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 E · Chapters 27–32 · First Edition

Storage and Recovery

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 E

Storage and Recovery

Trace a write into storage and learn how to recover after failures without inventing certainty.

This volume contains Chapters 27–32 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 E

Storage and Recovery

Trace a write into storage and learn how to recover after failures without inventing certainty.

27

CHAPTER

Pages, Buffers and the Storage Engine

27.0 What this chapter gives you

1. SQL describes logical records, but a storage engine moves and manages physical units. This chapter connects rows to pages, memory buffers, free space and the work triggered by a small update.
2. You will estimate page occupancy under stated assumptions, distinguish a database cache hit from a device read, and inspect storage metadata without pretending a toy page is a real PostgreSQL or SQLite file.
3. The arithmetic layouts below are teaching designs. Product-specific sizes and behaviours are labelled separately, and the labs operate only on synthetic data.

27.1 Logical rows and physical pages

27.1.1 PLAIN — in simple words

1. A row is a logical record. A page is a physical storage unit used by many database engines to organize records and indexes.
2. Several small rows can share a page. A large value may require an overflow or separate-storage mechanism. The relationship is not simply one row equals one file or one disk sector.
3. Reading one requested field can involve bringing a larger unit into memory. That is one reason data layout and access patterns affect performance even when the SQL result is tiny.

27.1.2 PLAIN — a picture in your head

1. Think of records as individual forms and pages as folders that carry several forms together. Fetching one form may require retrieving its folder.
2. A folder also has labels and indexing information, leaving less than its full capacity for form contents.
3. **Where the comparison breaks:** database pages have exact binary layouts, and the device can have different physical transfer units. A folder analogy does not define page atomicity or hardware write guarantees.

27.1.3 PLAIN — a worked example

1. Design a toy page of 4,096 bytes with a 64-byte header. Each stored row uses 96 payload bytes plus a 4-byte slot entry.
2. Available space is 4,096 − 64 = 4,032 bytes. At 100 bytes per row-plus-slot, the page holds at most 40 such rows, leaving 32 bytes unused under this simplified layout.
3. One thousand rows therefore need at least 25 data pages if all pages achieve that occupancy. Real engines may use more because of metadata, free space, variable-length values and update history.

4. These are not SQLite's or PostgreSQL's actual tuple-overhead figures. They are a transparent model whose assumptions can be changed and recalculated.

27.1.4 PLAIN — what is really happening inside

1. A page can contain a header, an array of item references, free space and record bytes. The engine interprets those bytes using its file-format and schema information.
2. An index page may organize keys and child pointers rather than business rows. A query can visit both index pages and table pages to produce one result.
3. Physical storage can change while logical identity stays the same. Compaction, updates and maintenance should not force the application to identify customers by their current byte offsets.

27.1.5 TECHNICAL — the engineer's version

1. PostgreSQL's standard heap layout commonly uses 8 kB pages, with a compile-time page-size choice. The documented page header is 24 bytes and item identifiers use 4 bytes each. These are implementation facts, not universal database constants. [S128]
2. SQLite's database page size is a power of two between 512 and 65,536 bytes under its file format. Header encoding treats the value 1 specially to represent 65,536. [S133]
3. Separate logical key, page identifier, slot reference and device block address in a design. Each names a different layer and may have different stability guarantees.

27.1.6 WORDS — remember these

1. **Database page:** a unit in the engine's physical layout — a fixed-size storage block containing records, index entries or other engine structures. **Slot entry:** a small reference locating an item within a page — metadata mapping an item identifier to its stored position and length. **Page occupancy:** how much of a page is usefully filled — the proportion consumed by payload and required structures under a stated layout.

27.2 Buffer pools

27.2.1 PLAIN — in simple words

1. A buffer pool keeps database pages in memory so repeated work can avoid fetching them again from lower storage layers.
2. Memory is limited. The engine must decide which pages to retain, which to replace and how to coordinate pages currently being used or changed.
3. A database-cache miss does not necessarily mean a physical device read. The operating system may already have a copy in its own cache.

27.2.2 PLAIN — a picture in your head

1. Mira keeps frequently used folders on her desk and retrieves others from a cabinet. A desk hit is faster than a cabinet trip.
2. The cabinet assistant may also have a trolley of recently fetched folders. Missing from Mira's desk does not mean the folder must travel all the way from the distant archive.
3. **Where the comparison breaks:** the layers contain copies and coordination metadata, not one movable physical folder. Cache coherence, pinning and dirty-state handling are more exact than ordinary desk tidying.

27.2.3 PLAIN — a worked example

1. A toy buffer pool has three slots and uses least-recently-used replacement. Read page IDs 1, 2, 3, 1, 4, 2.
2. The first three reads miss and fill the pool. Reading 1 hits and makes it most recent. Reading 4 evicts 2; the later read of 2 misses again.
3. The model records one hit and five misses. It demonstrates a replacement policy, not the policy or device-read count of a named engine.
4. A different access sequence or replacement algorithm can change the result. A single global hit ratio cannot tell you which important request suffered the expensive miss.

27.2.4 PLAIN — what is really happening inside

1. The engine locates the requested page in its buffer mapping or chooses a buffer to load it into. It coordinates concurrent access and prevents a page from being evicted while an operation still requires it.
2. A dirty page cannot be discarded as though it were merely an unchanged cached copy. Its modifications need the engine's recovery and flushing protocol.
3. Large scans can compete with frequently reused pages for cache capacity. Engines may use specialized strategies to avoid letting one scan disrupt every other workload.

27.2.5 TECHNICAL — the engineer's version

1. PostgreSQL I/O statistics distinguish operations at the database/kernel boundary but do not by themselves distinguish actual device reads from data already in the kernel's page cache. Combine them with appropriate operating-system evidence. [S144]
2. Cache metrics require a scope and interval. A counter since process start can hide a recent regression; ratios aggregated across workloads can hide a slow critical query.
3. The companion LRU exercise is a deterministic model with no real I/O, locks or dirty-page write-back. Do not use its hit count as a measured PostgreSQL buffer-pool result.

27.2.6 WORDS — remember these

1. **Buffer pool:** the database's in-memory page workspace — a managed cache of pages and metadata used by the storage engine. **Cache hit:** the needed item is already in the named cache — an access resolved at a specified cache layer. **Eviction:** remove a cached item to make room — replacement subject to usage, dirty-state and recovery constraints.

27.3 Row layout

27.3.1 PLAIN — in simple words

1. A stored row contains more than the visible values returned by SELECT. The engine needs information about field boundaries, missing values, versions and other implementation details.
2. Variable-length text cannot always be located by multiplying a column number by a fixed width. Earlier fields, alignment and length information can affect where it begins.
3. Large values may be compressed or stored elsewhere with a reference in the main row. Reading only small fields can therefore avoid some large-value work, depending on the engine and query.

27.3.2 PLAIN — a picture in your head

1. An application form has boxes of different sizes and a note saying that the attached document is in another folder. The visible answers are not the whole filing structure.
2. A short reference can stand in for a large attachment until someone actually needs it.
3. **Where the comparison breaks:** binary length headers, alignment and compression are not hand-written annotations. Reading raw bytes safely requires the exact format and bounds, not guess-work based on how a screen displays the row.

27.3.3 PLAIN — a worked example

1. Define a toy record with an 8-byte header, a 4-byte ID, a 2-byte text length and five UTF-8 bytes of text. Without alignment or other fields, its size is 8 + 4 + 2 + 5 = 19 bytes.
2. Replace the text with ten UTF-8 bytes and the record becomes 24 bytes. The number of displayed characters need not equal the encoded byte count.
3. If a format rounds total size up to a multiple of eight bytes, both 19 and 24 occupy 24 bytes, while 25 occupies 32. That rule belongs to the toy format only.
4. The exercise links Chapter 3's representations to physical space: units and encoding remain important after values enter a database.

27.3.4 PLAIN — what is really happening inside

1. Row metadata tells the engine how to interpret fields under the relevant schema and storage representation. Missing values may use bitmap information rather than a full ordinary payload value.
2. Schema evolution adds another layer: old and new row representations may coexist while the engine supplies the logical view promised to SQL.
3. Do not infer storage savings solely from a column's declared maximum length. Actual representation, compression, nulls and engine rules determine the physical result.

27.3.5 TECHNICAL — the engineer's version

1. PostgreSQL heap rows have a fixed header, optional null bitmap and aligned user data under the documented layout. Physical interpretation depends on type lengths and alignment metadata. [S128]
2. PostgreSQL TOAST handles eligible large values through compression and/or out-of-line storage. Ordinary tuples do not simply span arbitrary heap pages as one contiguous record. [S143]
3. SQLite records use their own header and serial-type representation. Treating a PostgreSQL tuple decoder as an SQLite parser would be a format error, even when both systems return similar SQL values. [S133]

27.3.6 WORDS — remember these

1. **Row header:** metadata around the visible field values — implementation information needed to locate, interpret or manage a stored row version. **Alignment:** place values at required byte boundaries — padding and layout rules supporting an implementation's representation or access requirements. **Out-of-line value:** a large value stored separately from its main row — payload reached through a reference rather than fully embedded in the ordinary tuple.

27.4 Dirty pages and flushing

27.4.1 PLAIN — in simple words

1. A dirty page is a memory copy containing changes that have not yet been written to the corresponding main data-file location.
2. Dirty does not mean corrupt. It means that the in-memory page and its lower-level stored copy differ in a way the engine is managing.
3. A transaction can commit before every changed data page reaches its final data-file location if the recovery protocol has durably recorded enough information to reconstruct the accepted changes.

27.4.2 PLAIN — a picture in your head

1. Mira corrects a working folder on her desk and records the correction in a protected journal. The cabinet copy can be updated later because the journal tells the recovery clerk what changed.
2. The order is essential: losing the desk copy must not also lose the only description of an accepted correction.
3. **Where the comparison breaks:** a recovery log uses engine-specific records, ordering and commit rules. A casual handwritten note is not enough to recover every possible partial write.

27.4.3 PLAIN — a worked example

1. A toy database page on persistent storage contains stock 5. A transaction changes its buffered version to 2 and creates sufficient recovery information for that accepted update.
2. If the log and required commit evidence are durable while the data page still contains 5, recovery can redo the committed change under the toy protocol.
3. If the engine overwrites the data page before making the required recovery information durable, a crash can leave an incomplete state with no reliable reconstruction path.
4. Chapter 28 makes the toy log rules explicit. This chapter does not claim that merely writing the text “stock=2” implements a real WAL engine.

27.4.4 PLAIN — what is really happening inside

1. The engine schedules writeback and coordinates it with recovery records. A background writer or checkpoint can move dirty pages toward their data-file locations.
2. A page may be modified again while flushing is in progress. Correct engines track the relevant synchronization and dirtiness state rather than assuming one write makes every future version clean.
3. Flushing to the operating system and forcing data through lower caches are distinct boundaries. A successful function call must be interpreted under its actual contract.

27.4.5 TECHNICAL — the engineer’s version

1. Write-ahead logging requires relevant log records to be persisted before the corresponding data-file modifications are allowed to become durable under the engine’s protocol. This permits redo-based recovery without forcing every data page at each commit. [S129]
2. PostgreSQL checkpoints bound the WAL region needed for crash recovery while creating their own I/O work. WAL archiving and other retention requirements can keep older log material relevant beyond local checkpoint needs. [S135] [S131]
3. A database buffer flush is not a universal proof against device loss or dishonest hardware acknowledgements. Chapter 30 follows the persistence chain explicitly. [S130]

27.4.6 WORDS — remember these

1. **Dirty page:** a changed cached page awaiting corresponding writeback — a buffer whose managed contents differ from its lower stored representation. **Flush:** push pending changes toward a specified lower layer — an operation whose persistence guarantee depends on the API and storage contract. **Redo:** reconstruct an accepted change from recovery information — replay that restores data not fully reflected in the main data files before failure.

27.5 Free space and fragmentation

27.5.1 PLAIN — in simple words

1. Deleting a row does not necessarily make the database file smaller immediately. Its space may become reusable within the file while the file retains its allocated size.
2. Free space can be scattered across pages. A large new record may need a suitable contiguous region or an overflow mechanism even when total free bytes look sufficient.
3. Maintenance can reorganize storage, but it consumes resources and may require locks or extra temporary space. Rebuilding a file is not a harmless cosmetic operation.

27.5.2 PLAIN — a picture in your head

1. Removing forms from several folders creates spare room without removing any folder from the cabinet. A new thick form may still not fit in any one folder.
2. Repacking the folders can help, but someone must read, move and rewrite the contents while preserving their identities.
3. **Where the comparison breaks:** engines have exact slot and free-space rules, and some can reuse fragmented regions efficiently. "Fragmented" is a diagnosis requiring measurements, not a general explanation for every slow query.

27.5.3 PLAIN — a worked example

1. Four toy pages each have 100 free bytes. The total is 400 bytes, but a 250-byte item cannot fit on any one page if the format requires the item to fit entirely within one page.
2. A compaction or overflow strategy might change the result. The sum of free bytes alone cannot tell you which strategy exists.
3. Now delete half the rows from a 20-page toy file. The logical row count halves, but the file can remain 20 pages while its internal free space becomes available for later inserts.
4. Measure logical rows, allocated pages and reusable space separately rather than expecting all three to shrink together.

27.5.4 PLAIN — what is really happening inside

1. Engines keep metadata about pages with reusable space and may compact items within a page or recycle entirely free pages.
2. Old row versions and long snapshots can delay when space becomes reusable. A DELETE's commit is not always the end of its storage consequences.
3. Rebuilding or vacuuming operations differ by product. A command with the same name can have different locking, file-size and operational effects in two engines.

27.5.5 TECHNICAL — the engineer's version

1. PostgreSQL ordinary VACUUM and SQLite VACUUM are not interchangeable operations. PostgreSQL routine vacuuming commonly reclaims reusable space, while SQLite VACUUM rebuilds the database file and has its own space and transaction constraints. [S123] [S145]
2. Include indexes and large-value storage when measuring total footprint. A compact heap with oversized indexes is not a small database merely because one table-size metric looks low.
3. The lab's page arithmetic and SQLite metadata observations do not justify production maintenance commands. Capture workload, free-space evidence, backup/restore readiness and acceptable lock duration before a real intervention.

27.5.6 WORDS — remember these

1. **Reusable space:** allocated storage available for later records — free capacity inside an existing database structure rather than necessarily returned to the filesystem. **Fragmentation:** useful space is distributed inconveniently — a layout condition whose performance effect depends on the engine and access pattern. **Rebuild:** write a reorganized physical representation — maintenance that can improve layout while requiring I/O, temporary capacity and coordination.

27.6 Reading engine-specific evidence

27.6.1 PLAIN — in simple words

1. A useful storage investigation begins with a question: are we reading too much, missing a cache, retaining old versions, waiting for writes or running out of space?
2. Choose observations that distinguish those possibilities. A single file size, hit ratio or elapsed time rarely identifies the cause by itself.
3. Observe through supported interfaces first. Editing a live database file with a hex editor is not a reasonable first diagnostic step.

27.6.2 PLAIN — a picture in your head

1. A mechanic first checks the dashboard, service record and accessible measurements. They do not drill a hole into a running engine because a noise sounds unfamiliar.
2. Storage metadata is the instrument panel, not a complete explanation of every failure.
3. **Where the comparison breaks:** database files can contain private records and credentials. Diagnostic copies, logs and screenshots need access controls as well as technical care.

27.6.3 PLAIN — a worked example

1. In a fresh SQLite teaching database, record `PRAGMA page_size`, `PRAGMA page_count` and `PRAGMA freelist_count` before and after a bounded synthetic workload.
2. Multiply page size by page count to describe the logical main-file page allocation under that observation. Do not add a claim that this equals every physical byte consumed by WAL, journals, temporary files and backups.
3. Compare row counts and expected totals too. A smaller file containing missing records is not an optimization success.
4. The companion records actual runtime and query outputs; no page count is invented in the prose because it can vary with the engine and exact fixture.

27.6.4 PLAIN — what is really happening inside

1. Statistics can be cumulative, delayed, reset or sampled. Their observation interval and reset point are part of their meaning.
2. Product documentation can also change. The official SQLite WAL page consulted for this edition discloses a 2026 WAL-reset bug affecting older releases under specific concurrent WAL write/checkpoint conditions.
3. The historical lab runtime is SQLite 3.46.1. Its successful bounded tests are not a recommendation to deploy that release or a test of the disclosed concurrent WAL bug. The new exercises do not attempt that workload. [S132]

27.6.5 TECHNICAL — the engineer's version

1. Record engine version, journal/recovery mode, relevant settings, connection count, exact statements and measurement interval. Use supported metadata and a disposable copy for low-level experiments. [S75] [S144]
2. As consulted on 25 September 2026, SQLite documents the WAL-reset fix in 3.51.3 and later, with named backports. Verify the current supported release and applicable advisories before real deployment; do not infer safety from this book's historical runtime. [S132]
3. Corruption investigations require evidence preservation. Keep the original data and associated recovery files intact under a reviewed acquisition procedure; Chapter 32 explains why an attempted repair can destroy the information needed to diagnose the problem. [S139]

27.6.6 WORDS — remember these

1. **Storage observation:** a measured fact about one physical configuration — metadata or instrumentation interpreted with version, scope and timing. **Freelist:** storage recorded as available for reuse — an engine-managed collection of free pages or blocks. **Runtime provenance:** which software actually produced the result — the recorded engine and interpreter versions and relevant configuration of an experiment.

