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 D · Chapters 21–26 · First Edition
Correct Changes

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 D

Correct Changes

Keep related changes consistent when requests overlap, fail or arrive more than once.

This volume contains Chapters 21–26 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 D

Correct Changes

Keep related changes consistent when requests overlap, fail or arrive more than once.

21

CHAPTER

Transactions: One Change, All or Nothing

21.0 What this chapter gives you

1. A useful business action can require several database changes. This chapter explains how to give those changes one commit boundary without pretending that a database transaction controls the whole outside world.
2. You will distinguish atomicity, consistency, isolation and durability; recover from a failed step; use savepoints; and identify effects that remain outside rollback.
3. The stock and request examples introduced here are separate synthetic exercises. They do not explain the earlier unresolved expected-stock-10/observed-stock-9 discrepancy or alter the canonical agreed orders.

21.1 The boundary of a change

21.1.1 PLAIN — in simple words

1. Suppose accepting a reservation must both reduce available stock and record which request received it. Saving only one of those changes leaves an incomplete account of the operation.
2. A transaction gives a selected collection of database work one all-or-nothing boundary. The application decides which related operations belong together and whether their results satisfy its rules.
3. Putting statements inside BEGIN and COMMIT does not make a wrong rule right. A transaction can reliably commit the wrong amount if the application supplied the wrong calculation.

21.1.2 PLAIN — a picture in your head

1. Mira prepares a reservation slip and the corresponding stock adjustment inside one envelope. The envelope is either accepted as a complete unit or rejected before becoming the official record.
2. The envelope boundary makes it harder to accept the slip while forgetting its stock change.
3. **Where the comparison breaks:** a transaction is not literally a hidden envelope containing every effect. Network calls, printed receipts and handed-over goods are outside its ordinary database boundary. Visibility also depends on the engine's isolation rules.

21.1.3 PLAIN — a worked example

1. In a new teaching inventory, product TX-PEN has five available units. Request TX-R1 asks for three. A successful operation should leave two units and one accepted request record for three units.
2. The incomplete outcomes are easy to name: stock becomes two with no request record, or the request is accepted while stock remains five. Either could mislead later reconciliation.

3. Group the guarded stock change and request insertion into one owned transaction. If either step fails, roll back both transactional changes. If both succeed and all required checks pass, commit.
4. This models reservation records, not physical delivery, payment, authentication or a production checkout.

21.1.4 PLAIN — what is really happening inside

1. The database tracks the transaction's changes and coordinates their visibility and persistence under its documented mechanisms.
2. The application must inspect outcomes inside the boundary. An UPDATE affecting zero rows is often a successful SQL statement but an unsuccessful business precondition.
3. Keep the transaction no larger than necessary for the invariant. Waiting for a human to answer a form while holding a transaction open can retain locks, snapshots and resources unnecessarily.

21.1.5 TECHNICAL — the engineer's version

1. Define a transaction's read set, write set, invariants and completion condition. The transaction boundary should include the reads and decisions on which its protected writes depend, subject to a concurrency-control strategy. [S117]
2. State who owns BEGIN, COMMIT and ROLLBACK. A helper that unexpectedly commits its caller's transaction can split a larger intended atomic operation.
3. In SQLite, explicit transaction forms and driver transaction settings interact. The educational code uses a deliberate connection configuration and owns one transaction rather than relying on an unstated autocommit assumption. [S25] [S86]

21.1.6 WORDS — remember these

1. **Transaction boundary:** the work accepted or rejected together — the scope of a database transaction's reads, writes and completion decision. **Business precondition:** what must be true before an operation is accepted — an application rule that may require checking affected rows or queried state. **Transaction owner:** the code responsible for completion — the layer that controls commit and rollback for a defined unit of work.

21.2 Commit and rollback

21.2.1 PLAIN — in simple words

1. Commit asks the database to accept the transaction's changes. Rollback abandons changes still inside an uncommitted transaction.
2. A statement succeeding does not mean the whole transaction committed. A later constraint check, communication failure or other error can change the outcome.
3. After a commit acknowledgement is lost, the client may not know whether the database committed. "I did not receive success" is not the same as "nothing happened."

21.2.2 PLAIN — a picture in your head

1. A clerk sends an envelope for registration. Before registration, it can be withdrawn. After registration, tearing up the clerk's local copy does not erase the official record.
2. If the return receipt is lost, the clerk needs to check the registration identity rather than register the same operation again under a new name.

3. **Where the comparison breaks:** database completion involves protocol and persistence guarantees, not a human registrar. The important analogy is the difference between an uncommitted change and an unknown acknowledgement outcome.

21.2.3 PLAIN — a worked example

```
db.execute("BEGIN IMMEDIATE")
try:
    changed = db.execute(
        "UPDATE tx_stock SET available = available - ? "
        "WHERE product_id = ? AND available >= ?",
        (3, "TX-PEN", 3),
    ).rowcount
    if changed != 1:
        raise ValueError("reservation precondition not satisfied")
    db.execute(
        "INSERT INTO tx_requests(request_id, product_id, quantity) "
        "VALUES (?, ?, ?)",
        ("TX-R1", "TX-PEN", 3),
    )
    db.execute("COMMIT")
except BaseException:
    if db.in_transaction:
        db.execute("ROLLBACK")
    raise
```

1. This is a scoped SQLite teaching pattern for an owned connection and separately created exercise tables. It does not implement idempotent replay; a duplicate request key can reject the insertion and roll back the stock change.
2. Inject an ordinary Python exception after the UPDATE but before the INSERT or COMMIT. After rollback, verify both stock and requests, not just the raised error.
3. This injection tests application-level rollback. It does not simulate device power loss or a lost network acknowledgement.

21.2.4 PLAIN — what is really happening inside

1. Transaction state can differ after different failures. Some errors abort one statement; others invalidate more work or leave a transaction needing explicit recovery. Read the engine and driver contract.
2. Deferred constraints can reject COMMIT after earlier statements appeared successful. The application must handle that failure rather than return success before commit completes.
3. Once a change is committed, a correction is normally another transaction. Rollback is not a time machine for previously committed business actions.

21.2.5 TECHNICAL — the engineer's version

1. SQLite documents that COMMIT can fail, including due to busy conditions, and that transaction state after errors needs deliberate handling. PostgreSQL has its own failed-transaction and savepoint behaviour. Do not generalise one engine's exception handling to all engines. [S25] [S117]
2. A communication exception during commit can be an **indeterminate outcome** from the client's perspective. Use stable request identity and a reconciliation path; Chapter 26 develops stored idempotency outcomes.
3. Cleanup should not disguise the original failure. Production libraries need careful exception handling if rollback itself fails, and must not return a possibly damaged or still-active connection to a pool without a defined recovery policy.

21.2.6 WORDS — remember these

1. **Commit:** accept a transaction's database changes — the completion operation whose acknowledgement has engine- and configuration-specific guarantees. **Rollback:** abandon uncommitted transactional work — restoration of the transaction's database effects under its supported semantics. **Indeterminate outcome:** the caller cannot yet tell whether it completed — uncertainty commonly caused by losing communication around a commit or external effect.

21.3 Atomicity and consistency

21.3.1 PLAIN — in simple words

1. Atomicity concerns whether the selected changes take effect together. Consistency concerns the rules that valid states must satisfy.
2. The database enforces the constraints you actually declared and the mechanisms its transaction model provides. It does not infer every intended business rule from a table's name.
3. A transaction that subtracts four units for a three-unit request can be perfectly atomic and still wrong. Both changes agree only if the application checks the correct relationship.

21.3.2 PLAIN — a picture in your head

1. An envelope can contain a complete but incorrect invoice. Keeping every page together prevents missing pages; it does not correct the arithmetic.
2. A reviewer still needs rules about quantities, prices and the meaning of the document.
3. **Where the comparison breaks:** some consistency rules are mechanically enforceable as constraints, while others need application logic or coordination. A human reviewer is not an automatic feature of every transaction.

21.3.3 PLAIN — a worked example

1. Return to the separate capacity example from Chapter 11: capacity is ten, and two allocations are seven and six. Each allocation is positive, so a positive-value CHECK can accept both while the sum reaches thirteen.
2. Wrapping both INSERTs in one transaction makes their acceptance all-or-nothing. It does not add the missing sum-within-capacity rule.
3. A correct design can represent a guarded available quantity, coordinate through a shared record, or use another explicitly justified invariant strategy. The chosen strategy must also work when requests overlap.
4. The earlier counterexample remains evidence of absent protection in that lab; it is not retroactively repaired by this explanation.

21.3.4 PLAIN — what is really happening inside

1. A valid-state rule can concern one value, one row, several rows or an external fact. Different enforcement mechanisms cover different scopes.
2. Transactional atomicity helps keep a multi-row rule intact once the application has a correct algorithm and suitable isolation or locking. It does not supply that algorithm on its own.
3. Verification should include both valid and invalid transitions. Tests that only confirm successful insertion never establish that forbidden states are rejected.

21.3.5 TECHNICAL — the engineer's version

1. In ACID terminology, atomicity, consistency, isolation and durability describe distinct aspects of transactional behaviour. Consistency depends on declared constraints and correctly implemented invariants; it is not a general truth detector.
2. A transaction can preserve structural constraints while violating a business policy omitted from them. Chapter 11's cross-row and historical-price counterexamples demonstrate this distinction with concrete accepted states.
3. Prove the isolated transition preserves the invariant, then analyse overlapping transitions under the chosen concurrency model. A proof that assumes serial execution is incomplete when the deployment permits non-serialisable interactions.

21.3.6 WORDS — remember these

1. **Atomicity:** the selected changes take effect together — all-or-nothing transactional acceptance of the covered database work. **Invariant:** a rule every accepted state must preserve — a condition maintained by constraints and operation protocols across permitted transitions. **Consistency:** accepted states obey the specified rules — validity relative to actual constraints and invariants, not an assurance that every stored claim is true.

21.4 Isolation and durability

21.4.1 PLAIN — in simple words

1. Isolation controls what overlapping transactions can observe and how their actions interact. Durability concerns what an acknowledged change survives.
2. These properties answer different questions. A change may be protected from partial visibility yet configured with weaker persistence acknowledgements. A durable wrong answer remains wrong.
3. State the failure being discussed: a process crash, an operating-system crash, power loss, a destroyed device or a lost region are not equivalent events.

21.4.2 PLAIN — a picture in your head

1. Isolation is about several clerks working without seeing an unacceptable mixture of one another's unfinished edits. Durability is about which official records remain after a failure.
2. A fireproof cabinet does not decide which edits clerks may combine, and a careful editing protocol does not make paper fireproof.
3. **Where the comparison breaks:** actual guarantees depend on software, configuration, storage and replication protocols. Neither "transactional" nor "saved" specifies all failure domains by itself.

21.4.3 PLAIN — a worked example

1. A reader totals two fields while another transaction changes them together. The isolation question is whether the reader can observe an unacceptable mixture of before and after values under its snapshot rules.
2. A server acknowledges that the change committed, then its process exits. The durability question is whether a restarted engine recovers the acknowledged state under its configured persistence path.

3. A memory-only teaching database can demonstrate transactional visibility and rollback while losing all data when the process ends. It is not a durable storage deployment merely because COMMIT works.
4. Chapters 23 and 30 examine the two questions separately rather than hiding them behind one ACID label.

21.4.4 PLAIN — what is really happening inside

1. Isolation can use locks, versions, conflict detection and other coordination. A snapshot is a visibility rule, not necessarily a physical copy of the entire database.
2. Durability usually involves recovery records, ordered persistence requests and assumptions about the operating system and device. Replication can extend the failure boundary only under its actual acknowledgement rules.
3. Testing a Python exception proves less than testing an abrupt process crash, which proves less than a controlled power-loss experiment. Name the experiment actually performed.

21.4.5 TECHNICAL — the engineer's version

1. PostgreSQL's isolation levels and SQLite's writer/reader behaviour are product-specific contracts. Do not infer a universal anomaly table from a generic level name. [S66] [S54]
2. SQLite's atomic-commit documentation explicitly discusses filesystem and device assumptions. The educational in-memory examples do not exercise those assumptions. [S53]
3. A durability claim should identify acknowledgement point, configured persistence mode, recovery procedure and covered failure domain. Later replication acknowledgements must be analysed with the same precision.

21.4.6 WORDS — remember these

1. **Isolation:** rules for overlapping work — guarantees governing visibility and interaction among concurrent transactions. **Durability:** acknowledged work survives specified failures — persistence under a named configuration, recovery model and failure boundary. **Failure domain:** what one failure can affect together — a process, host, device, availability zone or other explicitly bounded component group.

21.5 Savepoints

21.5.1 PLAIN — in simple words

1. A savepoint marks a position inside a transaction. Rolling back to it can discard later transactional changes while retaining earlier uncommitted work.
2. It is not necessarily an independent committed transaction. Releasing an inner savepoint does not make its changes immune to a later rollback of the outer transaction.
3. Savepoints help recover from optional or tentative steps, but they should not be used to silently accept an incomplete business operation whose steps were all required.

21.5.2 PLAIN — a picture in your head

1. While drafting an order, Mira bookmarks a version before trying an optional annotation. She can return to that bookmark without discarding the whole draft.
2. The draft still is not an accepted order merely because the annotation's bookmark was removed.

3. **Where the comparison breaks:** savepoints have precise engine-specific stack and error-recovery semantics. They do not restore external files, user-interface state or every nontransactional counter.

21.5.3 PLAIN — a worked example

1. In a disposable notes table, begin a transaction and insert a required note A. Create savepoint `optional_note`, then insert tentative note B.

```
BEGIN;
INSERT INTO tx_notes(note_id, body) VALUES ('A', 'required');
SAVEPOINT optional_note;
INSERT INTO tx_notes(note_id, body) VALUES ('B', 'tentative');
ROLLBACK TO optional_note;
RELEASE optional_note;
COMMIT;
```

2. The committed result contains A but not B. The exercise explicitly defines B as optional; this is not permission to drop a failed stock adjustment from a required reservation.
3. Repeat with a full ROLLBACK instead of COMMIT at the end. Then neither A nor B remains from the transaction, even after RELEASE.

21.5.4 PLAIN — what is really happening inside

1. Savepoints form nested recovery boundaries within the transaction's supported semantics. Rolling back to one abandons subsequent work and affects later savepoints according to the engine's rules.
2. A released inner boundary no longer provides that local recovery point, but its changes remain part of the outer transaction until outer completion.
3. SQLite permits an outermost SAVEPOINT to start a transaction, so releasing that outermost boundary can commit. Distinguish this case from releasing an inner savepoint inside an existing transaction. [S127]

21.5.5 TECHNICAL — the engineer's version

1. PostgreSQL SAVEPOINT and ROLLBACK TO support partial recovery inside a transaction, including recovery from eligible failed suboperations. The outer transaction remains the final acceptance boundary. [S121]
2. SQLite RELEASE has stack semantics; an inner release is not a durable commit, while release of the outermost transaction savepoint has a different effect. Code must know which boundary it owns. [S127]
3. A savepoint strategy should document which failures are optional, which invalidate the whole business action and which leave the connection unusable. Catching every exception and continuing is not a valid general recovery policy.

21.5.6 WORDS — remember these

1. **Savepoint:** a local return point inside transaction work — a named boundary to which later transactional changes can be rolled back. **Partial rollback:** discard only a selected later portion — rollback to a savepoint rather than abandonment of the entire outer transaction. **Outer transaction:** the enclosing acceptance boundary — the transaction whose final commit or rollback governs changes retained from inner savepoint work.

21.6 External effects outside the transaction

21.6.1 PLAIN — in simple words

1. Sending an email, charging an external payment service, printing a document or handing over goods is not normally undone by rolling back a local database transaction.
2. Performing the external effect before commit can leave an effect with no accepted database record. Performing it after commit can leave accepted work with an effect that was never completed.
3. The solution is an explicit cross-boundary protocol, not a claim that one local transaction magically covers everything. Chapter 26 introduces durable delivery intent; Chapter 38 covers compensation.

21.6.2 PLAIN — a picture in your head

1. Dev sends a customer a reservation message, then discovers that the database transaction failed. The message is already in the customer's inbox.
2. Erasing the draft on Dev's desk does not recall that message.
3. **Where the comparison breaks:** some external providers support idempotency keys or transactional integration, but those are additional contracts. Their existence must be verified for the exact operation, not assumed from the word API.

21.6.3 PLAIN — a worked example

1. Schedule A: update stock; send confirmation; request insertion fails; roll back. The customer saw a confirmation for a reservation that did not commit.
2. Schedule B: update stock and request; commit; process crashes before sending. The reservation exists but no confirmation was sent.
3. An outbox stores the intention to send alongside the business changes in one database transaction. A later worker retries delivery from that committed intention.
4. This closes the local "committed work with no recorded intent" gap. It does not by itself prevent duplicate external delivery when a worker sends successfully and crashes before recording acknowledgement. [S125]

21.6.4 PLAIN — what is really happening inside

1. Each independent system has its own acceptance point. A local database cannot roll back a remote side effect merely because the application uses a try/except block.
2. Stable operation identity helps reconcile uncertain outcomes. A correction or compensation is a new action with its own evidence, not a claim that the first action never happened.
3. Even within PostgreSQL, sequence allocation has special non-rollback behaviour. Gaps in generated values therefore do not prove that a business record was deleted or lost. [S122]

21.6.5 TECHNICAL — the engineer's version

1. Distinguish local transactional atomicity from distributed atomic commit and from a saga of compensating actions. They provide different failure semantics.
2. An outbox atomically binds business state to durable delivery intent when both writes share the same effective database transaction. Delivery, ordering and consumer deduplication remain separate obligations. [S125]

3. The educational tests inject failures at named points and inspect database state. They do not send live messages, perform payments or prove end-to-end exactly-once effects.

