Right:** missing and known zero have different meanings.
4. **Wrong:** a window removes detail rows like GROUP BY. **Right:** window calculations can retain each detail row.
5. **Wrong:** ordering inside OVER guarantees display order. **Right:** the final result needs its own ORDER BY.
6. **Wrong:** a calendar day always means 24 elapsed hours. **Right:** calendar and timezone rules define the bucket.
7. **Wrong:** DISTINCT discovers business identity. **Right:** it applies equality to the expressions supplied.
8. **Wrong:** a matching grand total completes reconciliation. **Right:** identities, populations and explained exclusions matter too.

19.99 Chapter summary in 20 lines

1. A useful report begins with a defined population and grain.

2. Grouping changes what one output row represents.
3. WHERE qualifies input rows and HAVING qualifies groups.
4. Count rows, known values, identities and units deliberately.
5. An average's denominator is part of its meaning.
6. Weighted means preserve the intended contribution sizes.
7. Subgroup means need weights or sufficient statistics before recombination.
8. Missing prices do not become known zero prices.
9. Empty input and observed zero can require different presentation.
10. Windows calculate across related rows while retaining detail.
11. Partition, ordering and frame are separate choices.
12. Peer-inclusive frames differ from row-by-row frames.
13. Ranking functions treat ties differently.
14. Time buckets need a clock basis, timezone and boundary policy.
15. Half-open adjacent intervals avoid double-counting their shared boundary.
16. Late arrivals require an explicit report-revision policy.
17. Distinct expression values are not automatically distinct business events.
18. Deduplication requires scoped identity and conflict handling.
19. Reconcile contribution keys and intermediate totals, not only the grand total.
20. Preserve source and transformation versions so the report can be explained later.

20

CHAPTER

Measuring Performance Without Fooling Yourself

20.0 What this chapter gives you

1. A performance result is an observation about a defined workload, environment and measurement boundary. Without those definitions, a fast number can be true and still misleading.
2. You will specify a benchmark, calculate latency percentiles under an explicit convention, separate throughput from latency, and report errors and limitations instead of selecting only attractive runs.
3. The companion experiment is a local educational microbenchmark, not a capacity test of KedByte, a comparison of database brands or a claim about production hardware.

20.1 Define the workload

20.1.1 PLAIN — in simple words

1. Before timing, say what the system must do correctly. Two queries that return different populations are not competing implementations of the same task.
2. Specify data size and distribution, query parameters, selected fields, concurrency and the mix of reads and writes. "One million rows" alone leaves most of the workload undefined.
3. Decide where the timer starts and stops. Measuring only server execution differs from measuring connection setup, result transfer, decoding and application work as well.

20.1.2 PLAIN — a picture in your head

1. Timing a runner requires a marked course and finish line. One runner cannot stop after 80 metres while another completes 100 and still be compared as though they ran the same race.
2. A race with hurdles is also different from an empty track, even when the distance is equal.
3. **Where the comparison breaks:** database workloads can include many simultaneous operations and changing state. There may be no single finish line for the entire service, so define per-operation and experiment-level boundaries separately.

20.1.3 PLAIN — a worked example

1. Our query experiment compares the same fixed product-like lookup on separately generated teaching rows before and after a candidate index. It checks the complete ordered result against an independent expected list.
2. Dataset creation and index construction are recorded separately from repeated query execution. The query timer includes fetching all result rows into the local client process.

3. Rare, common and absent keys form separate cases. A combined summary, when used, must specify their weighting rather than silently letting the fastest case dominate.
4. The four canonical order lines remain a correctness fixture. They are not expanded by pretending the same four sales happened thousands of times in the story.

20.1.4 PLAIN — what is really happening inside

1. A workload generator determines arrivals and parameters. The database executes work, while the client observes successes, failures and durations within its timer boundary.
2. A closed-loop generator waits for one response before issuing more work at a fixed number of clients. An open-loop generator schedules arrivals independently of previous completion. Under overload, they expose different queueing behaviour.
3. If the generator slows down whenever the system slows down, it can omit the waiting that independently arriving users would experience. State the arrival model rather than claiming that every benchmark measures the same service demand.

20.1.5 TECHNICAL — the engineer's version

1. A benchmark contract includes schema, indexes, data-generation code and seed, parameter distribution, operation mix, concurrency, arrival process, warm-up, timer boundary, sample count and correctness oracle.
2. Use a monotonic timer for elapsed intervals. Python's `perf_counter_ns()` avoids floating-point timestamp subtraction and returns integer nanoseconds; its unit is not an accuracy guarantee. [S102]
3. Retain exact runtime and library versions. A reproducible dataset does not make two machines' execution conditions identical, so reproducibility of inputs and portability of timing results are separate claims.

20.1.6 WORDS — remember these

1. **Benchmark contract:** the rules of the measurement — a specification of inputs, operations, environment, timing boundaries and success criteria. **Closed-loop load:** clients wait before issuing their next request — a workload in which completion controls later arrivals. **Open-loop load:** arrivals follow an independent schedule — a workload whose offered demand does not automatically slow with each response.