27.97 Practice and worked answers

1. **Count toy rows per page.** A 4,096-byte page has 64 header bytes and 100 bytes per row-plus-slot. **Answer:** floor(4,032/100)=40 rows, with 32 bytes left.
2. **Trace the three-slot LRU cache.** Read 1,2,3,1,4,2. **Answer:** one hit and five misses under the stated toy policy, not five proven device reads.
3. **Apply alignment.** Round record lengths 19,24,25 up to multiples of eight. **Answer:** 24,24,32 bytes under that chosen layout.
4. **Interpret dirty.** A buffered page differs from its main-file copy. **Answer:** this is managed pending writeback, not necessarily corruption.
5. **Find the space trap.** Four pages each have 100 free bytes; a 250-byte item must fit on one page. **Answer:** total free space is insufficiently localized; no page can hold the item without another mechanism.
6. **Interpret a DELETE.** Row count halves but file size does not. **Answer:** storage may be reusable internally, retained for old versions or still allocated for engine structures.
7. **Name the cache boundary.** PostgreSQL reports an I/O read. **Answer:** that does not by itself distinguish a device access from an operating-system cache hit.

8. **Limit the runtime claim.** The old SQLite runtime passes the educational tests. **Answer:** this does not establish current production suitability or exercise the disclosed concurrent WAL-reset bug.

27.98 Common wrong ideas

1. **Wrong:** one row occupies one page. **Right:** pages can contain many rows, while large values can use separate storage.
2. **Wrong:** page size is a universal database constant. **Right:** formats and builds differ.
3. **Wrong:** a database cache miss proves a disk read. **Right:** lower caches may satisfy it.
4. **Wrong:** dirty means damaged. **Right:** it normally means changed and awaiting managed write-back.
5. **Wrong:** COMMIT must flush every changed data page. **Right:** a correct recovery protocol can persist log evidence and redo later.
6. **Wrong:** deleted rows immediately shrink the file. **Right:** allocated and reusable space are different measurements.
7. **Wrong:** all VACUUM commands have the same semantics. **Right:** consult the specific engine and operation.
8. **Wrong:** passing old-runtime tests certifies current deployment safety. **Right:** version advisories and real workload validation remain necessary.

27.99 Chapter summary in 20 lines

1. Logical records and physical storage units belong to different layers.
2. Pages organize rows, index entries and engine metadata.
3. Headers and slot entries reduce space available for payload.
4. Toy occupancy calculations must state their layout assumptions.
5. Large values can use compression or out-of-line storage.
6. Physical row locations should not replace stable business keys.
7. Buffer pools retain pages in memory for reuse.
8. Cache replacement depends on workload and engine policy.
9. A database-cache miss is not necessarily a physical-device read.
10. Row representations include metadata, lengths, null information and alignment.
11. Dirty pages contain managed changes awaiting corresponding writeback.
12. Recovery ordering connects log persistence to later data-page flushing.
13. A committed transaction need not have every data page at its final location immediately.
14. Checkpoints and background writeback have measurable resource costs.
15. Free space inside a file differs from space returned to the filesystem.
16. Scattered free bytes do not guarantee room for one large item.
17. Maintenance commands differ in semantics across products.
18. Supported metadata is the starting point for storage diagnosis.
19. Record runtime, settings and observation intervals with every experiment.
20. Preserve recovery evidence and verify current advisories before real deployment.

28

CHAPTER

Write-Ahead Logs and Recovery

28.0 What this chapter gives you

1. A crash can erase memory while leaving only part of the recent work in storage. Recovery needs a protocol for deciding what those surviving bytes mean.
2. This chapter explains write-ahead ordering, log positions, checkpoints and redo. A small logical recovery model makes the steps visible without pretending to implement PostgreSQL's or SQLite's physical log format.
3. The model has explicit simplifying assumptions: a known starting state, ordered valid records, serial teaching transactions and an explicitly selected surviving log prefix. Its tests are not process-crash or power-loss experiments.

28.1 Why recovery information exists

28.1.1 PLAIN — in simple words

1. An application can change several pages before the storage system finishes writing all of them. A crash in the middle can leave a mixture of old and new data.
2. Recovery information records enough about changes and their acceptance boundaries to reconstruct a valid state under the engine's rules.
3. A log is useful only when its own validity, ordering and persistence are protected. An incomplete or corrupted note cannot be treated as trustworthy merely because it is called a log.

28.1.2 PLAIN — a picture in your head

1. Before updating several filing cabinets, a clerk records the required changes in a protected journal. After an interruption, another clerk uses the journal to finish or resolve the interrupted work.
2. The journal must distinguish an accepted instruction from a draft that was never approved.
3. **Where the comparison breaks:** real logs often describe physical or physiological engine changes, not ordinary business sentences. Their records, checksums and transaction metadata must be interpreted by the matching engine.

28.1.3 PLAIN — a worked example

1. A toy operation changes stock from 5 to 2 and records a reservation. If the stock page reaches storage but the reservation page does not, a crash leaves an incomplete local operation.
2. A recovery protocol can avoid forcing both pages at the exact same instant by preserving sufficient log evidence and a defined commit boundary.
3. After restart, the engine follows its recovery rules rather than guessing from whichever page happens to look newest.

4. A business audit log containing "reservation accepted" may lack the physical information needed to rebuild pages. Conversely, a WAL record may lack the human explanation needed for an audit. The two logs serve different purposes.

28.1.4 PLAIN — what is really happening inside

1. The engine coordinates data-page modifications, log generation and transaction completion. Log persistence and page writeback obey a required order.
2. Recovery can encounter work from committed, aborted or incomplete transactions. How it handles each depends on the engine's storage and visibility design.
3. A correct recovery process must also recognize incomplete or invalid log tails. It cannot blindly execute every byte remaining in a file as a valid operation.

28.1.5 TECHNICAL — the engineer's version

1. PostgreSQL's WAL supports redo-based recovery of data-file changes. SQLite WAL uses page frames and commit information under its own format; SQLite rollback journaling is a different mechanism. [S129] [S132] [S133]
2. The logical model in this chapter replays only complete committed teaching transactions. That is not a description of PostgreSQL's exact redo algorithm: physical recovery can replay changes whose transaction visibility is subsequently governed by transaction-status and MVCC rules.
3. Keep recovery logs, application event logs and audit records distinct in a system inventory. Similar append-only shapes do not imply identical contents, retention or authority.

28.1.6 WORDS — remember these

1. **Recovery log:** information needed to reconstruct a valid stored state — engine records supporting recovery after incomplete persistence or interruption. **Log tail:** the final portion of the recorded sequence — a region that can be incomplete or require validity checks after failure. **Recovery protocol:** the rules for interpreting surviving evidence — a specified procedure connecting log validity, transaction status and stored data.

28.2 Log before data

28.2.1 PLAIN — in simple words

1. Write-ahead means that the required recovery information reaches the promised persistent boundary before the corresponding data-file change can make recovery depend on it.
2. This ordering is more important than the fact that a log file exists. A log written too late cannot explain a data change after the only in-memory explanation disappears.
3. Commit acknowledgement adds another condition: the engine must meet the configured durability promise before telling the caller that the transaction is accepted under that promise.

28.2.2 PLAIN — a picture in your head

1. A clerk must secure the correction instruction before altering the official cabinet copy. Otherwise an interruption can leave a changed cabinet and no reliable account of the intended operation.
2. Securing the instruction can allow the cabinet changes to be completed later in a more efficient batch.
3. **Where the comparison breaks:** "secure" means a precise storage contract, not merely placing a note in another memory buffer. Operating-system and device caches may still be volatile.

28.2.3 PLAIN — a worked example

1. Consider four milestones: create the log record in memory, persist the required log record, flush the changed data page, and acknowledge commit under the selected contract.
2. A valid WAL design can persist the log and required commit information, acknowledge a durable commit, and flush some data pages later. Recovery uses the surviving log if the later flush never happened.
3. An invalid ordering flushes a changed page while its required recovery record exists only in volatile memory. Losing that memory can make the page change impossible to interpret or complete safely.
4. This is an ordering argument, not a claim that every engine uses exactly four system calls or one log record per business action.

28.2.4 PLAIN — what is really happening inside

1. Log records are often appended in a sequence, making it possible to synchronize a contiguous prefix efficiently.
2. Several commits can share one lower-level synchronization when the engine ensures that the required records for each are included. This is group commit, not a relaxation of each acknowledged transaction's selected promise.
3. Durability settings can intentionally change when acknowledgement occurs. Such a change belongs in the application contract and benchmark report, not hidden in a performance tweak.

28.2.5 TECHNICAL — the engineer's version

1. PostgreSQL's WAL ordering allows data pages to be written after the necessary log records have been flushed. Its documentation explains that a WAL synchronization can cover multiple concurrent commits. [S129]
2. PostgreSQL asynchronous commit can acknowledge before the transaction's WAL reaches durable storage, allowing recent acknowledged transactions to be lost after a crash while preserving database consistency under the documented design. This differs from disabling `fsync`, which can threaten consistency. [S136]
3. An experiment comparing commit latency must record these settings. A faster result obtained by weakening the durability contract is not a like-for-like optimization.

28.2.6 WORDS — remember these

1. **Write-ahead ordering:** recovery evidence must precede dependent data persistence — the ordering constraint linking log durability and data-page writes. **Group commit:** one synchronization supports several commits — batching log persistence while meeting each transaction's selected acknowledgement condition. **Asynchronous commit:** acknowledge before full local log persistence — a configured contract that can trade recent-transaction durability for lower latency.

28.3 Log sequence positions

28.3.1 PLAIN — in simple words

1. A log position identifies a place in the ordered recovery stream. It lets the engine say how far it has generated, written, flushed or replayed information.

2. Those progress points are different. A record can be generated but not yet persisted, or persisted on a replica but not yet applied there.
3. A position is not automatically a timestamp, a transaction number or a count of rows. Interpret it within the log stream and timeline to which it belongs.

28.3.2 PLAIN — a picture in your head

1. A long instruction roll has marked positions. One clerk has written through position 100, another has secured through 90, and a recovery clerk has applied through 80.
2. Saying "we are at 100" hides which clerk and which boundary you mean.
3. **Where the comparison breaks:** real log sequence numbers may refer to byte positions rather than numbered instructions. Comparing positions across unrelated streams or timelines needs additional identity information.

28.3.3 PLAIN — a worked example

1. Our toy log numbers records 1 through 11 for easy hand tracing. Suppose records through 9 survive a simulated failure; records 10 and 11 exist only in the discarded model memory.
2. Recovery must use the surviving prefix ending at 9. It cannot infer the contents of later records from the fact that the application intended to append them.
3. A checkpoint at position 6 supplies a known base state. Replaying valid records 7 through 9 can advance that state without rereading the model's earlier completed prefix.
4. These small integers are record ordinals in the teaching model, not actual PostgreSQL LSN encodings.

28.3.4 PLAIN — what is really happening inside

1. Engine metadata can associate pages with log positions so recovery knows which changes a page already reflects.
2. The engine must make redo safe under its actual record semantics. Simply replaying an increment twice would be wrong unless the protocol recognizes that the corresponding change is already applied.
3. Position continuity matters. A missing required segment is not repaired by skipping to the next available filename and hoping the data agrees.

28.3.5 TECHNICAL — the engineer's version

1. PostgreSQL page headers include a WAL-related LSN field. Recovery and replication use log positions with engine-specific meanings; the toy ordinal notation deliberately avoids imitating their byte-level format. [S128] [S131]
2. Distinguish generated, written, flushed and replayed progress. A lag measurement must name both endpoints and whether the difference is expressed in bytes, time or another unit. [S144]
3. Retain stream identity and timeline context with recovery positions. A numerically similar offset in an unrelated history is not interchangeable evidence.

28.3.6 WORDS — remember these

1. **Log sequence number:** a position in a named recovery stream — an engine-defined offset used to identify and compare WAL progress. **Durable prefix:** the surviving accepted beginning of a log — the valid ordered region known to have reached the required persistence boundary. **Re-**

play position: how far recovery has applied the stream — progress measured under the engine's interpretation of log records.

28.4 Checkpoints

28.4.1 PLAIN — in simple words

1. A checkpoint records a recovery starting boundary so the engine need not reconstruct everything from the beginning of time after every crash.
2. It coordinates data-page persistence and recovery metadata. It is not simply the same thing as a user transaction committing.
3. Older logs may still be needed for backups, replicas or other consumers even after local crash recovery can start later. "Checkpoint complete" is not permission to delete arbitrary recovery files.

28.4.2 PLAIN — a picture in your head

1. A clerk periodically produces a verified cabinet state and marks where the journal continues from that state. A later recovery starts from the cabinet and reads the remaining instructions.
2. An archive keeper may still need older instructions to reconstruct an earlier date.
3. **Where the comparison breaks:** real checkpoints can coordinate work over an interval rather than freeze the entire system at one instant. Their correctness depends on the engine's detailed ordering rules.

28.4.3 PLAIN — a worked example

1. In the toy model, start with P=5. T1 commits P=2 and T2 aborts its proposed P=9. After position 6, no teaching transaction is active and the accepted state is P=2.
2. Save that state as a model checkpoint with position 6. Later T3 commits P=4 at position 9; T4 remains incomplete at position 11.
3. Recovery from the checkpoint and valid suffix returns P=4. The checkpoint does not make T4's incomplete proposed P=1 committed.
4. This model deliberately checkpoints only at a simple quiescent boundary. It is not an implementation of a real fuzzy checkpoint algorithm.

28.4.4 PLAIN — what is really happening inside

1. More frequent checkpoints can reduce later redo work but increase current writeback and related log overhead.
2. Less frequent checkpoints can improve some write workloads while increasing recovery work and retained log volume. The best interval depends on recovery objectives and actual measurements.
3. Long readers can also affect checkpoint progress in engines such as SQLite WAL. A checkpoint that cannot complete under its mode should not be reported as a fully reclaimed log merely because the call returned.

28.4.5 TECHNICAL — the engineer's version

1. PostgreSQL checkpoints establish redo boundaries and spread writeback work under configurable limits. Full-page-image behaviour interacts with checkpoint frequency, so tuning one parameter can change WAL volume as well as recovery time. [S135]

2. SQLite WAL checkpointing transfers eligible page content back to the main database and has modes with different blocking/completion behaviour. Active readers can prevent complete reset of the WAL. [S132]
3. Checkpoint retention and archive retention are different. Keep every segment required by the actual backup, recovery and replication contracts. [S131]

28.4.6 WORDS — remember these

1. **Checkpoint:** a recorded starting boundary for recovery — coordinated data and metadata progress that limits required redo under an engine's protocol. **Redo horizon:** the earliest recovery position still needed — the log boundary from which reconstruction must begin for a particular purpose. **Checkpoint pressure:** current work caused by reaching that boundary — I/O and related overhead generated by checkpoint activity.

28.5 Redo and rollback concepts

28.5.1 PLAIN — in simple words

1. Redo means reconstructing changes that the recovery protocol says must be represented. Rollback means preventing an unsuccessful transaction's changes from becoming its accepted result.
2. Engines can achieve those goals differently. Some recovery designs use explicit undo information; others combine physical redo with transaction visibility so incomplete transactions do not become visible committed work.
3. Do not assume that every database literally runs an opposite SQL statement for every failed statement. Recovery operates at the engine's own representation and transaction rules.

28.5.2 PLAIN — a picture in your head

1. A recovery clerk can finish approved corrections and exclude unapproved drafts. The exact paperwork determines how each is recognized.
2. One office keeps before-and-after copies; another preserves new pages with approval metadata. Both need a coherent protocol, but their recovery steps differ.
3. **Where the comparison breaks:** "undo" is not necessarily a business compensation. Refunding a delivered purchase is a new real-world action, not erasing an uncommitted database write.

28.5.3 PLAIN — a worked example

1. The toy model stages SET records by transaction identity. A COMMIT applies that transaction's staged after-values; an ABORT discards them. Incomplete staged work at the end is ignored.
2. For T1: BEGIN, SET P=2, COMMIT changes accepted state from 5 to 2. For T2: BEGIN, SET P=9, ABORT leaves accepted state at 2.
3. Replaying the same complete log from the same initial state produces the same result. That is determinism of the model, not permission to apply arbitrary increments repeatedly to an already advanced state.
4. The lab validates record ordering and transaction state. A COMMIT for an unknown transaction is malformed input, not a harmless record to skip silently.

28.5.4 PLAIN — what is really happening inside

1. Recovery must know both the base state's provenance and the log region being applied. Replaying a suffix against the wrong checkpoint can produce plausible but incorrect records.