21.6.6 WORDS — remember these

1. **External effect:** something changes beyond the local transaction — an action in another system or the physical world not automatically reversed by database rollback. **Outbox:** committed work carries a delivery instruction — a transactional record of an event or message to be delivered by a separate process. **Compensation:** a new action addresses an earlier effect — an explicit corrective operation, not erasure of the original history.

21.97 Practice and worked answers

1. **Question:** Stock decreases but request insertion fails before commit. What should the owned transaction do? **Answer:** Roll back both covered changes and verify that neither partial outcome remains.
2. **Question:** An UPDATE affects zero rows without a SQL error. Is the reservation accepted? **Answer:** Not under the specified one-row precondition. The application must inspect the affected-row result.
3. **Question:** Can a transaction atomically commit an allocation total above capacity? **Answer:** Yes, if the capacity invariant is not enforced by the schema or operation protocol. Atomicity does not invent the rule.
4. **Question:** Is a lost commit acknowledgement proof of rollback? **Answer:** No. The outcome may be indeterminate to the caller; reconcile using stable operation identity.
5. **Question:** Does releasing an inner savepoint protect its changes from outer rollback? **Answer:** No. Those changes remain inside the outer transaction.
6. **Question:** Does an in-memory COMMIT establish power-loss durability? **Answer:** No. The storage and failure boundary were not exercised.
7. **Question:** Will database rollback recall an email already sent? **Answer:** Not ordinarily. The external effect requires its own protocol or corrective action.
8. **Question:** Why do sequence gaps not prove missing orders? **Answer:** Sequence allocation can survive transaction rollback and has other allocation behaviours. Business-event completeness must be checked against business records, not inferred from gapless numbering.

21.98 Common wrong ideas

1. **Wrong:** successful statements mean the transaction committed. **Right:** final completion and deferred checks still matter.
2. **Wrong:** zero affected rows is always a database error. **Right:** it can be a successful statement whose business precondition failed.
3. **Wrong:** ACID means every business rule is automatically known. **Right:** omitted rules remain omitted.
4. **Wrong:** rollback reverses earlier committed work. **Right:** a later correction is another transaction.
5. **Wrong:** releasing any savepoint commits independently. **Right:** inner and outer boundaries differ.
6. **Wrong:** saved in memory means durable on a device. **Right:** persistence guarantees need a named failure model.

7. **Wrong:** an external API call belongs to the local transaction because it is inside the same function. **Right:** code scope is not a shared commit protocol.
8. **Wrong:** an outbox alone guarantees one external effect. **Right:** delivery ambiguity and consumer behaviour still need handling.

21.99 Chapter summary in 20 lines

1. A transaction groups selected database work under one completion boundary.
2. The application chooses the business action and its invariants.
3. Required reads and decisions need a concurrency strategy as well as grouped writes.
4. One layer should clearly own commit and rollback.
5. A zero-row update can signal a failed business precondition.
6. Commit can fail after earlier statements succeeded.
7. A lost acknowledgement can leave the caller uncertain about completion.
8. Rollback abandons uncommitted work, not all past actions.
9. Atomicity does not correct wrong arithmetic or omitted rules.
10. Consistency is relative to actual enforced constraints and protocols.
11. Isolation governs overlapping observations and changes.
12. Durability concerns survival under specified failures.
13. In-memory tests do not establish persistent-device durability.
14. Savepoints provide local recovery inside an enclosing transaction.
15. Releasing an inner savepoint is not independent final acceptance.
16. Optional-step recovery must not silently drop required business work.
17. External effects usually remain outside local rollback.
18. An outbox binds committed business changes to recorded delivery intent.
19. Sequence gaps are not proof of missing business events.
20. Every guarantee needs a clear boundary and evidence matching that boundary.

22

CHAPTER

Two People Change the Same Record

22.0 What this chapter gives you

1. A program can be correct when run alone and wrong when another program changes the same facts between its steps. This chapter teaches you to write down that overlap instead of explaining it away as a mysterious race.
2. You will trace lost updates, use conditional writes, distinguish a uniqueness rule from an absence check, and design a bounded stock-reservation transaction. The examples concern database records, not an assurance that the physical shelf matches them.
3. Every inventory schedule in this chapter is new synthetic teaching data. None establishes the cause of the earlier unexplained stock discrepancy at Mira's Corner.

22.1 Interleaving actions

22.1.1 PLAIN — in simple words

1. Two people do not need to press a button at exactly the same instant to interfere with one another. It is enough for one person's work to happen between another person's reading and writing.
2. A request that appears to be one action on a screen may contain several separate steps: read a value, decide what to do, calculate a result, write it, and confirm completion.
3. The important question is which of those steps may overlap and what the database promises about the overlap. Counting buttons or adding a loading spinner does not answer it.

22.1.2 PLAIN — a picture in your head

1. Mira reads the number on a stock card, walks away to calculate, and comes back to write her answer. While she is away, Dev reads the same card and begins his own calculation.
2. Each person remembers a different moment, even though both later write on the same card. The card cannot reconstruct the missing conversation by itself.
3. **Where the comparison breaks:** a database may block, reject or reorder operations according to its concurrency controls. A handwritten schedule describes a proposed sequence; it is not evidence that every database permits that sequence.

22.1.3 PLAIN — a worked example

1. Name the requests A and B and split each into read, calculate and write. One possible schedule is A-read, B-read, A-calculate, A-write, B-calculate, B-write.
2. Another is A-read, A-calculate, A-write, B-read, B-calculate, B-write. The second completes A before B reads; it may therefore give B different input.

3. Before claiming an anomaly, state the initial records, each statement, the transaction boundaries, the isolation level, the connection arrangement and the observed results. A drawing with none of these details is a hypothesis to investigate.
4. In a pure model we can deliberately enumerate schedules. In an engine experiment we must record which actions actually succeeded, waited or failed.

22.1.4 PLAIN — what is really happening inside

1. The server and operating system schedule work independently of the order in which people remember clicking. Network delays can also change arrival and response order.
2. A connection may hold one transaction while another connection executes its own. A pool is not a guarantee that related statements use the same connection or transaction unless the application arranges that explicitly.
3. Correctness reasoning follows dependencies: this write used that earlier observation; this decision required that predicate to remain protected. Wall-clock timestamps alone often cannot establish all such relationships.

22.1.5 TECHNICAL — the engineer's version

1. An interleaving is an ordering of elementary operations consistent with the order required inside each participating process or transaction. A schedule model should state which operations are atomic in that model.
2. Engine isolation constrains the legal schedules and visible versions. PostgreSQL 17 and SQLite do not offer identical execution models; a PostgreSQL snapshot example should not be advertised as a reproduced SQLite anomaly. [S66] [S54]
3. Use deterministic barriers or explicit two-connection coordination for a controlled experiment where feasible. A sleep can encourage overlap but does not prove the intended ordering. Retain a step log and transaction outcome for every participant.

22.1.6 WORDS — remember these

1. **Interleaving:** the steps of different requests mixed together — an operation ordering that preserves each participant's required local order. **Race condition:** an answer that depends on an unsafe overlap — behaviour whose correctness changes with the relative execution order of competing operations. **Schedule evidence:** the recorded sequence that actually ran — observations of operations, waits, errors and outcomes under a stated concurrency configuration.

22.2 Lost updates

22.2.1 PLAIN — in simple words

1. A lost update happens when a later write accidentally replaces the effect of another change. Both people may receive a success message even though the final record represents only part of their work.
2. A common cause is calculating a replacement value from an old read. The replacement says "set stock to six," not "subtract the four units this request reserved."
3. The database cannot always infer the intention behind a supplied number. Six might be a deliberate correction, an obsolete calculation or an entirely wrong input. The application must express the intended operation and its conditions.

22.2.2 PLAIN — a picture in your head

1. Two editors download the same paragraph. One corrects a spelling error; the other adds a sentence. Uploading the second whole paragraph can restore the spelling error by replacing the first editor's version.
2. The second upload is not necessarily larger or later in meaning. It is merely the last replacement accepted under that editing protocol.
3. **Where the comparison breaks:** text-merging tools sometimes combine edits automatically. A stock counter needs a precisely defined numeric rule; an automatic text-style merge is not an inventory policy.

22.2.3 PLAIN — a worked example

1. A new model begins with RACE-STOCK equal to 10. Request A wants 3 units and request B wants 4. Both read 10 before either writes.
2. A calculates 10 – 3 = 7. B independently calculates 10 – 4 = 6. A writes 7, then B writes 6.
3. The final value is 6, but accepting both reservations should have left 10 – 3 – 4 = 3. A's three-unit subtraction was overwritten. The error is not that subtraction was calculated incorrectly; it was calculated from an observation that was no longer safe to replace.
4. This deliberately unsafe read-then-replace model does not establish that a particular transaction configuration permits both writes. Some engines or isolation levels would reject or serialize relevant operations instead.

22.2.4 PLAIN — what is really happening inside

1. An UPDATE can faithfully store the literal value supplied to it. Atomic execution of that one UPDATE does not retroactively protect a separate earlier SELECT.
2. Beginning a transaction around the SELECT and UPDATE is important but does not, by itself, specify sufficient isolation for every read-dependent rule. The chosen engine and isolation level determine the remaining risks.
3. Different intentions require different controls. An increment can often be expressed as a relative update. Replacing a person's edited document may require a version check and a conflict response rather than silently adding or merging values.

22.2.5 TECHNICAL — the engineer's version

1. Distinguish `SET available = :old_value - :quantity` from `SET available = available - :quantity`. The first binds a client-derived replacement; the second asks the engine to derive the new value from the row it updates.
2. In PostgreSQL 17 Read Committed, a competing updater may wait and then re-evaluate its WHERE condition against an updated row. This supports useful single-row guarded-update patterns, not an unrestricted proof for arbitrarily complex cross-row rules. [S66]
3. A version-based compare-and-set is another strategy: require the expected revision and advance it with the write. Chapter 25 develops that approach and its limits, including deletion and recreation of an identity.

22.2.6 WORDS — remember these

1. **Lost update:** one accepted change is overwritten by another — a write anomaly in which a later write fails to preserve an effect that should remain represented. **Relative update:** describe the

change rather than an old replacement — an expression such as `value = value - amount` evaluated by the database. **Stale read:** an observation that no longer describes the relevant current state — a previously read version that may be unsafe as an unconditional write basis.

22.3 Conditional writes

22.3.1 PLAIN — in simple words

1. A conditional write asks the database to make a change only while a stated condition is satisfied. For stock, the condition may be "this product has at least the requested quantity."
2. The condition and the change must belong to the protected operation. Checking stock in one request and changing it later without protection leaves the gap open.
3. A statement affecting no rows needs interpretation. It may mean insufficient stock, a missing product, a stale version or a forbidden scope, depending on the statement and application contract. It is not automatically a database malfunction.

22.3.2 PLAIN — a picture in your head

1. A ticket dispenser checks whether tickets remain and releases one through the same controlled mechanism. It does not hand you a note saying that a ticket existed several seconds ago.
2. Combining the check with the action prevents another buyer from treating the same old observation as a separate promise.
3. **Where the comparison breaks:** a database condition concerns stored records. The record may still disagree with damaged, missing or uncounted physical goods. Concurrency control does not replace stocktaking.

22.3.3 PLAIN — a worked example

1. In a separate RES-PEN exercise there are 5 available units. A asks for 3 and B asks for 3. Both cannot be accepted while available stock stays non-negative.
2. The guarded operation subtracts 3 only when at least 3 remain. One successful update leaves 2. A later evaluated condition sees that 2 is not enough and affects zero rows.

```
UPDATE reservation_stock
SET available = available - ?
WHERE product_id = ? AND available >= ?;
```

3. Bind `(3, 'RES-PEN', 3)` after validating that quantity is a positive bounded integer. Accept the stock step only if exactly one row was affected under the helper's stated schema and driver behaviour.
4. This statement alone does not create the request record, protect an external payment or distinguish a replay. Those responsibilities require the surrounding transaction and request protocol.

22.3.4 PLAIN — what is really happening inside

1. The engine coordinates the qualifying row and its update. Other relevant operations may wait or receive a conflict instead of both acting on the same old value.
2. Keep the product's complete identity in the predicate. Once branches or tenants exist, a product code alone may match the wrong scope or multiple rows.
3. Separate input validation from database enforcement. Reject zero, negative, boolean-like or excessively large quantities according to the API's explicit type contract; retain database constraints as another boundary.

22.3.5 TECHNICAL — the engineer's version

1. A guarded update protects only the predicate and resources covered by the engine's concurrency semantics. It does not automatically enforce a total across other rows or across databases. [S66]
2. With SQLite, serialize the owned write transaction deliberately, for example using `BEGIN IMMEDIATE` in these bounded exercises. Handle contention and inspect transaction state rather than assuming simultaneous independent writers. [S25] [S54]
3. Record the row-count contract for the driver and statement. For the supplied SQLite UPDATE examples, the code checks `cursor.rowcount`; more complex statements, triggers and different drivers require their own documented interpretation. [S86]

22.3.6 WORDS — remember these

1. **Conditional write:** change a record only when its requirement still holds — a write guarded by a predicate evaluated under the database's concurrency rules. **Affected-row check:** verify how many target records changed — inspection of the statement result against an expected cardinality. **Complete scope:** all identity dimensions needed to select the intended record — the tenant, branch and local key components required by the data model.

22.4 Uniqueness under competition

22.4.1 PLAIN — in simple words

1. Asking whether a key exists and receiving "no" is an observation, not a reservation of that key. Another request may receive the same answer before either inserts.
2. A uniqueness constraint gives the database an enforceable rule about which keys may coexist. The application must still decide how to handle the losing request.
3. The meaning of the key matters. One key per event, one key per customer and one key per retryable request describe different things. A constraint cannot repair a confused definition of identity.

22.4.2 PLAIN — a picture in your head

1. Two visitors see an empty seat and each announce that it is theirs. Looking at the seat did not allocate it.
2. A controlled reservation desk can accept one seat assignment and reject the conflicting assignment. The desk must also know whether two messages came from one visitor retrying or from two different visitors.
3. **Where the comparison breaks:** some database uniqueness rules treat missing values differently from ordinary values. A nullable field is not automatically the same as a required reservation identifier.

22.4.3 PLAIN — a worked example

1. A and B both execute a lookup for `(branch='BR-A', receipt_no=17)` and see no row. Both then try to insert that pair.
2. With the required composite unique key, both cannot commit conflicting rows with that key. One may succeed while the other waits, fails or takes an explicitly programmed conflict path.
3. `(BR-B,17)` is different and may be valid. A global uniqueness constraint on receipt number alone would reject legitimate records from another branch.

4. Receiving a uniqueness error does not prove that the competing record contains the same request payload. Before returning an existing outcome as a replay, check the intended request identity and its recorded meaning.

22.4.4 PLAIN — what is really happening inside

1. The engine's constraint mechanism participates in concurrency control; the application does not implement uniqueness merely by querying first.
2. A pre-check can improve a user message, but it cannot replace the final enforcement point. Handle a conflict even when the pre-check passed.
3. Blindly retrying a permanently conflicting insert wastes resources. A new attempt with the same impossible key and unchanged policy is likely to encounter the same problem again.

22.4.5 TECHNICAL — the engineer's version

1. Define the unique constraint on the complete non-null identity required by the model. Consult the engine's treatment of NULL, deferred constraints and conflict clauses rather than assuming universal behaviour. [S74] [S61] [S73]
2. PostgreSQL's serialization-retry guidance distinguishes transient concurrency situations from persistent uniqueness failures. A uniqueness error is not a universal instruction to retry automatically. [S120]
3. An upsert is a chosen write policy, not proof of semantic equivalence. Replacing an existing payload on conflict may destroy the very evidence needed to identify incompatible uses of a request key.

22.4.6 WORDS — remember these

1. **Check-then-insert race:** two requests act on the same observed absence — an unsafe gap between an existence query and an unprotected insertion decision. **Composite uniqueness:** a combination must be distinct — a constraint enforcing uniqueness over a tuple of columns. **Conflict policy:** what to do when a rule prevents a write — an explicit decision to reject, reconcile, return an equivalent result or apply a defined update.

22.5 Read-modify-write traps

22.5.1 PLAIN — in simple words

1. Some rules depend on more than the row being changed. A branch may allow at most ten active reservations across many request rows. Each individual row can be valid while their total is too large.
2. Protecting one writer's own row does not necessarily protect the shared total. Two writers can change different rows after reading the same acceptable total.
3. The remedy must match the invariant. It may use a shared guarded capacity record, an appropriate locking protocol or serializable transactions with correct retry handling. Choosing a familiar keyword without tracing the rule is not a design.

22.5.2 PLAIN — a picture in your head

1. Two staff members each have a locked drawer for booking forms, but both sell space in the same ten-seat room. Locking the drawers does not coordinate the room's remaining capacity.
2. The scarce thing is shared, even though the pieces of paper are separate.

3. **Where the comparison breaks:** database locks operate on specific engine resources, not abstract business concepts. A design must identify the actual row, predicate or coordination protocol representing the shared capacity.

22.5.3 PLAIN — a worked example

1. Revisit CAP-DEMO as a separate rule demonstration: capacity is 10 and allocations of 7 and 6 are individually positive. Their sum is 13, exceeding capacity by 3.
2. A CHECK requiring each allocation to be between 1 and 10 accepts both. It checks each row, not the combined allocation rule.
3. A possible redesign uses a capacity row with available=10. Every allocating path must atomically reduce that same available value under a sufficient-stock condition and record its allocation in the same transaction.
4. The redesign is incomplete if an administrator, import job or cancellation path can bypass the coordination protocol. Enumerate all mutation paths before claiming the invariant is enforced.

22.5.4 PLAIN — what is really happening inside

1. The read set and write set can differ. A request may read a sum of many rows but write only its new allocation. Another request can have a disjoint write set while depending on the same shared read condition.
2. A lock on a row that does not exist cannot casually be treated as a universal lock on future matching rows. Engines have different predicate and range-locking behaviours.
3. Computing a decision outside the protected transaction and merely repeating the final INSERT inside it can preserve the original unsafe assumption. Recompute or validate the decision at the correct boundary.

