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 F · Chapters 33–39 · First Edition

Data Across Machines

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 F

Data Across Machines

Understand copies, partitions, coordination and the limits of promises made over a network.

This volume contains Chapters 33–39 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 F

Data Across Machines

Understand copies, partitions, coordination and the limits of promises made over a network.

33

CHAPTER

Replication: More Than One Copy

33.0 What this chapter gives you

1. A second copy of a database can help answer questions, survive some failures or move data closer to readers. It also creates a new question: which copy knows about which changes?
2. You will distinguish physical from logical replication, receiving from applying, synchronous from asynchronous acknowledgement, and a useful replica from a recoverable historical backup.
3. The examples use a separate fictional replication register, not a live shop cluster. Positions such as 40 and 41 are teaching sequence numbers, not PostgreSQL WAL addresses. The companion model demonstrates prefixes and stale reads; it does not implement a network replication protocol.

33.1 Why copies exist

33.1.1 PLAIN — in simple words

1. Keeping one copy makes the meaning of "current" relatively straightforward, but it puts much responsibility on one machine. If that machine is unavailable, readers may have nowhere else to ask.
2. A replica is another maintained copy. It might support recovery from a machine failure, serve read-only reports or keep frequently requested information near a distant branch.
3. Those purposes are not identical. A server configured for delayed reporting may be a poor emergency replacement. A nearby replacement may share the same power, building or administrative failure as the original.

33.1.2 PLAIN — a picture in your head

1. Mira keeps an order notebook and asks Dev to maintain a second notebook from numbered change slips. Dev can answer some questions while Mira serves another customer.
2. If slips are delayed, Dev's notebook is not necessarily wrong about the past; it is simply behind the latest accepted changes. Problems begin when that older view is presented as current.
3. **Where the comparison breaks:** a database replica follows an engine-specific protocol with transaction and recovery rules. It is not an independent human witness confirming that a sale physically occurred.

33.1.3 PLAIN — a worked example

1. Suppose a hypothetical service receives 900 read requests and 100 write requests each second. Two followers could share some of the read work while the leader remains responsible for writes.

2. Splitting 900 reads equally would assign 300 to each of three machines. That arithmetic is a proposed allocation, not a measured threefold speedup: the machines also spend resources transferring and applying changes.
3. If 200 of the reads must immediately observe a just-completed write, routing those indiscriminately to lagging followers can violate the application requirement. Read traffic must be classified before it is distributed.
4. The benefit therefore depends on the workload, the freshness contract and the failure being addressed. "Three servers" alone specifies none of them.

33.1.4 PLAIN — what is really happening inside

1. A replication design chooses an authoritative source or conflict protocol, captures changes, transports them and brings each destination to a defined state.
2. Additional readers can increase useful capacity only if the cost of serving them is not replaced by a larger bottleneck in change generation, transmission or replay.
3. The operator must monitor both the original and the copies. A silently disconnected replica can look healthy to a simple process check while holding increasingly old data.

33.1.5 TECHNICAL — the engineer's version

1. Separate high availability, disaster recovery, read scaling and geographic placement as requirements. Their recovery objectives, consistency needs and failure-domain assumptions should be explicit.
2. PostgreSQL 17 distinguishes warm standby operation from hot standby read-only queries. Streaming and log shipping maintain a recovery-capable history under their documented configuration and retention requirements. [S147]
3. Replication adds ongoing work and additional privileged data paths. Capacity planning must include replay, retained logs, connection management and recovery rehearsals, not just foreground queries. [S144] [S147]

33.1.6 WORDS — remember these

1. **Replica:** another maintained copy — a data instance updated through a defined replication mechanism. **Read scaling:** sharing question-answering work — distributing eligible read operations across resources without assuming identical freshness. **Failure domain:** things that can fail together — a boundary such as process, host, power supply, site or administrative authority.

33.2 Physical and logical approaches

33.2.1 PLAIN — in simple words

1. One way to maintain a copy is to reproduce changes in the database's physical storage machinery. Another is to describe changes to named records and apply them to a compatible destination.
2. Physical replication closely follows the source engine's representation. Logical replication works with data objects and their identities, allowing more choice about which records are copied and how the destination is organized.
3. Neither label means "copy absolutely everything." You must inspect the mechanism's actual coverage, including schemas, sequences, permissions and objects that are not ordinary table rows.

33.2.2 PLAIN — a picture in your head

1. Copying each amended notebook page resembles the physical approach. Sending "change the quantity on order line X" resembles the logical approach.
2. Page copying expects a compatible notebook layout. Record instructions require both people to agree on what identifies X and what each field means.
3. **Where the comparison breaks:** physical replication commonly transports recovery records, not photographs or repeated whole-file copies. Logical replication may preserve transaction boundaries rather than sending unrelated individual edits.

33.2.3 PLAIN — a worked example

1. A source contains `order_id`, `line_no`, `product_id`, `quantity` and `unit_price_paise`. The agreed line identity is the pair `(order_id, line_no)`.
2. A logical update naming only `product_id` would not identify an order line: the same product occurs in many orders. Replication needs an appropriate identity for the action it is applying.
3. Suppose an analytical subscriber intentionally receives only order lines and products. A row count matching those tables does not prove that customers, roles or sequence state have been copied.
4. In PostgreSQL 17, built-in logical replication does not automatically replicate DDL or sequence state. A promotion or schema-change plan must account for those omissions rather than inferring completeness from successfully copied rows. [S164]

33.2.4 PLAIN — what is really happening inside

1. A destination normally needs an initial consistent data boundary and a stream of changes beginning at a compatible position. A gap between the initial copy and the change stream loses work; uncontrolled overlap can repeat work.
2. The apply process interprets the stream using its configured schema and identity rules. Incompatible changes or local writes can stop application or create conflicts.
3. Physical formats place tighter compatibility requirements on engine versions and storage layout. Logical formats avoid some of those constraints but introduce their own schema, identity and semantic requirements.

33.2.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 logical replication uses publication/subscription and replication identity. Its transactional consistency guarantee is scoped to publications within a single subscription, not arbitrary independent subscriptions observed at unrelated positions. [S148]
2. Physical standby compatibility and logical object coverage are product-specific. Do not generalize a logical subscriber into a complete physical failover target without a separate assessment. [S147] [S164]
3. Record initial-snapshot boundaries, schema versions, identity configuration and apply errors. These are evidence needed to diagnose a divergent or stalled subscriber; the transport connection alone is insufficient.

33.2.6 WORDS — remember these

1. **Physical replication:** following the storage engine's changes — replication tied to physical representation and recovery information. **Logical replication:** following changes to data objects — replication based on record identity and logical operations rather than identical physical place-

ment. **Replication identity:** the handle for locating a changed row — the key or supported old-row representation used to apply updates and deletions.

33.3 Synchronous and asynchronous paths

33.3.1 PLAIN — in simple words

1. An asynchronous path can report success before a remote copy has reached the required point. That can reduce waiting, but a later failure may expose a gap between what was acknowledged and what the survivor knows.
2. A synchronous path waits for specified remote confirmation. The important word is "specified": confirmation of receipt, durable storage and application to readable state are different milestones.
3. Waiting for one follower does not imply waiting for every follower. A reader sent to a different, slower machine may still see older data.

33.3.2 PLAIN — a picture in your head

1. Mira can finish with a customer after putting a change slip in her own tray, after Dev says "received," or after Dev says "entered in my notebook." Each acknowledgement means something different.
2. If Dev has merely received the slip, a query against his notebook can still return the old quantity. The message arrived, but the visible state has not changed.
3. **Where the comparison breaks:** durable confirmation relies on storage and recovery guarantees, not a person's verbal assurance. A remote operating-system write and a durable flush are not interchangeable.

33.3.3 PLAIN — a worked example

1. At a synthetic instant, the leader has accepted sequence 41. Follower A has received and durably stored 41 but has applied only through 40. Follower B has applied through 39.
2. A policy waiting for A's durable storage can be satisfied while A's read view is still at 40. A policy waiting for A to apply 41 cannot be satisfied yet.
3. Even after A applies 41, a request sent to B can remain stale. The acknowledgement scope must match the read-routing decision.
4. For a hypothetical remote path with 35 ms round-trip delay, a commit that must await that remote response cannot complete that confirmation in 2 ms. This is a lower-bound illustration, not a benchmark of a particular service.

33.3.4 PLAIN — what is really happening inside

1. The leader tracks progress reported by each configured follower. A transaction waits until its selected acknowledgement condition is met.
2. Slow or missing followers can therefore delay success responses. Changing the policy to stop waiting may restore responsiveness while also changing the protection promised to future requests.
3. Configuration is part of the data contract. A per-transaction override can matter even when a global setting appears strict.