2. Physical redo may rebuild pages containing versions that are not visible as committed rows. Visibility and physical reconstruction are related but separate engine responsibilities.
3. A recovery tool should fail clearly when required evidence is absent or incompatible, rather than inventing a clean state by dropping unexplained records.

28.5.5 TECHNICAL — the engineer's version

1. The toy recovery function uses logical after-values and serial transaction fixtures. It does not model page LSN checks, torn pages, interleaved conflicting writers, transaction-ID wraparound or an engine's log parser.
2. PostgreSQL crash recovery is redo-oriented; MVCC and transaction-status information govern visibility of tuple versions. Do not equate the toy's "apply only committed transactions" loop with the exact PostgreSQL implementation. [S129] [S119]
3. SQLite rollback journals preserve original page content, whereas WAL frames record revised page content. These mechanisms illustrate why recovery terminology must be attached to a concrete format. [S133]

28.5.6 WORDS — remember these

1. **After-value:** the state an accepted update should leave — a logical replacement used by this teaching recovery model. **Undo information:** evidence for reversing unaccepted changes — recovery data used by designs that require explicit restoration of earlier state. **Recovery base:** the known state from which replay begins — a checkpoint or backup identified together with its matching log boundary.

28.6 Crash-recovery traces

28.6.1 PLAIN — in simple words

1. A useful recovery trace lists what survived, what did not, and what the protocol can conclude. It does not fill missing evidence with the application's intention.
2. Testing different surviving prefixes exposes the importance of commit boundaries. An update record alone is not necessarily an accepted transaction.
3. A simulated prefix is a teaching experiment. A real crash test additionally needs control over processes, storage, flush settings and the failure being injected.

28.6.2 PLAIN — a picture in your head

1. Imagine the journal is cut at a chosen line. The recovery clerk may use only the lines still present and the known starting cabinet state.
2. A line that would have been written next is not evidence merely because the clerk remembers planning it.
3. **Where the comparison breaks:** real failures do not always leave a neat cut at a record boundary. Checksums, framing, partial writes and device behaviour determine which bytes are valid.

28.6.3 PLAIN — a worked example

1. Use the following complete synthetic log. SET means a staged after-value, and the initial state is P=5.

```
1 BEGIN T1       2 SET T1 P=2       3 COMMIT T1
4 BEGIN T2       5 SET T2 P=9       6 ABORT T2
```

```
7 BEGIN T3       8 SET T3 P=4       9 COMMIT T3
10 BEGIN T4     11 SET T4 P=1       [no commit]
```

2. A surviving prefix ending at 2 recovers P=5. Ending at 3 recovers P=2. Ending at 5 or 6 still recovers P=2.
3. Ending at 9 recovers P=4. Ending at 11 also recovers P=4 because T4 lacks a commit. Recovering from the position-6 checkpoint P=2 plus the remaining suffix also yields P=4.
4. Every expected result follows from the model's explicit rules. The example makes no claim that a real device preserved exactly those prefixes during a power cut.

■ 28.6.4 PLAIN — what is really happening inside

1. A real recovery investigation preserves the original files, relevant log segments, configuration and diagnostic output before attempting changes.
2. A successful restart proves that the engine reached a state it accepted under its checks. Business reconciliation and expected-record checks are still needed to establish the recovery target's meaning.
3. Failure to find a record can mean it was never committed, not included in the surviving history, restored to an earlier target or queried incorrectly. The trace must distinguish these possibilities rather than assigning a convenient cause.

■ 28.6.5 TECHNICAL — the engineer's version

1. The companion tests every listed prefix and malformed transaction ordering in memory. No process is killed, no power is cut, and no hardware durability claim follows from those tests.
2. Before a real failure experiment, state the fault model: process termination, operating-system crash, abrupt power loss, device loss or a network partition. Each exercises different layers of the persistence chain. [S130]
3. For PostgreSQL recovery from backups, use matching physical base-backup and continuous WAL evidence under the documented procedure. A logical SQL dump is not a physical WAL-replay base. [S131]

■ 28.6.6 WORDS — remember these

1. **Crash trace:** the ordered evidence surrounding an interruption — a record of persisted state, surviving log material and recovery decisions. **Fault model:** the failures an experiment assumes or injects — the specific process, operating-system, device or communication conditions being tested. **Recovery target:** the intended state or boundary to restore — a chosen time, log position or other engine-supported objective with matching evidence.

28.97 Practice and worked answers

1. **Find the unsafe ordering.** A changed data page reaches storage while its required log record remains only in memory. **Answer:** a crash can remove the only recovery explanation; the write-ahead requirement is violated under the stated model.
2. **Interpret group commit.** One sync covers five transactions' required records. **Answer:** batching can preserve each selected durability promise; five separate sync calls are not inherently required.
3. **Trace prefix 2.** T1 has begun and staged P=2 but not committed. **Answer:** recover P=5 from the known initial state.
4. **Trace prefix 5.** T1 committed P=2; T2 staged P=9 without commit. **Answer:** recover P=2.

5. **Trace prefix 11.** T3 committed P=4; T4 remains incomplete. **Answer:** recover P=4.
6. **Use a checkpoint.** Base P=2 at position 6, then replay 7–11. **Answer:** P=4 under the same model and matching boundary.
7. **Classify a malformed log.** COMMIT appears for a transaction never begun. **Answer:** reject the invalid model input; do not silently invent a transaction.
8. **Limit the experiment.** All prefix tests pass. **Answer:** the logical recovery model behaved as expected. Physical log parsing, process crashes and device persistence remain untested.

28.98 Common wrong ideas

1. **Wrong:** any append-only application log is a database recovery log. **Right:** contents and recovery guarantees differ.
2. **Wrong:** write-ahead means only that the log file was created first. **Right:** the required persistence ordering must hold for dependent changes.
3. **Wrong:** every COMMIT forces every data page. **Right:** durable recovery evidence can permit later page writeback.
4. **Wrong:** a log position is always a timestamp or row count. **Right:** its meaning belongs to the named stream and implementation.
5. **Wrong:** a checkpoint makes all older logs disposable. **Right:** backups, replicas and other recovery objectives can still require them.
6. **Wrong:** all engines recover by the toy committed-transaction loop. **Right:** physical redo, undo and visibility designs differ.
7. **Wrong:** a restart proves every expected business record survived. **Right:** reconcile the restored state against the intended recovery target.
8. **Wrong:** a simulated log cut is a power-loss test. **Right:** it exercises only the stated logical model.

28.99 Chapter summary in 20 lines

1. Recovery needs evidence about changes that may be only partly reflected in data files.
2. A recovery log has a different purpose from an ordinary business audit log.
3. Log validity, ordering and persistence are part of the protocol.
4. Required recovery records must precede dependent data-page persistence.
5. Commit acknowledgement must match the configured durability promise.
6. Group commit can synchronize several transactions together.
7. Asynchronous commit changes the recent-transaction durability contract.
8. Log positions identify progress within a named stream and history.
9. Generated, written, flushed and replayed positions are different boundaries.
10. Recovery cannot use records that did not survive.
11. Checkpoints provide a matching base and redo starting boundary.
12. Checkpoint frequency trades current I/O against later recovery work.
13. Older logs can remain necessary for other recovery and replication purposes.
14. Redo reconstructs required state; rollback prevents unaccepted transactional effects.
15. Engine recovery mechanisms are not universally identical.
16. The toy model stages after-values until a valid COMMIT appears.
17. Its incomplete T4 does not change the recovered P=4 result.

18. A checkpoint must be paired with the correct suffix and provenance.
19. Preserve original evidence before attempting recovery interventions.
20. State which fault model was actually tested and which remains unverified.

29

CHAPTER

LSM Trees and Compaction

29.0 What this chapter gives you

1. Some storage engines accept new writes into memory and later produce sorted immutable files instead of continually updating one in-place search tree. This chapter explains that family of designs.
2. You will follow a key through a memory table, sorted runs, older versions and deletion markers, then examine how compaction trades write work, read work and temporary space.
3. The companion model is an in-memory sorted-run exercise, not a RocksDB implementation. It has no real WAL, concurrent snapshots, file manifest or crash-safe installation protocol.

29.1 Sequential writes

29.1.1 PLAIN — in simple words

1. A system receiving many scattered key updates can collect them before writing larger organized groups to storage. That changes the physical work performed for each incoming request.
2. Writing sequentially can avoid some scattered access costs, but the work does not disappear. The engine must later find the latest values and reorganize accumulated files.
3. An LSM design is therefore not "writes are free." It moves and batches work, creating trade-offs that depend on the workload and storage system.

29.1.2 PLAIN — a picture in your head

1. Rather than walking to the filing cabinet for every arriving note, Mira collects notes in an organized tray and periodically files a sorted batch.
2. Later searches may need to consult several batches until the office merges them into a cleaner arrangement.
3. **Where the comparison breaks:** real engines need durable logs, ordered installation of files and concurrency controls. An ordinary tray that vanishes in a fire does not provide durable acknowledged writes.

29.1.3 PLAIN — a worked example

1. Ten thousand synthetic updates each contain 100 bytes of logical key/value payload. The incoming payload totals 1,000,000 bytes, or 1 MB decimal.
2. If storage writes those values first into new sorted files and later rewrites them during compaction, total bytes written can exceed 1 MB. WAL, indexes, filters, compression and metadata add further differences.

3. Define write amplification as physical bytes written divided by logical bytes written over a stated interval and boundary. If the measured boundary writes 5 MB for 1 MB of logical input, the ratio is 5.
4. This arithmetic does not predict RocksDB's ratio. The real ratio depends on configuration, key distribution, updates and which storage layers the measurement includes.

■ 29.1.4 PLAIN — what is really happening inside

1. New entries are organized in a memory structure and may also be logged for recovery. A flush creates a sorted persistent run under the engine's durability protocol.
2. Older runs remain available while newer runs accumulate. A lookup must choose the correct visible version across these structures.
3. Background compaction merges selected runs, discards eligible obsolete versions and installs replacement files safely. This is ongoing maintenance work, not a one-time setup step.

■ 29.1.5 TECHNICAL — the engineer's version

1. RocksDB's documented architecture includes a memtable, log files and sorted-string-table files. It is an embedded storage-engine library, not by itself a complete networked database service. [S140]
2. Log-structured merge designs batch updates into sorted runs and reorganize them over time. Their compaction policy determines the shape of the stored levels and which runs reads may need to consult. [S134]
3. Distinguish the engine's write amplification from a flash device's internal write amplification. Multiplying or comparing ratios requires compatible boundaries and denominators.

■ 29.1.6 WORDS — remember these

1. **LSM tree:** organize writes through memory and merged sorted runs — a log-structured merge family of storage designs with deferred reorganization. **Write amplification:** storage writes exceed logical input — a measured ratio of bytes written at a named layer to logical bytes accepted over a stated interval. **Sorted run:** a sequence already ordered by its search key — a file or logical collection that can be searched and merged using key order.

29.2 Memory tables and immutable files

■ 29.2.1 PLAIN — in simple words

1. A memory table holds recent entries in a searchable structure. When it reaches a chosen limit, the engine can freeze it and flush its contents into an immutable sorted file.
2. Immutable means the installed file is not edited in place by ordinary key updates. Later updates appear in newer structures, and compaction eventually replaces groups of files.
3. An immutable file is not automatically a complete immutable history. Old files can be retired, and the engine can discard obsolete versions under its retention rules.

■ 29.2.2 PLAIN — a picture in your head

1. Mira writes into today's tray. When it fills, she seals it for filing and starts a new tray so incoming work can continue.
2. The sealed batch does not receive handwritten corrections. New correction notes go into the active tray and take precedence under the office's version rules.

3. **Where the comparison breaks:** a real flush can fail partway. The engine needs metadata identifying which files are complete and installed, not merely files whose names happen to exist in a directory.

29.2.3 PLAIN — a worked example

1. Let the active table hold entries `(A,1,'red')`, `(B,2,'blue')` and `(A,3,'green')`, where the middle number is a strictly ordered model version.
2. A current lookup for A returns green because version 3 supersedes version 1. A snapshot lookup at version 2 returns red under the model's visibility rule.
3. After flush, the sorted representation groups A's versions and B's entry. The same logical answers should hold before and after the flush.
4. A flush that accidentally drops version 1 while a permitted version-2 snapshot still exists would violate that snapshot contract.

29.2.4 PLAIN — what is really happening inside

1. Engines can keep active and immutable memory tables concurrently while background workers flush completed batches.
2. Recovery logging protects recent acknowledged work according to the chosen settings. Disabling or weakening logging changes the failure contract; sorted files alone do not protect entries still only in memory.
3. A manifest or equivalent metadata identifies the active file set and related state. Installing new files and retiring old ones must survive interruption without exposing an arbitrary mixture.

29.2.5 TECHNICAL — the engineer's version

1. RocksDB describes memtable flushes producing sorted table files and associated log lifecycle management. Its configurable recovery and synchronization behaviour must be read separately from its high-level architecture. [S140]
2. Internal keys commonly combine user-key identity with sequence or version information in LSM implementations. The exact encoding and comparison order are product details; this book's tuples are a teaching representation.
3. A correct flush preserves every version required by active snapshots and the current visibility contract. A latest-value-only export is not a valid substitute when historical snapshot reads remain supported.

29.2.6 WORDS — remember these

1. **Memtable:** the searchable workspace for recent writes — an in-memory structure holding entries before or alongside persistent sorted files. **Immutable table file:** a sorted file replaced rather than updated in place — an installed run whose ordinary contents do not change after creation. **Manifest:** the record of which storage files belong to the current state — metadata coordinating installed runs and their lifecycle.

29.3 Sorted runs

29.3.1 PLAIN — in simple words

1. Sorting makes each run easier to search and combine with another run. But a key may appear in several runs, so finding one occurrence is not necessarily enough.

2. The engine must select the version visible to the reader. A newer deletion marker can override an older value that still exists in a lower run.
3. Range queries must merge ordered candidates and resolve duplicates or versions while preserving the requested key order.

29.3.2 PLAIN — a picture in your head

1. Three alphabetized card boxes contain successive batches of corrections. Looking up a name requires considering the newest relevant card, not accepting the first old card you find.
2. Printing all names in order requires merging the boxes and removing superseded copies under a clear rule.
3. **Where the comparison breaks:** engine lookups use indexes, filters and metadata to skip irrelevant runs. They do not necessarily scan every card in every box.

29.3.3 PLAIN — a worked example

1. Run R1 contains A at version 1 with value red and C at version 2 with value black. Run R2 contains A at version 3 with value green and B at version 4 with value blue.
2. A current ordered merge returns A=green, B=blue, C=black. It does not return A twice merely because two physical versions exist.
3. A permitted snapshot at version 2 returns A=red and C=black; B did not yet exist in that snapshot. The version limit changes the answer even though the same files may be consulted.
4. The model explicitly orders versions numerically. Wall-clock timestamps from unsynchronized writers are not silently substituted for this ordering rule.

29.3.4 PLAIN — what is really happening inside

1. Key-range metadata can rule out runs that cannot contain the requested key. Within a run, an index can locate the relevant block.
2. A Bloom filter can help reject absent keys without reading their data blocks, subject to its construction and lookup contract. A possible match still needs verification against real entries.
3. Snapshot visibility, key comparison and deletion handling must agree across memory tables, file indexes and compaction. A mismatch can create lost or resurrected records.

29.3.5 TECHNICAL — the engineer's version

1. A merge iterator can combine ordered runs using a priority structure while grouping equal user keys and selecting the highest visible version under the defined comparator.
2. In the simple model with k runs and n emitted candidates, a heap-based merge can organize candidate selection in approximately O(n log k) comparisons, excluding decoding, I/O and version-resolution costs. This is an algorithmic model, not a device-latency estimate.
3. Real compaction strategies constrain run overlap and hence lookup work. RocksDB's documentation distinguishes leveled and tiered/universal approaches rather than one universal LSM layout. [S134]

29.3.6 WORDS — remember these

1. **Merge iterator:** read several ordered runs as one ordered sequence — an iterator that combines candidates while resolving key and version rules. **Visible sequence limit:** the newest version a snapshot may use — a model or engine boundary excluding later updates from that reader. **Run**

overlap: several files cover some of the same key range — a layout property affecting how many candidates a lookup may need to inspect.

29.4 Read amplification

29.4.1 PLAIN — in simple words

1. A request for one small value can require several checks and reads across memory, indexes, filters and runs. That extra work is part of read amplification.
2. The term needs a definition. It can refer to bytes read relative to bytes returned, or to the number of storage probes required for one logical lookup. These are different metrics.
3. Optimizing only write speed can leave readers doing excessive work later. A storage design should be judged against the complete workload, not one convenient operation.

29.4.2 PLAIN — a picture in your head

1. Filing each arriving batch immediately is easy, but finding one customer's latest note becomes harder if the office keeps hundreds of overlapping boxes.
2. A directory and "not in this box" filters reduce unnecessary opening, while merging boxes reduces the number of places to check.
3. **Where the comparison breaks:** one physical read can bring many useful entries into cache. Counting boxes without naming the cache and block boundary can misrepresent actual device work.