22.5.5 TECHNICAL — the engineer's version

1. Express the invariant mathematically, such as `sum(active quantities for capacity_key) <= capacity`. Then specify a protocol that every competing writer follows.
2. PostgreSQL Serializable can detect dependency patterns that threaten serializability, but applications must retry the whole transaction when required. Row-locking alone and snapshot isolation alone do not imply the same guarantee. [S66] [S120]
3. Avoid holding a transaction open across an unbounded external call to preserve a read. Separate durable intent and external execution where possible; Chapter 26 explains the outbox boundary.

22.5.6 WORDS — remember these

1. **Cross-row invariant:** a rule about several records together — a correctness condition involving a set of rows rather than one row's fields. **Shared guard record:** one coordination point for a shared limit — a record whose guarded mutation serializes changes to the represented resource. **Mutation path:** a way records can change — an application route, job, import, administrator action or other writer that must respect the invariant.

22.6 A stock-reservation case

22.6.1 PLAIN — in simple words

1. A reservation should have a clear outcome: accepted with a specific quantity, rejected for a stated business reason, or unresolved because the caller has not learned the result.

2. "The connection disappeared" belongs to the third category. It does not prove either acceptance or rejection. The database may have committed before the response was lost.
3. Build the operation in layers: validate the request, coordinate the stock change, record the outcome, commit, and expose a way to resolve retries. Do not let a success message outrun the evidence supporting it.

22.6.2 PLAIN — a picture in your head

1. The reservation desk records a numbered decision before sending a confirmation. A customer who loses the confirmation can ask about that same decision rather than buying another ticket by accident.
2. A decision number is useful only if the desk keeps its meaning stable and can tell a retry from a new purchase.
3. **Where the comparison breaks:** database and notification systems may fail separately. Recording a decision does not guarantee that an email was delivered or that the physical item was handed over.

22.6.3 PLAIN — a worked example

1. Start a new exercise with RES-PEN=5. R-A requests 3; R-B requests 3. An accepted R-A leaves 2 and a matching accepted reservation record. R-B cannot acquire another 3 from that state.
2. Inject a failure after reducing stock but before recording R-A. Rolling back the owned transaction should restore stock to 5 and leave no accepted R-A record.
3. Next allow R-A to commit but pretend its response was lost. Simply submitting another subtraction is unsafe. Chapter 26 records a scoped request identity and stable outcome so the same intent can be resolved without allocating twice.
4. Reconcile the accepted reservation quantities with the stock change in the exercise. This proves agreement within the model, not agreement with an unobserved real shelf.

22.6.4 PLAIN — what is really happening inside

1. The database transaction supplies a local atomic boundary. The request protocol adds a durable connection between the caller's intent and the recorded outcome.
2. Failure handling must know whether a transaction is still active, rolled back or committed. Where the client cannot determine that state remotely, it needs reconciliation rather than a guessed status.
3. Cancellation is another business operation, not deletion of inconvenient history. Define whether, when and how it releases stock and whether a repeated cancellation has any additional effect.

22.6.5 TECHNICAL — the engineer's version

1. Minimum tests include success, insufficient stock, missing scoped product, invalid quantity, duplicate request identity, failure before commit and reconciliation after a lost response. Add actual coordinated competing-connection tests before claiming an engine-specific concurrency result.
2. The companion code separates deterministic schedule models from real SQLite transactions. A sequential model demonstrates why a protocol is needed; a passing in-memory transaction test is not multi-host or power-loss evidence.
3. Request deduplication requires a stable identifier and an explicit equivalence contract. Atomic recording of that identity, its outcome and the business write is developed in Chapter 26. [S126]

22.6.6 WORDS — remember these

1. **Reservation:** a recorded allocation under defined rules — a business state transition that reduces available capacity while linking the allocation to a request. **Unknown outcome:** the caller lacks reliable completion evidence — a state in which a request may have committed even though its response was not received. **Reconciliation:** compare related records to resolve disagreement — a procedure that checks quantities, identities and outcomes against stated invariants and evidence.

22.97 Practice and worked answers

1. **Trace the lost update.** Start at 10. A reads 10 and reserves 3; B reads 10 and reserves 4; A writes 7 and B writes 6. **Answer:** the model ends at 6, whereas both accepted reservations require 3. Three units of A's effect were overwritten.
2. **Change the schedule.** Let A finish before B reads. **Answer:** B reads 7 and calculates 3. This safe-looking schedule does not prove that the unprotected program is safe under every permitted overlap.
3. **Interpret zero affected rows.** The guarded update selects by product and sufficient stock. **Answer:** zero alone cannot distinguish a missing product from insufficient stock. The API must define a safe response or perform an appropriately scoped diagnostic read.
4. **Find the incomplete key.** A multi-branch UPDATE filters only by product code. **Answer:** include the branch and, where applicable, tenant identity. A unique local product code is not a globally complete key.
5. **Evaluate a pre-check.** Two requests each see receipt 17 absent in BR-A. **Answer:** the lookup does not reserve the key. The composite uniqueness constraint must decide conflicting insertions, and the losing path must handle its result.
6. **Repair the capacity argument.** Each of two allocation rows passes `quantity <= 10`. **Answer:** that does not enforce their sum. Coordinate a shared resource or use another proven protocol covering all participating writers.
7. **Classify a disconnected caller.** The caller did not receive COMMIT's result. **Answer:** the outcome may be unknown. Resolve the original request identity rather than assuming rollback and issuing an unrelated new intent.
8. **State the evidence boundary.** A deterministic Python schedule produces a lost update. **Answer:** it proves the model's arithmetic and ordering, not that PostgreSQL or SQLite under an unspecified configuration admitted that schedule.

22.98 Common wrong ideas

1. **Wrong:** a race requires exactly simultaneous clicks. **Right:** an unsafe overlap between dependent steps is enough.
2. **Wrong:** an atomic UPDATE protects an earlier unprotected SELECT. **Right:** the earlier observation needs its own appropriate concurrency protocol.
3. **Wrong:** BEGIN and COMMIT imply serializable behaviour. **Right:** the engine and isolation level determine the actual guarantee.
4. **Wrong:** a pre-check implements uniqueness. **Right:** a database constraint must enforce competing claims at the write boundary.
5. **Wrong:** a locked individual row protects every total involving it. **Right:** the shared invariant may involve other rows or future insertions.

6. **Wrong:** a timeout proves that nothing was saved. **Right:** the caller may have an unknown outcome after a committed operation.
7. **Wrong:** retries should always create a new request key. **Right:** retrying one intent should preserve its defined identity; a new intent is a separate decision.
8. **Wrong:** correct reservation records prove correct physical inventory. **Right:** the shelf remains an external source of evidence.

22.99 Chapter summary in 20 lines

1. Competing requests interleave at their internal steps, not only at button presses.
2. State the initial records, statements and transaction boundaries before analysing an overlap.
3. A schedule model is different from a recorded engine experiment.
4. A replacement calculated from a stale read can overwrite another accepted change.
5. Starting at 10, stale replacements of 7 then 6 lose the intended combined result of 3.
6. A relative update expresses a change instead of an obsolete replacement.
7. A guarded write combines a condition with the protected update.
8. Inspect affected rows against the operation's expected cardinality.
9. Validate positive bounded quantities before attempting the database mutation.
10. Select records using every required tenant, branch and local identity component.
11. Seeing that a key is absent does not reserve it.
12. Enforce uniqueness in the database and handle the losing request deliberately.
13. A conflict does not prove that two payloads represent the same intent.
14. Cross-row limits need protocols that protect the shared invariant.
15. Every mutation path must follow that protocol, including jobs and administrative writers.
16. Record the reservation and stock change within the intended atomic boundary.
17. A failure before commit should not leave one half of the local operation accepted.
18. A lost response after commit can leave the caller uncertain rather than rejected.
19. Resolve retries through stable request identity and recorded outcomes.
20. Database correctness and physical-stock accuracy require different evidence.

23

CHAPTER

Isolation Levels and What You Can Observe

23.0 What this chapter gives you

1. A transaction needs an answer to a deceptively simple question: which changes made by other transactions may it see? Isolation levels describe parts of that answer and the anomalies that remain possible.
2. You will distinguish statement snapshots from transaction snapshots, trace non-repeatable reads and write skew, and understand what serializable execution promises without mistaking it for a single physically sequential worker.
3. Product-specific statements in this chapter refer to PostgreSQL 17 unless explicitly marked otherwise. SQLite examples are kept separate. The written schedules are teaching models, not fabricated server traces.

23.1 Visibility rules

23.1.1 PLAIN — in simple words

1. While Mira is changing an order, Dev may be reading it. The database needs rules about whether Dev sees the old version, the new version or a wait before receiving an answer.
2. "A transaction hides changes" is too vague. We must ask whose changes, until which boundary, and whether two statements in the same transaction can see different committed worlds.
3. An isolation level does not decide whether a recorded sale actually occurred. It governs interactions among database operations. Good visibility rules cannot transform invented input into a verified real-world event.

23.1.2 PLAIN — a picture in your head

1. Imagine viewing a noticeboard through a window while someone updates its cards. One rule gives you a fresh photograph whenever you ask a question. Another gives you one photograph to use throughout a consultation.
2. Both avoid watching half-erased handwriting, but they do not give the same answers after the board changes.
3. **Where the comparison breaks:** transactions can normally see their own earlier writes, so their view is not simply a frozen photograph of everyone else's board. Locking and conflict checks may also affect what happens when they write.

23.1.3 PLAIN — a worked example

1. Suppose product ISO-PEN has price 100 in a separate example. Transaction A changes it to 120 but has not committed. Transaction B asks for the price.

2. In the PostgreSQL levels discussed here, B does not read A's uncommitted replacement as an ordinary visible committed value. What B later sees after A commits depends on B's snapshot and statement timing.
3. If A rolls back, its transactional price replacement must not become an accepted committed price. A read that relied on an uncommitted replacement would have relied on a change that never became committed history.
4. Keep this separate from sequences: PostgreSQL sequence advancement has special non-rollback behaviour. A generated number is not itself proof of a committed business record. [S122]

23.1.4 PLAIN — what is really happening inside

1. Engines combine snapshots, locks, row versions and conflict detection in different ways. An isolation-level name is not a substitute for the relevant product documentation.
2. The SQL standard's traditional anomaly categories are useful vocabulary, but implementations can offer stronger guarantees than the minimum associated with a name.
3. A practical contract names the engine, version, level, transaction boundaries and participating writers. It also states what the application does when the engine refuses a conflicting transaction.

23.1.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 implements Read Uncommitted as Read Committed. Its Repeatable Read prevents phantom reads as well as non-repeatable reads, while serialization anomalies remain possible. Serializable adds protection against those anomalies through its concurrency-control implementation. [S66]
2. A dirty read observes data written by an uncommitted transaction. A non-repeatable read obtains a different value when rereading a row after another transaction commits a change. A phantom concerns changed membership of a repeated predicate query. These categories describe observations, not every possible application failure.
3. SQLite normally serializes writers; WAL mode permits readers to retain snapshots while a writer works. Its documented exceptions and upgrade failures must be treated as SQLite-specific behaviour, not inferred from PostgreSQL names. [S54]

23.1.6 WORDS — remember these

1. **Isolation level:** the chosen rules for overlapping transactions — a named contract restricting visibility and permitted concurrency anomalies. **Dirty read:** seeing a change before it commits — observation of another transaction's uncommitted data. **Phantom:** a repeated condition matches a changed set — a predicate-query membership change caused by another transaction's committed work.

23.2 Statement and transaction snapshots

23.2.1 PLAIN — in simple words

1. A statement snapshot gives one query a consistent view of relevant committed data at a particular point. A later query may receive a newer view.
2. A transaction snapshot lets a transaction continue using an earlier view of other transactions' committed work. This helps related reads agree, but it does not mean the transaction can safely make every possible decision from that view.

3. Snapshot age also matters operationally. A long-running reader can keep old information relevant to the engine long after ordinary short requests have moved on.

23.2.2 PLAIN — a picture in your head

1. For each question, Mira either asks for today's newly printed report or keeps consulting the report she collected when the meeting began.
2. A fresh report can include later corrections; a retained report supports consistent comparisons within the meeting. Neither is automatically preferable for every purpose.
3. **Where the comparison breaks:** a database snapshot is not usually a complete copied report. Visibility is determined using version and transaction metadata, and the transaction can see changes it made itself.

23.2.3 PLAIN — a worked example

1. Start with ISO-PEN=100. Reader R runs its first SELECT and sees 100. Writer W then commits a change to 120. R runs the same SELECT again.
2. Under PostgreSQL 17 Read Committed, ordinary separate SELECT statements can return 100 and then 120 because the second statement obtains a later snapshot.
3. Under PostgreSQL 17 Repeatable Read, R's snapshot begins with its first non-transaction-control statement. If that snapshot precedes W's commit, the repeated ordinary read continues to see 100, excluding R's own changes for simplicity.
4. Neither result is inherently a lie. The question is which visibility contract the application needed and actually selected. [S66]

23.2.4 PLAIN — what is really happening inside

1. The engine checks whether a row version is visible to the snapshot. A later committed replacement can exist while an older transaction still reads an earlier visible version.
2. Read Committed does not mean every operation inside one statement behaves like a simplistic immutable photograph. PostgreSQL's modifying statements have rules for waiting on and rechecking concurrently updated rows.
3. A report requiring several related totals should explicitly choose how their common observation boundary is established. Repeated independent queries cannot be assumed to describe one shared instant.

23.2.5 TECHNICAL — the engineer's version

1. In PostgreSQL 17, Read Committed ordinary SELECT uses a snapshot as of the query's start and sees the transaction's own preceding writes. Repeatable Read uses the snapshot established at the transaction's first non-transaction-control statement. [S66]
2. A stable snapshot is a visibility mechanism, not a full serializability proof. Transactions can read the same condition and write disjoint rows in ways that violate a combined invariant.
3. For reproducible demonstrations, record when the snapshot-establishing statement ran. Writing only BEGIN in a timeline can assign the snapshot to the wrong moment in this implementation.

23.2.6 WORDS — remember these

1. **Statement snapshot:** a view chosen for one statement — the visibility boundary used by a query under statement-level snapshot rules. **Transaction snapshot:** a retained view for related state-

ments — a visibility boundary reused through a transaction under the chosen implementation. **Snapshot establishment:** the event that chooses the view — the implementation-defined point at which a transaction's snapshot is acquired.

23.3 Non-repeatable reads

23.3.1 PLAIN — in simple words

1. A non-repeatable read means that reading the same row twice can produce different values because another transaction committed between the reads.
2. This is sometimes useful: a screen refresh should often reveal a newly changed value. It is dangerous when a calculation assumes that an earlier value remained unchanged across several steps.
3. The solution is not always "use the strongest level everywhere." First decide whether the operation needs a common snapshot, a guarded write, a lock or a different workflow.

23.3.2 PLAIN — a picture in your head

1. Mira checks the price card, walks to the till, and checks it again after Dev updates the card. The second answer differs because the world she is consulting changed.
2. A quotation process must decide whether the first observed price was merely information or an agreement that should be preserved.
3. **Where the comparison breaks:** a database snapshot can preserve an earlier stored view, but it cannot create a customer agreement that the application never recorded. Historical meaning and transaction visibility are related but different responsibilities.

23.3.3 PLAIN — a worked example

1. Consider an analytical exercise with A's balance=60 and B's balance=40. A transfer moves 10 from A to B in one transaction, leaving 50 and 50. The total is 100 before and after.
2. A reader first queries A before the transfer commits and receives 60. It then separately queries B after the transfer commits and receives 50. Adding the two observations gives 110.
3. Neither query necessarily returned an individually incorrect committed value. The reader combined different observation boundaries and called the result one total.
4. A single appropriately defined query or a suitable retained snapshot can align the reads. This example is fictional arithmetic, not a banking implementation or financial advice.

23.3.4 PLAIN — what is really happening inside

1. The transfer's atomicity protects its own related writes. It does not automatically force a separate reader's two statements to share a snapshot.
2. A transaction can therefore be locally atomic while a poorly designed multi-statement report mixes versions. Different guarantees protect different boundaries.
3. Document whether a report is "as of one snapshot," "latest available per query," or an eventually assembled view. The label affects how readers should interpret discrepancies.

23.3.5 TECHNICAL — the engineer's version

1. Under PostgreSQL Read Committed, successive commands can see different committed data. This makes it unsuitable for a multi-command invariant calculation that silently assumes a stable view unless additional controls establish the required condition. [S66]

2. Predicate membership also matters. Repeating `WHERE status='active'` may change the rows returned even when no previously returned row changes value. Analyse both row values and the set being counted.
3. Do not "fix" a mixed-snapshot report by clamping an unexpected total to a preferred number. Preserve evidence, identify the observation boundary and rerun the defined query under a suitable contract.

23.3.6 WORDS — remember these

1. **Non-repeatable read:** rereading a row yields a later committed value — a permitted observation change under some isolation contracts. **Mixed-snapshot calculation:** combining facts observed at different boundaries — a derived result that may correspond to no single database state. **Observation contract:** the stated point or interval a result describes — the temporal and visibility meaning assigned to a query or report.

23.4 Write skew

23.4.1 PLAIN — in simple words

1. Write skew is a different trap from two people overwriting the same row. Two transactions can change different rows and still break a rule about those rows together.
2. Each transaction may see a stable, internally consistent snapshot. The trouble is that both make decisions without seeing the other's eventual change.
3. Protecting only the changed rows may miss the shared condition. The rule, not just the UPDATE target, must guide the concurrency design.

23.4.2 PLAIN — a picture in your head

1. Two assistants are on duty, and the rule requires at least one to remain. Each sees that the other is present and decides to leave.
2. They lock their own lockers carefully, but neither coordinates the shared requirement that somebody stay.
3. **Where the comparison breaks:** real staffing has human communication and policy enforcement outside a database. This is a small logical model of a cross-row invariant, not a recommended scheduling system.