33.3.5 TECHNICAL — the engineer's version

1. With synchronous standbys configured, PostgreSQL 17 `synchronous_commit=on` waits for their required durable WAL confirmation; `remote_write` waits for a weaker remote file-system write; `remote_apply` additionally waits for replay visibility. `local` does not wait for replication. [S149]
2. When no synchronous standby is configured, these names do not create a remote guarantee. Confirm the active standby-selection policy and actual transaction setting. [S149]
3. Synchronous replication is not, by itself, a complete failover, consensus or multi-record application-correctness protocol. Those claims require their own mechanisms and failure assumptions.

33.3.6 WORDS — remember these

1. **Asynchronous replication:** remote progress may follow acknowledgement — a configuration that does not wait for a selected remote replication milestone before reporting success. **Synchronous acknowledgement:** success waits for specified remote progress — a policy whose exact receipt, flush or apply condition must be named. **Apply position:** how far changes affect the readable replica — the prefix already replayed into the destination's visible state.

33.4 Lag and staleness

33.4.1 PLAIN — in simple words

1. Lag is a gap in progress. Staleness describes how old an answer is relative to the information the application requires. The two are related but not identical.
2. A replica can be many bytes behind because one large update is pending, or only a small number of bytes behind while missing a very important cancellation.
3. A time since the last replayed transaction can also be misleading when the source has been idle. An old last-change time is not proof that new changes are waiting.

33.4.2 PLAIN — a picture in your head

1. Dev's last change slip is an hour old. If Mira made no changes during that hour, his notebook may be fully caught up.
2. If Mira wrote a crucial new slip ten seconds ago, Dev can be behind even though his last update sounds recent.
3. **Where the comparison breaks:** real systems expose several progress counters with different update intervals and clock assumptions. A single wall-clock subtraction is not a universal freshness measure.

33.4.3 PLAIN — a worked example

1. Suppose a fictional stream has generated 120 MB, sent 115 MB, remotely stored 110 MB and applied 100 MB at one observation boundary.
2. The unsent gap is 5 MB; the sent-but-not-stored gap is 5 MB; the stored-but-not-applied gap is 10 MB. The total generated-to-applied difference is 20 MB.
3. At a net catch-up rate of 4 MB/s, clearing a fixed 20 MB backlog would take 5 seconds. If the source generates another 4 MB/s while the receiver only applies 4 MB/s, the backlog does not shrink.

4. These decimal-byte calculations assume a consistent stream and compatible observation positions. They do not identify which business records are missing or provide a guaranteed catch-up deadline.

33.4.4 PLAIN — what is really happening inside

1. Separate generation, transmission, receipt, flush and apply bottlenecks. More network bandwidth will not resolve a replay process blocked by an incompatible schema or resource shortage.
2. Retained log data allows a disconnected destination to catch up only while the required history remains available. Keeping everything indefinitely can exhaust the source's disk.
3. Monitoring therefore needs both freshness evidence and retention-pressure evidence. A slot that preserves history can protect a follower while threatening the primary's remaining storage.

33.4.5 TECHNICAL — the engineer's version

1. PostgreSQL exposes WAL positions and replication statistics for distinguishing progress stages. Interpret counters within the same history and account for observation timing rather than subtracting unrelated positions. [S144] [S147]
2. Replication slots can retain WAL for disconnected consumers. Limits and monitoring are necessary because retained WAL can consume the available storage; losing required history may force reinitialization. [S147]
3. An application freshness requirement should identify a minimum causal position, acceptable age under stated clock assumptions, or authoritative read path. "Lag looks low" is not a complete correctness condition.

33.4.6 WORDS — remember these

1. **Replication lag:** one replication stage is behind another — a difference measured using compatible positions, bytes or carefully interpreted timing evidence. **Stale read:** an answer lacks required newer information — a read that does not meet the application's freshness or consistency contract. **Catch-up rate:** how quickly a backlog shrinks — destination progress minus continuing source generation over the measured interval.

33.5 Read routing

33.5.1 PLAIN — in simple words

1. Choosing where to read is part of application correctness. A recently changed order should not appear unchanged merely because the next screen used a slower copy.
2. Some questions can tolerate an older view. Others need to observe the user's own completed action or a particular confirmed state before they proceed.
3. The design must state what happens when the preferred copy is unavailable: wait, use an authoritative alternative, show explicitly older data, or decline the operation. Quietly weakening the requirement hides the problem.

33.5.2 PLAIN — a picture in your head

1. Mira gives a customer a receipt for change slip 41. The customer asks Dev about that change and shows the receipt number.
2. Dev can answer only after his notebook includes at least slip 41, or he can direct the customer to someone who can. Guessing from slip 39 does not satisfy the request.

3. **Where the comparison breaks:** a progress token must be meaningful for the actual history, tenant and operation. A bare number from an unrelated stream is not evidence of causal inclusion.

33.5.3 PLAIN — a worked example

1. Start a separate teaching register at position 40 with quantity 5. A committed change at 41 sets quantity to 2. The follower remains at 40.
2. An unrestricted follower read returns 5. A read requiring position 41 must not report that value as satisfying the requirement.
3. After replaying the contiguous entry at 41, the follower can return 2 with applied position 41. The model rejects a missing predecessor rather than pretending that receiving position 42 establishes an uninterrupted history.
4. The companion tests exercise these states as a pure model. They do not prove a real deployment's failover-safe session consistency or cross-database token handling.

33.5.4 PLAIN — what is really happening inside

1. A session may carry a minimum observed position, or the application may route selected reads to the current authority. Each approach requires explicit handling of role changes and unavailable dependencies.
2. Waiting on a follower must be bounded operationally. A timeout should remain visible; it must not silently convert a required-fresh read into an old answer labelled current.
3. Transaction snapshots also matter. A session that already holds an older snapshot may not see a newly applied change merely because another observer reports that replay has advanced.

33.5.5 TECHNICAL — the engineer's version

1. Read-your-writes and monotonic-read requirements are session-level contracts, distinct from a generic claim that replicas eventually converge. Track the scope and validity of any causal token.
2. PostgreSQL `remote_apply` can support useful visibility conditions on the selected synchronous standbys, but arbitrary follower selection or an older transaction snapshot can invalidate a broader inference. [S149] [S66]
3. Reader routing, transaction isolation and failover authority must be reviewed together. Routing to a machine called "leader" is insufficient if stale authority is possible; Chapter 34 develops that boundary.

33.5.6 WORDS — remember these

1. **Read-your-writes:** seeing your own completed changes — a session guarantee that later eligible reads include the session's prior acknowledged writes. **Monotonic reads:** not going backward in observed progress — a session constraint preventing later reads from using an older relevant state. **Causal token:** evidence of a required prior position — a scoped marker used to ensure that a read includes specified earlier work.

33.6 Replication is not a restore plan

33.6.1 PLAIN — in simple words

1. A replica normally follows changes, including mistaken deletions and harmful updates. Copying a mistake faithfully is successful replication, not successful recovery.

2. A backup preserves a state for a recovery purpose. Retained logs may let that state be advanced to a chosen point before a mistake, subject to the supported recovery procedure.
3. Both mechanisms can be valuable. Neither replaces an inventory of failure scenarios, protected copies and tested restoration steps.

33.6.2 PLAIN — a picture in your head

1. If Mira crosses out the wrong page and Dev copies the crossing-out, both notebooks now contain the same mistake.
2. An earlier protected copy may help reconstruct what existed before the error. A second notebook that always follows the first does not provide that historical boundary by itself.
3. **Where the comparison breaks:** restoring a database also involves transaction histories, credentials, external effects and compatible software. It is not merely undoing ink on a page.

33.6.3 PLAIN — a worked example

1. A synthetic deletion is accepted at sequence 42. Both leader and follower eventually apply 42 and no longer contain the deleted record.
2. A protected snapshot at 40 plus complete retained changes through 41 can support a separate recovery exercise stopping before 42. A snapshot without the needed log or encryption key may not.
3. If the restored service replays old outbox messages without controls, it can repeat downstream actions even when the database restoration itself is correct.
4. A recovery plan therefore verifies data, identity, access and external-effect policy before reconnecting the restored instance to ordinary traffic.

33.6.4 PLAIN — what is really happening inside

1. Replication reduces some recovery delays by maintaining a prepared copy. Backup retention protects selected historical states against different failures.
2. Separate administrative and storage failure domains where the requirements demand it. A single compromised authority able to erase the primary, replicas and all backups defeats mere copy counting.
3. Rehearse both failover and restoration. Their procedures, evidence and acceptable data-loss windows are not the same.