29.4.3 PLAIN — a worked example

1. A toy absent-key lookup checks four overlapping runs. Filters reject three runs, leaving one candidate block read that ultimately finds no matching key.
2. Without those filters, the same model might read four candidate blocks. The logical result is still "not found"; the physical work differs.
3. If each block is 4,096 bytes and the model reads four, it transfers 16,384 bytes at that modeled boundary. A real cache can reduce device reads below this count.
4. A filter false positive costs extra checking. A false negative caused by a broken filter protocol can hide a real record and is a correctness failure, not merely a performance cost.

29.4.4 PLAIN — what is really happening inside

1. Read cost depends on run count, overlap, filters, block indexes, cache state and how many obsolete versions must be examined.
2. Negative lookups can be especially revealing because they may need to establish absence across all relevant structures.
3. Range scans have different behaviour from point lookups. A filter designed for exact-key membership may help less when the request spans a broad interval.

29.4.5 TECHNICAL — the engineer's version

1. Define separate metrics for run probes, data-block reads, bytes read and user-visible latency. Report warm and cold cache assumptions rather than treating these metrics as interchangeable.
2. Leveled and tiered compaction make different read/write/space trade-offs; workload skew and implementation optimizations can change the practical result. [S134]

3. The companion records model probe counts only. It does not measure RocksDB file I/O or claim a particular production read-amplification factor.

29.4.6 WORDS — remember these

1. **Read amplification:** extra storage work for a logical read — a defined ratio or probe count measured at a specified layer. **Negative lookup:** a search that establishes absence — a lookup returning no visible matching key after checking relevant structures. **Filter false positive:** a filter suggests a candidate that is absent — extra verification work without a false final membership claim.

29.5 Compaction and tombstones

29.5.1 PLAIN — in simple words

1. Compaction merges selected runs and removes versions that are no longer needed under the visibility and retention rules.
2. Deletion often begins as a tombstone: a newer marker saying that an older value must not be returned. The older bytes may still exist until safe cleanup occurs.
3. Dropping the marker too early can resurrect an older value from an unmerged file. Deletion correctness depends on which older versions remain reachable.

29.5.2 PLAIN — a picture in your head

1. A new card says "this account entry is deleted," while an older card remains in a lower archive box. Throwing away only the deletion card makes the old card look current again.
2. The cleanup clerk must know whether all older relevant copies have been covered before removing the marker.
3. **Where the comparison breaks:** real deletion also involves snapshots, backups and replicas. Safe removal from one LSM structure does not prove erasure from every system copy.

29.5.3 PLAIN — a worked example

1. R1 contains `(A,1,'red')`. R2 contains `(A,3,TOMBSTONE)`. A current lookup returns no A; a permitted snapshot at version 2 returns red.
2. Compacting only R2 and dropping its tombstone while R1 remains would wrongly expose red to a current lookup.
3. In a latest-only toy mode with no older snapshots and all relevant runs included, compaction can remove both the obsolete value and its deletion marker. The assumptions are part of the result.
4. If a later `(A,4,'green')` is added, current A is green while a version-3 snapshot remains deleted. Recreation is another versioned state, not evidence that the earlier deletion never occurred.

29.5.4 PLAIN — what is really happening inside

1. Compaction reads selected runs, resolves versions and writes replacement runs. The old files remain necessary until the new file set is safely installed and no reader still requires them.
2. This can require temporary space for both input and output files. A nearly full disk can prevent the cleanup work intended to reclaim space.
3. Tombstone removal requires knowledge of older versions and snapshot requirements. A simple age threshold is not universally sufficient without the protocol's supporting assumptions.

29.5.5 TECHNICAL — the engineer's version

1. Leveled compaction typically trades more rewriting for tighter run organization, while tiered/universal approaches can reduce some write amplification at the cost of read and space amplification. These are design tendencies, not universal numeric rankings. [S134]
2. The model exposes two operations: a visibility-preserving merge retaining versions, and latest-only compaction allowed only when the caller declares complete run coverage and no retained snapshots. Tests demonstrate the unsafe tombstone-drop counterexample.
3. Neither operation implements crash-safe file installation, concurrent reader references or a real deletion guarantee across backups. Those are separate protocol layers.

29.5.6 WORDS — remember these

1. **Compaction:** merge and reorganize stored runs — background work that installs a new representation and discards only eligible obsolete state. **Tombstone:** a newer marker hiding an older value — a deletion record retained until older visible copies and snapshot requirements permit removal. **Resurrection:** an old deleted value becomes visible again — a correctness failure caused by losing the deletion information while older state remains reachable.

29.6 Workload trade-offs

29.6.1 PLAIN — in simple words

1. A useful storage choice asks what the application actually does: frequent small updates, large range scans, mostly reads, repeated changes to a few hot keys or mostly new keys.
2. Background maintenance must keep up with the long-term write rate. A short benchmark can look fast while leaving a growing backlog of compaction work.
3. The right comparison includes steady-state latency, throughput, space, recovery and operational effort. One peak write number is not a complete description of the system.

29.6.2 PLAIN — a picture in your head

1. An office can accept incoming boxes quickly by stacking them in the corridor. That speed is not sustainable if nobody can sort the boxes and the corridor eventually fills.
2. A steady-state test asks whether incoming work and cleanup can continue together without an ever-growing backlog.
3. **Where the comparison breaks:** engines can throttle or stall writes deliberately to avoid exhausting memory or storage. A slowdown can be a protective response to accumulated work, not arbitrary malfunction.

29.6.3 PLAIN — a worked example

1. A synthetic service accepts 10 MB/s of logical writes. At a chosen measured storage boundary, steady-state write amplification is 6, implying about 60 MB/s of writes before other uncounted work.
2. If the device and competing workload leave only 40 MB/s for that work, the long-run demand exceeds the available budget by 20 MB/s under these assumptions.
3. A brief test may temporarily absorb the difference in memory and pending files. It cannot continue indefinitely without reducing demand, increasing capacity or changing the layout and workload.

4. These values are illustrative. Measure actual amplification and sustained capacity rather than selecting a product from this arithmetic alone.

29.6.4 PLAIN — what is really happening inside

1. Compaction competes with foreground reads and writes for CPU, memory, bandwidth and temporary space.
2. Skew matters: repeatedly updating a small key range can behave differently from uniformly rewriting the entire dataset. Compression and value sizes also change the balance.
3. A benchmark should include sufficient duration for maintenance to occur and should report pending work, not merely completed foreground requests.

29.6.5 TECHNICAL — the engineer's version

1. Report workload distribution, key/value sizes, update ratio, point/range reads, cache state, compaction policy, durability settings and storage headroom. Keep acknowledged-write guarantees equivalent across comparisons. [S134] [S140]
2. Test restore and recovery alongside steady-state performance. An engine that meets latency targets but cannot meet the required recovery objective does not meet the complete application requirement.
3. The book's model develops the mechanisms; it is not a benchmark ranking of LSM and B-tree products. Both families have sophisticated implementations and workload-dependent trade-offs.

29.6.6 WORDS — remember these

1. **Compaction backlog:** reorganization work waiting to be completed — pending maintenance that can increase read cost, space use or write stalls. **Steady state:** workload and maintenance remain sustainable together — an operating regime without unbounded growth of deferred work under the stated conditions. **Space amplification:** physical footprint exceeds live logical data — a ratio whose scope includes the selected files, versions and temporary maintenance state.

29.97 Practice and worked answers

1. **Compute write amplification.** One MB of logical input causes five MB of writes at the measured boundary. **Answer:** the ratio is 5; name whether WAL and device-internal writes are included.
2. **Read versions.** A has red at sequence 1 and green at sequence 3. **Answer:** latest reads green; a sequence-2 snapshot reads red under the model.
3. **Merge runs.** R1 has A1=red,C2=black; R2 has A3=green,B4=blue. **Answer:** latest ordered output is A=green,B=blue,C=black.
4. **Interpret filter work.** Three filters reject and one candidate block is read. **Answer:** one modeled block read, not necessarily one physical-device read.
5. **Find resurrection.** Drop A's sequence-3 tombstone while its sequence-1 value remains in another run. **Answer:** a current read can wrongly expose the old value.
6. **Allow latest-only cleanup.** All relevant runs are covered and no older snapshots are supported. **Answer:** the model can remove obsolete versions and a fully covered tombstone; the assumptions must remain explicit.
7. **Budget sustained work.** Ten MB/s logical input with amplification six needs sixty MB/s at the named boundary, but forty is available. **Answer:** demand exceeds that budget by twenty MB/s; temporary buffering does not solve the long-run mismatch.

8. **Limit the model.** Does the sorted-run exercise validate RocksDB crash recovery? **Answer:** no. It lacks real files, logging, manifest installation and concurrency.

29.98 Common wrong ideas

1. **Wrong:** LSM writes create no later work. **Right:** flushing and compaction move and rewrite data.
2. **Wrong:** immutable files mean eternal immutable history. **Right:** files and obsolete versions can be retired.
3. **Wrong:** finding any key occurrence gives its current value. **Right:** visibility and newer versions or tombstones matter.
4. **Wrong:** a filter's possible match proves a record exists. **Right:** verify against the actual entries.
5. **Wrong:** a tombstone can be discarded immediately. **Right:** older reachable values and snapshots can still require it.
6. **Wrong:** compaction always reduces instantaneous disk usage. **Right:** input and replacement files can coexist temporarily.
7. **Wrong:** a short burst benchmark demonstrates sustainable throughput. **Right:** deferred maintenance must keep up in steady state.
8. **Wrong:** LSM or B-tree is universally faster. **Right:** compare complete implementations under the actual workload and guarantees.

29.99 Chapter summary in 20 lines

1. LSM designs organize recent writes in memory and persistent sorted runs.
2. Batching changes the timing and shape of work rather than eliminating it.
3. Recovery logging protects recent work according to the selected durability contract.
4. Immutable files are replaced through controlled installation rather than ordinary in-place edits.
5. A manifest identifies which runs belong to the installed state.
6. Multiple versions of one key can coexist across memory and files.
7. A reader chooses the highest version visible to its snapshot.
8. Sorted runs support ordered search and merging.
9. Filters and range metadata can avoid some unnecessary probes.
10. Read amplification needs a defined metric and cache boundary.
11. Write amplification compares physical writes with logical input at a named layer.
12. Compaction reorganizes runs and removes only eligible obsolete versions.
13. A tombstone prevents an older deleted value from being returned.
14. Dropping a tombstone too early can resurrect old data.
15. Older snapshots can require versions that current readers no longer need.
16. Compaction can require temporary space for old and new files together.
17. Leveled and tiered strategies trade read, write and space costs differently.
18. Background maintenance must keep up with the sustained workload.
19. Measure backlog, latency, throughput and recovery, not just peak ingestion.
20. The educational run model is not a production storage engine or crash-safety proof.

30

CHAPTER

Durability From Application to Device

30.0 What this chapter gives you

1. "Saved" can mean copied into an application buffer, accepted by the operating system, acknowledged by a storage device or preserved across a specified failure. Those meanings are not interchangeable.
2. This chapter follows the acknowledgement chain and teaches you to state exactly which failures a durability claim covers. It separates process failure, operating-system failure, power loss, device loss and correlated loss of copies.
3. The supplied exercises do not cut power, reset hardware or modify a user's storage configuration. They test models and bounded local file/database operations, with their limits stated explicitly.

30.1 The acknowledgement chain

30.1.1 PLAIN — in simple words

1. A user sees a success message only after several layers have exchanged requests and responses. Each response confirms something at its own boundary.
2. A server saying "accepted into memory" is different from a database saying "committed under this durability setting." Both can be useful contracts, but the interface must not describe one as the other.
3. A lost response creates uncertainty. The operation may have completed before the reply disappeared, so the caller's silence cannot be treated as proof of rollback.

30.1.2 PLAIN — a picture in your head

1. A shop hands a document to a clerk, the clerk hands it to a filing department, and the department places it in a protected archive. Each handover can produce a receipt.
2. The first receipt does not prove that the archive step finished unless the organization explicitly waits for that boundary before issuing it.
3. **Where the comparison breaks:** software layers can buffer, reorder and batch work. The chain is a protocol, not necessarily one person waiting motionless at each step.

30.1.3 PLAIN — a worked example

1. Define three API statuses for a hypothetical service: queued, locally committed and externally confirmed. Queued means the request is retained under the queue's stated contract; locally committed means the database accepted the transaction; externally confirmed means a named external participant acknowledged its defined action.

2. A request can reach locally committed while the client never receives that response. It can also be locally committed while the outbox delivery remains pending.
3. The API should expose the original request identity so the caller can reconcile these states rather than creating an accidental second operation.
4. The status names are a proposed teaching contract, not claims about KedByte's live website or a deployed service.

30.1.4 PLAIN — what is really happening inside

1. The application maps lower-level results into business responses. Incorrect mapping can weaken a strong database guarantee by showing success too early or can hide an already committed result behind a generic failure.
2. Database settings decide which lower-level persistence conditions COMMIT waits for. Replication settings can add remote acknowledgement requirements, each with its own meaning.
3. Document the exact success boundary and make tests observe it. A diagram saying "disk" without naming the promised failure survival is incomplete.

30.1.5 TECHNICAL — the engineer's version

1. A durability contract should name the acknowledged operation, persistence boundary, fault model and assumptions about storage correctness. PostgreSQL's reliability documentation explicitly discusses the cache layers between memory and non-volatile storage. [S130]
2. Asynchronous commit intentionally changes when acknowledgement occurs relative to WAL persistence. It is not equivalent to ordinary durable commit merely because both statements eventually return success. [S136]
3. Request idempotency and outcome lookup address uncertain completion, not the physical persistence of the underlying storage. Both protocols are needed when the service requires both properties.

30.1.6 WORDS — remember these

1. **Acknowledgement chain:** the sequence of accepted handovers — responses from application, database, operating system and storage layers with distinct meanings. **Success boundary:** the event a positive response promises has occurred — the precise acceptance or persistence condition represented to the caller. **Durability contract:** what accepted data is promised to survive — a guarantee defined against specified failures and storage assumptions.

30.2 Process memory and operating-system caches

30.2.1 PLAIN — in simple words

1. A program can hold data in its own memory before passing it to the operating system. The operating system can then hold data in a cache before writing it to lower storage.
2. Ending one program does not necessarily erase the operating system's cache. Therefore a process-kill test is not the same as a sudden power-loss test.
3. Conversely, a successful buffered write call may only mean that bytes were copied into an intermediate buffer. The program must understand the interface it used.

30.2.2 PLAIN — a picture in your head

1. Mira's notebook, the office's outgoing tray and the archive cabinet are three different places. Losing Mira's notebook does not destroy papers already in the office tray; losing the whole office's power-protected storage is another event.
2. Testing only the notebook's loss does not establish the archive's behaviour in every disaster.
3. **Where the comparison breaks:** memory and cache failures do not follow simple room boundaries. Hardware, kernel, filesystem and controller behaviour all matter.

30.2.3 PLAIN — a worked example

1. A Python file object receives text in a user-space buffer. Calling its `flush()` pushes buffered output through the underlying stream interface; it is not by itself a universal guarantee that the device has durably stored the bytes.
2. On a supported ordinary file, an application can request a lower persistence boundary using the platform's synchronization API, such as `os.fsync()` after flushing the language buffer.
3. Even then, inspect errors and the platform contract. The API cannot promise survival of the complete loss of the only storage device holding the data.
4. This is an explanation of layers, not a replacement for a database engine's carefully designed logging and file-management protocol. [S43] [S64] [S141]

30.2.4 PLAIN — what is really happening inside

1. Buffered interfaces combine small writes for efficiency. A lower write can be partial or fail later, so robust software cannot assume that every requested byte was accepted merely because it attempted one call.
2. Operating-system writeback may occur after the application has continued. Delayed errors need to be surfaced at the relevant synchronization or closing boundary according to the platform.
3. Database engines already implement these details under their supported configurations. Application authors should not bypass the engine by editing its files directly.

30.2.5 TECHNICAL — the engineer's version

1. Linux `write()` can return fewer bytes than requested. Successful return does not, by itself, guarantee that the data has been committed to persistent storage. [S142]
2. Language-buffer flushing and file synchronization are separate operations. A correct low-level file protocol accounts for partial writes, synchronization errors, file metadata and directory entry persistence where required. [S141]
3. Tests must identify whether they terminate only a process, crash the operating system or remove power. The surviving cache layers differ between those fault models.

30.2.6 WORDS — remember these

1. **User-space buffer:** data waiting inside the program or library — memory used to batch output before passing it to the operating system. **Page cache:** operating-system memory holding file data — a cache between application I/O and lower storage devices. **Partial write:** fewer bytes were accepted than requested — an I/O result requiring explicit handling rather than an assumption of complete transfer.