23.4.3 PLAIN — a worked example

1. A new table contains `(Mira,on_duty=1)` and `(Dev,on_duty=1)`. The invariant is `count(on_duty=1) >= 1`.
2. T1's snapshot sees two assistants and sets Mira's row to 0. T2's snapshot also sees two and sets Dev's row to 0. Their writes concern different rows.
3. If both transactions are allowed to commit under this snapshot model, the final count is 0. Either serial order would have made the second transaction see only one assistant and refuse its departure.
4. The example illustrates a serialization anomaly possible under snapshot isolation, including PostgreSQL's Repeatable Read model. It is not claimed as a reproduced SQLite two-writer trace. [S66]

23.4.4 PLAIN — what is really happening inside

1. Each transaction's decision depends on a fact changed by the other. Those cross-dependencies create a cycle that cannot be explained by a safe serial order of the stated decision logic.
2. Locking only Mira's row in T1 and only Dev's row in T2 still leaves distinct protected targets. A protocol must cover the shared duty condition, not merely each private update.
3. Possible designs include a common guard record, locking an appropriately defined shared set with consistent ordering, or serializable transactions with full retries. Each proposal must include every path that can change the duty state.

23.4.5 TECHNICAL — the engineer's version

1. Snapshot isolation can prevent same-row concurrent update patterns while still allowing write skew through disjoint write sets and intersecting read dependencies. Stable reads therefore do not imply serializability. [S66]
2. A correct serializable execution of the stated decision logic cannot commit both departures from the initial two-person state. At least one transaction must observe the changed condition, wait appropriately or abort and retry under the implementation's protocol.
3. A serializable transaction containing faulty decision logic can still commit a bad business outcome. Isolation cannot enforce an invariant that the program neither checks nor represents correctly.

23.4.6 WORDS — remember these

1. **Write skew:** separate writes jointly break a shared rule — an anomaly in which transactions act on overlapping read conditions while changing different records. **Read dependency:** a decision relies on an observed fact — a relationship between a transaction's read and another transaction's relevant write. **Serializable outcome:** a result explainable by a legal one-at-a-time execution — an outcome equivalent to some serial order of the participating transactions.

23.5 Serialisable execution

23.5.1 PLAIN — in simple words

1. Serializable means that committed transactions behave as though they occurred in some one-at-a-time order. The engine may still run their internal work concurrently.
2. The word "some" matters. Serializability alone is not a universal promise about real-time order observed by external callers across every system component.
3. An engine may preserve the guarantee by refusing a transaction and asking the application to retry. A refusal is part of the concurrency contract, not proof that the database has become unreliable.

23.5.2 PLAIN — a picture in your head

1. Several clerks prepare paperwork in parallel. The final accepted collection must be explainable as though the desk handled each complete case in a valid sequence.
2. If two prepared cases cannot both fit such a sequence, one may need to be prepared again using newer information.
3. **Where the comparison breaks:** the engine does not necessarily find and print a literal serial schedule for the application. Implementations use locks or dependency detection to prevent forbidden outcomes.

23.5.3 PLAIN — a worked example

1. Return to the duty model. If Mira's departure is treated first, one assistant remains. Dev's later decision must then refuse to leave. Reverse the order and Mira must stay instead.
2. These are two different acceptable serial outcomes. The system need not choose the same assistant every time unless a separate fairness policy requires that.
3. Committing both departures is unacceptable under the stated serial decision logic. A retry must repeat the read and decision, not merely resubmit the old "set off duty" statement.
4. The operation's correctness includes the retry path because that path may be the normal way the engine resolves competition.

23.5.4 PLAIN — what is really happening inside

1. PostgreSQL's serializable implementation tracks potentially dangerous read/write dependencies and may abort a transaction to prevent a serialization anomaly.
2. Its predicate-lock information is not the same as an ordinary blocking row lock. Treating every lock-related term as "another request must wait here" gives the wrong mental model.
3. Results read inside a transaction that later aborts cannot be presented as though that transaction's business decision committed. Keep externally visible success aligned with the final outcome.

23.5.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 Serializable uses Serializable Snapshot Isolation. Its `SIReadLock` predicate-lock records support dependency checking; they do not block ordinary operations or cause deadlocks in the manner of blocking locks. [S66]
2. Correctness arguments require appropriate participation by the transactions that interact with the invariant. Mixing arbitrary bypass writers into a supposedly protected application can invalidate the application's proof.
3. Distinguish serializability from strict serializability and linearizability. Chapter 36 introduces real-time ordering explicitly; do not advertise stronger guarantees merely because a transaction used the Serializable label.

23.5.6 WORDS — remember these

1. **Serializability:** concurrent work has a valid serial explanation — equivalence of committed transaction effects to an execution in some serial order. **Serialization failure:** a transaction is refused to preserve the isolation guarantee — an error requiring an appropriate full-operation retry or failure response. **Predicate tracking:** remember the read conditions that matter — implementation metadata used to detect dependencies involving rows or ranges a transaction read.

23.6 Retries and operational cost

23.6.1 PLAIN — in simple words

1. Stronger isolation can make the application simpler in one respect: it may avoid some fragile manual coordination. It does not remove the need to handle failures and retries.
2. A retry consumes work and can repeat unsafe external effects. Re-running a database decision is different from sending another payment instruction or email.
3. The operational question is whether the complete protocol meets its correctness and performance requirements under the actual workload. A label alone cannot answer that question.

23.6.2 PLAIN — a picture in your head

1. A clerk whose draft conflicts with another case returns to the beginning, reads the current facts and prepares a new decision. Re-signing the old draft would preserve the original conflict.
2. The clerk also checks whether a confirmation was already dispatched. Starting again must not mean blindly sending everything twice.
3. **Where the comparison breaks:** software needs explicit error classes, deadlines and request identities. A human's memory of "I already handled that" is not a durable concurrency protocol.

23.6.3 PLAIN — a worked example

1. Suppose 100 submitted operations produce 100 initial transaction attempts, and 8 encounter retryable serialization failures. If each of those succeeds on one additional attempt, the database executed 108 attempts for 100 completed operations.
2. The retry count is 8, not an 8% business failure rate. But the extra attempts consumed resources and increased some callers' latency.
3. If 2 operations exhaust their retry deadline, report 98 completions and 2 failures separately. Do not compute latency only for the fastest successes and call it the service's complete performance.
4. These figures are invented arithmetic for reporting definitions, not a measured property of Serializable isolation.

23.6.4 PLAIN — what is really happening inside

1. A retry must re-enter a clean transaction and repeat every read and decision that depended on the previous attempt's view.
2. Classify errors. A serialization failure and an invalid required field have different remedies. Repeating an invalid field cannot make it valid.
3. Bound retry attempts by a deadline and cancellation policy, and use suitable backoff when contention warrants it. Keep a stable business request identity so uncertainty does not become accidental duplication.

23.6.5 TECHNICAL — the engineer's version

1. PostgreSQL recommends retrying the complete transaction after SQLSTATE `40001`, including application logic that selected the statements and values. Deadlock error `40P01` can also justify a retry, while other errors require context-specific classification. [S120]
2. Measure attempt counts, conflict classes, wait time, successful-operation latency and deadline exhaustion. Do not equate higher raw attempt throughput with more useful business completions.
3. The companion exercises model anomaly schedules and test bounded SQLite transaction behaviour. PostgreSQL isolation examples here are documentation-grounded, not executed server experiments. Independent engine-specific testing remains necessary before deploying a matching protocol.

23.6.6 WORDS — remember these

1. **Full-transaction retry:** repeat the decision from fresh protected reads — a new attempt that does not reuse an invalidated transaction's conclusions blindly. **Attempt amplification:** extra work caused by retries — the ratio or count of transaction attempts relative to completed business operations. **Retry deadline:** the point beyond which another attempt is not allowed — an operation-level bound on time spent resolving transient failures.

23.97 Practice and worked answers

1. **Choose the snapshot boundary.** PostgreSQL Repeatable Read begins at 09:00, but its first ordinary statement runs at 09:01. **Answer:** the documented snapshot is established by the first non-transaction-control statement, not merely by BEGIN.
2. **Trace two reads.** R reads 100, W commits 120, R reads again under Read Committed. **Answer:** 100 then 120 is permitted for ordinary separate SELECT statements.
3. **Explain a total of 110.** A transfer preserves 60+40=50+50=100, but a reader combines old A=60 with new B=50. **Answer:** the reader mixed observation boundaries; the transfer's own atomicity did not force the reader's statements to share a snapshot.
4. **Identify write skew.** Two transactions see two on-duty assistants and each disables a different assistant. **Answer:** disjoint writes can jointly violate the shared minimum-one invariant under the stated snapshot model.
5. **Assess a row-lock proposal.** Each assistant locks only their own row. **Answer:** that does not by itself protect the shared count. Specify a protocol covering the invariant and all writers.
6. **Interpret Serializable.** Must the engine run every transaction physically one after another? **Answer:** no. Committed results must have a valid serial explanation; internal execution may overlap.
7. **Retry correctly.** A serialization failure occurs after a stock decision. **Answer:** start a clean transaction and repeat the relevant reads and decision. Do not resubmit only the final write with stale assumptions.
8. **Report attempts.** One hundred operations need 108 attempts and all complete. **Answer:** report 100 completions and 8 extra attempts, with latency and retry distribution. Do not call the extra attempts eight failed business operations.

23.98 Common wrong ideas

1. **Wrong:** isolation names have identical behaviour in every engine. **Right:** consult the chosen implementation and version.
2. **Wrong:** Repeatable Read universally permits phantoms. **Right:** PostgreSQL 17 provides stronger snapshot behaviour than that minimum description.
3. **Wrong:** a stable snapshot prevents every business anomaly. **Right:** write skew can involve disjoint writes based on a shared snapshot condition.
4. **Wrong:** an atomic transfer guarantees every separately assembled report sees one total. **Right:** the reader needs its own observation contract.
5. **Wrong:** Serializable means one physical worker and no parallelism. **Right:** it constrains the explanation of committed outcomes.
6. **Wrong:** a serialization failure can be fixed by replaying only the last UPDATE. **Right:** repeat the entire dependent decision in a new transaction.
7. **Wrong:** every database error deserves a retry. **Right:** permanent validation and identity conflicts need different handling.
8. **Wrong:** a documented PostgreSQL schedule is a measured SQLite result. **Right:** implementation claims and experiment evidence must remain separate.

23.99 Chapter summary in 20 lines

1. Isolation describes how overlapping transactions interact and what they may observe.
2. Dirty reads concern another transaction's uncommitted work.

3. Non-repeatable reads concern changed committed values across repeated reads.
4. Phantoms concern changed membership of a repeated predicate query.
5. PostgreSQL 17 treats Read Uncommitted as Read Committed.
6. PostgreSQL Read Committed ordinary SELECT obtains a statement-start snapshot.
7. Successive statements can therefore see different committed states.
8. PostgreSQL Repeatable Read retains its first non-control-statement snapshot.
9. A transaction can see its own preceding writes under the documented rules.
10. Snapshot consistency is not a universal serializability guarantee.
11. Mixing snapshots can produce a total corresponding to no single state.
12. Write skew can break a shared rule through changes to different rows.
13. Protect the invariant, not merely each writer's individual target.
14. Serializable outcomes must be explainable by a valid serial order.
15. Serializable execution need not be physically sequential.
16. PostgreSQL's dependency tracking can abort a transaction to preserve that guarantee.
17. Repeat the complete dependent decision after a retryable serialization failure.
18. Keep external effects outside blind transaction retries.
19. Measure retry work and deadline failures alongside successful-operation latency.
20. Match every claim to the engine, version, configuration and evidence actually used.

24

CHAPTER

Locks, Deadlocks and Safe Retries

24.0 What this chapter gives you

1. A lock is a coordination mechanism, not a synonym for failure. Waiting can be the correct way to protect a record; waiting forever or repeating the wrong operation is not.
2. This chapter separates lock scope, compatibility, blocking, deadlock, timeouts and retries. You will draw a wait graph, choose a consistent acquisition order and identify effects that must not be repeated blindly.
3. Lock names and error codes are implementation-specific. PostgreSQL 17 examples are documented illustrations. The supplied SQLite exercises use fresh teaching databases and do not establish PostgreSQL lock behaviour.

24.1 What is locked

24.1.1 PLAIN — in simple words

1. A lock reserves a kind of access to a resource while an operation is in progress. The resource might be a row, a table or an application-defined coordination key.
2. Saying "the database is locked" hides the important details. Which operation is waiting, which resource does it need, who currently holds conflicting access, and when should that holder finish?
3. A lock is effective only within the protocol that honours it. An application-defined lock does not magically stop an unrelated writer that ignores that agreement.

24.1.2 PLAIN — a picture in your head

1. A staff member places a sign on a particular filing drawer while correcting its contents. Other staff follow the drawer-access rule until the sign is removed.
2. Locking the drawer is different from closing the entire room. A room closure affects far more work.
3. **Where the comparison breaks:** database locks are not all exclusive signs, and normal readers may use older visible row versions without waiting for a writer. Some lock names also describe tables even when they contain the word "row."

24.1.3 PLAIN — a worked example

1. Request A is updating product LOCK-PEN. Request B wants to update the same product. Request C is reading an unrelated product. A useful diagnosis identifies whether B conflicts with A and whether C needs a conflicting resource at all.
2. A table-wide schema change may have a broader lock requirement than the ordinary row update. It can therefore wait behind activity that would not block another ordinary row change.

3. Record the actual lock mode and object identifier in an engine investigation. Do not infer a table lock from a slow screen or infer a row lock solely from a statement's WHERE clause.
4. The example names are synthetic; it does not assert a particular observed server lock trace.

■ 24.1.4 PLAIN — what is really happening inside

1. The engine checks requested access against existing incompatible locks. It may grant the request, queue it, time it out or later select a transaction for abort.
2. Locks can be acquired implicitly by SQL statements or explicitly through locking clauses. A statement may acquire more than one kind of lock.
3. The business resource and engine resource need not match one-to-one. Protecting "remaining branch capacity" requires identifying the concrete record or protocol representing that capacity.

■ 24.1.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 has table-level and row-level lock modes. Names such as ROW EXCLUSIVE in the table-lock family still denote table-level locks; the name alone does not establish row granularity. [S118]
2. PostgreSQL advisory locks coordinate application-defined resources. Their usefulness depends on every relevant participant following the convention; they are not substitutes for ordinary data constraints or authorization. [S118]
3. SQLite's database-level writer coordination differs substantially from PostgreSQL row-locking patterns. Describe its transaction and journal mode explicitly rather than translating a PostgreSQL lock diagram unchanged. [S54]

■ 24.1.6 WORDS — remember these

1. **Lock scope:** the resource access being coordinated — the row, table, key or other engine/application object covered by a lock. **Lock holder:** the participant currently retaining protected access — the session or transaction owning a granted lock. **Advisory lock:** a coordination agreement writers must choose to follow — an application-defined locking facility not automatically tied to every relevant data mutation.

24.2 Lock modes and duration

■ 24.2.1 PLAIN — in simple words

1. Not every lock excludes every other lock. A compatibility rule determines which kinds of access may coexist.
2. Duration matters as much as scope. A brief conflict can be harmless; a transaction left open while someone answers a phone can hold resources long after useful database work stops.
3. Know what releases a lock. Transaction completion, statement completion, session termination and explicit release are different boundaries.

■ 24.2.2 PLAIN — a picture in your head

1. A reading room may allow several readers at once but stop new visitors during a structural repair. The rules depend on what each person is doing, not just whether someone is present.
2. A repair permit that lasts until the repair finishes is different from a building key retained by a staff member all day.

3. **Where the comparison breaks:** an engine's compatibility matrix is exact and product-specific. Everyday "reader" and "writer" labels are not precise enough to reproduce all lock modes.

24.2.3 PLAIN — a worked example

1. In PostgreSQL 17, an ordinary SELECT obtains ACCESS SHARE on its referenced table. ACCESS EXCLUSIVE conflicts with it; most other table-lock modes do not conflict with ACCESS SHARE.
2. Thus a schema operation requiring ACCESS EXCLUSIVE can be delayed by a long-running reader, even when that reader is not changing any row.
3. Conversely, a row locked for an update does not necessarily block an ordinary MVCC SELECT from reading an appropriate visible version. "A writer exists" is not enough to predict every reader's wait. [S118]
4. The practical response is to inspect the actual requested and held modes, not to memorize that all readers either always block or never block writers.

24.2.4 PLAIN — what is really happening inside

1. Transaction-scoped locks generally remain until the transaction ends, subject to documented savepoint and implementation behaviour.
2. Application code can accidentally lengthen that interval through network calls, slow loops, user interaction or an abandoned transaction in a pooled connection.
3. Session-level advisory locks have different lifetime rules from transaction-level advisory locks. Returning a connection to a pool without clearing session state can pass unwanted coordination state to another logical request.

24.2.5 TECHNICAL — the engineer's version

1. PostgreSQL distinguishes session-level advisory locks from transaction-level advisory locks. Session-level advisory locks do not follow ordinary transaction rollback semantics; transaction-level advisory locks are released at transaction end. [S118]
2. Define a maximum intended transaction duration and observe time spent idle inside transactions separately from active query work. A short individual query does not imply a short lock-holding transaction.
3. Savepoint rollback can release locks acquired after that savepoint under PostgreSQL's documented rules. Do not extrapolate this into a claim that any arbitrary external resource acquired in the same code block is also released. [S118] [S121]

24.2.6 WORDS — remember these

1. **Lock compatibility:** which access modes can coexist — the engine's conflict relation between requested and held lock modes. **Lock lifetime:** how long protected access remains held — the release boundary determined by transaction, statement, session or explicit protocol. **Idle in transaction:** a transaction remains open without active useful work — a state that can retain locks and snapshots while the application waits elsewhere.

24.3 Blocking versus deadlock

24.3.1 PLAIN — in simple words

1. Blocking means one operation is waiting for access held incompatibly by another. The holder may finish normally and release the resource.