20.2 Latency distributions

20.2.1 PLAIN — in simple words

1. Latency is not one fixed property. The same operation can complete quickly most of the time and occasionally take much longer.
2. An average compresses those differences. Percentiles describe positions in an ordered sample, but their calculation convention and sample size matter.
3. Keep failed and timed-out attempts visible. Reporting only successful durations can make an overloaded system look faster by discarding its worst experiences.

20.2.2 PLAIN — a picture in your head

1. Nineteen customers wait ten seconds and one waits a thousand. Saying "the average wait was 59.5 seconds" is mathematically correct but does not describe either group's experience well.

2. Sorting the waits lets us ask what a chosen fraction of customers experienced.
3. **Where the comparison breaks:** a small observed sample does not establish the long-run distribution. A percentile estimated from twenty requests is not reliable evidence about rare one-in-ten-thousand delays.

20.2.3 PLAIN — a worked example

1. Take an invented sample of twenty latencies: nineteen at 10 ms and one at 1,000 ms. The mean is `(19×10+1,000)/20 = 59.5 ms`.
2. This lab uses the nearest-rank convention: for fraction p with `0 < p <= 1`, select sorted position `ceil(pN)`, counting from one. The median under this convention is 10 ms; p95 selects position 19 and is 10 ms; p99 selects position 20 and is 1,000 ms.
3. Another percentile interpolation convention can give a different numerical p95. Label the convention instead of treating one library's output as the only mathematically possible percentile.
4. These are calculated teaching values, not measured database latencies. The benchmark's raw observed samples are stored separately in its evidence output.

20.2.4 PLAIN — what is really happening inside

1. Latency combines active work and waits within the chosen boundary. Cache misses, locks, scheduling, storage and result size can create different parts of the distribution.
2. Histograms and raw samples preserve more shape than a single mean. Histogram bucket boundaries limit the precision of derived quantiles; raw samples still have sampling uncertainty.
3. A timeout is a censored observation: the client knows it waited at least until its deadline without obtaining the desired completion. Recording the deadline as though it were the actual completion time understates what is unknown.

20.2.5 TECHNICAL — the engineer's version

1. State whether a percentile describes all attempts, successful attempts or another population. Retain attempt counts, success counts, timeout counts and error classes alongside duration summaries.
2. Quantile convention, sample size and workload stationarity affect interpretation. Do not average p99 values from separate groups and call the result the combined p99; combine compatible underlying observations or histogram counts instead.
3. User-oriented monitoring benefits from latency distributions together with traffic, errors and saturation, rather than one average alone. [S103]

20.2.6 WORDS — remember these

1. **Percentile:** a position in an ordered population or sample — a quantile defined under a stated convention, such as nearest rank. **Tail latency:** the unusually slow end of the distribution — high-percentile delays whose meaning depends on the measured population and sample size. **Censored observation:** completion time is only partly known — an observation limited by a timeout or stopping rule rather than a measured final duration.

20.3 Throughput and saturation

20.3.1 PLAIN — in simple words

1. Throughput counts completed work per unit time. Latency measures how long an operation takes. A system can complete many operations per second while each operation waits in a long queue.
2. As demand approaches a bottleneck's capacity, waiting can grow. Adding clients does not guarantee proportionally more completed work.
3. Count useful, correct completions separately from errors or retries. A server returning failures very quickly has high response throughput but may provide little successful service.

20.3.2 PLAIN — a picture in your head

1. One checkout counter can serve customers only so quickly. More people joining the queue increase waiting, not necessarily the number served each minute.
2. Opening a second counter helps only if the shared payment terminal, stock lookup or packing station is not the real bottleneck.
3. **Where the comparison breaks:** computer workloads can overlap CPU, storage and network work, and requests have different costs. One checkout's fixed service time is an intentionally simplified model.

20.3.3 PLAIN — a worked example

1. In a stable hypothetical system, 100 completed requests per second spend an average 0.2 seconds within a defined service boundary. Little's-law reasoning gives an average `100 × 0.2 = 20` requests inside that boundary, under its steady-state accounting assumptions.
2. That count includes any waiting represented by the latency boundary. It is not automatically twenty actively executing CPU tasks or twenty database connections.
3. Suppose offered load increases to 200 requests per second while sustainable completion remains 100 and nothing rejects or slows arrivals. The queue grows by roughly 100 requests per second in this simplified interval.
4. A bounded system must instead apply some policy: admission limits, backpressure, deadlines, rejection or additional capacity. Infinite queues are not a service-quality strategy.

20.3.4 PLAIN — what is really happening inside

1. Bottlenecks can move as load changes. CPU may dominate one workload, storage another, and locks or a connection pool another.
2. Retry traffic can amplify overload. Counting retries as independent fresh business operations can also overstate useful throughput and create correctness problems.
3. Plot or tabulate offered demand, successful completions, errors, queueing and latency together. A throughput plateau with growing latency suggests saturation, but further evidence is needed to identify the constrained resource.