30.3 Flush requests

30.3.1 PLAIN — in simple words

1. A flush request asks a layer to advance pending work toward a lower boundary. Different APIs use the word for different guarantees.
2. Synchronizing file contents does not always synchronize the directory entry that makes a newly created or renamed file discoverable after a crash.
3. Atomic visibility and durability are also different. A rename can appear all at once to concurrent readers while still requiring a persistence protocol for the promised crash behaviour.

30.3.2 PLAIN — a picture in your head

1. The archive stores a document securely, but its catalogue card naming the document has not yet been secured. After an interruption, the document and its discoverable name can have different states.
2. A reliable filing procedure considers both the contents and the catalogue.
3. **Where the comparison breaks:** filesystem rules differ, and not every platform accepts the same directory-synchronization operations. Copying a Linux recipe into another operating system without verification can be wrong.

30.3.3 PLAIN — a worked example

1. A conceptual safe-publication workflow writes a new file, verifies all intended bytes were written, synchronizes the file under the platform contract, installs its name using the appropriate atomic operation, and synchronizes required directory metadata.
2. If the process crashes before the name installation, readers should still use the previous installed version under the chosen design. If the installation completes durably, readers can use the new complete version.
3. The exact sequence and guarantees depend on the filesystem, operating system and overwrite semantics. This paragraph is a protocol outline, not executable cross-platform recovery code.
4. Database backup and storage tools should use their documented procedures rather than a home-made file replacement routine applied to a live database.

30.3.4 PLAIN — what is really happening inside

1. The filesystem coordinates data and metadata, while the device may have its own caches and flush commands. The application observes a returned result, not the physical movement of each electron.
2. Error handling is part of correctness. A failed sync is not a warning to ignore while still reporting the promised durable success.
3. Repeated sync calls do not fix a fundamentally incompatible or dishonest lower storage contract. Verify the supported stack and its documented assumptions.

30.3.5 TECHNICAL — the engineer's version

1. Linux `fsync()` synchronizes file data and associated metadata and waits for the device to report completion. Its manual explicitly notes that synchronizing a file does not necessarily synchronize its containing directory entry. [S141]

2. `fdatasync()` has a related but narrower metadata requirement: it synchronizes metadata needed for subsequent data retrieval, rather than all unrelated metadata. This distinction is platform-specific API detail, not a generic database setting. [S141]
3. Do not confuse an I/O synchronization guarantee with application-level transaction atomicity. Persisting two independent files successfully does not make their updates one atomic business operation.

30.3.6 WORDS — remember these

1. **File synchronization:** request persistence of pending file state — an operating-system operation with documented data, metadata and completion semantics. **Directory durability:** preserve the file's naming relationship — persistence of directory metadata needed to find the intended file after failure. **Atomic visibility:** observers do not see an intermediate replacement state — a concurrency property distinct from surviving a crash or power loss.

30.4 Device promises

30.4.1 PLAIN — in simple words

1. A storage device can acknowledge a write while retaining data in a volatile cache unless the protocol and configuration require a stronger boundary.
2. Power-loss protection can help preserve certain cached state, but its scope must be understood. It does not protect against every device fault or the loss of the entire storage location.
3. A database depends on the lower stack honouring its promises. No database setting can make a failed or dishonest device become reliable merely by naming it durable.

30.4.2 PLAIN — a picture in your head

1. A courier says a parcel is secured, but it is actually still in an unlocked van. Whether the receipt is trustworthy depends on what "secured" promised and whether the courier fulfilled it.
2. A backup power supply may keep the van's lock working, but it does not prevent the van itself from being destroyed.
3. **Where the comparison breaks:** device persistence has precise command semantics and firmware behaviour. Consumer metaphors cannot substitute for tested hardware and supported filesystem configurations.

30.4.3 PLAIN — a worked example

1. A database writes an 8,192-byte page. Suppose the storage path can fail after only part of that page reaches its final location. The surviving page may contain a mixture of old and new portions.
2. This is a torn-page risk, not merely the loss of the entire latest page. Recovery needs sufficient information to reconstruct or detect the damaged state under its design.
3. PostgreSQL's full-page-image mechanism addresses partial-page-write recovery within its documented assumptions. A page checksum can detect some damage but does not itself contain the original page needed to repair it. [S130] [S137]
4. The numerical page size here matches the common PostgreSQL page size; it does not claim that all devices provide 8,192-byte atomic writes.

30.4.4 PLAIN — what is really happening inside

1. Controllers, devices and filesystems can reorder or cache work. Flush and barrier semantics constrain that behaviour when correctly implemented and configured.
2. A battery-backed cache has maintenance requirements. An exhausted battery or an unverified firmware behaviour can invalidate assumptions that previously held.
3. Avoid casually disabling synchronization or barriers to obtain a better benchmark. First identify exactly which guarantee would change and whether the application can tolerate that loss.

30.4.5 TECHNICAL — the engineer's version

1. PostgreSQL's reliability guidance discusses operating-system, controller and device caches, full-page writes and the need for a trustworthy storage subsystem. Those assumptions are part of its durability story. [S130]
2. Checksums support detection, not general reconstruction or authentication. A checksum-valid page can still encode a logically wrong business change. [S137]
3. The educational tests do not validate a device's flush implementation, controller battery, firmware, atomic-write unit or power-loss protection. Such claims require separate hardware-specific evidence.

30.4.6 WORDS — remember these

1. **Volatile write cache:** acknowledged data still depends on power — intermediate storage that can lose pending writes when its power disappears. **Torn page:** only part of a page update survives — a mixed old/new physical page caused by an interrupted multi-part write. **Power-loss protection:** preserve specified device state during power interruption — a hardware capability whose scope and health must be verified.

30.5 Failure domains

30.5.1 PLAIN — in simple words

1. Two copies are useful only to the extent that the failures you care about do not destroy both together.
2. Two files on one disk are separate filenames but share the disk's failure. Two disks in one machine share other risks, such as a mistaken deletion command or a stolen machine.
3. Replication can spread copies across machines while faithfully spreading an accidental deletion. Recovery requires both independent failure planning and a way to return to an earlier valid state.

30.5.2 PLAIN — a picture in your head

1. Keeping two photocopies in the same folder helps if one sheet is smudged. It does not help if the folder is lost.
2. Keeping another copy elsewhere changes the risk, but someone must still verify that the remote copy is readable and current enough for its purpose.
3. **Where the comparison breaks:** digital copies can fail through shared credentials, software defects and automation, not only physical disasters. Geography alone does not make failures independent.

30.5.3 PLAIN — a worked example

1. A hypothetical system has a primary database and a replica in the same building. It tolerates some individual-machine failures but not necessarily a building-wide outage.
2. Add a backup in a separate administrative and storage boundary. This can improve recovery options, but its access credentials, encryption keys, retention and restore procedure must also survive the incident.
3. If the same compromised administrator can erase the primary, replica and backup, the copies share a destructive authority domain despite their different locations.
4. The example is a design exercise, not a claim that any named provider guarantees independence between its products.

30.5.4 PLAIN — what is really happening inside

1. Failure domains include processes, hosts, devices, racks, sites, regions, identities, software versions and operational procedures.
2. Independence is an assumption to test and document, not something proved by drawing two boxes on a diagram.
3. A complete recovery plan also includes dependencies such as configuration, secrets, keys and application versions. Restored database bytes can remain unusable if the only decryption key was lost.

30.5.5 TECHNICAL — the engineer's version

1. Build a fault matrix listing which copies and dependencies survive each proposed failure. Distinguish availability during the fault from recoverability after it.
2. Include logical corruption and operator error as well as hardware loss. A replica of the latest state is not necessarily a restore point preceding a bad transaction. [S131]
3. Avoid multiplying independent-failure probabilities when the independence assumption is unsupported. Shared infrastructure and authority can dominate the practical risk.

30.5.6 WORDS — remember these

1. **Failure domain:** components that can fail together — a shared physical, software, administrative or operational dependency boundary. **Correlated failure:** one cause affects several copies — a failure pattern that invalidates an assumption of independent redundancy. **Recovery dependency:** something required to make restored data usable — keys, configuration, software, access and other prerequisites beyond database bytes.

30.6 Testing a durability claim

30.6.1 PLAIN — in simple words

1. A test should state the failure it injects, the data acknowledged before that failure and the evidence checked afterward.
2. A clean shutdown is not a crash test. A process kill is not a power cut. A restored in-memory copy is not an off-site disaster rehearsal.
3. Small tests are still useful when their claims stay small. The problem is not limited evidence; it is describing limited evidence as proof of a larger untested guarantee.

30.6.2 PLAIN — a picture in your head

1. Testing whether a door closes is useful. It does not prove that the building survives a flood.
2. A test report becomes more useful when it says exactly which door, under which condition and with what observed result.
3. **Where the comparison breaks:** data systems can hide failure until later reads or reconciliation. "The program restarted" is only one observation in a recovery test.

30.6.3 PLAIN — a worked example

1. Prepare a disposable database with known records and a separate record of acknowledged request identities. Run a bounded operation, induce the approved fault, then recover into an isolated inspection environment.
2. Check structural integrity, required relationships, acknowledged outcomes, request replay behaviour and business totals. Record missing or uncertain evidence explicitly.
3. The book's actual companion performs in-memory rollback, backup/restore and logical surviving-prefix tests. It does not execute the hardware-fault procedure above.
4. A real power-loss test must use authorized disposable equipment and a reviewed safety procedure; this book does not ask readers to interrupt power to their ordinary computer or production database.

30.6.4 PLAIN — what is really happening inside

1. A test harness must avoid becoming the source of misleading evidence. Recording acknowledgements only in the same volatile memory that the test destroys can leave no trustworthy oracle afterward.
2. Failure injection timing and reproducibility matter. Randomly killing a process once may miss the vulnerable window entirely.
3. Preserve logs and configuration together with the outcome. A test that passed under stronger synchronization settings does not validate a later weakened configuration.

30.6.5 TECHNICAL — the engineer's version

1. Separate safety properties, such as "no acknowledged durable transaction is lost under the stated fault," from liveness and recovery-time properties. A test can establish one observation without proving all schedules or failures.
2. Record the exact engine/runtime, filesystem, device, settings, fault point, acknowledged set, recovered set and reconciliation procedure. Compare identities and values, not just row counts.
3. PostgreSQL and SQLite document storage and recovery assumptions that must remain part of the claim. Successful educational checks are not independent hardware certification. [S130] [S53] [S139]

30.6.6 WORDS — remember these

1. **Failure injection:** deliberately create a specified fault — a controlled experiment against an authorized disposable system. **Recovery oracle:** independent evidence of the expected result — identities and values used to judge what should remain after the tested failure. **Durability evidence:** observations supporting a bounded persistence claim — a test record tied to configuration, fault model and verification procedure.

30.97 Practice and worked answers

1. **Read the response contract.** An API acknowledges "queued in memory." **Answer:** this is not a promise of survival after process or power loss unless another documented mechanism establishes it.
2. **Separate buffers.** Python `flush()` succeeds. **Answer:** language-buffer progress alone is not a universal device-persistence guarantee.
3. **Handle a short write.** A low-level write returns 600 for a 1,000-byte request. **Answer:** only 600 bytes were reported accepted; the remaining bytes and error conditions require explicit handling.
4. **Find missing metadata.** A new file's data is synchronized but its directory entry is not covered by the platform's persistence procedure. **Answer:** content durability and discoverable-name durability may differ.
5. **Classify a torn page.** Half an updated page survives. **Answer:** recovery needs appropriate reconstruction information; a checksum alone can detect some damage but cannot recreate the lost original.
6. **Identify correlated authority.** One compromised credential can erase all three copies. **Answer:** the copies share a destructive administrative failure domain.
7. **Limit a process-kill test.** The process dies but the operating system continues. **Answer:** lower caches can survive; this is not a power-loss test.
8. **Choose a recovery oracle.** Compare only row counts after restore. **Answer:** counts can match while identities or values differ. Check the acknowledged set, relationships and defined business totals.

30.98 Common wrong ideas

1. **Wrong:** every success response means the same kind of saved. **Right:** name the acknowledgement and failure-survival boundary.
2. **Wrong:** a process kill and a power cut test the same thing. **Right:** different cache and storage layers survive.
3. **Wrong:** successful write means all bytes are permanently stored. **Right:** partial writes and persistence semantics must be handled explicitly.
4. **Wrong:** atomic rename automatically proves crash durability everywhere. **Right:** platform data and metadata synchronization rules still matter.
5. **Wrong:** power-loss protection prevents every storage failure. **Right:** its scope is narrower than total device or site survival.
6. **Wrong:** three copies are three independent failure domains. **Right:** infrastructure and authority can correlate their loss.
7. **Wrong:** a checksum repairs damaged data. **Right:** detection and reconstruction require different evidence.
8. **Wrong:** passing bounded local tests certifies production durability. **Right:** the tested configuration and fault model define the claim.

30.99 Chapter summary in 20 lines

1. Saved data passes through several acknowledgement boundaries.
2. The application must map those boundaries into honest business statuses.
3. A lost response can leave a committed operation unresolved to its caller.

4. User-space buffers, operating-system caches and device caches are separate layers.
5. Language-buffer flushing is not the same as device synchronization.
6. Low-level writes can be partial and require error handling.
7. File data and directory metadata have distinct persistence requirements.
8. Atomic visibility does not by itself establish durability.
9. Synchronization depends on the lower stack honouring its documented contract.
10. Volatile controller and device caches affect power-loss assumptions.
11. Partial-page writes require appropriate recovery protection.
12. Checksums detect some damage but do not reconstruct missing content.
13. Power-loss protection has a specific scope and maintenance requirements.
14. Copies can share physical, software and administrative failure domains.
15. Replication does not replace a recoverable earlier version.
16. Keys, configuration and application dependencies belong in recovery planning.
17. Tests must state the exact fault they inject.
18. Preserve an independent record of acknowledged identities and expected values.
19. Reconcile recovered meaning as well as structural validity.
20. Report bounded evidence without extending it to untested hardware or failures.

31

CHAPTER

Backups, Restore and Point-in-Time Recovery

31.0 What this chapter gives you

1. A backup is a copy kept for a recovery purpose. Its value depends on whether the required state can actually be restored, validated and made usable within the intended limits.
2. This chapter distinguishes logical exports, physical backups, snapshots, log archives and restore rehearsals. You will calculate recovery objectives and identify dependencies that a database file alone does not contain.
3. The actual lab restores synthetic SQLite data into a fresh isolated destination. PostgreSQL point-in-time recovery is explained from its documentation, not claimed as an executed server restore.

31.1 Copies with a recovery purpose

31.1.1 PLAIN — in simple words

1. A backup is not simply any second copy. It has a defined source, time or recovery boundary, retention policy and procedure for turning it back into usable data.
2. A replica usually follows current changes, including mistakes. A backup can preserve an earlier state needed after accidental deletion or corruption.
3. An export can be useful without containing everything required for a complete service recovery. Know what it includes and what it omits.

31.1.2 PLAIN — a picture in your head

1. A photocopy of yesterday's ledger can help recover from today's mistaken erasure. A second clerk copying today's erasure immediately may leave two equally damaged ledgers.
2. The recovery copy needs a date, an identity and enough context to interpret its contents.
3. **Where the comparison breaks:** digital backups may contain physical engine structures, logical SQL or incremental changes. They are not interchangeable sheets that can be opened by any database version.

31.1.3 PLAIN — a worked example

1. Mira exports the four canonical order lines. Their total is 28,650 paise across six units. That export can help reconstruct or verify those lines.
2. It does not automatically include customers, permissions, product metadata, outbox state, schema definitions, secrets or the application version needed to run the service.

3. A complete recovery inventory lists every required component and how it is acquired. Some dependencies should be backed up separately under stronger access restrictions rather than bundled into a public teaching archive.
4. The canonical export contains invented data only. No live customer records or credentials are included in this book's files.

31.1.4 PLAIN — what is really happening inside

1. Logical backups describe objects and values in a form the target can reconstruct. Physical backups preserve the engine's storage representation under its supported consistency procedure.
2. The two approaches have different portability, granularity and recovery characteristics. A logical export can support selected-object restoration, while a physical base backup can participate in a matching WAL-replay recovery chain.
3. Backups themselves need monitoring. A scheduled job that stopped producing usable copies weeks ago is not a current recovery capability.

31.1.5 TECHNICAL — the engineer's version

1. PostgreSQL distinguishes SQL dumps from filesystem-level base backups. `pg_dump` output is logical and is not a physical base for continuous WAL replay. [S67] [S131]
2. SQLite's online backup API supports making a consistent database copy through the engine. Copying only a live main file without the appropriate journal/WAL coordination can produce an invalid or incomplete backup. [S55] [S139]
3. A backup manifest should name source identity, engine version, acquisition method, completion state, checksums and required accompanying material. A filename containing a date is not sufficient provenance.

31.1.6 WORDS — remember these

1. **Backup:** a copy retained for a defined recovery purpose — data and metadata acquired under a procedure that supports a stated restoration objective. **Logical backup:** reconstruct objects from their logical definitions and values — an export such as SQL or another supported logical representation. **Physical backup:** preserve an engine's storage representation — a copy requiring the matching consistency, version and recovery procedure.