2. A deadlock is a cycle of waits: each participant needs something held by another participant in the cycle, so none can finish without intervention.
3. A long wait is not automatically a deadlock. It may be an overloaded server, a slow holder, an abandoned transaction or an external dependency.

24.3.2 PLAIN — a picture in your head

1. Mira waits for Dev to finish using the label printer. Dev is printing and will finish. That is blocking.
2. Now Mira holds the label printer and needs Dev's scanner, while Dev holds the scanner and needs Mira's printer. Neither releases the held device until acquiring the other. That is a deadlock in this model.
3. **Where the comparison breaks:** databases can detect some lock cycles and abort a participant. A cycle spanning outside services may not be visible to one database's detector.

24.3.3 PLAIN — a worked example

1. Let A hold resource X and request Y. Let B hold Y and request X. Draw an arrow from the waiter to the holder: A → B and B → A.
2. The arrows form a cycle. Neither can obtain its requested resource while both retain their current locks under the stated protocol.
3. Compare A → B → C with no arrow returning from C. That chain is blocking, not a cycle. C may complete and release the chain.
4. The companion graph exercise checks directed cycles in a small synthetic wait graph. It is not a full model of an engine's lock manager, lock upgrades or distributed deadlock detection.

24.3.4 PLAIN — what is really happening inside

1. A database may inspect waits and choose a victim transaction to abort. Releasing that transaction's locks lets another participant continue.
2. The application must handle the victim's failure even when its business request was otherwise valid. It should not assume that the first or newest request is always chosen.
3. A statement timeout or lock timeout can terminate waiting without proving that a cycle existed. Preserve the actual error classification in logs and retry decisions.

24.3.5 TECHNICAL — the engineer's version

1. A wait-for graph represents transactions as vertices and incompatible-resource waits as directed edges. A cycle is the key condition in the simplified single-resource ownership model used here.
2. PostgreSQL automatically detects deadlocks and aborts one involved transaction. Its documentation advises against relying on a predictable choice of victim. [S118]
3. A client-side deadline, server statement timeout, lock timeout and detected deadlock are different events. Keep their error codes and transaction-state consequences separate instead of collapsing them into "database busy."

24.3.6 WORDS — remember these

1. **Blocking:** waiting for conflicting access to be released — a resource wait that need not contain a dependency cycle. **Deadlock:** a cycle of participants waiting on one another — a condition requiring some participant to release, abort or otherwise break the cycle. **Wait-for graph:** a map of who waits for whom — a directed graph used to reason about resource dependencies.

24.4 Detection and ordering

24.4.1 PLAIN — in simple words

1. A consistent rule for acquiring multiple resources can prevent many deadlocks. Everyone asks for them in the same agreed order instead of choosing an order independently.
2. This works only when the protocol is complete. Hidden trigger work, alternate code paths or lock upgrades can introduce additional dependencies.
3. Prevention and recovery belong together. Even a carefully ordered application should handle deadlock errors because other participants and maintenance operations may not follow the same small model.

24.4.2 PLAIN — a picture in your head

1. Whenever two drawers are needed, staff open the lower-numbered drawer first. Someone may wait, but two staff following that rule cannot each hold the higher drawer while waiting for the lower one in the simple two-drawer model.
2. The order gives everyone a shared coordination language.
3. **Where the comparison breaks:** real statements can touch rows through plans, constraints or triggers. Sorting application IDs is not by itself a complete proof of all engine lock acquisition orders.

24.4.3 PLAIN — a worked example

1. A transfer-like teaching operation needs accounts AC-02 and AC-09. A second operation needs the same pair in the opposite business direction.
2. Both acquire their required coordination locks in key order: AC-02, then AC-09. One may wait for AC-02 before holding AC-09, avoiding the earlier reversed-order pattern.
3. Business direction remains independent of lock order. The destination account can be locked first if its key sorts first; locking order does not change the meaning of the transfer.
4. Include complete scoped keys in the ordering rule. Ordering only by a local number while ignoring tenant or branch can create ambiguous protocol descriptions.

24.4.4 PLAIN — what is really happening inside

1. A common acquisition order removes certain cycles from the allowed protocol. It is a structural argument, not merely an attempt to make collisions less likely.
2. Keep transactions short after acquiring locks. Holding the first lock while performing unrelated expensive work increases waiting even if no cycle forms.
3. During diagnosis, capture the waiting statement, holder, transaction age, lock objects and recent changes. Avoid killing arbitrary sessions without understanding the rollback and business-outcome consequences.

24.4.5 TECHNICAL — the engineer's version

1. PostgreSQL recommends acquiring locks on multiple objects in a consistent order and acquiring the most restrictive needed mode first where appropriate. This reduces common deadlock patterns but does not eliminate the need for error handling. [S118]
2. Document explicit locking SQL separately from the business updates. Include trigger, foreign-key and index-related interactions in a real review rather than treating them as invisible implementation magic.

3. A diagnostic procedure should favour bounded observation before intervention. Record which transaction was cancelled, which request it represented and how its outcome will be reconciled after rollback or connection loss.

24.4.6 WORDS — remember these

1. **Acquisition order:** the sequence in which resources are requested — a shared ordering protocol intended to prevent cyclic waits. **Lock upgrade:** requesting stronger access while already holding weaker access — a mode transition that can introduce additional conflicts. **Deadlock victim:** the transaction selected to break a cycle — a participant aborted by the engine so other work can proceed.

24.5 Bounded retries

24.5.1 PLAIN — in simple words

1. A retry is another attempt at the same intended operation, not permission to repeat every action blindly until something works.
2. Some failures are temporary, such as a chosen serialization conflict. Others are permanent until the input or policy changes, such as an invalid quantity or an incompatible request-key reuse.
3. A retry needs a stopping rule. Without one, a failing system can spend more and more resources repeating work while the caller receives no useful outcome.

24.5.2 PLAIN — a picture in your head

1. A clerk who finds the desk briefly occupied waits a little and returns. A clerk whose form lacks a required signature does not solve the problem by submitting the same unsigned form a hundred times.
2. The reason for failure determines the next action.
3. **Where the comparison breaks:** software cannot rely on intuition about whether a failure is temporary. It needs documented error classification, transaction cleanup, a deadline and recorded attempt outcomes.

24.5.3 PLAIN — a worked example

1. Define a teaching retry policy allowing at most four total attempts within a two-second operation deadline. The first attempt is not called a retry; at most three further attempts remain.
2. Suggested wait ceilings might grow from 25 to 50 to 100 milliseconds, with a bounded random delay beneath each ceiling. These are invented parameters for illustrating the policy, not recommended universal production values.
3. Before each new attempt, check cancellation and remaining deadline. Start a clean transaction and reread the facts used in the decision.
4. If the deadline expires, return a defined failure or unresolved status as appropriate. Do not silently extend the caller's deadline just because the retry counter has room left.

24.5.4 PLAIN — what is really happening inside

1. Backoff reduces immediate repeated competition. Jitter spreads retry timing so a crowd does not wake and collide again in perfect synchrony.
2. Retrying only the failed final statement can preserve stale decisions from the previous transaction. Retry the entire dependent operation where the engine's error contract requires it.

3. Record attempts separately from business requests. The same request may have several failed attempts and one committed outcome; those counts answer different operational questions.

24.5.5 TECHNICAL — the engineer's version

1. PostgreSQL SQLSTATE `40001` requires retrying the complete transaction, including application decision logic. `40P01` deadlock detection can also justify a full retry. Uniqueness failures need context: some arise from competing decisions, while others are persistent conflicts. [S120]
2. A robust retry wrapper takes a monotonic deadline, maximum attempts, an allowlist of retryable failure classes and an operation that owns its transaction. Cleanup must complete before a new attempt starts.
3. A lost connection around commit creates a different problem from a confirmed abort. Use request identity and reconciliation to resolve uncertain completion rather than assuming the next attempt starts from an uncommitted world. [S126]

24.5.6 WORDS — remember these

1. **Backoff:** wait longer before another attempt — a retry-spacing strategy that reduces repeated contention or overload. **Jitter:** deliberately vary the waiting time — bounded randomness used to avoid synchronized retries. **Retry budget:** the allowed additional effort — limits on attempts, elapsed time or resources spent resolving transient failures.

24.6 Avoiding repeated external effects

24.6.1 PLAIN — in simple words

1. Database rollback does not recall an email, unprint a receipt or reverse an already accepted external instruction.
2. A retryable transaction that performs such an effect midway can cause it again when the transaction restarts. Even moving the effect after commit leaves a crash gap before sending it.
3. The design needs a durable connection between the accepted business action and the external work still to be attempted. An outbox records that intent, but delivery and consumer behaviour still need their own rules.

24.6.2 PLAIN — a picture in your head

1. Mira writes "send this confirmation" into the accepted reservation record before a messenger takes responsibility for delivery. Another messenger can later find unfinished deliveries.
2. A messenger who delivered the letter but failed to mark the list may cause another delivery attempt. The list solves forgetting, not every duplicate.
3. **Where the comparison breaks:** a recipient's system can sometimes deduplicate by a stable identifier, but a real person's inbox or an arbitrary external service may not offer the exact atomicity required by your application.

24.6.3 PLAIN — a worked example

1. Attempt A sends a confirmation, then encounters a serialization failure before committing its reservation. A full retry sends the confirmation again. Two messages now describe a reservation that committed only on the second attempt, or perhaps never committed at all.
2. Instead, record the reservation and an outbox row in one transaction. If that transaction aborts, neither accepted reservation nor accepted delivery intent remains.

3. A dispatcher later sends the outbox message. If it crashes after sending but before marking completion, it may send again. Give the message a stable identity and define the receiver's duplicate-handling contract.
4. Chapter 26 demonstrates a local inbox record and database effect committed together. That bounded design does not prove exactly-once email delivery.

24.6.4 PLAIN — what is really happening inside

1. A transaction boundary ends at the participating storage system. An HTTP call to another service is not automatically enlisted in that transaction.
2. The outbox transforms an immediate external side effect into durable local intent. Delivery becomes a separate state machine with attempts, acknowledgements, retries and unresolved outcomes.
3. A retry framework should therefore receive an operation designed for repetition, not wrap arbitrary business code and hope rollback covers everything it did.

24.6.5 TECHNICAL — the engineer's version

1. The transactional outbox pattern records business changes and message intent in the same local transaction. Dispatch can still produce duplicate messages, so consumers must handle the delivery semantics explicitly. [S125]
2. Provider-supported idempotency keys are useful only under the provider's actual scope, payload-equivalence and retention contract. A key header with no such contract is just an uninterpreted string. [S126]
3. Tests should inject failures before commit, after commit, before send and after send but before acknowledgement recording. State which components are models and which are actual processes; a simulated crash point is not a power-loss experiment.

24.6.6 WORDS — remember these

1. **External effect:** a change outside the local transaction — an email, print, remote API action or other operation not undone by ordinary database rollback. **Outbox:** a durable list of external work implied by accepted changes — message-intent records committed atomically with local business state. **Delivery gap:** uncertainty between an external action and its recorded acknowledgement — a failure window that can require replay or reconciliation.

24.97 Practice and worked answers

1. **Read a lock name.** PostgreSQL reports ROW EXCLUSIVE in its table-lock family. **Answer:** this is a table-level mode despite its name. Determine scope from the documented lock family and object.
2. **Explain a waiting schema change.** A long SELECT holds ACCESS SHARE while a change needs ACCESS EXCLUSIVE. **Answer:** those modes conflict; a reader can delay the broader operation without changing rows.
3. **Find a cycle.** A → B, B → C, C → A. **Answer:** the graph contains a cycle. A → B → C with no return edge is only a chain in this simplified model.
4. **Separate a timeout.** A client gives up after two seconds. **Answer:** elapsed time alone does not prove deadlock. Inspect the actual failure class and holder state.

5. **Choose an order.** Opposite-direction operations need AC-02 and AC-09. **Answer:** both should follow the same resource-acquisition order, independent of business direction, while reviewing all additional implicit locks.
6. **Count attempts.** A maximum of four attempts succeeds on the fourth. **Answer:** one original attempt and three retries occurred. Record four attempts and one completed business operation.
7. **Classify an invalid request.** Quantity is negative. **Answer:** reject or correct the input under the contract; contention backoff cannot make it valid.
8. **Find the external-effect gap.** A dispatcher sends then crashes before marking the outbox row delivered. **Answer:** replay may duplicate delivery. Stable message identity and receiver-side handling are still needed.

24.98 Common wrong ideas

1. **Wrong:** any lock is a database failure. **Right:** coordination often requires temporary incompatible access.
2. **Wrong:** all readers block all writers. **Right:** compatibility and MVCC behaviour depend on the operation and engine.
3. **Wrong:** every lock ends at COMMIT. **Right:** session-scoped advisory locks have different lifetime rules.
4. **Wrong:** a long wait proves a deadlock. **Right:** a deadlock requires the relevant dependency cycle, not merely elapsed time.
5. **Wrong:** sorting two IDs proves the whole application deadlock-free. **Right:** implicit locks and alternate paths still matter.
6. **Wrong:** retry every error. **Right:** distinguish transient conflicts, permanent rejection and unknown completion.
7. **Wrong:** retrying only the last statement always preserves the business operation. **Right:** dependent reads and decisions may need to run again.
8. **Wrong:** an outbox guarantees exactly-once external delivery. **Right:** it records local intent atomically; delivery duplicates remain a separate problem.

24.99 Chapter summary in 20 lines

1. A lock coordinates access to a specified resource.
2. Diagnose the waiter, holder, resource, mode and expected release boundary.
3. Lock names do not always reveal their granularity.
4. Compatibility matrices decide which modes can coexist.
5. A broad schema lock can wait behind ordinary reading activity.
6. Row locks need not block ordinary MVCC reads of visible versions.
7. Transaction duration controls how long many locks remain relevant.
8. Idle transactions can retain resources without doing useful work.
9. Session-level advisory locks differ from transaction-scoped locks.
10. Blocking is a wait; deadlock is a cycle of waits under the relevant model.
11. A timeout alone does not establish a deadlock.
12. A detector may abort a victim so other transactions can proceed.
13. Consistent resource-acquisition order prevents common cyclic patterns.

14. Include implicit locks and all mutation paths in a real protocol review.
15. Retry only failures classified as retryable under the actual contract.
16. Repeat dependent reads and decisions inside a clean transaction.
17. Bound retries by attempts, a monotonic deadline and cancellation.
18. Backoff and jitter can reduce synchronized repeated contention.
19. Database rollback does not undo ordinary external effects.
20. Atomic outbox intent still requires a delivery and duplicate-handling protocol.

25

CHAPTER

Versions, Snapshots and Optimistic Concurrency

25.0 What this chapter gives you

1. A record can have more than one relevant version: the engine's internal versions, a saved business revision and the copy someone is editing on a screen. Confusing these versions creates subtle bugs.
2. You will trace snapshot visibility, understand why old readers affect cleanup, implement an explicit revision check and choose a conflict policy that does not silently discard someone's work.
3. The examples introduce a new draft quotation and do not change the agreed historical prices in O-1042 or O-1043. Engine version metadata is not treated as an everlasting business audit log.

25.1 Old and new row versions

25.1.1 PLAIN — in simple words

1. When a database changes a record, an earlier version may remain available for readers that still need it. The engine can decide which version each reader is allowed to see.
2. This helps readers and writers coexist without requiring every reader to wait for every change. It does not mean that all old versions are kept forever.
3. A business history has a different purpose. If the shop must explain who approved an agreed price, it needs deliberately recorded history and retention rules rather than assuming the database's temporary internal versions will answer later.

25.1.2 PLAIN — a picture in your head

1. Imagine a shared document where a reader can finish the edition they opened while an editor prepares a newer edition for later readers.
2. The printing office keeps old sheets only as long as its current readers require them. An archive keeper has a separate job if editions must be preserved for years.
3. **Where the comparison breaks:** a database does not necessarily copy an entire document for each edit. Physical version storage and visibility checks vary by engine, and cleanup can reclaim old storage.

25.1.3 PLAIN — a worked example

1. In a new draft example, D-EDIT has quantity 3 at business revision 7. A transaction changes it to quantity 4 at revision 8.

2. An older permitted snapshot may still read the earlier quantity while a later snapshot reads the committed replacement. The application revision identifies the edit state chosen by the application; the engine may use other internal metadata to decide visibility.
3. After old snapshots are no longer relevant, the engine may reclaim obsolete versions. A future request for "the reason revision 7 existed" cannot rely on that reclaimed physical version.
4. If reasons and approvals matter, record them in an explicit history model with a defined scope and retention policy.

■ 25.1.4 PLAIN — what is really happening inside

1. Multi-version concurrency control, or MVCC, associates versions with transactional visibility information. Readers evaluate visibility according to their snapshot rules.
2. An UPDATE can therefore create storage and cleanup work beyond changing one displayed number. Index maintenance and space reuse also depend on the engine's implementation.
3. MVCC is not one universal algorithm. Use the general concept to ask useful questions, then consult the actual engine for version layout, cleanup and conflict behaviour.

■ 25.1.5 TECHNICAL — the engineer's version

1. PostgreSQL uses MVCC so statements can see appropriate snapshots while concurrent transactions modify data. Its internal tuple metadata participates in visibility, but it is not a substitute for application-level history. [S119] [S124]
2. PostgreSQL's `ctid` identifies a physical row-version location and can change. Transaction identifiers also have lifecycle and wraparound considerations. Neither should casually replace an immutable business key or an explicitly designed revision protocol. [S124] [S123]
3. An application revision should have a documented domain, increment rule and generation boundary. Its correctness depends on every relevant writer advancing or otherwise respecting it.

■ 25.1.6 WORDS — remember these

1. **MVCC:** let readers use suitable versions while changes occur — multi-version concurrency control using versioned data and transactional visibility rules. **Business revision:** the edit state named by the application — an explicit version value used for conflict detection or documented history. **Physical row location:** where one stored version currently resides — engine metadata such as PostgreSQL's `ctid`, not an enduring logical identity.