33.6.5 TECHNICAL — the engineer's version

1. Distinguish replica promotion from point-in-time recovery. PostgreSQL's continuous-archiving procedure requires compatible base data and sufficient WAL coverage for the selected recovery target. [S131]
2. Backup verification should include actual restore rehearsals, not only manifest checking. A healthy replication stream does not establish that a historical recovery point is usable. [S138]
3. State the tested failure scope. The book's prefix model and local backup tests are educational evidence, not a certification of multi-host durability, operational failover or off-site disaster recovery.

33.6.6 WORDS — remember these

1. **Promotion:** making a replica an active authority — a role transition requiring a supported procedure and protection against competing writers. **Historical recovery point:** a selected earlier

usable state — a state reconstructible from compatible retained recovery material. **Recovery rehearsal:** trying the recovery procedure in isolation — a test of restoration steps, evidence and operational assumptions before an incident.

33.97 Practice and worked answers

1. **A follower has stored position 41 but applied 40. Can it necessarily answer a query requiring 41?** No. Durable receipt and query visibility are separate milestones.
2. **A standby's last replay timestamp is an hour old. Is it definitely behind?** No. First establish whether the source produced any later work and compare compatible progress evidence.
3. **A 20 MB backlog is processed at 8 MB/s while the source adds 4 MB/s. What is the simple catch-up estimate?** The net rate is 4 MB/s, so approximately 5 seconds under these fixed-rate assumptions.
4. **The leader waited for follower A. May a later query use follower B without another check?** Only if B independently satisfies the required read contract. A's acknowledgement is not B's progress.
5. **Does copying table rows through PostgreSQL 17 logical replication copy every sequence and DDL operation?** No. Those boundaries require a separate compatibility and promotion plan.
6. **A mistaken deletion appears on all replicas. Did replication fail?** Not necessarily. It may have reproduced the accepted operation correctly; recovery from the mistake is a different responsibility.
7. **What should happen if a required-fresh follower read reaches its deadline?** Return the documented timeout, route to an authorized alternative that meets the requirement, or expose an explicitly weaker permitted mode. Do not conceal the downgrade.
8. **What does the companion model prove?** Its contiguous-prefix, stale-read and minimum-position rules behave as tested. It does not execute a PostgreSQL cluster or model every network and storage failure.

33.98 Common wrong ideas

1. **Wrong:** more copies automatically mean more truth. **Right:** replicas can faithfully repeat incorrect source data.
2. **Wrong:** received means readable. **Right:** receipt, durable storage and application are distinct stages.
3. **Wrong:** synchronous means every replica is current. **Right:** acknowledgement follows a configured scope and milestone.
4. **Wrong:** low byte lag proves every important record is fresh. **Right:** a small missing change may be operationally critical.
5. **Wrong:** a connected replica is caught up. **Right:** it may be blocked or accumulating backlog.
6. **Wrong:** logical replication copies every database object. **Right:** inspect supported object and schema coverage.
7. **Wrong:** a replica eliminates backups. **Right:** current copies and protected recovery points serve different purposes.
8. **Wrong:** a successful local model certifies failover. **Right:** real authority, routing, retention and failure handling remain separate validation work.

33.99 Chapter summary in 20 lines

1. Replication maintains additional copies for explicitly chosen purposes.
2. Read scaling, high availability and historical recovery are different requirements.
3. Physical replication follows engine-level representation and recovery rules.
4. Logical replication follows identified data objects and changes.
5. Initial copies and change streams need a compatible boundary.
6. Schema and object coverage must be inspected rather than assumed.
7. Asynchronous acknowledgement may precede remote progress.
8. Synchronous acknowledgement waits for a named remote milestone.
9. Receipt, flush and application are different milestones.
10. Waiting for one follower does not establish another follower's freshness.
11. Lag can be measured at several stages of the replication path.
12. Time since the last change is not a complete lag measurement.
13. Catch-up requires progress faster than continuing input.
14. Retained logs protect catch-up but consume storage.
15. Read routing is part of the application's consistency contract.
16. Minimum-position reads require compatible history and snapshot handling.
17. Timeouts must not conceal a weaker freshness guarantee.
18. Replicas can faithfully reproduce harmful accepted operations.
19. Backups and retained logs support separately tested historical recovery.
20. State exactly which replication and recovery properties were demonstrated.

34

CHAPTER

Leaders, Followers and Failover

34.0 What this chapter gives you

1. A leader is not simply the machine with the newest-looking screen. It is a participant currently authorized by a defined protocol to coordinate particular work.
2. You will trace a role change through failure suspicion, selection, fencing, client routing and restoration of redundancy. You will also learn why a missing reply leaves an operation's outcome uncertain.
3. The examples use fictional nodes A, B and C and a small authority model. They do not install failover software, promote a real PostgreSQL server or establish production high availability.

34.1 Write authority

34.1.1 PLAIN — in simple words

1. A single-leader design gives one participant responsibility for coordinating writes to a defined set of data. Other participants may copy those writes or serve eligible reads.
2. This is a rule about authority, not a statement that one computer is permanently special. A safe design can transfer that role without allowing incompatible authorities to operate at the same time.
3. The scope matters. Different partitions may have different leaders, and a machine coordinating database writes may have no authority over a payment service or a delivery system.

34.1.2 PLAIN — a picture in your head

1. Mira appoints one counter supervisor to approve reservations from a shared stock allocation. Other staff forward requests to that supervisor rather than independently promising the same last item.
2. If the supervisor changes, the old approval badge must stop working. Giving a new person a badge while the old badge remains valid creates two competing authorities.
3. **Where the comparison breaks:** a database protocol cannot rely on everyone noticing a human announcement. Delayed messages and paused processes can continue carrying obsolete authority.

34.1.3 PLAIN — a worked example

1. In a separate teaching system, node A holds authority generation 7 for resource `RESERVE-BR-A`. Its writes include that resource identity and generation.
2. Node B may be a follower for the same resource while leading an unrelated reporting task. Calling B "a follower" without naming the resource obscures its actual responsibilities.

3. A role transition to generation 8 must define when generation 7 ceases to authorize changes and how the affected resource enforces that decision.
4. Merely changing a dashboard label from A to B does not alter what an old connection to A can still do.

34.1.4 PLAIN — what is really happening inside

1. Authority may come from a consensus group, a carefully managed failover controller or another explicit coordination mechanism. The storage and request paths must cooperate with it.
2. Clients also need a way to find the current service endpoint and recover from stale routing. Connection pools may retain old sessions after a network address changes.
3. The design must connect selection, enforcement and discovery. Implementing only one of these leaves gaps between who was chosen and who can actually modify data.

34.1.5 TECHNICAL — the engineer's version

1. Separate control-plane authority from data-plane enforcement. The former chooses a valid owner or epoch; the latter rejects operations that do not satisfy the current authorization condition.
2. PostgreSQL 17 provides standby promotion mechanisms but does not itself supply the complete external failure-detection and notification system described in its failover documentation. [S150]
3. A single-leader label is not a complete safety proof. Review the failure model, durable state, resource boundaries and stale-client behaviour of the actual implementation.

34.1.6 WORDS — remember these

1. **Leader:** the current coordinator for specified work — a role granted under a protocol with a defined authority scope. **Control plane:** the machinery choosing and configuring behaviour — coordination mechanisms such as role assignment and membership management. **Data plane:** where requests produce actual effects — the operational path that must enforce the control plane's decisions.

34.2 Failure suspicion

34.2.1 PLAIN — in simple words

1. When a machine stops replying, it may be dead, slow, disconnected or unable to reach the observer. The observer has evidence of missing communication, not complete knowledge of the machine's state.
2. Waiting forever prevents useful recovery. Acting too quickly can replace a healthy but temporarily unreachable leader. A failover design must handle both risks explicitly.
3. A timeout is therefore a trigger for a protocol, not proof that the previous authority has ceased to exist.

34.2.2 PLAIN — a picture in your head

1. Dev calls Mira and hears nothing. Her phone may be off, the mobile network may be down, or she may be busy with a customer.
2. Dev cannot safely infer that every approval she previously held has disappeared. A separate rule is needed to transfer responsibility and invalidate obsolete approvals.

3. **Where the comparison breaks:** systems use timers, persistent terms and communication protocols rather than human judgement. The analogy explains uncertainty, not a recommended failure detector.

34.2.3 PLAIN — a worked example

1. Suppose a fictional controller expects a heartbeat every second and suspects failure after three missed intervals. At time 3 seconds it may initiate investigation or an election.
2. In one schedule, A crashed at time 0. In another, A is still running but its messages to the controller are delayed for 5 seconds. The observer sees the same missing heartbeats during the first 3 seconds.
3. Promoting B at time 3 without fencing A can allow both machines to accept incompatible writes between times 3 and 5, or longer if the network split persists.
4. These timers are invented for reasoning. They are not suggested production settings or a measured failure-detection guarantee.