20.3.5 TECHNICAL — the engineer's version

1. Under suitable steady-state conditions and consistent boundaries, Little's law is `L = λW`: average number in the system equals average effective arrival/completion rate times average time in the system. It is an accounting relation, not a universal capacity forecast for transient overload.
2. Distinguish offered rate, admitted rate, completed-attempt rate and successfully completed business-operation rate. Retries and rejected requests separate these quantities.

3. Report saturation signals alongside user outcomes. Resource utilisation alone can miss queueing at a logical bottleneck, and high utilisation is not automatically harmful when latency and reliability requirements remain satisfied. [S103]

20.3.6 WORDS — remember these

1. **Throughput:** completed work per unit time — a rate whose counted operation and success condition must be defined. **Saturation:** a limiting resource is at or near effective capacity — a condition in which additional demand tends to create waiting, rejection or deterioration rather than proportional useful work. **Backpressure:** make upstream work respect downstream limits — a mechanism that slows or bounds production when consumers cannot keep up.

20.4 Warm versus cold state

20.4.1 PLAIN — in simple words

1. A repeated operation can benefit from work and data retained by earlier runs. This warm state may be realistic for frequent requests, but it must be named.
2. Cold state is not one universal condition. A new connection can still use database, operating-system or device caches warmed by another process.
3. Do not clear caches on a live machine merely to manufacture a benchmark. Use an isolated test environment and record what was actually reset.

20.4.2 PLAIN — a picture in your head

1. Dev's first trip finds the right cupboard and brings folders to the desk. A second question about those folders avoids the trip.
2. Starting a new notebook for timings does not move the folders back to a remote archive.
3. **Where the comparison breaks:** there are several cache layers, each with its own lifetime and replacement policy. A single "desk empty" analogy cannot establish the state of all of them.

20.4.3 PLAIN — a worked example

1. In the educational microbenchmark, both variants receive a stated warm-up phase before measured repetitions. The comparison is labelled warm, in-process query execution on generated data.
2. Reopening an in-memory SQLite database with newly inserted rows includes a different preparation history from reopening a file already read by the operating system. Neither should be called a device-cold test without further evidence.
3. Alternate the order of variants or use a recorded random order to reduce systematic first-run advantage. Save all samples, including the first measured one.
4. Index construction remains a separately reported preparation cost. Excluding it from steady-state query timing is acceptable only when that boundary is explicit.

20.4.4 PLAIN — what is really happening inside

1. Warm state can include pages, prepared statements, code paths, allocator state, connection establishment and storage caches. It can also include undesirable accumulated state such as a growing queue.
2. Background work can occur between runs: checkpoints, cleanup or other tenants' activity. A warm-up phase does not make the rest of the environment stationary.

3. Compare the use case you need. Startup latency, infrequent cold retrieval and sustained warm service are different workloads, each potentially important.

20.4.5 TECHNICAL — the engineer's version

1. Define reset scope precisely: process, connection, database buffer pool, filesystem cache, device state or dataset reconstruction. Do not claim a wider reset than the procedure demonstrates.
2. In-memory SQLite timing omits network transport and does not exercise persistent-device durability. That can be appropriate for a teaching comparison but cannot establish server or storage capacity.
3. Counterbalancing run order reduces one source of bias; it does not eliminate all thermal, scheduling or shared-host effects. Record those limitations rather than reporting false environmental control.

20.4.6 WORDS — remember these

1. **Warm state:** earlier work remains useful — cached data or prepared resources available to later operations under a specified scope. **Reset scope:** what was actually returned to a starting condition — the exact layers or resources reinitialised before a measurement. **Counterbalancing:** vary which option runs first — an experimental ordering technique intended to reduce systematic sequence effects.

20.5 Fair comparisons

20.5.1 PLAIN — in simple words

1. Compare equal work with equal correctness requirements. Disabling durability, dropping validation or returning fewer fields changes the service being measured.
2. Use enough repetitions to expose variation, but do not confuse many nearly identical local samples with evidence about every production condition.
3. Preserve unsuccessful runs and reasons for exclusion. A result selected after seeing which measurements look attractive is not a neutral comparison.

20.5.2 PLAIN — a picture in your head

1. Two delivery services cannot be compared fairly if one includes packaging and insurance while the other only moves an unwrapped box across the room.
2. The cheaper service may still be useful, but the difference must be stated rather than hidden inside the word "faster."
3. **Where the comparison breaks:** software guarantees can be subtle. Two functions returning the same value once may differ under retries, concurrent access or failure, so equal happy-path output is necessary but not sufficient.

20.5.3 PLAIN — a worked example

1. Before timing a query variant, compare its ordered rows with an independently computed expected result. A variant that loses one valid duplicate fails correctness even if its duration is lower.
2. Record the same generated dataset, parameter cases and selected fields for both variants. Keep schema and settings differences limited to the proposed intervention unless additional differences are explicitly part of the comparison.

3. Use a fixed number of warm-ups and measured repetitions decided before inspecting the results. Summarise the samples with the same convention.
4. A result such as "variant B was faster in these warm local samples" is narrower and more defensible than "B scales to ten thousand users." The latter requires a different workload and capacity experiment.