31.2 Consistent snapshots

31.2.1 PLAIN — in simple words

1. A recovery copy must represent a state the database can interpret consistently. Copying files one after another while they change can mix incompatible moments.
2. A supported snapshot or backup protocol establishes the required boundary, sometimes using recovery logs to reconcile files copied over an interval.
3. A storage snapshot is not automatically an application-consistent service snapshot. Other databases, object stores and external actions may have separate boundaries.

31.2.2 PLAIN — a picture in your head

1. Copying the first half of a ledger before a transfer and the second half afterward can create an impossible combined total.

2. A controlled copying procedure either fixes the view or preserves the instructions needed to reconstruct the matching state.
3. **Where the comparison breaks:** database backups can be consistent even when file copying spans time, provided the engine's backup and log protocol supplies the required recovery evidence. They need not all be instantaneous photographs.

31.2.3 PLAIN — a worked example

1. A toy service stores an order in one database and its attachment in an object store. The database backup records attachment key ATT-7, but the object-store snapshot predates ATT-7's upload.
2. Each component can be internally valid while the restored service contains a missing attachment reference.
3. The recovery plan must define the relationship: coordinate acquisition, retain referenced object versions or verify and reconcile missing objects under an explicit policy.
4. Replacing the missing attachment with a blank file would conceal the failure rather than restore the original evidence.

31.2.4 PLAIN — what is really happening inside

1. Recovery consistency has layers: file-format validity, transaction consistency, cross-store references and business meaning.
2. Engine-supported backup procedures specify which logs, metadata and completion markers are required. A partially completed backup must not be labelled usable merely because many files exist.
3. The destination should be isolated while it is validated. Starting ordinary outbound workers immediately after restore can repeat historical notifications or other external effects.

31.2.5 TECHNICAL — the engineer's version

1. PostgreSQL continuous archiving combines a supported physical base backup with the required WAL sequence. The base copy can span time because replay resolves the corresponding physical inconsistencies under the documented procedure. [S131]
2. SQLite's backup API operates through database connections rather than treating a changing file as an uncoordinated byte source. The companion uses this API for its actual synthetic restore test. [S55] [S86]
3. Define application-consistency checks beyond engine startup: foreign keys, expected objects, request/outbox relationships, attachment references and reconciliation totals.

31.2.6 WORDS — remember these

1. **Consistent backup boundary:** the state the recovery procedure can validly reconstruct — a snapshot or log-supported acquisition point with matching metadata. **Application consistency:** related service components agree — correctness of references and workflows beyond one engine's internal file validity. **Restore isolation:** keep the recovered copy from affecting live systems — a controlled environment with outbound effects and production access disabled during validation.

31.3 Log retention

31.3.1 PLAIN — in simple words

1. Point-in-time recovery starts from an earlier supported base and replays the required changes only as far as the chosen target.
2. The necessary log chain must be continuous. A missing required segment can prevent reaching the target even if newer segments remain available.
3. Retaining a base backup without its required logs, or retaining logs without a usable matching base, can leave a collection of files that cannot meet the promised recovery objective.

31.3.2 PLAIN — a picture in your head

1. A verified ledger at Monday morning plus every accepted correction through Thursday can reconstruct Thursday's state. Missing Tuesday's corrections cannot be repaired by reading Friday's entries alone.
2. The base and the journal belong to one history, including any later branches created by recovery.
3. **Where the comparison breaks:** real log segments have engine-specific identities, timelines and validation rules. Sorting filenames lexically is not a recovery algorithm.

31.3.3 PLAIN — a worked example

1. A synthetic recovery chain has a valid base at 10:00 and complete archived changes through 10:30. An incident occurs at 10:40.
2. The chain can support an appropriate target no later than its verified coverage, subject to the engine's transaction boundaries. It cannot claim recovery through 10:40 merely because the primary generated changes after 10:30.
3. If the intended target is 10:25, the plan must preserve every required part from the base through that target. A missing segment at 10:15 cannot be ignored.
4. These times illustrate coverage; actual WAL recovery targets and inclusive/exclusive settings require the documented engine procedure.

31.3.4 PLAIN — what is really happening inside

1. Archiving is a transfer and retention workflow with its own failures. Generated WAL is not the same as successfully archived, verified and retrievable WAL.
2. Recovery can create a new timeline or history branch. Later backups and log references must identify which history they continue.
3. Retention deletion should evaluate dependencies before removing a base or log range. Deleting the oldest-looking file can break several otherwise retained restore points.

31.3.5 TECHNICAL — the engineer's version

1. PostgreSQL PITR requires a continuous sequence of archived WAL extending back at least to the required start of the base backup. Its documentation separately explains recovery targets and timelines. [S131]
2. Record archive success, failures, coverage gaps and restore-tested ranges. A counter saying that some archive command succeeded does not prove every desired target remains recoverable. [S144]
3. Do not replay a physical log against a different logical export or unrelated cluster. Match base identity, version compatibility and timeline evidence before attempting recovery.

31.3.6 WORDS — remember these

1. **Point-in-time recovery:** restore a supported earlier state — replay from a matching base to a chosen engine-supported recovery target. **Archive coverage:** the recovery history actually retained and retrievable — verified continuity of required log material over a stated range. **Recovery timeline:** a named branch of physical recovery history — metadata distinguishing histories after recovery or promotion under the engine's rules.

31.4 Recovery objectives

31.4.1 PLAIN — in simple words

1. Recovery point objective, or RPO, describes the tolerated loss of recent data measured against a defined incident boundary. Recovery time objective, or RTO, describes how quickly the required service should be restored.
2. An objective is a requirement, not evidence that the current backup system achieves it. A restore rehearsal measures actual performance under stated conditions.
3. The service is not necessarily recovered when file copying finishes. Validation, application startup, access checks and controlled return to users can consume substantial time.

31.4.2 PLAIN — a picture in your head

1. Mira says, "We can tolerate losing at most fifteen minutes of accepted work, and we need usable service within one hour." These are two different targets.
2. Finding a recent ledger quickly does not help if nobody can unlock the cabinet or run the software that reads it.
3. **Where the comparison breaks:** recovery objectives need precise definitions for accepted work, incident start, degraded service and completion. Everyday phrases such as "back online" can hide incompatible meanings.

31.4.3 PLAIN — a worked example

1. An incident occurs at 10:40. The verified recoverable state reaches 10:30, so the recent-data gap is ten minutes under this simplified timeline. That fits a fifteen-minute RPO only if the coverage and acceptance definitions truly match.
2. A rehearsal takes 32 minutes to restore, 18 minutes to validate and 12 minutes to return the required service safely. Total recovery time is 62 minutes.
3. A one-hour RTO is missed by two minutes. Reporting only the 32-minute copy time would conceal the operational result.
4. These invented values show how to calculate the measures; they are not a promise about the book's lab or any hosting plan.

31.4.4 PLAIN — what is really happening inside

1. Restore time depends on data volume, retrieval bandwidth, decompression, replay, index work, validation and operational coordination.
2. A backup in low-cost archival storage can have a retrieval delay that dominates the recovery timeline. Keys or approvals can also become bottlenecks.
3. Objectives can differ by service function. A read-only emergency view may return before full write service, but the completion definition must state that distinction honestly.

31.4.5 TECHNICAL — the engineer's version

1. A lower-bound transfer calculation for 1 TB decimal at a sustained 100 MB/s is 10,000 seconds, about 2.78 hours, before replay, validation or contention. It is not an end-to-end restore estimate.
2. Define RPO relative to acknowledged operations and verified recoverable history, not merely backup-job start times. Define RTO's start and finish events before measuring it.
3. Keep objectives, measured rehearsal results and residual assumptions in separate fields. A green backup-job status is not a measured RTO.

31.4.6 WORDS — remember these

1. **RPO:** how much recent accepted work may be lost — a recovery-point requirement defined against a specified incident and acceptance boundary. **RTO:** how long required service restoration may take — a recovery-time requirement with explicit start and completion events. **Restore rehearsal:** actually exercise the recovery procedure — a controlled test measuring whether the required state and service can be recovered.

31.5 Restore rehearsals

31.5.1 PLAIN — in simple words

1. A backup that has never been restored is an untested recovery input. Reading its filename and checking that it is large enough is not a substitute for trying the procedure.
2. Rehearse in isolation, verify the restored state and record the results. Keep outbound side effects disabled until their replay risks are understood.
3. Tests should include failure cases: missing files, wrong keys, incompatible versions and a target beyond the retained log coverage.

31.5.2 PLAIN — a picture in your head

1. An emergency key is useful only if it opens the right door. A rehearsal tries it before the emergency rather than admiring the key's label.
2. The rehearsal also checks what is behind the door and whether the staff can perform the required work.
3. **Where the comparison breaks:** restoring historical state can reactivate pending jobs, old permissions or stale request records. A restored service can create new harm if connected to live systems without review.

31.5.3 PLAIN — a worked example

1. The companion creates the canonical synthetic shop, makes an engine-supported SQLite backup into a fresh destination and checks that order totals remain O-1042=17,100 and O-1043=11,550 paise.
2. It also checks the combined 28,650-paise total, foreign-key violations and structural integrity under the available SQLite checks.
3. A later separate mutation of the source should not silently mutate the already completed backup destination. The test verifies separation under this local setup.
4. This is an actual bounded backup/restore exercise, not an off-site recovery rehearsal, encrypted-archive test or PostgreSQL PITR run.

31.5.4 PLAIN — what is really happening inside

1. Structural verification and business verification answer different questions. A readable database can still contain the wrong target time or missing expected records.
2. Compare identities and values, not just totals: two offsetting errors can leave a sum unchanged. Include representative application workflows and boundary tests.
3. Record the exact backup input, software versions, procedure, timings and checks. A later successful restore from another file does not retroactively validate an earlier failed backup.

31.5.5 TECHNICAL — the engineer's version

1. PostgreSQL `pg_verifybackup` checks a base backup against its manifest and performs defined WAL-related validation. Its documentation explicitly states that this does not replace test restores and application-data verification. [S138]
2. A restore acceptance report should include acquisition identity, hashes, target boundary, isolated environment, executed commands, checks, timing and unresolved findings.
3. Restored outbox and idempotency records require special review before external dispatch resumes. A backup can precede actions already performed outside the restored database; reconciliation must prevent unsafe replay.

31.5.6 WORDS — remember these

1. **Backup verification:** check the retained input against defined expectations — structural, manifest or checksum validation that may precede a full restore. **Restore acceptance:** decide whether the recovered state meets its objective — a documented result based on actual recovery and relevant application checks. **Replay review:** inspect historical pending work before reactivation — reconciliation of restored tasks with effects that may already exist outside the backup boundary.

31.6 Access and retention of backups

31.6.1 PLAIN — in simple words

1. Backups can contain sensitive information long after the live application has changed or deleted it. They need access controls, retention rules and a way to prevent old data from quietly returning to normal use.
2. Encryption helps protect stored bytes, but recovery also depends on keeping the required keys available to authorized responders.
3. Retention should preserve the required restore points without claiming that every copy lasts forever. Storage cost, purpose and deletion obligations need an explicit policy.

31.6.2 PLAIN — a picture in your head

1. An old ledger in a locked archive is still a ledger containing people's information. Locking it does not make its contents disappear.
2. Destroying the only key may prevent recovery, while leaving a spare key on the door defeats the access control.
3. **Where the comparison breaks:** digital deletion and encryption-key lifecycle are complex across replicas, caches and services. A local file deletion is not proof that every retained copy was erased.

31.6.3 PLAIN — a worked example

1. A teaching policy retains daily backups for 30 days and monthly backups for a separately defined period. The policy is an example, not legal advice or a universal retention recommendation.
2. A deletion request affects the live system today, while an older restricted backup still contains the record until its permitted retention expires.
3. A restore procedure must reapply the relevant deletion or suppression decisions before that old record becomes available to ordinary users again, under the organization's approved policy.
4. Keep evidence of which stores were checked and which retained exceptions remain. Do not report universal erasure based only on the live-table DELETE.

31.6.4 PLAIN — what is really happening inside

1. Backup credentials should be scoped to their tasks. Separate the ability to create backups from the ability to erase every historical copy where the storage system and operational model support that separation.
2. Retention automation needs dependency checks so it does not delete logs required by retained base backups. It also needs monitoring for failures and unexpectedly retained data.
3. Recovery access should be rehearsed under realistic restrictions. A plan that works only with an unavailable administrator's personal credentials is operationally fragile.

31.6.5 TECHNICAL — the engineer's version

1. Record backup classes, purpose, retention, encryption-key dependencies, allowed operators and restore-time suppression procedures. Chapter 48 develops lineage and deletion boundaries in more detail.
2. Protect manifests and integrity metadata against unauthorized replacement. A checksum stored beside a file can detect accidental damage but does not authenticate the file if an attacker can rewrite both.
3. The book does not prescribe a jurisdiction-specific retention period. Legal and contractual requirements must be checked for the actual organization and data classes; the technical protocol should expose those decisions rather than invent them.

31.6.6 WORDS — remember these

1. **Backup retention:** how long recovery copies remain available — a policy governing restore-point preservation and eventual retirement. **Key dependency:** recovery requires particular cryptographic material — an access and availability requirement for decrypting retained data. **Restore-time suppression:** prevent retired data from re-entering ordinary use — reapplication of approved deletion or access decisions after historical recovery.

31.97 Practice and worked answers

1. **Classify an export.** A CSV contains only four order lines. **Answer:** it is a useful logical data extract, not automatically a complete service backup.
2. **Choose a PITR base.** Can a PostgreSQL SQL dump be used directly as the physical base for WAL replay? **Answer:** no; use the supported matching physical-backup procedure.
3. **Find cross-store inconsistency.** The database references ATT-7, absent from the object-store restore. **Answer:** component validity does not establish application consistency; reconcile the missing dependency.

4. **Calculate the data gap.** Incident at 10:40, verified coverage through 10:30. **Answer:** ten minutes under the stated acceptance and coverage assumptions.
5. **Calculate recovery time.** Restore 32 minutes, validate 18, return service 12. **Answer:** 62 minutes, missing a one-hour objective by two minutes.
6. **Compute a transfer floor.** One TB decimal at 100 MB/s. **Answer:** 10,000 seconds, about 2.78 hours before other recovery work.
7. **Interpret manifest success.** `pg_verifybackup` reports success. **Answer:** perform an actual test restore and application checks; the tool does not establish every recovery property.
8. **Review restored workers.** Historical outbox rows appear pending. **Answer:** keep dispatch isolated until external effects and replay identities are reconciled.

31.98 Common wrong ideas

1. **Wrong:** any second copy is a usable backup. **Right:** provenance, consistency and a tested restore procedure matter.
2. **Wrong:** replication protects against every mistaken deletion. **Right:** replicas can copy the mistake.
3. **Wrong:** individually valid component snapshots form a valid service snapshot. **Right:** cross-store references can disagree.
4. **Wrong:** newer WAL can compensate for missing required older segments. **Right:** the recovery chain must be continuous.
5. **Wrong:** RPO and RTO are achieved because they are written in a policy. **Right:** objectives need measured evidence.
6. **Wrong:** copying finished means recovery finished. **Right:** validation and safe service return also take time.
7. **Wrong:** matching row counts prove correct restoration. **Right:** identities, values and business relationships must be checked.
8. **Wrong:** deleting live data proves every backup copy is gone. **Right:** retained copies and restore-time controls need separate accounting.

31.99 Chapter summary in 20 lines

1. A backup has a defined recovery purpose and provenance.
2. Logical exports and physical backups support different procedures.
3. A PostgreSQL logical dump is not a physical WAL-replay base.
4. Supported backup interfaces coordinate changing data safely.
5. Cross-store application consistency extends beyond one database's validity.
6. Keep restored environments isolated while validating them.
7. Point-in-time recovery requires a matching base and continuous required logs.
8. Generated log data is not automatically archived and retrievable coverage.
9. Recovery timelines distinguish branches of physical history.
10. Retention deletion must preserve dependencies of retained restore points.
11. RPO defines tolerated recent-data loss under a stated boundary.
12. RTO defines the allowed time to restore required service.
13. Objectives are different from measured rehearsal results.
14. Transfer, replay, validation and controlled service return all consume time.

15. Manifest verification does not replace a test restore.
16. Check identities, values, constraints and business reconciliation after recovery.
17. Historical outbox tasks need replay review before external dispatch resumes.
18. Backups require access, encryption-key and retention controls.
19. Restoring old data must not silently undo approved deletion or suppression decisions.
20. Report exactly which recovery procedure was executed and which remains untested.

32

CHAPTER

Corruption, Reconciliation and Repair

32.0 What this chapter gives you

1. A wrong answer can come from damaged bytes, an invalid relationship, a faulty query, a mistaken business entry or a misunderstanding of the report. These are different problems with different evidence.
2. This chapter teaches a disciplined path from suspected disagreement to verified diagnosis and bounded repair. It explains checksums, structural checks, reconciliation and evidence preservation without treating a repair command as a substitute for understanding.
3. The exercises use synthetic records and disposable byte strings. They do not modify a live database, bypass engine safeguards or claim to recover a user's damaged files.