34.2.4 PLAIN — what is really happening inside

1. Failure detectors trade responsiveness against false suspicion under the timing conditions they assume. A process pause can resemble a network outage from another participant's perspective.
2. A witness or third participant may help a protocol decide which side can proceed, but only if its role and failure assumptions are correctly integrated. An arbitrary extra ping target is not a proof of safety.
3. Good incident records preserve observations from multiple paths, the timer policy and the authority transition. They do not retroactively turn an initial suspicion into a fact that was known at the time.

34.2.5 TECHNICAL — the engineer's version

1. Distinguish crash failure from communication failure and timing uncertainty. Raft uses election timeouts to initiate a new term, while safety depends on voting and log rules rather than the timeout proving that a leader died. [S154]
2. PostgreSQL's failover guidance explicitly discusses heartbeat mechanisms, possible witnesses and the need to prevent an old primary from continuing as primary. [S150]
3. Record the actual detection-to-service-restoration interval, including selection, promotion, routing and readiness. A three-second suspicion threshold does not imply a three-second recovery time.

34.2.6 WORDS — remember these

1. **Heartbeat:** a repeated liveness signal — periodic communication used to detect absence or delay under a configured policy. **Failure suspicion:** evidence that a participant may be unavailable — an observation requiring protocol handling rather than certain knowledge of a crash. **False suspicion:** a healthy participant is treated as failed — a classification possible under delays, partitions or process pauses.

34.3 Election and promotion

34.3.1 PLAIN — in simple words

1. Selection chooses a participant to take responsibility. Promotion changes a replica's operational role. These steps must agree with the data history and the authority mechanism.
2. Choosing the first machine that answers a ping is not enough. The candidate must have suitable data, compatible configuration and a legitimate path to ownership.
3. A system may intentionally stop rather than promote a candidate that would violate its acknowledged-data or authority requirements.

34.3.2 PLAIN — a picture in your head

1. Two assistants have different prefixes of the order notebook. One is available immediately but is missing the latest approved reservations; another has the necessary history but needs a moment to finish applying it.
2. The shop must decide what evidence a replacement needs, not merely who can reach the counter fastest.
3. **Where the comparison breaks:** consensus election rules are mathematical protocol conditions. Human preference for the most confident assistant is not an equivalent selection rule.

34.3.3 PLAIN — a worked example

1. Suppose A acknowledged a synthetic write at sequence 41, B has durable history through 41 and C has only through 40. A is now unreachable.
2. Promoting C can lose the acknowledged change unless the system first obtains the missing history or its documented policy accepts that loss. The fact that C is running does not fill the gap.
3. If B was the required synchronous destination for that acknowledgement, the failover procedure must preserve the relevant durability assumptions and select a valid survivor. An unrelated asynchronous copy is not automatically equivalent.
4. In a Raft-style protocol, the election's log-freshness rule is specific: compare the last entry's term first, then its index when terms are equal. "Largest row count wins" is not the rule. [S154]

34.3.4 PLAIN — what is really happening inside

1. A candidate may need recovery replay before serving traffic. A process accepting TCP connections does not establish application readiness or complete recovery.
2. The transition must also handle routing, credentials, replication slots, monitoring and downstream consumers. These are practical components of service continuity, not optional cosmetic tasks.
3. Once a follower becomes primary, the system may temporarily have fewer healthy copies. Restoring redundancy is part of finishing the incident, even after requests start succeeding again.

34.3.5 TECHNICAL — the engineer's version

1. Failover safety depends on the acknowledged commit set, eligible candidates and the mechanism preventing competing primaries. PostgreSQL streaming replication and Raft are different mechanisms; do not attribute Raft's election rules to a plain PostgreSQL primary/standby pair. [S147] [S150] [S154]
2. Readiness checks should cover recovery completion, intended role, required schema, protected write access and the ability to perform a bounded representative operation.

3. The procedure must record any accepted data-loss window. Fast role switching with undisclosed loss is not equivalent to a verified no-loss transition.

34.3.6 WORDS — remember these

1. **Election:** selecting a coordinator under protocol rules — an authority decision with explicit membership and eligibility conditions. **Candidate eligibility:** the conditions a replacement must satisfy — requirements covering authority, history and readiness rather than mere reachability. **Degraded redundancy:** fewer healthy copies than intended — an operational state that may persist after the service resumes.

34.4 Fencing stale writers

34.4.1 PLAIN — in simple words

1. A paused old leader can wake up believing it still owns the job. Fencing prevents its obsolete authority from producing effects.
2. A common pattern attaches a changing generation to authority. The protected resource checks that generation and rejects obsolete operations.
3. The check must occur where the effect is accepted. A generation number written only in a log message provides evidence, not enforcement.

34.4.2 PLAIN — a picture in your head

1. The stock room changes its approval register from badge generation 7 to generation 8. The door attendant checks the register for every withdrawal.
2. An old supervisor returning with badge 7 is refused even if the badge once worked. Merely asking the supervisor to remember to stop would be weaker.
3. **Where the comparison breaks:** digital enforcement must be atomic with the protected change and must validate the actual authority source. A caller must not gain ownership merely by inventing a larger number.

34.4.3 PLAIN — a worked example

1. Our toy protected resource stores current generation 7 and value 5. A valid generation-7 request changes the value to 4.
2. A separate trusted authority transition atomically advances the resource to generation 8 before new-owner traffic is admitted. A delayed generation-7 write proposing value 99 is rejected; the value remains 4.
3. A valid generation-8 write changes it to 3. The model checks generation equality, not "accept any number larger than the last number." Untrusted callers cannot advance authority through an ordinary write.
4. This is a deterministic teaching model. Authentication, consensus-based generation allocation, durable fencing across hosts and actual device isolation are outside it.

34.4.4 PLAIN — what is really happening inside

1. If enforcement relies on remembering the highest generation ever seen, the exact acquisition and first-write protocol matters. A stale write arriving before the newer generation is registered can still be accepted unless the larger design prevents it.

2. Physical or infrastructure fencing can instead remove the old node's ability to affect the resource. That requires supported, authorized operational controls and verification, not an assumption that a failed ping means isolation.
3. Every relevant path must be covered. Fencing database writes does not automatically fence an old worker's email, object-storage update or command to another service.

34.4.5 TECHNICAL — the engineer's version

1. Chubby's original design describes sequencers containing lock generation information and recipient-side validation to reject obsolete lock-protected requests. The recipient's participation is essential. [S163]
2. PostgreSQL's failover documentation requires a mechanism preventing the old primary from continuing as primary after replacement. The exact fencing implementation is deployment-specific. [S150]
3. A correct design specifies epoch allocation, durable acceptance state, replay rules, atomic effect checks and all downstream resources. Do not describe a standalone monotonic integer as a complete distributed lock.

34.4.6 WORDS — remember these

1. **Fencing:** preventing obsolete authority from acting — enforcement at a protected resource or infrastructure boundary that rejects stale owners. **Epoch:** a generation of authority or configuration — a value distinguishing successive valid ownership periods under a defined protocol. **Sequencer:** evidence describing a lock generation — a token that a recipient validates before accepting a protected operation.

34.5 Lost acknowledgements

34.5.1 PLAIN — in simple words

1. A request can succeed even when the caller never receives the reply. The network can fail after the change is committed but before the success message arrives.
2. The caller must therefore distinguish "known failed" from "outcome unknown." Treating every timeout as failure can repeat an already completed operation.
3. A stable request identity and stored outcome help the caller reconcile what happened across a retry or leader change.

34.5.2 PLAIN — a picture in your head

1. Mira approves a reservation and files the record, but the phone call drops before Dev hears the answer.
2. Dev should ask about that same request number. Sending a brand-new request for the same goods can reserve them a second time.
3. **Where the comparison breaks:** duplicate detection must be part of the durable operation, not an informal memory of the last caller. Its retention and authority boundaries must survive the relevant failures.

34.5.3 PLAIN — a worked example

1. Request `IDEM-A` reserves three units from five, leaving two. The request record, reservation and notification intent commit together, as in Chapter 26.

2. The success reply is lost. After routing changes, the caller retries `IDEM-A` with the same validated semantic payload. A survivor containing that committed history returns the stored outcome without reserving again.
3. If an incorrectly selected replacement lacks the request record and stock change, the same retry can be treated as new. Local idempotency does not repair a failover procedure that discarded its evidence.
4. Reusing `IDEM-A` with quantity four remains a conflict, not a convenient way to edit the original request.

34.5.4 PLAIN — what is really happening inside

1. Request identity, business mutation and durable outcome must share an appropriate transaction boundary. Otherwise one can survive without the others.
2. Recovery also has to preserve enough request history for the declared retry horizon. Deleting deduplication evidence changes which old requests can safely be recognized.
3. A client may need to query an authoritative operation-status endpoint rather than blindly repeat an external action. The endpoint itself must enforce tenant and permission boundaries.