20.5.4 PLAIN — what is really happening inside

1. A benchmark harness itself consumes resources. Data generation, logging, hashing results and printing can dominate a tiny operation if placed inside its timer boundary.
2. Conversely, timing only a cursor creation can omit the work performed when rows are fetched. Ensure the measurement includes the actual completion condition.
3. A trusted correctness check can be outside the repeated timing loop when the workload is deterministic, but it must still run for every variant and relevant parameter case. Record where checks occur.

20.5.5 TECHNICAL — the engineer's version

1. Predefine the estimand: the quantity the comparison aims to estimate. Examples include median local fetch latency, successful throughput under a fixed arrival process or p99 end-to-end latency at a stated admitted load.
2. Avoid a durability or isolation mismatch between variants. Different guarantees can be compared as different services, but not silently labelled equivalent implementations.
3. Do not impose a universal speed assertion in a functional test. The educational suite checks result equality and valid measurement records; host scheduling can change timing without making the algorithm incorrect.

20.5.6 WORDS — remember these

1. **Estimand:** the exact quantity a study seeks — the defined performance or outcome measure to which observations are intended to speak. **Harness overhead:** work caused by the measurement setup — time and resources consumed by generation, instrumentation and reporting rather than the target operation alone. **Correctness oracle:** a separate expected-answer check — a mechanism that rejects a fast but semantically wrong variant.

20.6 Reporting limitations

20.6.1 PLAIN — in simple words

1. A useful performance report makes its boundaries easy to see. Readers should know what was measured, what was excluded and what cannot be concluded.
2. Include actual versions, dataset construction, sample counts, result checks and failures. Label simulated or calculated figures as such.
3. Do not turn a local educational run into a claim about production readiness, security, recovery or the number of schools a server can safely host.

20.6.2 PLAIN — a picture in your head

1. A laboratory label says which material was tested, under what load and for how long. It does not simply say "strong."
2. Performance evidence needs the same discipline: the label travels with the result.

3. **Where the comparison breaks:** software and workloads change quickly. A benchmark's input files and version identifiers are often necessary to understand a result later, even when the machine's model name remains unchanged.

20.6.3 PLAIN — a worked example

1. A minimal record includes: Python and SQLite versions; generated row count and distribution; query text and parameter case; schema/index variant; warm-up and repetition counts; timer boundary; raw nanosecond samples; ordered result digest; and any errors.
2. The result digest is a convenient reproducibility check, not an authentication or semantic proof by itself. The independently expected rows remain the correctness basis.
3. Add an interpretation such as: "Warm single-process SQLite query comparison; no network, multi-user load, persistent-device failure or PostgreSQL server tested."
4. This explicit limitation does not make the result worthless. It tells the reader exactly which small question was answered and which larger experiments remain.

20.6.4 PLAIN — what is really happening inside

1. Reproducibility records separate stable inputs from variable observations. The same code and seed can reproduce the data while observed timings differ on another run.
2. Keep raw measurements so summaries can be recalculated under another convention. Preserve original failures rather than overwriting them with a later passing run.
3. A benchmark informs a decision together with correctness, maintainability, operational costs and service requirements. It is not the entire decision.

20.6.5 TECHNICAL — the engineer's version

1. The companion benchmark emits a machine-readable record and makes no host-independent speed guarantee. Functional tests validate its schema, sample count, non-negative durations and result equivalence, not a predetermined winner.
2. Report measurement uncertainty and environmental limits at the level the experiment supports. A narrow microbenchmark does not warrant an invented confidence interval for production throughput.
3. Future capacity work should connect latency, successful throughput, errors and saturation under realistic demand, including failure and retry behaviour. [S103]

20.6.6 WORDS — remember these

1. **Raw sample:** an individual retained observation — an unaggregated measurement from which summaries can be recomputed. **Measurement boundary:** what the timer includes — the explicitly chosen start and completion conditions of an observed operation. **External validity:** whether a result transfers beyond its test — the justified scope of applying observations to other workloads, environments or populations.

20.97 Practice and worked answers

1. **Question:** Nineteen requests take 10 ms and one takes 1,000 ms. Find the mean and nearest-rank p99. **Answer:** The mean is 59.5 ms; p99 selects the twentieth observation and is 1,000 ms.
2. **Question:** Why can two libraries disagree on p95 for the same twenty samples? **Answer:** Quantile interpolation conventions differ. State the convention and retain raw samples.