32.1 Detecting disagreement

32.1.1 PLAIN — in simple words

1. Disagreement is a reason to investigate, not a diagnosis by itself. A report total can be wrong even when every stored byte is intact.
2. Begin by stating the expected relationship and the observed difference. Then identify which layer could explain it: query, input, schema, transaction, storage or outside-world measurement.
3. Preserve uncertainty. The earlier expected-stock-10/observed-stock-9 example remains unexplained unless new evidence actually establishes a cause.

32.1.2 PLAIN — a picture in your head

1. Two calculators show different totals. One may contain a different list, one may use a different tax rule, or one may be broken. The disagreement alone does not identify which.
2. Comparing the inputs and rules is often more informative than replacing the calculator immediately.
3. **Where the comparison breaks:** a database may have several simultaneous faults. Finding one query error does not prove that every underlying record is healthy.

32.1.3 PLAIN — a worked example

1. The canonical agreed lines total 28,650 paise. A report joining lines to product tags produces 51,300 paise because some lines appear once per tag.
2. The difference is 22,650 paise, exactly the repeated notebook contribution in that fixture. The stored agreed prices need not be corrupted for the report to be wrong.
3. A repair aimed at reducing stored order totals to compensate for the report would damage correct source data. The query's grain and join must be fixed instead.

4. This demonstrates a verified logical cause in the synthetic fixture; it does not resolve any unrelated real inventory discrepancy.

32.1.4 PLAIN — what is really happening inside

1. Reproduce the smallest relevant observation using a stable data boundary. A changing live dataset can make two correctly run queries disagree because they observed different states.
2. Keep the exact query, parameters, schema version and result. A screenshot of one number without its population or exclusions is weak evidence.
3. Classify findings separately: suspected, reproduced, explained, repaired and revalidated. These are stages, not interchangeable labels for "done."

32.1.5 TECHNICAL — the engineer's version

1. Distinguish physical corruption, structural inconsistency, violated business invariants and incorrect derived queries. A diagnostic plan should select tests that discriminate among these classes.
2. A stable snapshot or isolated copy can support reproducible comparisons, subject to the acquisition procedure. Query plans and joins must be reviewed at their actual grain. [S58] [S66]
3. Do not modify authoritative rows merely to make a dashboard match a preferred number. Identify the source of authority and preserve the original evidence before proposing a correction.

32.1.6 WORDS — remember these

1. **Disagreement:** two observations or expectations do not match — a finding that requires diagnosis rather than automatically proving storage damage. **Physical corruption:** stored representation is damaged or invalid — a failure involving bytes or engine structures rather than merely an undesirable business value. **Logical inconsistency:** records or calculations violate their intended meaning — a mismatch that can exist even when all files are structurally readable.

32.2 Checksums and their limits

32.2.1 PLAIN — in simple words

1. A checksum or digest summarizes bytes so a later comparison can detect many changes. It answers a question about byte identity under the chosen algorithm.
2. It does not explain whether the original bytes were correct, whether a transaction was authorized or how to reconstruct missing information.
3. If someone can replace both a file and its stored checksum, a matching pair does not authenticate the file's origin. Integrity comparison and trusted provenance are separate requirements.

32.2.2 PLAIN — a picture in your head

1. A parcel's inventory code helps detect that its contents changed after packing. It does not prove that the packer selected the right goods.
2. A dishonest person who repacks the parcel and rewrites the code can defeat a comparison against that same untrusted label.
3. **Where the comparison breaks:** cryptographic digests have formal security properties and collision limits. An everyday inventory code is only an analogy, not an implementation of those properties.

32.2.3 PLAIN — a worked example

1. The lab hashes a fixed synthetic byte string, changes one byte in a separate copy and hashes it again. The observed digests differ for that fixture.
2. It also verifies that hashing the unchanged bytes reproduces the original digest. These checks establish deterministic behaviour and the detected alteration in the example.
3. They do not prove that no two different byte strings can ever share the digest or that the original string represented a true sale.
4. A trusted manifest must bind the expected digest to the intended source and version through an appropriate authority or custody process.

32.2.4 PLAIN — what is really happening inside

1. Checksums are computed over a defined byte range. Omitted files, metadata or fields are outside that range's protection.
2. Database page checksums can detect some storage damage when pages are checked, but a logically valid wrong UPDATE can produce a perfectly checksum-valid page.
3. Verification timing matters. A successful check yesterday does not prove that a file is unchanged today unless the relevant bytes were checked again or protected by another justified mechanism.

32.2.5 TECHNICAL — the engineer's version

1. PostgreSQL data checksums provide page-level corruption detection within their documented scope. They are not a general repair facility or a proof of business correctness. [S137]
2. Cryptographic digest comparison, such as SHA-256 through Python's `hashlib`, can identify exact artifact bytes when paired with a trusted expected digest. It does not supply authentication by itself. [S99] [S100]
3. Define the manifest's coverage and protect its provenance. Hashing a ZIP after delivery identifies that ZIP's bytes; it does not imply every internal claim or test result is true.

32.2.6 WORDS — remember these

1. **Checksum coverage:** the bytes included in an integrity calculation — the exact range or file set a checksum can help verify. **Trusted expected digest:** an integrity reference with justified provenance — a digest bound to an intended artifact through a trusted source or process. **Authentication:** establish a source or actor under a security protocol — a property not supplied merely by an unkeyed matching checksum.

32.3 Structural checks

32.3.1 PLAIN — in simple words

1. A structural check asks whether the database's internal representation satisfies the checker's defined rules. It can detect problems invisible to an ordinary SELECT.
2. Different checks cover different things. A general integrity check may not test every foreign-key relationship or every business invariant.
3. A successful result means the selected checks found no covered problem. It does not mean that every possible fault has been ruled out.

32.3.2 PLAIN — a picture in your head

1. A building inspection can verify that shelves are stable without verifying that every form is filed under the right customer. A separate filing audit checks those relationships.
2. Both checks matter, and neither should be described as the other.
3. **Where the comparison breaks:** database checkers have precise implementation scopes, may require permissions and can consume substantial resources. Running every checker on a busy production system is not automatically safe.

32.3.3 PLAIN — a worked example

1. In an isolated SQLite demonstration, a child row can exist without its intended parent if foreign-key enforcement was bypassed during insertion.
2. A general `PRAGMA integrity_check` and a separate `PRAGMA foreign_key_check` answer different questions. The earlier labs demonstrate that a structurally accepted database can still contain the orphan relationship.
3. Enabling foreign keys afterward does not retroactively create the missing parent or prove all old rows valid. Inspect existing relationships explicitly.
4. Repairing the orphan requires a business decision based on evidence; inventing a parent record just to silence the checker is not necessarily correct.

32.3.4 PLAIN — what is really happening inside

1. Checkers traverse selected engine structures and compare them against invariants encoded in the checker.
2. They can miss application-level rules because those rules may not exist in the schema. A capacity total exceeding ten is invisible to a checker that only knows each row's positive quantity constraint.
3. Checker errors should be retained exactly, together with version and options. Summarizing them as "database broken" loses information needed for a bounded diagnosis.

32.3.5 TECHNICAL — the engineer's version

1. SQLite documents `integrity_check` and `foreign_key_check` separately. Use both when their respective coverage is needed, while respecting the workload and acquisition boundary. [S75] [S59]
2. PostgreSQL `amcheck` provides defined table/index consistency checks. Its options, lock implications and limitations must be reviewed before use; it is not a universal repair command. [S146]
3. A checker's successful exit should be recorded with the exact object scope. Testing one index or one database does not certify every other store in the service.

32.3.6 WORDS — remember these

1. **Structural integrity check:** inspect engine representation rules — a diagnostic procedure for covered page, tree or table/index invariants. **Referential check:** verify relationships between records — a test that child references resolve according to the schema's key rules. **Checker scope:** the objects and properties actually examined — the boundary limiting what a successful diagnostic result establishes.

32.4 Business reconciliation

32.4.1 PLAIN — in simple words

1. Business reconciliation compares records that should agree under a defined rule. It can reveal missing, duplicated or misclassified work even when storage is structurally healthy.
2. Start with grain and inclusion rules. Comparing all-time accepted orders with today's delivered items creates disagreement even when both lists are correct for their own purpose.
3. A matching total is useful but not sufficient. Two opposite errors can cancel numerically while the wrong customers or products remain affected.

32.4.2 PLAIN — a picture in your head

1. Mira compares each reservation slip with its stock allocation, not just the total number of slips in two piles.
2. Ten slips in each pile can still refer to different reservations.
3. **Where the comparison breaks:** real reconciliation may involve delayed arrivals and approved exceptions. An unmatched record is not automatically fraud, corruption or a completed failure until timing and scope are examined.

32.4.3 PLAIN — a worked example

1. The canonical dataset has four agreed lines, six units and a total of 28,650 paise. Verify each complete `(order_id,line_no)` key and its product, quantity and agreed price.
2. Suppose one copied line is increased by 100 paise and another decreased by 100 paise. The combined total can still match while both line values are wrong.
3. A record-level comparison catches the mismatch. A total-only comparison does not.
4. For reservations, compare each accepted request to its reservation and stock effect. Keep rejected requests and pending delivery intents in their own categories rather than forcing every count to be equal.

32.4.4 PLAIN — what is really happening inside

1. Reconciliation often joins independent representations at explicit identity boundaries. Missing matches, duplicate keys and incompatible values should be categorized separately.
2. Define the allowed timing lag and late-arrival policy. A recently committed order may legitimately await a downstream report refresh under the system's contract.
3. Keep the source of truth explicit for each field. A derived report cannot automatically overrule the original agreement merely because the report was generated later.

32.4.5 TECHNICAL — the engineer's version

1. A reconciliation specification includes population, grain, keys, units, time boundary, expected equations, allowed exceptions and authoritative sources.
2. Use anti-joins or complete-key comparisons to identify missing and extra records before comparing aggregate totals. Chapter 13's join-grain rules apply directly. [S58]
3. Reconciliation findings should be reproducible from captured inputs or a defined snapshot. A live query whose population changes during investigation needs an explicit observation contract. [S66]

32.4.6 WORDS — remember these

1. **Business reconciliation:** compare representations that should agree — an identity- and meaning-aware check of records, totals and state transitions. **Authoritative field source:** the record allowed to decide a disputed value — a defined authority for a particular fact, not necessarily one universal source for every field. **Offsetting errors:** wrong values cancel in an aggregate — discrepancies that preserve a total while corrupting individual records.

32.5 Evidence-preserving repair

32.5.1 PLAIN — in simple words

1. Repair changes evidence. Before changing a damaged or disputed system, preserve enough of the original state to explain what was found and why the proposed correction is justified.
2. A repair should be bounded: identify the records, preconditions, exact changes, expected post-conditions and a safe stopping rule.
3. When the evidence does not establish the correct value, the repair must not invent one. Escalate the unresolved decision or restore from an appropriate verified source.

32.5.2 PLAIN — a picture in your head

1. An archivist photographs a damaged page before repairing it and records which words were restored from a verified duplicate. They do not silently fill illegible words with a plausible story.
2. The repair record distinguishes observed text from reconstructed text and unresolved gaps.
3. **Where the comparison breaks:** copying a live database incorrectly can itself create an inconsistent artifact. Evidence acquisition needs a supported procedure that includes relevant recovery files and metadata.

32.5.3 PLAIN — a worked example

1. In a disposable copy, identify one imported row whose source record and import trace prove that quantity 2 was mistakenly stored as 3. Preserve the source evidence, original row and acquisition identity.
2. Propose a conditional correction targeting the complete key and expected old value. Check that exactly the intended row changes and that related totals reconcile afterward.
3. If the old value no longer matches, stop and review rather than broadening the UPDATE until it affects something. The precondition protected the repair from silently overwriting intervening work.
4. This is a teaching repair pattern, not authorization to change the canonical agreed orders or any live customer record.

32.5.4 PLAIN — what is really happening inside

1. Evidence preservation includes original files, logs, schema, version, configuration and observations relevant to the finding. A checksum records artifact identity but does not independently validate the diagnosis.
2. Some storage-level repairs can discard data to regain structural readability. Such a result must be reported as recovery with known loss, not a complete restoration.
3. Perform proposed changes on an isolated copy first where possible. Validate them before a separately authorized production intervention with a tested recovery path.

32.5.5 TECHNICAL — the engineer's version

1. A repair manifest should contain input hashes, object scope, expected preimages, patch or statements, transaction boundary, postcondition checks, rollback/recovery plan and decision authority. Do not invent reviewer identities or approvals.
2. SQLite's corruption guidance warns about unsafe file operations, journal handling and interference with its locking assumptions. Preserve associated recovery material rather than deleting unfamiliar sidecar files. [S139]
3. Prefer restoring verified data or rebuilding derived structures from authoritative sources when that is the justified remedy. Never present a generic low-level edit as universally safe for an unknown damaged database.

32.5.6 WORDS — remember these

1. **Preimage:** the exact state expected before a repair — captured values or bytes used to verify that a proposed change targets the reviewed input. **Repair manifest:** the record of a bounded correction — evidence, scope, changes, checks and authority supporting an intervention. **Evidence-preserving repair:** fix a problem without concealing its original state — a procedure retaining provenance and disclosing reconstructed or lost information.

32.6 Incident follow-through

32.6.1 PLAIN — in simple words

1. A repair is not complete merely because the first error message disappeared. Check whether the cause remains, whether downstream copies are consistent and whether the same fault can recur.
2. Separate immediate containment, restored service, verified data correction and long-term prevention. They can finish at different times.
3. A useful incident record states facts, uncertainty and actions without assigning unsupported motives or blame.

32.6.2 PLAIN — a picture in your head

1. Replacing a wet ledger page helps today, but a leaking roof can ruin the replacement tomorrow. The repair and the cause need separate attention.
2. Staff also need to know which reports used the damaged page before it was replaced.
3. **Where the comparison breaks:** software causes can be interacting conditions rather than one broken part. An incident may have several contributing factors and no single simple root cause.

32.6.3 PLAIN — a worked example

1. A synthetic report is corrected from the tag-multiplied 51,300 paise to the properly scoped 28,650 paise. The source order lines remain unchanged.
2. Follow-through identifies other reports using the same faulty join, adds a regression fixture with multiple tags, and records which exported results may need correction.
3. The incident closes only after the agreed scope is checked and remaining unreviewed consumers are disclosed. Fixing one dashboard does not prove every downstream spreadsheet was replaced.
4. The original unresolved physical stock discrepancy remains a separate finding, not absorbed into this explained reporting error.

32.6.4 PLAIN — what is really happening inside

1. Data flows create downstream effects: caches, reports, exports, messages and decisions. Lineage helps identify which outputs depended on the disputed input or query.
2. A regression test should reproduce the actual failure condition, not merely repeat the happy path that already passed before the incident.
3. Monitor the corrected invariant over an appropriate period and preserve the old and new definitions. Silent metric-definition changes can make a problem appear solved without changing the underlying behaviour.

32.6.5 TECHNICAL — the engineer's version

1. An incident report should include detection, impact scope, evidence, timeline, containment, correction, validation, remaining uncertainty and prevention actions with owners. Owners are actual designated people or roles, not invented identities.
2. Track which claims were verified: byte integrity, structural validity, record-level reconciliation, downstream regeneration and operational recovery. Do not collapse them into one unqualified PASS.
3. The final book package's own validation follows the same principle: checksums identify files, tests exercise examples and layout checks inspect exports. None alone proves that every sentence is perfect or independently reviewed.

32.6.6 WORDS — remember these

1. **Containment:** limit further harm while investigating — an action that can precede full diagnosis or repair. **Regression fixture:** data and steps reproducing a past failure — a test case intended to prevent recurrence of the specific defect. **Downstream impact:** outputs and decisions depending on disputed data — the propagation scope that must be reviewed after a correction.

32.97 Practice and worked answers

1. **Classify the inflated total.** Joining order lines to multiple tags yields 51,300 instead of 28,650 paise. **Answer:** the fixture demonstrates query-grain multiplication, not necessarily damaged stored prices.
2. **Limit a digest.** A file and its digest are both replaced by an attacker. **Answer:** their agreement does not authenticate the original source; trusted provenance is missing.
3. **Separate checks.** SQLite integrity checking reports no covered structural issue, but a foreign-key check finds an orphan. **Answer:** the checks have different scopes and can legitimately produce those results.
4. **Find offsetting errors.** Two values differ by +100 and −100. **Answer:** the total can match while record-level comparison fails.
5. **Protect a correction.** The reviewed old quantity is 3, but the live row now contains 4. **Answer:** stop the conditional repair and re-review; do not overwrite the intervening change blindly.
6. **Handle an unknown value.** No evidence establishes whether stock should be 9 or 10. **Answer:** retain the uncertainty and obtain a justified observation or decision rather than inventing a repair.
7. **Follow a query fix.** One dashboard is corrected. **Answer:** inspect other consumers, exports and regression coverage before claiming complete downstream correction.