34.5.5 TECHNICAL — the engineer's version

1. The Raft paper explicitly separates log replication from client duplicate suppression: a retried command may execute again without request identifiers and retained results in the state machine. [S154]
2. Idempotency contracts must bind the request key to caller scope and semantic intent. A timeout after a possible commit requires reconciliation under that contract, not automatic assumption of rollback. [S126]
3. End-to-end guarantees include failover history selection and external-effect handling. A database commit record does not establish that a remote notification was delivered exactly once.

34.5.6 WORDS — remember these

1. **Unknown outcome:** the caller lacks enough evidence to classify completion — a state distinct from confirmed success or confirmed rollback. **Operation reconciliation:** checking a prior request's authoritative result — resolving uncertainty through stable identity and retained outcome evidence. **Retry horizon:** how long old requests remain recognizable — the interval over which a system retains the evidence needed by its idempotency contract.

34.6 Safe return of a former leader

34.6.1 PLAIN — in simple words

1. Bringing an old primary back is not the same as turning on another read-only copy. Its data may have diverged from the accepted history after the replacement took over.
2. Do not reconnect it as writable merely because its files are intact. First establish its role, compare histories and choose a supported resynchronization path.
3. Restoring redundancy is complete only when the returning participant follows the intended authority and can be observed doing so.

34.6.2 PLAIN — a picture in your head

1. An old supervisor returns with a notebook containing some private entries made while disconnected. The new supervisor has a different continuation of the shared notebook.
2. Stapling the two endings together arbitrarily can create contradictory reservations. The shop must determine the accepted history and preserve disputed evidence before rebuilding the old copy.
3. **Where the comparison breaks:** database histories include recovery positions and timelines. They cannot generally be merged by comparing visible rows or choosing the newest file timestamp.

34.6.3 PLAIN — a worked example

1. A and B share entries through 40. While isolated, A has an unaccepted local entry 41A; B becomes valid authority and commits a different entry 41B.
2. The matching sequence number does not make the entries interchangeable. The old node must not serve its 41A continuation as though it were the current history.
3. A supported rebuild or rewind procedure may return A as a follower of B, subject to its prerequisites and preservation requirements. A later integrity check alone cannot decide which business history was authorized.
4. The example is a history diagram, not permission to discard real data or run a destructive repair command.

34.6.4 PLAIN — what is really happening inside

1. Preserve the old state when investigation or reconciliation requires it. Establish the accepted authority and acknowledge any unresolved client outcomes before resynchronization.
2. Use the database's supported procedure for the exact version and configuration. Recovery must finish, and the node must be configured as the intended follower before it receives ordinary traffic.
3. Finally verify replay progress, role, read-routing behaviour, monitoring and the number of healthy copies. Record what was tested rather than declaring the cluster healthy from one successful connection.

34.6.5 TECHNICAL — the engineer's version

1. PostgreSQL `pg_rewind` can help resynchronize divergent clusters with a common history, but it has documented prerequisites involving WAL information and configuration. It is not a general merge tool. [S151]
2. Recovery replay and standby configuration remain necessary after rewind. The documentation warns that failures during the operation can leave the target unusable and that incorrect restart configuration can allow renewed divergence. [S151]
3. A failover runbook should include a safe-return phase, evidence preservation and a restoration-of-redundancy check. This book supplies reasoning templates, not a verified procedure for an unseen production cluster.

34.6.6 WORDS — remember these

1. **Divergent history:** copies contain incompatible continuations — a situation requiring authority and recovery analysis rather than simple row copying. **Reinitialization:** rebuilding a participant from valid source state — a controlled procedure restoring a compatible replica and its role. **Failover runbook:** the documented role-transition procedure — steps, prerequisites, checks and failure dispositions for transferring service authority.

34.97 Practice and worked answers

1. **Three heartbeats are missing. What is established?** Communication has not met the observation policy. A crash is possible but not proven.
2. **Why is a dashboard role change insufficient?** Existing processes, connections and downstream resources may still accept the old authority unless enforcement changes.
3. **A has committed 41, B contains 41 and C contains 40. Is C automatically a safe replacement?** No. The acknowledged-history and promotion requirements determine eligibility.
4. **A write carries generation 999. Should a resource accept it because 999 is large?** No. Generation allocation and authorization belong to the authority protocol, not an untrusted request field.
5. **A caller times out after a possible commit. What should it preserve?** The stable request identity, semantic payload and evidence needed to query or retry the same operation safely.
6. **Can local idempotency survive a failover that lost its request table?** Not automatically. The selected survivor must retain the relevant committed request and business state.
7. **Two copies have different entries numbered 41. May they be merged by number?** No. Positions are meaningful within histories and authority rules, not as universally interchangeable labels.
8. **When is the incident finished?** After the required service and redundancy checks, unresolved outcomes are recorded and the tested scope is explicit—not merely when a replacement accepts a connection.

34.98 Common wrong ideas

1. **Wrong:** timeout means dead. **Right:** it means the observer lacks a timely reply.
2. **Wrong:** the fastest responder should always become leader. **Right:** eligibility includes history and legitimate authority.
3. **Wrong:** promotion automatically fences the old primary. **Right:** stale authority needs an enforced prevention mechanism.
4. **Wrong:** a larger token is self-authenticating. **Right:** the token must originate from and be checked against valid authority.
5. **Wrong:** no reply means no write. **Right:** a committed operation can lose its acknowledgement.
6. **Wrong:** consensus automatically removes repeated business requests. **Right:** client identity and duplicate handling are additional state-machine responsibilities.
7. **Wrong:** an old primary can safely restart with its previous configuration. **Right:** role, history and recovery must be reconciled first.
8. **Wrong:** failover ends when traffic returns. **Right:** redundancy and safe return also require verification.

34.99 Chapter summary in 20 lines

1. Leadership is scoped authority, not a permanent machine identity.
2. Selection, enforcement and discovery are separate responsibilities.
3. A timeout creates suspicion rather than certainty about a crash.
4. Process pauses and network partitions can resemble failure.
5. Failure detection initiates a protocol instead of proving old authority vanished.
6. Replacement candidates need suitable history and configuration.
7. Acknowledged-data guarantees constrain failover selection.

8. PostgreSQL standby promotion is not itself a Raft election.
9. Fencing prevents obsolete owners from producing effects.
10. Enforcement must occur at every relevant protected resource.
11. Authority generations require legitimate allocation and validation.
12. A generation number alone is not a distributed lock.
13. Missing acknowledgements leave some outcomes unknown.
14. Stable request identities support safe reconciliation and replay.
15. Idempotency evidence must survive the promised failover and retry horizon.
16. External effects need their own outcome and duplicate policy.
17. A former leader may hold a divergent continuation.
18. Supported resynchronization is not arbitrary history merging.
19. Recovery and standby configuration must precede return to service.
20. Complete failover evidence includes restored redundancy and remaining limits.

35

CHAPTER

Partitioning: Dividing the Work

35.0 What this chapter gives you

1. Partitioning divides a collection into pieces. It can make some operations smaller, distribute capacity or align storage with the way records are managed. It can also make previously local questions cross several boundaries.
2. You will choose a partition key from a workload, trace range and hash placement, identify hot partitions and reason about movement, cross-partition queries and identity constraints.
3. The routing examples are small deterministic models. PostgreSQL table partitioning is documented separately from multi-host sharding; neither a Python routing function nor a partitioned table definition is a distributed database implementation.

35.1 Partition keys

35.1.1 PLAIN — in simple words

1. A partition key is the information used to decide which piece should hold a record. A date can place an order in a month; a branch identifier can place it with other records from that branch.
2. The choice should match the work. Keeping an order and its lines together can simplify order processing, while grouping by month can simplify bounded historical scans and retention.
3. No key makes every possible question local. A useful design chooses which operations deserve locality and records the cost imposed on the others.

35.1.2 PLAIN — a picture in your head

1. Mira can file invoices by month, branch or customer. Finding one month's invoices is easy in the first arrangement and may require opening many folders in the others.
2. Duplicating every invoice into every possible filing scheme creates maintenance and consistency work. The filing decision is therefore about priorities, not discovering a universally correct alphabet.
3. **Where the comparison breaks:** databases can maintain secondary indexes and distributed routing metadata. Physical placement and logical access paths need not be identical.

35.1.3 PLAIN — a worked example

1. Assume a fictional multi-branch workload where 85% of requests read or update one order within a known branch, 10% summarize one branch and 5% summarize all branches.
2. A branch-based key may keep much of the first 95% local. The all-branch report still needs a separate aggregation path or access to several partitions.

3. If one branch later produces 80% of all traffic, the same placement can become badly imbalanced. A workload percentage is a dated assumption to measure, not a permanent property of a branch name.
4. Record both record volume and request volume. Ten equally sized folders can still receive very unequal amounts of attention.