3. **Question:** At 100 completions per second and 0.2 seconds average time in a stable boundary, what average population does Little's law imply? **Answer:** Twenty operations within that boundary, not necessarily twenty active CPU workers.
4. **Question:** Can reopening a connection prove a cold-device benchmark? **Answer:** No. Other cache layers can remain warm.
5. **Question:** A variant is faster because it stops fetching after the first row. Is it comparable to a full-result variant? **Answer:** Only if the required task is explicitly first-row retrieval. Otherwise the completion boundaries differ.
6. **Question:** Should failed attempts disappear from a successful-latency chart? **Answer:** They may be excluded from that specifically labelled distribution, but their counts and outcomes must remain visible alongside it.
7. **Question:** Why should a functional test avoid asserting that an index is always faster? **Answer:** Timing depends on host and workload conditions. The test can verify correct results and valid measurement records without inventing a universal performance bound.
8. **Question:** What does the local in-memory benchmark not establish? **Answer:** Networked service capacity, multi-host behaviour, persistent-device durability, production isolation or a supported user count.

20.98 Common wrong ideas

1. **Wrong:** one average describes every user's experience. **Right:** distributions and failures reveal important differences.
2. **Wrong:** percentiles have one universal calculation convention. **Right:** label the method and sample population.
3. **Wrong:** more clients always increase useful throughput. **Right:** saturation can increase waiting and errors instead.
4. **Wrong:** a new connection means everything is cold. **Right:** reset scope must be demonstrated.
5. **Wrong:** a faster result is valid even when guarantees changed. **Right:** compare equal services or disclose the difference.
6. **Wrong:** the benchmark harness has no cost. **Right:** setup, fetching and instrumentation boundaries matter.
7. **Wrong:** enough local repetitions prove production scalability. **Right:** the workload and environment determine transferability.
8. **Wrong:** limitations weaken an otherwise strong result. **Right:** they prevent the result being used to support a claim it never tested.

20.99 Chapter summary in 20 lines

1. Define the required correct operation before timing it.
2. Record data distribution, parameters, concurrency and operation mix.
3. A timer boundary determines what the duration means.
4. Closed-loop and open-loop arrivals expose different overload behaviour.
5. Use a monotonic timer for elapsed intervals.
6. Nanosecond units do not imply nanosecond accuracy.
7. Latency is a distribution rather than one constant.
8. Percentile convention and sample size affect interpretation.

9. Keep timeouts and errors visible.
10. Throughput needs a defined successful unit of work.
11. Queueing can grow while completion rate stops increasing.
12. Little's law requires consistent boundaries and suitable steady-state assumptions.
13. Warm state exists at several layers.
14. Reset only what the procedure can legitimately establish.
15. Compare identical result and guarantee contracts.
16. Account for harness overhead and complete result retrieval.
17. Retain raw samples and predefined exclusion rules.
18. Do not build a universal speed assertion into a functional test.
19. Report actual versions, inputs, observations and limitations together.
20. A bounded experiment answers a bounded question, not every deployment decision.

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

Access method

the machinery behind a route — an engine-defined algorithm and operator interface used to organise and search index entries.

Read in context: 15.1.6

Adversarial input

data chosen to exploit assumptions — inputs deliberately selected to cause incorrectness, excessive resource use or other unwanted behaviour.

Read in context: 17.5.6

Aggregate state

information accumulated for a group — working values sufficient to compute an aggregate such as a sum and count.

Read in context: 19.1.6

As-of time

when the available information was evaluated — the observation or snapshot boundary attached to a report.

Read in context: 19.4.6

B

Backpressure

make upstream work respect downstream limits — a mechanism that slows or bounds production when consumers cannot keep up.

downstream limits affect upstream admission — a mechanism preventing work from accumulating without bound.

Read in context: 20.3.6

Baseline

the saved comparison point — a defined workload, configuration and observed result before an intervention.

Read in context: 18.6.6

Benchmark contract

the rules of the measurement — a specification of inputs, operations, environment, timing boundaries and success criteria.

Read in context: 20.1.6

Birthday effect

collisions become plausible before the space is full — pair growth causes collision probability to rise around the square root of a uniform output-space size.

Read in context: 17.5.6

Blocking operation

input must accumulate before output can proceed — a stage such as a full sort under its particular execution strategy.

Read in context: 18.4.6

Buffer hit

a needed page was already in the relevant cache — an engine-level observation, not a statement that all possible lower-level work vanished.

Read in context: 18.5.6

C

Cardinality estimate

a predicted row count — the planner's expected output size for an operation under its available statistics and model.

Read in context: 18.2.6

Censored observation

a boundary rather than an exact value; technically, partial information restricting a value to a range, such as an unobserved completion time known to exceed a deadline.

completion time is only partly known — an observation limited by a timeout or stopping rule rather than a measured final duration.

Read in context: 20.2.6

Closed-loop load

clients wait before issuing their next request — a workload in which completion controls later arrivals.

Read in context: 20.1.6

Collision

different inputs share one hash result — equality of hash outputs without equality of original inputs.

Read in context: 17.2.6

Collision resistance

hard to deliberately find two matching fingerprints — computational difficulty of finding distinct inputs with equal digests under the algorithm's security assumptions.

Read in context: 17.4.6

Composite index

one index built from several fields — an access structure over tuples of expressions in a specified order.

Read in context: 15.3.6

Content addressing

name bytes by a content-derived identifier — object identification based on a specified representation and digest scheme.

Read in context: 17.6.6

Contribution identity

which source fact a result uses — a stable scoped identifier for a counted or summed input contribution.