25.2 Snapshot visibility

■ 25.2.1 PLAIN — in simple words

1. A snapshot decides which committed work from other transactions belongs in a reader's view. "Newest record on disk" is not necessarily the version that reader should receive.
2. A reader can therefore obtain an older valid version without receiving a corrupt or random answer. The answer must be interpreted under the reader's observation contract.
3. A screen's age is another matter. After the database returns a result, the browser can keep displaying it indefinitely. Database snapshot guarantees do not continuously refresh every previously returned screen.

25.2.2 PLAIN — a picture in your head

1. A visitor enters an exhibition arranged as it was at opening time. New exhibits can be prepared for later visitors without rearranging the earlier visitor's tour halfway through.
2. Once the visitor leaves with a photograph, the exhibition staff no longer control how long they keep using that photograph.
3. **Where the comparison breaks:** snapshots also interact with a transaction's own writes and conflict checks. They are not merely wall-clock photographs, and not every engine starts them at BEGIN.

25.2.3 PLAIN — a worked example

1. Reader R establishes a retained snapshot while D-EDIT is at revision 7. Writer W commits revision 8. R may still read revision 7 under the retained-snapshot contract, while a new reader N sees revision 8.
2. R's user later submits an edit based on revision 7. The server should not assume the submitted revision is current merely because it once came from a valid database read.
3. A write protocol can require "apply this change only if the current business revision is still 7." If revision 8 now exists, return a conflict and preserve the user's proposed change for review.
4. This separates reading an old valid view from overwriting newer work without permission.

25.2.4 PLAIN — what is really happening inside

1. The snapshot belongs to the database operation or transaction, not to every future action of the client that received its result.
2. A revision returned with a resource is therefore useful as a precondition for a later update. It connects the update to the state on which the editor actually based the change.
3. A revision supplied by a client is still untrusted input. Validate its type and scope, and authorize the action independently before treating it as a precondition.

25.2.5 TECHNICAL — the engineer's version

1. PostgreSQL Read Committed and Repeatable Read establish and reuse snapshots differently, as detailed in Chapter 23. An ordinary earlier SELECT does not reserve its observed version for a later independent HTTP request. [S66]
2. SQLite WAL readers can retain a snapshot while another connection commits changes. A reader attempting to upgrade a stale snapshot to a writer can encounter `SQLITE_BUSY_SNAPSHOT`; restarting the transaction is different from overwriting the newer state. [S54] [S77]
3. Treat a client edit as `(resource identity, expected revision, proposed change)`, plus separately established authority. Do not use the revision field as a password or authorization token.

25.2.6 WORDS — remember these

1. **Visible version:** the row state allowed by a reader's snapshot — a version satisfying the engine's transaction-visibility rules. **Edit precondition:** the state an update expects to replace — a condition checked at the write boundary to detect intervening changes. **Stale client copy:** an earlier result still held outside the database — data whose original read validity does not guarantee current edit validity.

25.3 Cleanup and long readers

25.3.1 PLAIN — in simple words

1. Keeping every internal version forever would consume growing amounts of storage. Engines need to reclaim versions that no relevant reader or recovery rule still requires.
2. A long-running transaction can delay that cleanup because its older view may still need versions newer readers no longer use.
3. The operational cost is easy to miss: the reader may appear quiet while keeping old storage relevant. A dashboard showing only active query duration can overlook an idle open transaction.

25.3.2 PLAIN — a picture in your head

1. A library keeps an old edition on a return trolley because one registered reader still has a claim to consult it. Hundreds of newer editions cannot be fully cleared while that claim remains open.
2. Closing the reader's session can allow the cleanup process to proceed under its rules.
3. **Where the comparison breaks:** version cleanup is not simply deleting the oldest timestamp. Visibility horizons, replication and engine-specific transaction metadata can affect what is safe to reclaim.

25.3.3 PLAIN — a worked example

1. Suppose a synthetic workload replaces 50,000 rows per hour, each producing 200 bytes of obsolete row-version payload in a simplified accounting model. That is 10,000,000 bytes per hour before indexes, page overhead and reuse.
2. If a long reader prevents relevant cleanup for six hours, the simplified payload accumulation is 60,000,000 bytes, about 60 MB decimal. This is a planning calculation, not a measured PostgreSQL storage prediction.
3. Real storage growth must be measured because row layout, updates, indexes, fill space and vacuum behaviour change the result.
4. The lesson is not that every long reader causes exactly 60 MB of bloat. It is that reader lifetime can become a storage and maintenance input.

25.3.4 PLAIN — what is really happening inside

1. PostgreSQL vacuuming reclaims space from obsolete row versions when it is safe to do so and performs other essential maintenance, including transaction-ID-related work.
2. Ordinary VACUUM commonly makes space reusable inside the table rather than promising that the operating-system file immediately shrinks by the same amount.
3. Cancelling a long reader is an operational decision with consequences. Identify its owner, purpose and restart/recovery behaviour before treating it as disposable.

25.3.5 TECHNICAL — the engineer's version

1. PostgreSQL's routine vacuuming documentation covers dead-row-version cleanup, planner statistics and transaction-ID wraparound protection. Autovacuum is a maintenance system, not a guarantee that every workload remains healthy without monitoring. [S123]
2. Distinguish reusable free space, dead tuples, table-file size and live logical data size. They are related but not interchangeable measurements.

3. Monitor transaction age and relevant maintenance progress under the chosen engine. Avoid prescribing a global timeout or aggressive cleanup command without workload evidence and a reviewed rollback or restart plan.

25.3.6 WORDS — remember these

1. **Obsolete version:** an earlier stored state no longer needed by relevant readers — a version eligible for reclamation under the engine's visibility and maintenance rules. **Vacuuming:** maintenance that reclaims and manages old storage — PostgreSQL's process for handling obsolete tuples and other required maintenance tasks. **Cleanup horizon:** the boundary beyond which old state can be reclaimed — the visibility or retention limit constraining safe version removal.

25.4 Version columns

25.4.1 PLAIN — in simple words

1. A version column gives an editable record a counter or other explicit revision marker. The server advances it whenever a change relevant to the editing contract is accepted.
2. A client sends the revision it originally read. The server compares that expectation with the current revision before applying the proposed replacement.
3. The counter works only when its rules are followed consistently. A maintenance script that changes the record without advancing the revision can make an old client appear current.

25.4.2 PLAIN — a picture in your head

1. A quotation has an edition number printed in its corner. Dev's correction says, "This change is based on edition seven."
2. If Mira has already approved edition eight, the desk does not silently paste Dev's old replacement over it. The difference must be resolved.
3. **Where the comparison breaks:** an edition counter can wrap, reset or be reused after deletion. A complete protocol must define the resource's lifetime as well as its current number.

25.4.3 PLAIN — a worked example

1. D-EDIT starts with quantity=3 and revision=7. Mira submits quantity=4 with expected revision=7. The accepted update sets quantity=4 and revision=8.
2. Dev then submits quantity=5 with expected revision=7. The revision condition no longer matches, so his update affects zero rows and must not be reported as a successful replacement.
3. The response can return a conflict and the current revision, subject to access policy. Preserve Dev's proposed quantity so he can compare and intentionally resubmit against the newer state.
4. This policy rejects stale replacement; it does not decide whether quantity 4 or 5 is the better business choice.

25.4.4 PLAIN — what is really happening inside

1. The revision check and increment belong in the same protected write. Reading the revision first and then performing an unconditional replacement reintroduces the gap.
2. Define which changes advance the revision. A counter for editable quotation fields might intentionally ignore a background "last viewed" timestamp, but that decision must be explicit.

3. Bound the revision domain and handle exhaustion rather than wrapping silently into a previously used value. Large counters make exhaustion unlikely; they do not remove the need for a defined limit.

25.4.5 TECHNICAL — the engineer's version

1. A typical pattern is `UPDATE ... SET quantity=?, revision=revision+1 WHERE id=? AND revision=?`. The uniqueness of `id`, bounded revision domain and affected-row check are part of the protocol.
2. Add a generation identifier when a logical identifier can be deleted and recreated. Compare `(resource_id, generation_id, revision)` so a new object's revision 1 is not confused with the old object's revision 1.
3. Do not substitute PostgreSQL `xmin` or `ctid` casually for this application contract. Their implementation purpose and lifecycle differ from a deliberately stable business revision. [S124] [S123]

25.4.6 WORDS — remember these

1. **Version column:** a field naming the accepted edit state — an application-managed revision value checked and advanced by relevant writes. **Generation identifier:** distinguish separate lifetimes of a reused key — an immutable identity component that changes when a resource is recreated. **Revision exhaustion:** no unused value remains in the chosen domain — a boundary requiring an explicit policy rather than silent counter reuse.

25.5 Compare-and-set

25.5.1 PLAIN — in simple words

1. Compare-and-set means "make this change only if the current state still matches my expectation." It turns an old observation into an explicit precondition instead of an unconditional command.
2. Comparing only the displayed value may miss intervening changes that return to the same value. A revision can reveal that history of accepted edits even when the final quantity looks unchanged.
3. A successful comparison still does not authenticate the caller, validate every business rule or identify a repeated request. Those are separate controls.

25.5.2 PLAIN — a picture in your head

1. Dev leaves a note saying, "Change the sign only if it is still the edition I saw." That is stronger than saying, "Change it if it still happens to display the same number."
2. The sign could have changed twice and returned to that number while the edition advanced.
3. **Where the comparison breaks:** a counter records only changes that the protocol includes. It is not an omniscient record of every physical event or bypass mutation.

25.5.3 PLAIN — a worked example

1. Start with quantity=3, revision=7. An accepted edit changes it to quantity=4, revision=8. Another accepted edit changes it back to quantity=3, revision=9.
2. A stale client expecting quantity=3 passes a value-only comparison even though two edits intervened. This is the ABA pattern: A became B and later became A again.
3. A stale client expecting revision=7 fails against revision=9. The explicit revision detected intervening protocol-respecting edits.

4. If the record was deleted and recreated with revision=7 again, the revision alone could mislead the client. A fresh generation identifier separates the lifetimes.

25.5.4 PLAIN — what is really happening inside

1. The write predicate encodes the expectation; the database decides whether the current target satisfies it at the protected write boundary.
2. On failure, avoid guessing the reason from zero affected rows alone. Missing identity, different generation, stale revision or a scope condition can all make the predicate fail.
3. After a successful write whose response is lost, replaying the old expected revision will usually conflict. That conflict does not prove the original attempt failed. Stable request identity is needed to resolve the original outcome.

25.5.5 TECHNICAL — the engineer's version

1. The educational compare-and-set helper uses bound values, a complete identity, a positive bounded proposed quantity and an expected revision. It owns a transaction and accepts exactly one changed row under its schema.
2. The protocol protects stale replacement of that resource. It does not enforce unrelated cross-row limits or prevent authorized users from intentionally making undesirable but permitted changes.
3. Separate optimistic concurrency from idempotency: the former detects conflicting edit state; the latter identifies repeated attempts at the same intent and resolves their outcomes. Chapter 26 combines these concerns where appropriate. [S126]

25.5.6 WORDS — remember these

1. **Compare-and-set:** write only when the expectation still matches — an atomic conditional mutation based on an expected value, revision or generation. **ABA problem:** a value changes and returns, hiding intervening work — a limitation of equality-only checks that do not identify the relevant state history. **Optimistic concurrency:** detect a conflict when saving rather than holding a long edit lock — a protocol that validates an earlier observation at the write boundary.

25.6 Choosing a conflict policy

25.6.1 PLAIN — in simple words

1. Detecting a conflict is not the end of the user experience. The application must decide whether to reject, ask for review, merge safely or retry a newly computed operation.
2. Different data needs different treatment. Two independent additions to a counter can sometimes combine naturally; two replacements of a delivery address may require a person to choose.
3. "Last write wins" is a policy that discards something when changes compete. It can be appropriate for selected low-risk data, but it should never be mistaken for the absence of data loss.

25.6.2 PLAIN — a picture in your head

1. Two editors correct different spelling errors in a document; a careful merge may keep both. Two editors choose different delivery destinations; combining the street from one and the city from the other can create a destination neither intended.
2. The merge rule must understand the meaning of the fields, not just avoid a software error.
3. **Where the comparison breaks:** automatic text merging can still produce a syntactically valid but semantically wrong result. Business invariants need explicit validation after any merge.

25.6.3 PLAIN — a worked example

1. Mira changes a draft quantity from 3 to 4. Dev changes the same draft's delivery note, both from revision 7.
2. A whole-record revision policy rejects Dev's stale update even though the fields differ. This is conservative and simple, but may create avoidable review work.
3. A field-aware merge can preserve both only if the application checks that the changes are independent under its rules. If the note says "pack exactly three," it depends on quantity and is not independent after all.
4. Record the chosen resolution and resulting revision. Do not hide an automatic overwrite behind a generic "synchronised" message.

25.6.4 PLAIN — what is really happening inside

1. Conflict granularity is a design trade-off. One revision per document is easier to reason about; finer-grained revisions may reduce false conflicts while increasing metadata and invariant complexity.
2. A retry of a commutative operation, such as adding a separately identified increment, differs from retrying an absolute replacement. Duplicate delivery still needs its own identity handling.
3. Preserve the user's unsaved proposal and show the relevant current state only within their permissions. Conflict messages must not become a way to reveal another tenant's records.

25.6.5 TECHNICAL — the engineer's version

1. Specify the conflict response schema, retry eligibility, merge rules, audit requirements and authorization checks. A version mismatch should have a stable API meaning rather than being flattened into an unexplained server error.
2. Tests should cover two stale editors, ABA, identity recreation, bypass-writer assumptions, response loss, malformed revision and unauthorized scope. The provided lab exercises a trusted local data protocol, not a complete authenticated API.
3. Evaluate conflict rate and user resolution cost alongside throughput. Reducing visible conflicts by silently overwriting edits is not a correctness improvement.

25.6.6 WORDS — remember these

1. **Conflict resolution:** decide what to retain after incompatible changes — a policy for rejection, reviewed replacement, safe merge or recomputation. **Merge granularity:** the unit at which changes are compared — a document, row, field or operation boundary used by the conflict protocol. **Last-write-wins policy:** accept the selected later replacement — a conflict rule that can discard competing state and requires a defined ordering basis.

25.97 Practice and worked answers

1. **Separate version purposes.** Can internal MVCC versions replace an approval history? **Answer:** no. Cleanup and implementation lifecycles differ from a deliberate business-history and retention contract.
2. **Name a stable key.** Should PostgreSQL `ctid` identify a customer's permanent account? **Answer:** no. It identifies a physical row-version location, not an immutable business identity.
3. **Trace an edit conflict.** Quantity 3/revision 7 becomes 4/revision 8. A client still expects revision 7. **Answer:** the conditional replacement should fail without overwriting revision 8.

4. **Find ABA.** Quantity changes $3 \rightarrow 4 \rightarrow 3$. **Answer:** a quantity-only comparison can miss both edits. A consistently advanced revision distinguishes the states.
5. **Find identity reuse.** A deleted key is recreated with the old revision number. **Answer:** include a new generation identifier or otherwise prohibit ambiguous lifetime reuse.
6. **Interpret a lost response.** A saved edit advances revision 7 to 8, but the caller receives no reply. **Answer:** retrying expected revision 7 can conflict even though the original succeeded. Resolve request identity rather than treating the conflict as proof of original failure.
7. **Calculate simplified old-version payload.** Fifty thousand changes/hour at 200 bytes for six hours. **Answer:** 60,000,000 bytes before overhead and reuse; this is not a measured engine-size forecast.
8. **Review a merge.** One user changes quantity and another changes a note saying "pack exactly three." **Answer:** the fields have a semantic dependency. A blind field merge can violate the intended meaning.

25.98 Common wrong ideas

1. **Wrong:** MVCC means every historical row is kept forever. **Right:** internal versions are subject to visibility and cleanup rules.
2. **Wrong:** a physical row location is a permanent record identity. **Right:** storage location and business identity serve different purposes.
3. **Wrong:** an earlier valid read authorizes a later overwrite. **Right:** later writes need current preconditions and independent authority checks.
4. **Wrong:** VACUUM always shrinks the table file immediately. **Right:** ordinary maintenance often makes internal space reusable instead.
5. **Wrong:** comparing the current value detects every intervening edit. **Right:** ABA can return to the same value.
6. **Wrong:** a revision counter works even when some writers ignore it. **Right:** every relevant mutation must follow the revision contract.
7. **Wrong:** optimistic concurrency is idempotency. **Right:** stale-state detection and repeated-intent resolution solve different problems.
8. **Wrong:** automatic merging is always more correct than rejecting. **Right:** the merge must preserve the meaning and invariants of the data.

25.99 Chapter summary in 20 lines

1. Internal row versions, business revisions and client copies are different concepts.
2. MVCC uses versions and visibility rules to coordinate readers and writers.
3. An older visible version can be valid under a retained snapshot.
4. Internal version retention is not an everlasting business audit log.
5. Physical tuple locations are not permanent logical identities.
6. A client's earlier read does not reserve the record for a later edit.
7. Return explicit revisions when later writes need stale-state detection.
8. Cleanup reclaims obsolete versions when relevant visibility rules permit it.
9. Long readers can delay reclamation and affect maintenance cost.
10. Reusable space and operating-system file size are different measurements.
11. A version column requires a defined increment and exhaustion policy.

12. Every relevant writer must follow the same revision contract.
13. Compare the expected revision and mutate the record in one protected operation.
14. Affected-row failure must not be reported as successful replacement.
15. ABA can hide intervening changes from value-only comparisons.
16. Generation identifiers distinguish separate lifetimes of a reused key.
17. A lost response can make a successful original edit appear as a later conflict.
18. Request identity resolves retry outcomes separately from edit concurrency.
19. Choose rejection, review or merge according to the data's meaning.
20. Preserve user proposals and validate invariants after conflict resolution.