35.1.4 PLAIN — what is really happening inside

1. The application or engine maps the key to a partition. Routing must agree with the current layout version; otherwise a request can reach the wrong owner or miss a moved record.
2. Related constraints matter. A uniqueness rule spanning all records may become harder if each partition can independently accept the same value.
3. A partition key that changes during normal work can require moving records. That movement is a state transition with its own atomicity, routing and recovery requirements.

35.1.5 TECHNICAL — the engineer's version

1. Distinguish logical partitioning, physical placement and sharding across independent nodes. PostgreSQL declarative table partitioning divides a logical table into child relations; it does not automatically create a multi-host consensus or transaction layer. [S152]
2. Review query locality, write locality, cardinality, skew, key mutability and cross-partition invariants. These are workload properties, not consequences of choosing an integer key.
3. Complete identity must survive placement changes. A record should not acquire a new business identity merely because the partition directory assigns it to another host.

35.1.6 WORDS — remember these

1. **Partition key:** the value deciding placement — a field or expression mapping a record into a defined partition. **Locality:** related work stays close together — placement or access behaviour reducing the number of independent resources an operation must involve. **Shard:** an independently placed portion of data — commonly a partition hosted on a separate storage node or database, with product-specific semantics.

35.2 Range and hash approaches

35.2.1 PLAIN — in simple words

1. Range partitioning assigns intervals of ordered values to pieces. Hash partitioning transforms a key into a value used to select a piece.
2. Ranges can make date or identifier intervals easy to locate. Hashing can spread many distinct keys without preserving their original order.
3. Both methods need exact boundary rules. A record at a boundary must not disappear between two pieces or be treated as belonging to both.

35.2.2 PLAIN — a picture in your head

1. Range filing puts invoice numbers 1–99 in one drawer and 100–199 in another. Hash filing uses a repeatable sorting rule that may place neighbouring invoice numbers far apart.
2. The first helps with "all invoices from this interval." The second can spread separate lookups across drawers, but an interval question may need many of them.

3. **Where the comparison breaks:** a real hash function has defined encoding and distribution properties. The last digit of an identifier is not automatically a suitable general-purpose distribution scheme.

35.2.3 PLAIN — a worked example

1. Define ranges `[0,100)`, `[100,200)` and `[200,300)`. The left bracket includes the lower bound; the right parenthesis excludes the upper bound. Value 100 belongs only to the second range.
2. For a deliberately simple hash-routing example, place integer key `k` into `k mod 3`. Keys 0, 3, 6 and 9 go to partition 0; keys 1, 4, 7 and 10 go to 1; keys 2, 5, 8 and 11 go to 2.
3. Now change the rule to `k mod 4`. Of keys 0–11, only 0, 1 and 2 retain the same numerical destination. Nine of twelve move: 75% in this particular fixture.
4. This shows why changing the divisor is not a harmless configuration edit. The example's remainder function is pedagogical, not the PostgreSQL hash algorithm.

35.2.4 PLAIN — what is really happening inside

1. Range boundaries need coverage and non-overlap checks. Missing or default partitions require deliberate treatment, especially for new dates or previously unseen tenants.
2. Hash placement requires stable input encoding, algorithm and layout metadata. A process-randomized language hash must not silently define persistent cross-process routing.
3. Consistent-hash schemes and indirection through many small logical buckets can reduce how much placement changes when capacity is added. They do not eliminate the need to transfer records safely.

35.2.5 TECHNICAL — the engineer's version

1. PostgreSQL range partitions use inclusive lower and exclusive upper bounds; supported declarative methods also include list and hash partitioning. Their exact operator and constraint semantics are engine-specific. [S152]
2. The original Dynamo paper describes consistent hashing with virtual nodes. It is a historical design reference, not a specification for current DynamoDB or a universal recommendation for every workload. [S162]
3. A routing test should cover boundary values, missing keys, encoding changes and layout-version mismatches. Successful placement of a few ordinary keys does not establish safe rebalancing.

35.2.6 WORDS — remember these

1. **Range partition:** a piece owning an ordered interval — a partition defined by lower and upper key bounds. **Hash partition:** a piece selected through a repeatable mapping — placement based on a hash-derived value rather than original key order. **Layout version:** the identity of a placement map — metadata distinguishing successive assignments of logical partitions to owners.

35.3 Hot partitions

35.3.1 PLAIN — in simple words

1. A hot partition receives much more work than the others. The system can have spare capacity overall while that one piece is overloaded.

2. Evenly distributing record counts does not necessarily distribute reads, writes or expensive operations evenly. One popular product can receive more requests than thousands of quiet products combined.
3. Adding more partitions helps only if the hot work can actually be divided or served differently without breaking its correctness rules.

35.3.2 PLAIN — a picture in your head

1. Ten service counters each hold the same number of customer files, but nearly everyone needs the file kept at counter 1. Nine idle counters do not make that queue disappear.
2. Copying a read-only catalogue description may help. Allowing ten counters independently to promise the same final stock item creates a different problem.
3. **Where the comparison breaks:** some systems support coordinated counters, escrow-like allocations or specialized contention management. Those are additional designs, not consequences of drawing more boxes.

35.3.3 PLAIN — a worked example

1. Imagine 1,000 requests per second and ten partitions. One key attracts 700 requests per second; all other keys together attract 300.
2. Even if the remaining traffic distributes perfectly, the hot key's owner receives at least 700 requests per second while the average is only 100. A dashboard showing average load hides the concentration.
3. Hashing that same key more carefully does not split its requests: identical keys still map together under a deterministic placement rule.
4. A possible redesign could separate immutable product-description reads from authoritative stock reservations. Whether that helps depends on which part of the 700 requests is actually expensive and whether the new read path meets its freshness contract.

35.3.4 PLAIN — what is really happening inside

1. Measure per-partition and per-key workload where privacy and cardinality limits permit. Request counts, bytes, latency and lock contention reveal different kinds of concentration.
2. Sequential keys can concentrate recent writes at one range edge. A time-based partition may direct all current events to the newest piece.
3. Splitting a hot logical entity requires a merge or coordination rule for its state. Without one, the design has merely replaced a bottleneck with inconsistent independent answers.

35.3.5 TECHNICAL — the engineer's version

1. Distribution quality must be assessed against the request distribution, not only the key-space distribution. Hash uniformity cannot remove single-key contention.
2. Virtual partitions can improve placement flexibility and balancing across nodes, but the original Dynamo discussion also identifies non-uniform load and heterogeneous capacity as design concerns. [S162]
3. Report maxima and relevant latency distributions alongside averages. A useful intervention is tied to an observed bottleneck and includes a correctness-preserving way to divide or duplicate the work. [S103]

35.3.6 WORDS — remember these

1. **Hot partition:** one piece receives disproportionate work — a localized throughput, latency or resource bottleneck. **Skew:** an uneven distribution — concentration in key frequency, record size, cost or request rate. **Single-key contention:** many operations compete on one identity — a workload limitation that ordinary redistribution of unrelated keys cannot remove.

35.4 Rebalancing

35.4.1 PLAIN — in simple words

1. Rebalancing moves responsibility or data between pieces as the system grows or load changes. It is a migration, not simply editing an address.
2. The process must account for writes that happen while copying, requests using an older map, retries and incomplete transfers.
3. A successful copy is only one checkpoint. The system also needs a safe authority switch and evidence that the old owner can no longer create an incompatible continuation.

35.4.2 PLAIN — a picture in your head

1. Mira moves one filing drawer to a new branch while staff are still adding invoices. Copying the drawer at noon and changing the sign at one o'clock can miss everything added during that hour.
2. Staff holding an old directory may continue sending papers to the original branch. The move needs a forwarding or rejection rule and a clear cutover boundary.
3. **Where the comparison breaks:** physical papers do not normally arrive twice after a timeout. Distributed transfers must handle duplication and stale routing explicitly.

35.4.3 PLAIN — a worked example

1. In a toy directory, bucket 12 belongs to A under layout 4. A snapshot is copied to B at sequence 100, but A then accepts changes 101–105.
2. Switching bucket 12 to B at layout 5 before B has those changes would omit accepted work. The transfer must apply the required suffix or pause and reconcile under a controlled procedure.
3. At cutover, the design records a valid ownership transition and prevents A from accepting layout-4 writes as current. Stale clients receive a redirect or an explicit retryable routing error according to the protocol.
4. The copy and suffix examples describe prerequisites. The companion routing model does not implement an online rebalance service or prove a zero-downtime migration.