Read in context: 19.6.6

Correctness oracle

a separate expected-answer check — a mechanism that rejects a fast but semantically wrong variant.

Read in context: 20.5.6

Correlation

values vary together — a relationship that can invalidate an assumption that conditions are independent.

Read in context: 18.2.6

Counterbalancing

vary which option runs first — an experimental ordering technique intended to reduce systematic sequence effects.

Read in context: 20.4.6

Covering index

the guide carries the values this question needs — an index containing sufficient query attributes for an eligible index-only or covering access path.

Read in context: 15.4.6

D

Deduplication key

what identifies one contribution — the scoped fields used to recognise a replay rather than a separate event.

Read in context: 19.5.6

Denominator

what the division is over; technically, the quantity that defines the base of a ratio or average.

what the total is divided by — the explicitly defined count or weight underlying a ratio.

the population a rate is out of — the eligible count or exposure giving a numerator its meaning.

Read in context: 19.2.6

Digest

a compact fingerprint of bytes — a fixed-length output of a hash algorithm, whose meaning depends on the exact input bytes and algorithm.

Read in context: 17.1.6

Distinct count

count different values under a rule — cardinality of the included value set after the specified equality and NULL treatment.

Read in context: 19.5.6

E

Elapsed time

how long the chosen boundary took on a clock — wall-clock duration including waits within that boundary.

Read in context: 14.6.6

Eligible access path

a route that can implement the required semantics — an index or scan option permitted by the query, operators and engine rules.

Read in context: 15.6.6

Equality contract

the rule for deciding that two keys are the same — a domain-specific equivalence relation that hashing and comparison must respect.

Read in context: 17.3.6

Estimand

the exact quantity a study seeks — the defined performance or outcome measure to which observations are intended to speak.

the precise quantity a study seeks — a defined population-level contrast under specified interventions or conditions.

Read in context: 20.5.6

Exclusion register

explain what did not contribute — recorded counts and reasons for records outside the report's accepted population.

Read in context: 19.6.6

Execution plan

the chosen route to the answer — a structured set of access, join, calculation and ordering operations used by the executor.

Read in context: 18.1.6

Expected complexity

cost under specified distribution assumptions — an average or probabilistic bound, not a guarantee for every input.

Read in context: 17.3.6

External validity

whether a result transfers beyond its test — the justified scope of applying observations to other workloads, environments or populations.

Read in context: 20.6.6

F

False positive

a match accepted when the entities differ; technically, a positive decision for a negatively labelled case in the stated evaluation.

a possible match that is not actually present — a positive candidate result rejected by a later exact check.

Read in context: 17.6.6

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: 16.2.6

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: 17.1.6

Frequent value

one value appears in many records — a high-frequency member of a distribution relevant to selectivity estimation.

Read in context: 15.5.6

G

Grouping key

the fields defining each collection — expressions whose equal grouping values identify one aggregate group.

Read in context: 19.1.6

H

Half-open interval

include the start but not the end — an interval [a,b) useful for adjoining non-overlapping ranges.

Read in context: 16.5.6

Harness overhead

work caused by the measurement setup — time and resources consumed by generation, instrumentation and reporting rather than the target operation alone.

Read in context: 20.5.6

Hash function

a rule making an organising or checking value — a deterministic mapping under specified parameters and input representation.

Read in context: 17.1.6

Hash join

match through a temporary hashed key structure — a build-and-probe strategy that still checks equality and preserves required duplicates.

Read in context: 18.3.6

Hash partitioning

choose a data region from a key's hash — routing through a versioned mapping from derived values to partitions.

Read in context: 17.6.6

Histogram

a summary of how values are distributed — a bucketed representation used for approximate frequency or range reasoning.

Read in context: 18.2.6

Hypothesis-driven tuning

change what evidence suggests — performance investigation linking an explicit suspected cause to a controlled test.

Read in context: 18.6.6

I

Included column

payload carried alongside search keys — a non-key attribute stored in an index by an engine supporting that feature.

Read in context: 15.4.6

Index

an extra route to selected records — a maintained data structure supporting specified search, ordering or constraint operations.

Read in context: 15.1.6

Index maintenance

keeping the extra route correct — updates, cleanup and related work required as the underlying data changes.

Read in context: 15.2.6

Instrumentation

measurements added to execution — observation code collecting timings, counts or resource evidence, potentially with its own overhead.

Read in context: 18.5.6

K

Key conservation

no entry disappears or is invented — equality between the intended inserted entries and those represented after restructuring.

Read in context: 16.4.6

Known subtotal

a total over available values only — a partial calculation that must not be labelled complete when required observations are missing.

Read in context: 19.2.6

L

Latch

short-lived protection of an internal structure — a synchronisation mechanism distinct from a transaction's logical locking policy.

Read in context: 16.6.6

Late arrival

a record received after its event period — an observation whose processing time occurs later than the event-time window it describes.

Read in context: 19.4.6

Leading key

the first part of an ordered index key — the prefix that groups entries before later components are compared.