26

CHAPTER

Invariants, Idempotency and the Outbox

26.0 What this chapter gives you

1. A reliable operation must survive more than a clean first attempt. Requests can arrive twice, replies can disappear, and external delivery can fail after the database has committed.
2. This chapter combines three ideas: an invariant defining what must stay true, an identity defining which attempts represent one intent, and an outbox recording external work implied by an accepted local change.
3. The resulting pattern has precise boundaries. It can make a bounded local reservation protocol repeatable; it does not establish universal exactly-once execution, physical-stock accuracy, authentication or fault-free external delivery.

26.1 Rules across operations

26.1.1 PLAIN — in simple words

1. An invariant is a rule that must hold across the relevant accepted operations. "Available stock never becomes negative" is one possible rule. "Every accepted reservation has a matching stock allocation" is another.
2. A workflow can satisfy one while breaking the other. Stock can remain non-negative even when the application forgets to record who reserved it.
3. Write the rules before selecting the mechanism. A transaction, a unique key and a queue each protect different boundaries; none is a general replacement for understanding the intended state changes.

26.1.2 PLAIN — a picture in your head

1. Mira keeps a stock ledger, a reservation register and a dispatch list. A useful rule connects them: every accepted reservation explains a stock reduction and creates the required confirmation task.
2. Checking each notebook separately is not enough. Their relationships are part of the work.
3. **Where the comparison breaks:** the notebooks can represent promises about records, but actual goods and delivered messages remain outside the local database. A consistent ledger is evidence about the ledger first.

26.1.3 PLAIN — a worked example

1. In a new local teaching system, available stock begins at 5. Request IDEM-A asks for 3. Acceptance should leave available=2, one accepted request outcome for 3, one reservation and one outbox message identity.

2. A repeated delivery of IDEM-A should return its recorded outcome without reducing stock again or creating a second confirmation intent.
3. A different request IDEM-B asking for 3 should be rejected while only 2 remain. A duplicate of IDEM-B should follow the explicitly chosen rejection-replay policy rather than unexpectedly becoming accepted later.
4. These example quantities do not rewrite the canonical agreed orders or explain the earlier unknown stock discrepancy.

26.1.4 PLAIN — what is really happening inside

1. The protocol connects request identity to the state transition it authorizes. Unique constraints help prevent two concurrent records claiming the same identity, while a transaction groups the related local writes.
2. The code must handle accepted, rejected and exceptional outcomes distinctly. A deliberate business rejection can be recorded; an unexpected exception should not accidentally create a successful-looking outcome.
3. Cancellation, replenishment and correction need their own rules. Adding idempotency to one endpoint does not prove that every other mutation preserves the same invariant.

26.1.5 TECHNICAL — the engineer's version

1. Define the operation as a state transition over `(stock, requests, reservations, outbox)`. State the preconditions and which fields may change for accepted, rejected and replayed operations.
2. In the educational implementation, a single owned SQLite write transaction coordinates the guarded stock update and the relevant request/outbox records. SQLite writer serialization is part of this local implementation, not a claim about a distributed service. [S25] [S54]
3. Add reconciliation identities, such as initial stock plus replenishments minus accepted reservations plus valid releases equals expected available stock, under explicitly bounded record classes. Unrecorded physical loss remains outside that equation's evidence.

26.1.6 WORDS — remember these

1. **Invariant:** a rule accepted operations must preserve — a predicate over the relevant system state that the protocol is designed to maintain. **State transition:** one defined move from an old state to a new state — an operation with explicit preconditions, writes and outcomes. **Reconciliation identity:** an equation connecting related records — a check that quantities and recorded state changes agree under stated inclusions and exclusions.

26.2 Request identity

26.2.1 PLAIN — in simple words

1. A retry needs a stable name for the original intent. Without that name, the server may be unable to distinguish "try that same reservation again" from "make another identical reservation."
2. Two equal payloads are not automatically the same intent. A customer can legitimately place two separate orders for the same product and quantity.
3. Scope the name. The same local request string used by two different tenants should not make their operations collide or reveal one tenant's outcome to another.

26.2.2 PLAIN — a picture in your head

1. A customer receives a claim ticket for one request. Presenting the same ticket asks about the same decision; receiving a new ticket creates a separately identified request.
2. The ticket belongs to a particular desk and customer scope. Ticket 17 at another shop is not necessarily your ticket 17.
3. **Where the comparison breaks:** a request key is not automatically a secret or proof of ownership. Authorization must be established independently before the server reveals or changes anything associated with it.

26.2.3 PLAIN — a worked example

1. Tenant T-A submits key REQ-17 for branch BR-A, product P-DEMO and quantity 3. A network retry repeats the same scoped key and the same validated meaning.
2. A new purchase with the same branch, product and quantity uses a new intent identity, such as REQ-18. Deduplicating solely by a hash of the payload would wrongly collapse these legitimate separate purchases.
3. Tenant T-B can independently use REQ-17 under a key scoped by tenant. A request outcome lookup must include the tenant established by the trusted service boundary, not merely a client-supplied string.
4. The companion's local function accepts a trusted tenant parameter for teaching; it does not implement that authentication boundary.

26.2.4 PLAIN — what is really happening inside

1. Validate the input into a well-defined semantic representation before comparing attempts. Decide whether whitespace, field order or equivalent numeric spelling changes meaning under the API contract.
2. Store enough validated request meaning to detect incompatible reuse. A digest can help indexing or comparison, but a collision-prone shortcut must not replace the equivalence rule.
3. The server must define who generates keys, their allowed length, their retention and whether a key can ever be reused. An unlimited arbitrary string is also an input-resource concern.

26.2.5 TECHNICAL — the engineer's version

1. Model identity as a tuple such as `(tenant_id, operation_kind, request_key)`. Include additional scope only when required by the contract, and enforce the tuple's uniqueness in storage.
2. Idempotent APIs commonly distinguish caller-provided request intent from merely equal parameters. The equivalence and key-retention contract is central to safe retries. [S126]
3. Canonicalization is not generic "sort JSON and hash." It must preserve domain meaning, reject ambiguous duplicate members where relevant, represent units explicitly and avoid silently equating values that the application treats differently. Chapters 3, 5 and 7 supply the prerequisites.

26.2.6 WORDS — remember these

1. **Request identity:** the stable name of one intended operation — a scoped identifier linking repeated attempts to one business decision. **Payload equivalence:** when two attempts mean the same thing — an explicit comparison contract over validated request semantics. **Identity scope:** the domain in which a key is unique — the tenant, operation and other components defining a request key's namespace.

26.3 Replay detection

26.3.1 PLAIN — in simple words

1. When a known request returns, the server checks whether it represents the same intent and then returns the recorded outcome. It does not perform the business change again merely because the caller asked again.
2. Reusing the same key with different meaning is a conflict, not a request to rewrite history. The server should reject the incompatible reuse under a clear contract.
3. Decide what happens to recorded rejections. In this book's local protocol, a valid business rejection is stable for that request identity. A later change in stock does not silently change the old decision.

26.3.2 PLAIN — a picture in your head

1. A reservation desk keeps the decision attached to the claim ticket. Returning with the ticket retrieves that decision.
2. Changing the requested quantity while presenting the same ticket is not merely asking again. It is attempting to attach a different request to an existing identity.
3. **Where the comparison breaks:** some real APIs deliberately choose other policies, including retention limits or selected retryable rejections. Those choices must be documented; this chapter's policy is not a universal standard.

26.3.3 PLAIN — a worked example

1. IDEM-A with quantity 3 is accepted from stock 5, leaving 2. Replaying IDEM-A with quantity 3 returns the same accepted outcome and leaves stock 2.
2. IDEM-A with quantity 4 conflicts with the stored request meaning. It must neither allocate a fourth unit nor overwrite the original record.
3. IDEM-B with quantity 3 is rejected for insufficient stock. Suppose a separate replenishment later raises available stock to 7. Replaying IDEM-B still returns its original rejection under the chosen contract.
4. A caller who now intends a new allocation submits a new request identity. That is an explicit new decision, not an automatic retry that changes the meaning of the old outcome.

26.3.4 PLAIN — what is really happening inside

1. Replay detection must occur within a protocol that coordinates competing first attempts. A lookup followed by an unprotected insertion has the same check-then-insert race discussed in Chapter 22.
2. Storing an outcome separately after the stock commit leaves a gap: a replay could see no outcome even though stock was already reduced. Group the outcome and business mutation atomically.
3. Do not return another scope's result to resolve a conflict. Identity comparison, authorization and response filtering all remain relevant on the replay path.

26.3.5 TECHNICAL — the engineer's version

1. The local example persists both accepted and valid business-rejected outcomes, with normalized payload fields. A unique request key and an owned transaction prevent inconsistent duplicate outcome records within the tested database protocol.

2. Unexpected exceptions roll back that attempt's local changes. They are not converted into stable success or an invented business rejection. An externally unknown commit outcome is resolved by querying the same request identity.
3. Retention bounds the deduplication horizon. Deleting request records while late retries remain possible can allow a formerly recognized intent to be treated as new. State that limit explicitly. [S126]

26.3.6 WORDS — remember these

1. **Replay:** another attempt carrying the same defined intent — a repeated request that should resolve through its existing identity and outcome. **Incompatible key reuse:** one identity is presented with different meaning — a conflict requiring rejection or explicit reconciliation, not silent replacement. **Deduplication horizon:** how long repeated intent remains recognizable — the retention interval and protocol conditions under which replay records are available.

26.4 Atomic intent recording

26.4.1 PLAIN — in simple words

1. The reservation, its request outcome and its required message intent should be accepted together when the protocol requires all three.
2. This does not mean the message is already delivered. It means that a committed reservation has a durable local record of the delivery work that follows from it.
3. The order of application steps matters. A failure at any pre-commit point must not leave an accepted half-operation in the database.

26.4.2 PLAIN — a picture in your head

1. Mira accepts a reservation only when its stock entry, decision slip and dispatch instruction are filed as one case.
2. The messenger may arrive later, but the instruction no longer depends on Mira remembering it after a crash or interruption.
3. **Where the comparison breaks:** an in-memory educational database does not demonstrate physical durability. The protocol's atomic grouping can be tested locally while durable storage assumptions remain a separate chapter's responsibility.

26.4.3 PLAIN — a worked example

1. Begin an owned transaction. Look up the scoped request key. If it exists, compare meaning and return its recorded outcome without another allocation.
2. For a new valid request, attempt the guarded stock reduction. If accepted, insert the reservation, accepted outcome and one outbox row. If rejected under the business contract, insert the stable rejected outcome without a reservation or accepted-confirmation intent.
3. Commit only after every required local write succeeds. Inject an exception after stock reduction and verify that rollback removes all changes from that attempt.
4. The transaction is not allowed to call an email provider before commit. It records what should be sent, not an external send that rollback cannot undo.

26.4.4 PLAIN — what is really happening inside

1. An outbox record needs a stable message identity, type, relevant payload or durable reference, and enough state to manage delivery attempts under the chosen design.
2. Payload design has retention consequences. Copying an entire customer profile into every message makes later correction or deletion more complicated than sending a bounded required subset.
3. A worker that reads the outbox also needs a coordination protocol if several workers can claim the same work. This book's simplest local dispatcher is deliberately single-worker; it does not imply a production lease implementation.

26.4.5 TECHNICAL — the engineer's version

1. The transactional outbox places business state and delivery intent in the same local atomic boundary, avoiding the basic dual-write gap between a database commit and an unrelated message send. [S125]
2. Use constraints to connect outbox identity and request identity as required. Do not regenerate a new message identifier on every delivery attempt; that defeats consumer deduplication by message identity.
3. For multiple dispatchers, specify claiming, lease expiry, crash recovery, ordering and fencing where needed. Merely adding a `sending=true` flag can strand work permanently after a worker dies.

26.4.6 WORDS — remember these

1. **Atomic intent recording:** accept the business change and its follow-up instruction together — committing state and outbox intent in one local transaction. **Dual-write gap:** one system changes while another does not — a failure window between separately committed operations in different resources. **Dispatcher:** the worker attempting recorded external tasks — a process that reads delivery intent and manages sends and acknowledgements.

26.5 Outbox delivery and consumers

26.5.1 PLAIN — in simple words

1. A dispatcher can send the same message more than once if it loses certainty about a previous attempt. The receiver must know what repeated delivery means.
2. A database-backed consumer can record "I processed message M" together with its local database effect. Replaying M then finds the record and avoids applying that effect again.
3. This protects the consumer's participating local database work. It does not make a separate external effect, such as sending a physical parcel, part of that database transaction.

26.5.2 PLAIN — a picture in your head

1. The receiving desk stamps a message identity into a register at the same time it files the corresponding local instruction. A second copy with the same identity finds the stamp.
2. If the desk crashes before accepting either, it can safely try again. If it accepted both, it can recognize the replay.
3. **Where the comparison breaks:** a stamp made before an external action can falsely suggest completion, while a stamp made after it can leave a duplication gap. The atomic boundary must contain the actual effect you claim to deduplicate.

26.5.3 PLAIN — a worked example

1. Outbox message MSG-A represents IDEM-A's accepted reservation. The dispatcher sends it, and the consumer commits an inbox record plus a local notification-task record.
2. Before the dispatcher marks delivery complete, it crashes. On restart it sends MSG-A again.
3. The consumer finds MSG-A in its inbox with equivalent meaning and returns the recorded result without creating another notification task. There are two deliveries but one committed local task.
4. If that task later sends an email, email delivery remains another boundary. The local inbox result does not prove that exactly one email reached a person.

26.5.4 PLAIN — what is really happening inside

1. Inbox identity and effect must share the consumer's transaction. Marking a message processed first and performing the effect later can lose work; performing the effect first and marking later can duplicate it.
2. A duplicate identity carrying different payload is suspicious or incompatible under the protocol. Do not silently treat it as an equivalent replay simply because the identifier matches.
3. Order is separate from duplication. A consumer may need per-order sequence checks so "cancel reservation" does not apply before the reservation it references. A global message ID alone does not establish causal order.

26.5.5 TECHNICAL — the engineer's version

1. Consumer-side deduplication typically uses a unique inbox key and a transaction containing both the inbox decision and the local effect. Its horizon is bounded by retained identity records and participating storage guarantees. [S125]
2. An acknowledgement should correspond to the consumer's defined accepted boundary. A queue acknowledgement sent before the local effect commits can lose work if the consumer then fails.
3. Model delivery as at-least-once attempts with possible unknown outcomes unless the actual infrastructure contract establishes something else. "Exactly once" must name the effect, identity scope, storage boundary and retention interval it describes.

26.5.6 WORDS — remember these

1. **Inbox record:** remember which message identity was accepted — a consumer-side deduplication record committed with the participating local effect. **At-least-once delivery:** retries may deliver a message repeatedly — a delivery contract that prioritizes eventual attempts while permitting duplicates under its stated assumptions. **Acknowledgement boundary:** what a positive receipt actually confirms — the defined acceptance point represented by a queue, consumer or provider response.

26.6 End-to-end limits

26.6.1 PLAIN — in simple words

1. The complete journey contains several boundaries: caller intent, database acceptance, outbox dispatch, consumer acceptance and any later real-world action.
2. A local guarantee should be described at its own boundary. Extending it verbally to the whole journey does not make the missing coordination appear.

3. Strong engineering names the remaining uncertainty and provides recovery procedures. It does not hide uncertainty behind a confident success label.

26.6.2 PLAIN — a picture in your head

1. A parcel has separate records for booking, warehouse acceptance, courier collection and recipient delivery. A booking receipt is useful, but it is not proof that the parcel reached its destination.
2. Each handover needs a defined meaning and a way to investigate missing evidence.
3. **Where the comparison breaks:** this book's reservation lab does not implement a logistics system. The parcel story illustrates boundaries, not a tested physical delivery protocol.

26.6.3 PLAIN — a worked example

1. IDEM-A commits locally, but the response is lost. The caller retries the same identity and learns the accepted outcome. No second allocation occurs under the tested local protocol.
2. MSG-A is delivered twice, but the consumer's inbox and local task transaction apply one local task. The dispatcher eventually records its acknowledgement.
3. A later email provider returns an ambiguous timeout. Without a suitable provider identity contract or a reconciliation interface, the system cannot infer from that timeout whether the email was accepted.
4. The honest status separates "reservation accepted," "notification task recorded," "provider outcome unresolved" and "delivery verified," rather than collapsing all four into one invented certainty.

26.6.4 PLAIN — what is really happening inside

1. Retention, permissions and failure domains constrain the protocol. Erasing an inbox record can remove replay protection; losing both the business database and its only backup can remove outcome evidence.
2. Replays must remain authorized. Knowing an old request key does not grant permanent access after a user's rights change.
3. Operational recovery needs bounded procedures: identify stuck intents, verify payload identity, inspect consumer state, decide safe replay scope and preserve the evidence of the decision.

26.6.5 TECHNICAL — the engineer's version

1. The supplied implementation is a local educational state machine. It uses synthetic records and trusted function parameters; it is not a multi-process lease service, authenticated API, immutable audit store or proof of durable power-loss behaviour.
2. Test invariants after every injected failure point, replay and incompatible key reuse. Include stable rejection replay, inbox payload mismatch, tenant scope and stock reconciliation. A passing test means its stated behaviour occurred, not that unseen external systems were verified.
3. Before production, separately review transaction isolation, key retention, worker coordination, external provider contracts, privacy, monitoring and restore evidence. Idempotency is a protocol property with assumptions, not a decorative endpoint label. [S126]

26.6.6 WORDS — remember these

1. **End-to-end boundary:** the complete chain whose result is claimed — all participating systems and effects required for a stated business outcome. **Outcome reconciliation:** resolve uncertainty

using recorded identity and state — a procedure for determining what a prior attempt actually accepted. **Scoped guarantee:** a promise with explicit limits — a correctness claim tied to defined effects, participants, failures and retention assumptions.