35.4.4 PLAIN — what is really happening inside

1. A bounded pause can simplify a small migration: stop eligible writes, establish the final source boundary, finish the copy, verify, transfer authority and resume. Its availability cost must be declared.
2. An online design may capture changes during copying, but it then needs ordering, duplicate handling, compatible schemas and a valid final handover. Naive independent dual writes can leave destinations different.
3. Preserve rollback or roll-forward evidence. Once new authoritative writes occur at B, returning traffic to an old snapshot at A is not automatically a safe rollback.

35.4.5 TECHNICAL — the engineer's version

1. Treat data movement as an ownership protocol with snapshot position, catch-up position, routing version and fencing conditions. Atomicity of a local copy command does not establish atomic distributed cutover.
2. Rebalance bandwidth competes with foreground operations, replication and recovery. Rate limits and headroom belong in the plan, alongside correctness checks.
3. Engine-specific partition attachment and detachment have their own locking and validation rules. PostgreSQL's declarative partition operations are not equivalent to moving an independently writable shard between hosts. [S152]

35.4.6 WORDS — remember these

1. **Rebalancing:** changing where pieces are served — controlled movement of data or ownership to meet capacity and workload needs. **Cutover:** the point a new path becomes authoritative — a transition with explicit prerequisites and postconditions. **Catch-up suffix:** changes after an initial copy boundary — the remaining ordered work needed before a destination matches the selected source state.

35.5 Cross-partition questions

35.5.1 PLAIN — in simple words

1. A question involving several pieces needs a rule for combining their answers. Adding subtotals can be valid; averaging averages often is not.
2. The pieces may also be observed at different times. A report can mix incompatible states even when each local query is correct.
3. A distributed write is harder still: changing two independent owners does not automatically become one all-or-nothing operation. Chapter 38 develops that boundary.

35.5.2 PLAIN — a picture in your head

1. Mira asks each branch for sales totals. She must know whether all branches used the same date range, currency, exclusions and revision boundary.
2. If one branch reports "today so far" at noon and another at three o'clock, adding their numbers does not create a report for a single common instant.
3. **Where the comparison breaks:** distributed systems may support coordinated snapshots or timestamp protocols. These require explicit guarantees; they are not provided by sending requests at nearly the same time.

35.5.3 PLAIN — a worked example

1. Branch A has two hypothetical order totals, 100 and 300 paise. Its sum is 400, count 2 and average 200. Branch B has eight orders of 100 paise, giving sum 800, count 8 and average 100.
2. The overall average is `(400 + 800) / (2 + 8) = 120` paise. Averaging the branch averages gives `(200 + 100) / 2 = 150`, which answers a differently weighted question.
3. Correct partial aggregation carries the information needed for the final calculation: sum and count here, not only average.
4. These are separate fictional report values, not replacements for the canonical O-1042 and O-1043 amounts.

35.5.4 PLAIN — what is really happening inside

1. A scatter-gather query sends work to relevant partitions and combines responses. The slowest necessary response, retries and network traffic can dominate completion time.
2. Partial results need explicit labelling. Returning nine of ten partitions without disclosing the missing one can produce a plausible but incomplete business total.
3. Joins across placement boundaries may require moving rows or partial results. The optimizer and data design must account for bytes transferred as well as local computation.

35.5.5 TECHNICAL — the engineer's version

1. Aggregation must preserve grain, population and mergeable state. Sums and counts can be combined under compatible definitions; averages require their weights, and exact distinct counts cannot generally be added across overlapping populations. [S19]
2. A local transaction boundary does not imply a globally consistent snapshot across independent databases. Specify the supported distributed-read or reconciliation procedure.
3. For partitioned PostgreSQL tables, uniqueness constraints must satisfy documented partition-key restrictions. A local unique index on each independently managed shard is not a global uniqueness service. [S152]

35.5.6 WORDS — remember these

1. **Scatter-gather:** ask several pieces and combine their replies — a distributed query pattern with completeness and coordination requirements. **Partial aggregation:** summarize locally before combining — retaining sufficient intermediate state for the intended final calculation. **Global invariant:** a rule spanning several owners — a correctness condition not established by independent local checks alone.

35.6 Growth without permanent shortcuts

35.6.1 PLAIN — in simple words

1. A design chosen for today's scale should make its assumptions visible so they can be revisited. Hard-coding "all important customers are in partition 0" creates hidden dependency rather than a growth strategy.
2. It is often sensible to begin with fewer moving parts. The requirement is not to distribute everything immediately, but to avoid confusing a temporary placement decision with a permanent business truth.
3. Stable identities, documented boundaries and measured workloads make later changes easier to reason about.

35.6.2 PLAIN — a picture in your head

1. Mira labels folders with durable invoice identities and keeps a separate directory showing where they are stored. A moved folder changes the directory, not the invoice's identity.
2. Writing the shelf location into every invoice number would make every reorganization a renaming exercise with many opportunities for mismatch.
3. **Where the comparison breaks:** indirection itself has cost and failure modes. A directory must be available, versioned and protected; it is not a free extra layer.

35.6.3 PLAIN — a worked example

1. A toy ring has owners at positions 20, 60 and 90 on a 0–99 circle. A key goes to the first owner clockwise, wrapping to 20 after 90.
2. Key 25 belongs to owner 60. Adding an owner at 40 moves keys in the interval `(20,40]` to the new owner while leaving the other intervals' ownership unchanged in this model.
3. Key 25 now maps to 40; key 65 remains at 90; key 95 remains at 20. This is placement arithmetic, not evidence that the records were transferred or that concurrent requests were handled safely.
4. The lab tests the deterministic mapping and boundary convention separately from any migration claim.

35.6.4 PLAIN — what is really happening inside

1. Separate logical identity, partition identity and physical location. They may change on different schedules and should not be collapsed into one ambiguous field.
2. Record what would trigger a redesign: sustained hot-partition latency, recovery-time pressure, storage growth or an invariant that no longer fits the existing owner boundary.
3. Reassess the full workload after a change. Improving write distribution may worsen reports, backups, index maintenance or operational recovery.

35.6.5 TECHNICAL — the engineer's version

1. Consistent hashing provides a placement technique with limited remapping under membership changes, subject to its precise scheme. It does not itself supply replication, conflict resolution, migration atomicity or authorization. [S162]
2. Maintain versioned placement metadata and test invalid or stale maps. Treat routing configuration as part of the data system's correctness surface.
3. Growth decisions should use measured distribution and operational requirements. The companion examples demonstrate mapping and arithmetic; no production scalability or online-rebalance claim follows from them.

35.6.6 WORDS — remember these

1. **Indirection:** use a directory instead of embedding a location — a mapping layer separating stable identity from changing placement. **Consistent hashing:** a scheme limiting remapping when owners change — a family of placement techniques whose exact balance and movement properties depend on the design. **Placement invariant:** a rule about where records belong — a condition linking keys, layout versions and valid owners.

35.97 Practice and worked answers

1. **Which range owns 100 under `[0,100)` and `[100,200)`?** The second. Half-open intervals avoid overlap at the boundary.
2. **How many keys 0–11 change numerical destinations when `mod 3` becomes `mod 4`?** Nine. Only 0, 1 and 2 retain their old destination in this fixture.
3. **Will a uniform hash split 700 requests to one key across ten owners?** Not under ordinary deterministic single-owner placement. The same key continues to map together.
4. **What is missing from "copy bucket 12, then change its address"?** A boundary for concurrent writes, catch-up, ownership transfer, stale routing, validation and failure handling.

5. **How should averages from branches with different order counts be combined?** Combine compatible sums and counts, then divide. Do not take an unweighted average of averages unless that is the intended metric.
6. **May a report hide that one required partition failed to answer?** No. Completeness is part of the result contract.
7. **Does a local unique index on every shard ensure a globally unique email address?** No. Independent owners can accept the same value unless a broader mechanism or scoped identity rule prevents it.
8. **What does the ring example establish?** Its placement and limited-remapping behaviour under the stated boundary rule. It does not establish safe data movement or distributed correctness.

35.98 Common wrong ideas

1. **Wrong:** partitioning always means multiple machines. **Right:** a table can be partitioned within one database server.
2. **Wrong:** equal record counts mean equal load. **Right:** request frequency and operation cost can be highly skewed.
3. **Wrong:** a better hash fixes every hotspot. **Right:** single-key contention needs a different workload or coordination design.
4. **Wrong:** changing the partition count is just a configuration update. **Right:** it may change record placement and require a migration.
5. **Wrong:** copying data transfers authority. **Right:** authority and stale-writer prevention need explicit cutover rules.
6. **Wrong:** local correctness automatically composes globally. **Right:** cross-partition snapshots, writes and invariants need their own guarantees.
7. **Wrong:** partial query answers can be silently treated as complete. **Right:** missing partitions change the meaning of the result.
8. **Wrong:** consistent hashing is a complete database. **Right:** it is one placement technique among many other required mechanisms.