Read in context: 15.3.6

Leaf

the bottom searchable page — a node containing data entries rather than further child pages in this teaching B+ tree.

Read in context: 16.1.6

Lexicographic order

compare the first difference — tuple ordering determined by the earliest component on which two tuples differ.

Read in context: 15.3.6

Load factor

how full the hash structure is — entries divided by the relevant bucket or slot count under a stated convention.

Read in context: 17.2.6

Locality

useful records are close in the accessed representation — spatial or temporal concentration that can reduce repeated movement or improve cache reuse.

related work stays close together — placement or access behaviour reducing the number of independent resources an operation must involve.

Read in context: 15.5.6

Loop count

how often a node is invoked — an execution multiplicity that must be considered alongside per-loop measurements.

Read in context: 18.5.6

Lossy index entry

a representation that does not retain all distinguishing information — an entry requiring rechecking against fuller data to establish a match.

Read in context: 17.3.6

M

Materialisation

save an intermediate result for reuse — creation of a stored working representation rather than recomputing or streaming every use.

Read in context: 18.1.6

Measurement boundary

what the timer includes — the explicitly chosen start and completion conditions of an observed operation.

Read in context: 20.6.6

Membership check

confirm that the target actually exists — equality verification after locating a candidate or insertion position.

Read in context: 16.3.6

Merge fan-in

how many ordered streams are combined at once — the number of input runs a merge step can consume under its buffer and resource limits.

Read in context: 14.4.6

Merge join

match while advancing through ordered inputs — a strategy using compatible sorted join keys and correct handling of equal-key groups.

Read in context: 18.3.6

Mergeable summary

subgroup evidence can be combined correctly — a summary with a defined merge operation preserving the intended metric or error model.

Read in context: 19.5.6

Message authentication code

a keyed check on a message — an integrity and authenticity mechanism for parties sharing a key, not automatically a public signature.

Read in context: 17.4.6

N

Nested-loop join

repeat matching work for each outer record — a join strategy whose inner access can range from a full scan to a selective lookup.

Read in context: 18.3.6

O

Occupancy

how full a page is — the used share of its usable entry capacity under the actual layout.

Read in context: 16.2.6

Open-loop load

arrivals follow an independent schedule — a workload whose offered demand does not automatically slow with each response.

Read in context: 20.1.6

Oracle

an independent expected-result mechanism — a simpler model used to compare the behaviour of a tested implementation.

Read in context: 16.6.6

P

Page split

divide an overfull region — redistribution into two nodes with updated parent boundaries and relevant sibling links.

Read in context: 16.4.6

Partial index

a guide for a defined subset — an index whose entries include only rows satisfying its predicate.

Read in context: 15.4.6

Path property

useful information about an intermediate route — characteristics such as ordering, parameter dependence and estimated size used in planning.

Read in context: 18.4.6

Payload

the useful represented content — bytes under a stated boundary, excluding or including overhead only as explicitly defined.

Read in context: 14.2.6

Peer group

rows tied under a window's ordering — entries equal on all ordering expressions used to define peers.

Read in context: 19.3.6

Percentile

a position in an ordered population or sample — a quantile defined under a stated convention, such as nearest rank.

Read in context: 20.2.6

Plan regression

a changed execution choice worsens a defined workload — an observed performance deterioration, not merely a different plan diagram.

Read in context: 15.6.6

Plan selection

choosing among legal routes — the optimiser's decision based on estimated costs and available information.

Read in context: 15.6.6

Planner cost

an estimate for comparing execution choices — a model-specific quantity derived from statistics and configured operation costs.

Read in context: 14.6.6

Predicate

a condition used to choose records — an expression whose truth controls qualification in a query operation.

a condition used to select records — an expression whose truth determines query eligibility under the relevant semantics.

Read in context: 14.1.6

Preimage resistance

hard to work backwards from a digest — computational difficulty of finding an input producing a specified hash output.

Read in context: 17.4.6

Probe sequence

alternative places checked after a collision — a defined traversal of slots in an open-addressed hash table.

Read in context: 17.2.6

Projection

an answer built from other records — a derived representation produced by applying transformation rules to source data.

choose the fields needed — a relational or SQL operation producing selected expressions rather than every available attribute.

Read in context: 14.2.6

Q

Query rewrite

a semantics-preserving change in representation — transformation of a query before or during planning under the engine's rules.

Read in context: 18.1.6

Queueing delay

time waiting before work can proceed — delay caused by contention or scheduling rather than the operation's own active computation.

Read in context: 14.6.6

R

Range scan

walk entries between boundaries — ordered traversal beginning near a lower bound and stopping when an upper condition fails.

Read in context: 16.5.6

Raw sample

an individual retained observation — an unaggregated measurement from which summaries can be recomputed.

Read in context: 20.6.6

Record locator

information leading from an entry to its record — an engine-specific reference or identifying value used to retrieve associated data.

Read in context: 16.3.6

Regression envelope

cases checked for unintended deterioration — representative workloads beyond the one case an intervention aims to improve.