8. **Interpret test success.** A checksum check, integrity check and query test all pass. **Answer:** each supports its bounded claim; together they still do not prove every business fact or unseen system copy is correct.

32.98 Common wrong ideas

1. **Wrong:** every wrong total means corrupted storage. **Right:** query grain, timing and meaning can produce disagreement with intact bytes.
2. **Wrong:** a checksum proves the original data was true. **Right:** it helps compare bytes against a reference.
3. **Wrong:** a checksum can reconstruct a missing page. **Right:** repair needs suitable redundant information or an authoritative source.
4. **Wrong:** one integrity checker covers all constraints. **Right:** structural, referential and business checks differ.
5. **Wrong:** equal totals prove equal records. **Right:** offsetting errors and mismatched identities can hide underneath.
6. **Wrong:** a plausible replacement is a justified repair. **Right:** evidence or explicit authority must support the correction.
7. **Wrong:** deleting recovery sidecars is harmless cleanup. **Right:** they may contain information required for consistency or diagnosis.
8. **Wrong:** removing the symptom closes the incident. **Right:** cause, downstream impact and recurrence need follow-through.

32.99 Chapter summary in 20 lines

1. Disagreement is an observation, not an automatic diagnosis of corruption.
2. Separate physical damage, structural inconsistency and wrong business meaning.
3. Capture exact queries, parameters and observation boundaries.
4. The tag-join fixture inflates a correct 28,650-paise source total to 51,300.
5. Repair the faulty derivation rather than damaging correct source records.
6. Checksums compare covered bytes against an expected reference.
7. Matching digests do not by themselves authenticate origin or truth.
8. Page checksums detect some damage but do not reconstruct lost content.
9. Structural and foreign-key checks answer different questions.
10. Business reconciliation requires explicit grain, scope, keys and units.
11. Matching totals can conceal offsetting record-level errors.
12. Preserve original evidence before proposing a repair.
13. Bind each correction to reviewed preimages and a complete identity.
14. Stop when repair preconditions no longer match.
15. Do not invent missing values to make a checker pass.
16. Preserve recovery files and use supported acquisition procedures.
17. Validate proposed repairs in isolation before separately authorized intervention.
18. Review downstream reports, caches, exports and decisions after correction.
19. Add regression fixtures reproducing the actual failure condition.
20. Close only the verified scope and keep unresolved findings visible.

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

Acknowledgement chain

the sequence of accepted handovers — responses from application, database, operating system and storage layers with distinct meanings.

Read in context: 30.1.6

After-value

the state an accepted update should leave — a logical replacement used by this teaching recovery model.

Read in context: 28.5.6

Alignment

place values at required byte boundaries — padding and layout rules supporting an implementation's representation or access requirements.

Read in context: 27.3.6

Application consistency

related service components agree — correctness of references and workflows beyond one engine's internal file validity.

Read in context: 31.2.6

Archive coverage

the recovery history actually retained and retrievable — verified continuity of required log material over a stated range.

Read in context: 31.3.6

Asynchronous commit

acknowledge before full local log persistence — a configured contract that can trade recent-transaction durability for lower latency.

Read in context: 28.2.6

Atomic visibility

observers do not see an intermediate replacement state — a concurrency property distinct from surviving a crash or power loss.

Read in context: 30.3.6

Authentication

establish a source or actor under a security protocol — a property not supplied merely by an unkeyed matching checksum.

Read in context: 32.2.6

Authoritative field source

the record allowed to decide a disputed value — a defined authority for a particular fact, not necessarily one universal source for every field.

Read in context: 32.4.6

B

Backup

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

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

Read in context: 31.1.6

Backup retention

how long recovery copies remain available — a policy governing restore-point preservation and eventual retirement.

Read in context: 31.6.6

Backup verification

check the retained input against defined expectations — structural, manifest or checksum validation that may precede a full restore.

Read in context: 31.5.6

Buffer pool

the database's in-memory page workspace — a managed cache of pages and metadata used by the storage engine.

Read in context: 27.2.6

Business reconciliation

compare representations that should agree — an identity- and meaning-aware check of records, totals and state transitions.

Read in context: 32.4.6

C

Cache hit

the needed item is already in the named cache — an access resolved at a specified cache layer.

a lookup finds an eligible entry — an event that establishes presence, not automatically freshness or correctness.

Read in context: 27.2.6

Checker scope

the objects and properties actually examined — the boundary limiting what a successful diagnostic result establishes.

Read in context: 32.3.6

Checkpoint

how far processing has reached — a recorded boundary of incorporated source input.

a recorded starting boundary for recovery — coordinated data and metadata progress that limits required redo under an engine's protocol.

recorded safe progress — a recovery marker bound to preserved state and explicitly scoped effects.

Read in context: 28.4.6

Checkpoint pressure

current work caused by reaching that boundary — I/O and related overhead generated by checkpoint activity.

Read in context: 28.4.6

Checksum coverage

the bytes included in an integrity calculation — the exact range or file set a checksum can help verify.

Read in context: 32.2.6

Compaction

merge and reorganize stored runs — background work that installs a new representation and discards only eligible obsolete state.

Read in context: 29.5.6

Compaction backlog

reorganization work waiting to be completed — pending maintenance that can increase read cost, space use or write stalls.

Read in context: 29.6.6

Consistent backup boundary

the state the recovery procedure can validly reconstruct — a snapshot or log-supported acquisition point with matching metadata.

Read in context: 31.2.6

Containment

limit further harm while investigating — an action that can precede full diagnosis or repair.

Read in context: 32.6.6

Correlated failure

one cause affects several copies — a failure pattern that invalidates an assumption of independent redundancy.

Read in context: 30.5.6

Crash trace

the ordered evidence surrounding an interruption — a record of persisted state, surviving log material and recovery decisions.

Read in context: 28.6.6

D

Database page

a unit in the engine's physical layout — a fixed-size storage block containing records, index entries or other engine structures.

Read in context: 27.1.6

Directory durability

preserve the file's naming relationship — persistence of directory metadata needed to find the intended file after failure.

Read in context: 30.3.6

Dirty page

a changed cached page awaiting corresponding writeback — a buffer whose managed contents differ from its lower stored representation.

Read in context: 27.4.6

Disagreement

two observations or expectations do not match — a finding that requires diagnosis rather than automatically proving storage damage.

Read in context: 32.1.6

Downstream impact

outputs and decisions depending on disputed data — the propagation scope that must be reviewed after a correction.

Read in context: 32.6.6

Durability contract

what accepted data is promised to survive — a guarantee defined against specified failures and storage assumptions.

Read in context: 30.1.6

Durability evidence

observations supporting a bounded persistence claim — a test record tied to configuration, fault model and verification procedure.

Read in context: 30.6.6

Durable prefix

the surviving accepted beginning of a log — the valid ordered region known to have reached the required persistence boundary.

Read in context: 28.3.6

E

Eviction

remove a cached item to make room — replacement subject to usage, dirty-state and recovery constraints.

Read in context: 27.2.6

Evidence-preserving repair

fix a problem without concealing its original state — a procedure retaining provenance and disclosing reconstructed or lost information.

Read in context: 32.5.6

F

Failure domain

what one failure can affect together — a process, host, device, availability zone or other explicitly bounded component group.

components that can fail together — a shared physical, software, administrative or operational dependency boundary.

things that can fail together — a boundary such as process, host, power supply, site or administrative authority.

Read in context: 30.5.6

Failure injection

deliberately create a specified fault — a controlled experiment against an authorized disposable system.

Read in context: 30.6.6

Fault model

the failures an experiment assumes or injects — the specific process, operating-system, device or communication conditions being tested.

the failures considered — the boundary within which a mechanism's reasoning and tests are intended to apply.

Read in context: 28.6.6

File synchronization

request persistence of pending file state — an operating-system operation with documented data, metadata and completion semantics.

Read in context: 30.3.6

Filter false positive

a filter suggests a candidate that is absent — extra verification work without a false final membership claim.

Read in context: 29.4.6

Flush

push pending changes toward a specified lower layer — an operation whose persistence guarantee depends on the API and storage contract.

Read in context: 27.4.6

Fragmentation

useful space is distributed inconveniently — a layout condition whose performance effect depends on the engine and access pattern.

Read in context: 27.5.6

Freelist

storage recorded as available for reuse — an engine-managed collection of free pages or blocks.

Read in context: 27.6.6

G

Group commit

one synchronization supports several commits — batching log persistence while meeting each transaction's selected acknowledgement condition.

Read in context: 28.2.6

I

Immutable table file

a sorted file replaced rather than updated in place — an installed run whose ordinary contents do not change after creation.

Read in context: 29.2.6

K

Key dependency

recovery requires particular cryptographic material — an access and availability requirement for decrypting retained data.

Read in context: 31.6.6

L

Log sequence number

a position in a named recovery stream — an engine-defined offset used to identify and compare WAL progress.

Read in context: 28.3.6

Log tail

the final portion of the recorded sequence — a region that can be incomplete or require validity checks after failure.

Read in context: 28.1.6

Logical backup

reconstruct objects from their logical definitions and values — an export such as SQL or another supported logical representation.

Read in context: 31.1.6

Logical inconsistency

records or calculations violate their intended meaning — a mismatch that can exist even when all files are structurally readable.

Read in context: 32.1.6

LSM tree

organize writes through memory and merged sorted runs — a log-structured merge family of storage designs with deferred reorganization.

Read in context: 29.1.6

M

Manifest

the record of which storage files belong to the current state — metadata coordinating installed runs and their lifecycle.

an explicit inventory for a data version — metadata naming the files or entries that belong to a committed state.

Read in context: 29.2.6

Memtable

the searchable workspace for recent writes — an in-memory structure holding entries before or alongside persistent sorted files.

Read in context: 29.2.6

Merge iterator

read several ordered runs as one ordered sequence — an iterator that combines candidates while resolving key and version rules.

Read in context: 29.3.6

N

Negative lookup

a search that establishes absence — a lookup returning no visible matching key after checking relevant structures.

Read in context: 29.4.6

O

Offsetting errors

wrong values cancel in an aggregate — discrepancies that preserve a total while corrupting individual records.

Read in context: 32.4.6

Out-of-line value

a large value stored separately from its main row — payload reached through a reference rather than fully embedded in the ordinary tuple.

Read in context: 27.3.6

P

Page cache

operating-system memory holding file data — a cache between application I/O and lower storage devices.

Read in context: 30.2.6

Page occupancy

how much of a page is usefully filled — the proportion consumed by payload and required structures under a stated layout.

Read in context: 27.1.6

Partial write

fewer bytes were accepted than requested — an I/O result requiring explicit handling rather than an assumption of complete transfer.

Read in context: 30.2.6

Physical backup

preserve an engine's storage representation — a copy requiring the matching consistency, version and recovery procedure.

Read in context: 31.1.6

Physical corruption

stored representation is damaged or invalid — a failure involving bytes or engine structures rather than merely an undesirable business value.

Read in context: 32.1.6

Point-in-time recovery

restore a supported earlier state — replay from a matching base to a chosen engine-supported recovery target.

Read in context: 31.3.6

Power-loss protection

preserve specified device state during power interruption — a hardware capability whose scope and health must be verified.

Read in context: 30.4.6

Preimage

the exact state expected before a repair — captured values or bytes used to verify that a proposed change targets the reviewed input.

Read in context: 32.5.6

R

Read amplification

extra storage work for a logical read — a defined ratio or probe count measured at a specified layer.

reading more bytes than the requested logical data — additional work caused by layout, blocks, metadata or access patterns.

Read in context: 29.4.6

Rebuild

write a reorganized physical representation — maintenance that can improve layout while requiring I/O, temporary capacity and coordination.

Read in context: 27.5.6

Recovery base

the known state from which replay begins — a checkpoint or backup identified together with its matching log boundary.

Read in context: 28.5.6

Recovery dependency

something required to make restored data usable — keys, configuration, software, access and other prerequisites beyond database bytes.

Read in context: 30.5.6

Recovery log

information needed to reconstruct a valid stored state — engine records supporting recovery after incomplete persistence or interruption.

Read in context: 28.1.6

Recovery oracle

independent evidence of the expected result — identities and values used to judge what should remain after the tested failure.

Read in context: 30.6.6

Recovery protocol

the rules for interpreting surviving evidence — a specified procedure connecting log validity, transaction status and stored data.

Read in context: 28.1.6

Recovery target

the intended state or boundary to restore — a chosen time, log position or other engine-supported objective with matching evidence.

Read in context: 28.6.6

Recovery timeline

a named branch of physical recovery history — metadata distinguishing histories after recovery or promotion under the engine's rules.

Read in context: 31.3.6

Redo

reconstruct an accepted change from recovery information — replay that restores data not fully reflected in the main data files before failure.

Read in context: 27.4.6

Redo horizon

the earliest recovery position still needed — the log boundary from which reconstruction must begin for a particular purpose.

Read in context: 28.4.6

Referential check

verify relationships between records — a test that child references resolve according to the schema's key rules.

Read in context: 32.3.6

Regression fixture

data and steps reproducing a past failure — a test case intended to prevent recurrence of the specific defect.

Read in context: 32.6.6

Repair manifest

the record of a bounded correction — evidence, scope, changes, checks and authority supporting an intervention.

Read in context: 32.5.6

Replay position

how far recovery has applied the stream — progress measured under the engine's interpretation of log records.

Read in context: 28.3.6

Replay review

inspect historical pending work before reactivation — reconciliation of restored tasks with effects that may already exist outside the backup boundary.

Read in context: 31.5.6

Restore acceptance

decide whether the recovered state meets its objective — a documented result based on actual recovery and relevant application checks.

Read in context: 31.5.6

Restore isolation

keep the recovered copy from affecting live systems — a controlled environment with outbound effects and production access disabled during validation.

Read in context: 31.2.6

Restore rehearsal

actually exercise the recovery procedure — a controlled test measuring whether the required state and service can be recovered.

recovering and checking a separate destination — evidence about a specific recovery procedure and input.

Read in context: 31.4.6

Restore-time suppression

prevent retired data from re-entering ordinary use — reapplication of approved deletion or access decisions after historical recovery.

Read in context: 31.6.6

Resurrection

an old deleted value becomes visible again — a correctness failure caused by losing the deletion information while older state remains reachable.

Read in context: 29.5.6

Reusable space

allocated storage available for later records — free capacity inside an existing database structure rather than necessarily returned to the filesystem.

Read in context: 27.5.6

Row header

metadata around the visible field values — implementation information needed to locate, interpret or manage a stored row version.

Read in context: 27.3.6

RPO

how much recent accepted work may be lost — a recovery-point requirement defined against a specified incident and acceptance boundary.

Read in context: 31.4.6

RTO

how long required service restoration may take — a recovery-time requirement with explicit start and completion events.

Read in context: 31.4.6

Run overlap

several files cover some of the same key range — a layout property affecting how many candidates a lookup may need to inspect.

Read in context: 29.3.6

Runtime provenance

which software actually produced the result — the recorded engine and interpreter versions and relevant configuration of an experiment.

Read in context: 27.6.6

S

Slot entry

a small reference locating an item within a page — metadata mapping an item identifier to its stored position and length.

Read in context: 27.1.6

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: 29.1.6

Space amplification

physical footprint exceeds live logical data — a ratio whose scope includes the selected files, versions and temporary maintenance state.

Read in context: 29.6.6

Steady state

workload and maintenance remain sustainable together — an operating regime without unbounded growth of deferred work under the stated conditions.

Read in context: 29.6.6

Storage observation

a measured fact about one physical configuration — metadata or instrumentation interpreted with version, scope and timing.

Read in context: 27.6.6

Structural integrity check

inspect engine representation rules — a diagnostic procedure for covered page, tree or table/index invariants.

Read in context: 32.3.6

Success boundary

the event a positive response promises has occurred — the precise acceptance or persistence condition represented to the caller.

Read in context: 30.1.6

T

Tombstone

a newer marker hiding an older value — a deletion record retained until older visible copies and snapshot requirements permit removal.

Read in context: 29.5.6

Torn page

only part of a page update survives — a mixed old/new physical page caused by an interrupted multi-part write.

Read in context: 30.4.6

Trusted expected digest

an integrity reference with justified provenance — a digest bound to an intended artifact through a trusted source or process.

Read in context: 32.2.6

U

Undo information

evidence for reversing unaccepted changes — recovery data used by designs that require explicit restoration of earlier state.

Read in context: 28.5.6

User-space buffer

data waiting inside the program or library — memory used to batch output before passing it to the operating system.

Read in context: 30.2.6

V

Visible sequence limit

the newest version a snapshot may use — a model or engine boundary excluding later updates from that reader.

Read in context: 29.3.6

Volatile write cache

acknowledged data still depends on power — intermediate storage that can lose pending writes when its power disappears.

Read in context: 30.4.6

W

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: 29.1.6

Write-ahead ordering

recovery evidence must precede dependent data persistence — the ordering constraint linking log durability and data-page writes.

Read in context: 28.2.6

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