26.97 Practice and worked answers

1. **Name the invariant.** Stock falls from 5 to 2, but no reservation exists. **Answer:** non-negative stock holds, but the required allocation-to-reservation relationship fails. One invariant does not imply the other.
2. **Separate equal requests.** A customer intentionally buys the same three pens twice. **Answer:** equal payloads can represent two legitimate intents. Give the intents separate request identities.
3. **Replay acceptance.** IDEM-A for 3 is accepted once and delivered again. **Answer:** return its stored outcome without a second subtraction or a new outbox identity.
4. **Reject incompatible reuse.** IDEM-A returns with quantity 4. **Answer:** reject the mismatched meaning under the protocol; do not replace the original request record.
5. **Replay rejection.** IDEM-B was rejected, then stock was replenished. **Answer:** under this book's explicit policy, the old identity retains its rejection. A genuinely new allocation request uses a new intent identity.
6. **Inject a local failure.** An exception occurs after stock reduction but before outbox insertion. **Answer:** the owned transaction rolls back all of that attempt's local changes.
7. **Trace duplicate delivery.** MSG-A is sent twice after a dispatcher crash. **Answer:** an equivalent inbox identity and local effect in one consumer transaction can preserve one local task despite two deliveries.
8. **Limit the claim.** Does one local task prove exactly one delivered email? **Answer:** no. The provider and recipient boundaries require separate contracts and evidence.

26.98 Common wrong ideas

1. **Wrong:** a transaction automatically knows the business invariant. **Right:** the application must define and protect the relevant relationships.
2. **Wrong:** equal payload hashes always mean one intent. **Right:** identical purchases can be legitimate separate requests.
3. **Wrong:** a request key is an authorization token. **Right:** scope and permission checks remain independent.
4. **Wrong:** a duplicate key with different data should overwrite the old request. **Right:** incompatible identity reuse requires an explicit conflict policy.
5. **Wrong:** every rejected request should become successful when retried later. **Right:** replay semantics must be stable under the chosen contract.
6. **Wrong:** an outbox means the message has been delivered. **Right:** it records durable local intent within the participating storage boundary.
7. **Wrong:** a consumer inbox deduplicates arbitrary outside effects. **Right:** it protects only effects inside the same proven atomic boundary.
8. **Wrong:** exactly once is meaningful without naming the effect and scope. **Right:** define identity, participants, failure assumptions and retention horizon.

26.99 Chapter summary in 20 lines

1. Invariants define what accepted state transitions must preserve.
2. Related stock, reservation, outcome and message records need explicit relationships.
3. A retry must carry the stable identity of the original intent.
4. Equal payloads can represent different legitimate intents.
5. Scope request identity by the required tenant and operation namespace.
6. Validate and compare request meaning under a documented equivalence contract.
7. Matching identity with different meaning is a conflict, not an ordinary replay.
8. Store the business outcome with the local state change atomically.
9. A replay returns the recorded outcome without performing the allocation again.
10. This book's valid business rejections remain stable for their original request identity.
11. Unexpected pre-commit exceptions roll back the attempt's local changes.
12. A lost response after commit can be resolved through the same request identity.
13. An outbox records external work implied by an accepted local change.
14. Outbox intent is not evidence of completed external delivery.
15. Dispatch can duplicate messages after an ambiguous send or lost acknowledgement.
16. An inbox can deduplicate participating local consumer effects in one transaction.
17. Message identity does not by itself guarantee causal ordering.
18. Retention bounds how long replay protection and outcome evidence remain available.
19. Authorization and privacy checks apply to replays as well as first attempts.
20. State every guarantee at its actual boundary and preserve unresolved outcomes honestly.

Companion Labs

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

Start in an isolated folder

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

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

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

What is implemented

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

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

The advanced models

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

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

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

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

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

The capstone boundary

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

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

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

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

Reading the test record

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

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

A–Z Glossary

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

A

ABA problem

a value changes and returns, hiding intervening work — a limitation of equality-only checks that do not identify the relevant state history.

Read in context: 25.5.6

Acknowledgement boundary

what a positive receipt actually confirms — the defined acceptance point represented by a queue, consumer or provider response.

Read in context: 26.5.6

Acquisition order

the sequence in which resources are requested — a shared ordering protocol intended to prevent cyclic waits.

Read in context: 24.4.6

Advisory lock

a coordination agreement writers must choose to follow — an application-defined locking facility not automatically tied to every relevant data mutation.

Read in context: 24.1.6

Affected-row check

verify how many target records changed — inspection of the statement result against an expected cardinality.

Read in context: 22.3.6

At-least-once delivery

retries may deliver a message repeatedly — a delivery contract that prioritizes eventual attempts while permitting duplicates under its stated assumptions.

Read in context: 26.5.6

Atomic intent recording

accept the business change and its follow-up instruction together — committing state and outbox intent in one local transaction.

Read in context: 26.4.6

Atomicity

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

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

Read in context: 21.3.6

Attempt amplification

extra work caused by retries — the ratio or count of transaction attempts relative to completed business operations.

Read in context: 23.6.6

B

Backoff

wait longer before another attempt — a retry-spacing strategy that reduces repeated contention or overload.

Read in context: 24.5.6

Blocking

waiting for conflicting access to be released — a resource wait that need not contain a dependency cycle.

progress waits for unavailable coordination — a limitation distinct from a participant process being physically stopped.

Read in context: 24.3.6

Business precondition

what must be true before an operation is accepted — an application rule that may require checking affected rows or queried state.

Read in context: 21.1.6

Business revision

the edit state named by the application — an explicit version value used for conflict detection or documented history.

Read in context: 25.1.6

C

Check-then-insert race

two requests act on the same observed absence — an unsafe gap between an existence query and an unprotected insertion decision.

Read in context: 22.4.6

Cleanup horizon

the boundary beyond which old state can be reclaimed — the visibility or retention limit constraining safe version removal.

Read in context: 25.3.6

Commit

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

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

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

Read in context: 21.2.6

Compare-and-set

write only when the expectation still matches — an atomic conditional mutation based on an expected value, revision or generation.

Read in context: 25.5.6

Compensation

a later action that addresses an earlier effect — a domain-specific response that need not restore the exact original world.

a new action addresses an earlier effect — an explicit corrective operation, not erasure of the original history.

a new action addressing an earlier completed action — an application-defined correction distinct from local transaction rollback.

Read in context: 21.6.6

Complete scope

all identity dimensions needed to select the intended record — the tenant, branch and local key components required by the data model.

Read in context: 22.3.6

Composite uniqueness

a combination must be distinct — a constraint enforcing uniqueness over a tuple of columns.

Read in context: 22.4.6

Conditional write

change a record only when its requirement still holds — a write guarded by a predicate evaluated under the database's concurrency rules.

Read in context: 22.3.6

Conflict policy

what to do when a rule prevents a write — an explicit decision to reject, reconcile, return an equivalent result or apply a defined update.

Read in context: 22.4.6

Conflict resolution

decide what to retain after incompatible changes — a policy for rejection, reviewed replacement, safe merge or recomputation.

Read in context: 25.6.6

Consistency

accepted states obey the specified rules — validity relative to actual constraints and invariants, not an assurance that every stored claim is true.

Read in context: 21.3.6

Cross-row invariant

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

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

Read in context: 22.5.6

D

Deadlock

a cycle of participants waiting on one another — a condition requiring some participant to release, abort or otherwise break the cycle.

Read in context: 24.3.6

Deadlock victim

the transaction selected to break a cycle — a participant aborted by the engine so other work can proceed.

Read in context: 24.4.6

Deduplication horizon

how long repeated intent remains recognizable — the retention interval and protocol conditions under which replay records are available.

Read in context: 26.3.6

Delivery gap

uncertainty between an external action and its recorded acknowledgement — a failure window that can require replay or reconciliation.

Read in context: 24.6.6

Dirty read

seeing a change before it commits — observation of another transaction's uncommitted data.

Read in context: 23.1.6

Dispatcher

the worker attempting recorded external tasks — a process that reads delivery intent and manages sends and acknowledgements.

Read in context: 26.4.6

Dual-write gap

one system changes while another does not — a failure window between separately committed operations in different resources.

Read in context: 26.4.6

Durability

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

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

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

Read in context: 21.4.6

E

Edit precondition

the state an update expects to replace — a condition checked at the write boundary to detect intervening changes.

Read in context: 25.2.6

End-to-end boundary

the complete chain whose result is claimed — all participating systems and effects required for a stated business outcome.

Read in context: 26.6.6

External effect

something changes beyond the local transaction — an action in another system or the physical world not automatically reversed by database rollback.

a change outside the local transaction — an email, print, remote API action or other operation not undone by ordinary database rollback.

Read in context: 21.6.6, 24.6.6

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: 21.4.6

Full-transaction retry

repeat the decision from fresh protected reads — a new attempt that does not reuse an invalidated transaction's conclusions blindly.

Read in context: 23.6.6

G

Generation identifier

distinguish separate lifetimes of a reused key — an immutable identity component that changes when a resource is recreated.

Read in context: 25.4.6

I

Identity scope

the domain in which a key is unique — the tenant, operation and other components defining a request key's namespace.

Read in context: 26.2.6

Idle in transaction

a transaction remains open without active useful work — a state that can retain locks and snapshots while the application waits elsewhere.

Read in context: 24.2.6

Inbox record

remember which message identity was accepted — a consumer-side deduplication record committed with the participating local effect.

retained evidence of an accepted incoming message — a local identity and outcome record used to recognize duplicates and conflicts.

Read in context: 26.5.6

Incompatible key reuse

one identity is presented with different meaning — a conflict requiring rejection or explicit reconciliation, not silent replacement.

Read in context: 26.3.6

Indeterminate outcome

the caller cannot yet tell whether it completed — uncertainty commonly caused by losing communication around a commit or external effect.

Read in context: 21.2.6

Interleaving

the steps of different requests mixed together — an operation ordering that preserves each participant's required local order.

Read in context: 22.1.6

Invariant

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

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

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

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

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

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

Read in context: 21.3.6, 26.1.6

Isolation

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

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

Read in context: 21.4.6

Isolation level

the chosen rules for overlapping transactions — a named contract restricting visibility and permitted concurrency anomalies.

Read in context: 23.1.6

J

Jitter

deliberately vary the waiting time — bounded randomness used to avoid synchronized retries.

Read in context: 24.5.6

L

Last-write-wins policy

accept the selected later replacement — a conflict rule that can discard competing state and requires a defined ordering basis.

Read in context: 25.6.6

Lock compatibility

which access modes can coexist — the engine's conflict relation between requested and held lock modes.

Read in context: 24.2.6

Lock holder

the participant currently retaining protected access — the session or transaction owning a granted lock.

Read in context: 24.1.6

Lock lifetime

how long protected access remains held — the release boundary determined by transaction, statement, session or explicit protocol.

Read in context: 24.2.6

Lock scope

the resource access being coordinated — the row, table, key or other engine/application object covered by a lock.

Read in context: 24.1.6

Lock upgrade

requesting stronger access while already holding weaker access — a mode transition that can introduce additional conflicts.

Read in context: 24.4.6

Lost update

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

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

Read in context: 22.2.6

M

Merge granularity

the unit at which changes are compared — a document, row, field or operation boundary used by the conflict protocol.

Read in context: 25.6.6

Mixed-snapshot calculation

combining facts observed at different boundaries — a derived result that may correspond to no single database state.

Read in context: 23.3.6

Mutation path

a way records can change — an application route, job, import, administrator action or other writer that must respect the invariant.

Read in context: 22.5.6

MVCC

let readers use suitable versions while changes occur — multi-version concurrency control using versioned data and transactional visibility rules.

Read in context: 25.1.6

N

Non-repeatable read

rereading a row yields a later committed value — a permitted observation change under some isolation contracts.

Read in context: 23.3.6

O

Observation contract

the stated point or interval a result describes — the temporal and visibility meaning assigned to a query or report.

Read in context: 23.3.6

Obsolete version

an earlier stored state no longer needed by relevant readers — a version eligible for reclamation under the engine's visibility and maintenance rules.

Read in context: 25.3.6

Optimistic concurrency

detect a conflict when saving rather than holding a long edit lock — a protocol that validates an earlier observation at the write boundary.

Read in context: 25.5.6

Outbox

committed work carries a delivery instruction — a transactional record of an event or message to be delivered by a separate process.

a durable list of external work implied by accepted changes — message-intent records committed atomically with local business state.

atomically recorded delivery intent — a local record that survives with the business change but does not guarantee external completion.

Read in context: 21.6.6, 24.6.6

Outcome reconciliation

resolve uncertainty using recorded identity and state — a procedure for determining what a prior attempt actually accepted.

Read in context: 26.6.6

Outer transaction

the enclosing acceptance boundary — the transaction whose final commit or rollback governs changes retained from inner savepoint work.

Read in context: 21.5.6

P

Partial rollback

discard only a selected later portion — rollback to a savepoint rather than abandonment of the entire outer transaction.

Read in context: 21.5.6

Payload equivalence

when two attempts mean the same thing — an explicit comparison contract over validated request semantics.

Read in context: 26.2.6

Phantom

a repeated condition matches a changed set — a predicate-query membership change caused by another transaction's committed work.

Read in context: 23.1.6

Physical row location

where one stored version currently resides — engine metadata such as PostgreSQL's `ctid`, not an enduring logical identity.

Read in context: 25.1.6

Predicate tracking

remember the read conditions that matter — implementation metadata used to detect dependencies involving rows or ranges a transaction read.

Read in context: 23.5.6

R

Race condition

an answer that depends on an unsafe overlap — behaviour whose correctness changes with the relative execution order of competing operations.

Read in context: 22.1.6

Read dependency

a decision relies on an observed fact — a relationship between a transaction's read and another transaction's relevant write.

Read in context: 23.4.6

Reconciliation

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

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

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

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

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

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

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

Read in context: 22.6.6

Reconciliation identity

an equation connecting related records — a check that quantities and recorded state changes agree under stated inclusions and exclusions.

Read in context: 26.1.6

Relative update

describe the change rather than an old replacement — an expression such as `value = value - amount` evaluated by the database.

Read in context: 22.2.6

Replay

another attempt carrying the same defined intent — a repeated request that should resolve through its existing identity and outcome.

Read in context: 26.3.6

Request identity

the stable name of one intended operation — a scoped identifier linking repeated attempts to one business decision.

Read in context: 26.2.6

Reservation

a recorded allocation under defined rules — a business state transition that reduces available capacity while linking the allocation to a request.

Read in context: 22.6.6

Retry budget

the allowed additional effort — limits on attempts, elapsed time or resources spent resolving transient failures.

Read in context: 24.5.6

Retry deadline

the point beyond which another attempt is not allowed — an operation-level bound on time spent resolving transient failures.

Read in context: 23.6.6

Revision exhaustion

no unused value remains in the chosen domain — a boundary requiring an explicit policy rather than silent counter reuse.

Read in context: 25.4.6

Rollback

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

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

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

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

Read in context: 21.2.6

S

Savepoint

a local return point inside transaction work — a named boundary to which later transactional changes can be rolled back.

Read in context: 21.5.6

Schedule evidence

the recorded sequence that actually ran — observations of operations, waits, errors and outcomes under a stated concurrency configuration.

Read in context: 22.1.6

Scoped guarantee

a promise with explicit limits — a correctness claim tied to defined effects, participants, failures and retention assumptions.

Read in context: 26.6.6

Serializability

concurrent work has a valid serial explanation — equivalence of committed transaction effects to an execution in some serial order.

Read in context: 23.5.6

Serializable outcome

a result explainable by a legal one-at-a-time execution — an outcome equivalent to some serial order of the participating transactions.

Read in context: 23.4.6

Serialization failure

a transaction is refused to preserve the isolation guarantee — an error requiring an appropriate full-operation retry or failure response.

Read in context: 23.5.6

Shared guard record

one coordination point for a shared limit — a record whose guarded mutation serializes changes to the represented resource.

Read in context: 22.5.6

Snapshot establishment

the event that chooses the view — the implementation-defined point at which a transaction's snapshot is acquired.

Read in context: 23.2.6

Stale client copy

an earlier result still held outside the database — data whose original read validity does not guarantee current edit validity.

Read in context: 25.2.6

Stale read

an observation that no longer describes the relevant current state — a previously read version that may be unsafe as an unconditional write basis.

an answer lacks required newer information — a read that does not meet the application's freshness or consistency contract.

Read in context: 22.2.6

State transition

an allowed move between positions — a specified change from one model state to another.

one defined move from an old state to a new state — an operation with explicit preconditions, writes and outcomes.

Read in context: 26.1.6

Statement snapshot

a view chosen for one statement — the visibility boundary used by a query under statement-level snapshot rules.

Read in context: 23.2.6

T

Transaction boundary

the work accepted or rejected together — the scope of a database transaction's reads, writes and completion decision.

Read in context: 21.1.6

Transaction owner

the code responsible for completion — the layer that controls commit and rollback for a defined unit of work.

Read in context: 21.1.6

Transaction snapshot

a retained view for related statements — a visibility boundary reused through a transaction under the chosen implementation.

Read in context: 23.2.6

U

Unknown outcome

the caller lacks reliable completion evidence — a state in which a request may have committed even though its response was not received.

the caller lacks enough evidence to classify completion — a state distinct from confirmed success or confirmed rollback.

Read in context: 22.6.6

V

Vacuuming

maintenance that reclaims and manages old storage — PostgreSQL's process for handling obsolete tuples and other required maintenance tasks.

Read in context: 25.3.6

Version column

a field naming the accepted edit state — an application-managed revision value checked and advanced by relevant writes.

Read in context: 25.4.6

Visible version

the row state allowed by a reader's snapshot — a version satisfying the engine's transaction-visibility rules.

Read in context: 25.2.6

W

Wait-for graph

a map of who waits for whom — a directed graph used to reason about resource dependencies.

Read in context: 24.3.6

Write skew

separate writes jointly break a shared rule — an anomaly in which transactions act on overlapping read conditions while changing different records.

Read in context: 23.4.6

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