Read in context: 18.6.6

Report grain

what one result row describes — the precise unit represented after selection, joining and aggregation.

Read in context: 19.1.6

Reset scope

what was actually returned to a starting condition — the exact layers or resources reinitialised before a measurement.

Read in context: 20.4.6

Residual filter

a check made after finding candidates — a predicate evaluated after an access path has already visited candidate entries or rows.

Read in context: 14.5.6

Result cardinality

how many output rows there are — the row count at a specified stage, distinct from records examined or bytes read.

Read in context: 14.1.6

Root

the starting guide — the top node from which every lookup begins.

Read in context: 16.1.6

Row width

the amount carried by one row representation — estimated or measured bytes per row at a specified physical or logical stage.

Read in context: 14.2.6

S

Saturation

a limiting resource is at or near effective capacity — a condition in which additional demand tends to create waiting, rejection or deterioration rather than proportional useful work.

demand meeting a limiting resource — a condition producing increased waiting, rejection or reduced useful throughput.

Read in context: 20.3.6

Scan

visit data through an access path — sequential or otherwise structured traversal of records or index entries to produce candidates.

Read in context: 14.1.6

Search invariant

the condition kept true at every step — the target, if present, remains inside the selected subtree's permitted range.

Read in context: 16.3.6

Secondary index

an additional searchable organisation — an access path distinct from the primary physical or identifying organisation of the table.

Read in context: 15.1.6

Selectivity

the fraction that passes a condition — qualifying rows divided by a stated candidate population, either estimated or observed.

Read in context: 14.5.6

Separator key

a boundary directing the next step — an internal key dividing ordered child ranges under a defined comparison convention.

Read in context: 16.1.6

Skew

some values occur much more often than others — a non-uniform distribution that can concentrate work and invalidate uniform assumptions.

an uneven distribution — concentration in key frequency, record size, cost or request rate.

Read in context: 14.5.6

Sort key

the values deciding order — the ordered sequence of expressions and comparison rules used to arrange records.

Read in context: 14.3.6

Sorted run

one already ordered batch — an intermediate sequence used by an external merge process.

a sequence already ordered by its search key — a file or logical collection that can be searched and merged using key order.

Read in context: 14.4.6

Spill

working data moved out of the operation's memory allowance — temporary external storage used to continue an operation such as sorting or hashing.

Read in context: 14.4.6

Split propagation

a full parent also needs room — upward restructuring triggered when adding a child exceeds an internal node's capacity.

Read in context: 16.4.6

Startup cost

estimated work before the first result — a planner quantity relevant to operations that must prepare or consume input before emitting rows.

Read in context: 18.4.6

Structural validation

check that the representation obeys its rules — inspection of ordering, ranges, links, occupancy and depth invariants.

Read in context: 16.6.6

Successor link

the route to the next ordered region — a reference or traversal mechanism connecting neighbouring entries or leaf nodes.

Read in context: 16.5.6

T

Tail latency

the unusually slow end of the distribution — high-percentile delays whose meaning depends on the measured population and sample size.

Read in context: 20.2.6

Throughput

completed work per unit time — a rate whose counted operation and success condition must be defined.

Read in context: 20.3.6

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: 14.3.6

Time bucket

the period a record contributes to — a reporting interval defined by clock semantics, timezone and boundary rules.

Read in context: 19.4.6

Top-N

keep the requested best few — an optimisation that retains a bounded candidate set under an ordering, without necessarily avoiding a full input scan.

Read in context: 14.3.6

Transformation version

identify the calculation rules used — a versioned definition of parsing, filtering, joining and aggregation semantics.

which processing rule was used — an identifier binding output to a particular implementation and its parameters.

Read in context: 19.6.6

Tree height

how many levels a lookup crosses — the root-to-leaf depth under a stated counting convention.

Read in context: 16.2.6

U

Uniform model

assume every output is equally likely — a mathematical distribution assumption whose applicability must be justified separately.

Read in context: 17.5.6

W

Warm state

earlier work remains useful — cached data or prepared resources available to later operations under a specified scope.

Read in context: 20.4.6

Weighted mean

larger contributions receive proportionate influence — the sum of value-times-weight divided by total valid weight.

Read in context: 19.2.6

Window frame

the contributing region for one current row — a bounded subset of its partition under ROWS, RANGE or another supported frame mode.

Read in context: 19.3.6

Window function

calculate across related rows without collapsing each one — an SQL function evaluated over a partition and, where applicable, an ordered frame.

Read in context: 19.3.6

Workload envelope

the conditions a design was evaluated for — declared ranges of size, skew, query mix, concurrency and service requirements.

Read in context: 15.5.6

Workload mix

the combination of operations a system serves — their frequencies, parameters, sizes, concurrency and correctness requirements.

Read in context: 15.2.6

Write amplification

extra work caused by one logical change — a ratio of physical or internal writes to a clearly defined logical write baseline.

storage writes exceed logical input — a measured ratio of bytes written at a named layer to logical bytes accepted over a stated interval.

Read in context: 15.2.6

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