35.99 Chapter summary in 20 lines

1. Partitioning divides a logical collection into pieces.
2. A partition key should follow measured work and explicit priorities.
3. Record volume and request volume are different distributions.
4. Logical partitioning does not automatically imply multi-host sharding.
5. Range boundaries require complete and non-overlapping rules.
6. Hash placement requires stable encoding and mapping definitions.
7. Changing a modulus can move many keys.
8. Consistent hashing can limit remapping under a specified scheme.
9. Placement arithmetic does not move or validate records.
10. A single popular key can overload one otherwise balanced partition.
11. Monitor per-partition pressure rather than only averages.
12. Rebalancing is a data and authority migration.
13. Initial copies need compatible catch-up and cutover boundaries.

14. Stale routing and old writers need explicit handling.
15. Cross-partition reports require compatible populations and snapshots.
16. Combine sufficient aggregate state rather than blindly averaging averages.
17. Missing required partitions must remain visible in results.
18. Global invariants are not guaranteed by independent local constraints.
19. Separate stable business identity from changing physical location.
20. Revisit placement using workload evidence and operational requirements.

36

CHAPTER

Consistency, Availability and Network Partitions

36.0 What this chapter gives you

1. "The system is consistent" is incomplete unless we state which observations are allowed. A database can obey its local constraints while different clients observe incompatible versions of a shared value.
2. You will learn the meanings needed to reason about a network split: linearizability, weaker read contracts, availability, quorum intersection and the limits captured by CAP.
3. The examples are small histories and set calculations. They are not rankings of database products and do not prove that a real distributed deployment satisfies a consistency model.

36.1 Name the consistency model

36.1.1 PLAIN — in simple words

1. A consistency model describes what different readers and writers may observe. It is a contract about histories of operations, not simply whether all copies eventually contain the same bytes.
2. A strong single-object contract can make the object appear to change at one instant between each request and its reply. If one write finishes before a later read starts, that read cannot pretend the completed write never happened unless a later valid change explains the result.
3. Weaker contracts may allow older values or different observations under stated rules. They can be useful, but only when the application knows what it is accepting.

36.1.2 PLAIN — a picture in your head

1. Imagine a single reservation board watched by several people. In the strongest simple picture, every operation has one place in a shared timeline that respects actions already completed before another begins.
2. A system of copied boards may instead promise that everyone will eventually catch up, without promising that each immediate read is current.
3. **Where the comparison breaks:** the contract concerns observable operation histories. It does not require a physically instantaneous update to every machine or a universal perfectly synchronized wall clock.

36.1.3 PLAIN — a worked example

1. A register starts at 0. Client X writes 1 and receives success. Only afterwards, client Y invokes a read. No other write occurs.

2. A linearizable read must return 1. Returning 0 cannot be placed after the completed write in a valid single-register history.
3. If the read overlaps the write instead, either 0 or 1 can be consistent with a suitable ordering, depending on the rest of the history. Overlap is not the same as "the write finished first."
4. A later valid write of 0 changes the analysis again. Correctness is judged against the whole relevant history, not a rule that values must always increase.

36.1.4 PLAIN — what is really happening inside

1. Real-time precedence means the order of non-overlapping completed operations, not comparing arbitrary timestamps from different machines. The system can reason about messages and completion boundaries without assuming identical clocks.
2. Sequential consistency requires a shared order respecting each participant's program order but does not impose every cross-client real-time precedence constraint. Causal consistency preserves relevant cause-and-effect dependencies while allowing more freedom for unrelated work.
3. Eventual convergence is weaker still unless supplemented with precise guarantees. It needs conditions such as delivery, conflict resolution and an opportunity to finish propagating updates; the word "eventual" is not a deadline.

36.1.5 TECHNICAL — the engineer's version

1. The CAP discussion here uses atomic read/write object semantics, commonly described as linearizability: each operation appears at a point within its invocation-response interval. [S153]
2. Do not conflate single-object linearizability with transaction serializability or with the C in ACID. Transaction isolation concerns relationships among transactional operations; application consistency concerns invariants. [S66] [S117]
3. A specification should name object scope, session guarantees, visibility rules, failure behaviour and any staleness bound. "Strong" or "eventual" without those details is not sufficient for a design review.

36.1.6 WORDS — remember these

1. **Linearizability:** operations appear to take effect in one real-time-respecting order — a correctness condition placing each operation within its invocation-response interval. **Sequential consistency:** one shared order respects each client's own order — a model that does not additionally require all cross-client real-time precedence. **Causal consistency:** effects follow their relevant causes — a model preserving causal dependencies while allowing flexibility for concurrent unrelated operations.

36.2 What availability promises

36.2.1 PLAIN — in simple words

1. Availability in an operational dashboard often means the proportion of requests answered acceptably within a deadline. CAP uses a theoretical requirement: requests at non-failed participants eventually receive the required responses.
2. These are different measurements. A response after an hour might satisfy an eventual-response property while being useless for a checkout that must finish in two seconds.

3. Returning an error to every valid request does not magically satisfy the intended read/write service. The response must be interpreted under the service specification rather than counted as success merely because some bytes arrived.

36.2.2 PLAIN — a picture in your head

1. A shop that promises to answer every customer eventually has made a different promise from one that promises to answer 99.9% within two seconds.
2. A sign saying "unavailable" is a response in everyday language, but it does not provide the requested reservation or current stock answer.
3. **Where the comparison breaks:** formal availability is a property quantified over executions and requests. A month's observed uptime is empirical evidence for an operational target, not a proof over all possible executions.

36.2.3 PLAIN — a worked example

1. Consider a hypothetical one-hour test with 10,000 eligible reads. Of these, 9,990 return valid results within the agreed deadline, five return late and five fail.
2. Deadline-qualified success is 9,990 / 10,000 = 99.9%. Counting the late responses as acceptable would produce a different metric and must be declared.
3. That finite test does not establish CAP availability under every possible partition. It records the behaviour of one workload during one observation window.
4. Separate the denominator, deadline, excluded requests and response-validity rule before comparing systems or reporting service health.

36.2.4 PLAIN — what is really happening inside

1. An operation may require coordination with a remote participant. If the remote path is unavailable, the implementation must wait, use a supported weaker mode or reject according to its contract.
2. The ability to serve some operations does not imply the ability to serve all operations. A catalogue page may remain available while a stock reservation is deliberately paused.
3. Availability is therefore often operation-specific. A system-level percentage can hide that the most important action is the one failing.

36.2.5 TECHNICAL — the engineer's version

1. The formal CAP availability condition is eventual completion of requests under the specified service and failure model, not a latency percentile or a commercial uptime percentage. [S153]
2. Operational service-level indicators should count eligible events under a defined success predicate and time boundary. Their measurement does not establish a universal distributed-systems theorem. [S103]
3. Document degradation by operation: authoritative writes, stale-tolerant reads, status lookups and administrative changes may have different coordination requirements.

36.2.6 WORDS — remember these

1. **Theoretical availability:** required requests eventually complete — a liveness property under a stated distributed-system model. **Service-level indicator:** a measured service outcome — a defined quantity such as the fraction of eligible requests meeting a correctness and latency condition.

Degraded mode: a documented reduced service — an operational state with explicit limits rather than an undisclosed weakening of guarantees.

36.3 Partitioned communication

36.3.1 PLAIN — in simple words

1. A network partition prevents some participants from communicating with others. The machines on both sides may still be running and receiving client requests.
2. Each side can lack information about actions accepted on the other. Waiting longer may help when the interruption ends, but no finite wait can guarantee delivery during an indefinitely lasting split.
3. A system must decide which actions it can safely perform with the information it has. It cannot obtain missing knowledge merely by adding a timeout.

36.3.2 PLAIN — a picture in your head

1. Two branches lose their connecting phone line while both remain open. Each still has the last shared stock figure, but neither can learn about new promises made by the other.
2. If both sell from the same undivided last item, a later conversation may reveal that two customers were promised one object.
3. **Where the comparison breaks:** some applications can pre-allocate disjoint rights or use operations designed to merge safely. Those approaches change the coordination requirement rather than making communication failure disappear.

36.3.3 PLAIN — a worked example

1. Three replicas A, B and C begin with value 0. The network splits into `{A}` and `{B,C}`.
2. If B and C use an established majority protocol, that side may remain eligible to accept coordinated writes. A cannot obtain a majority by counting itself twice or inventing a replacement member.
3. A can still answer an explicitly stale-tolerant question from its local copy, but it cannot represent that answer as satisfying a current linearizable read without the necessary evidence.
4. Membership, quorum and read rules must already define the behaviour. The set diagram alone does not implement them.

36.3.4 PLAIN — what is really happening inside

1. Network problems can be asymmetric: A may reach B while B's replies do not reach A, or one client may reach an old leader while another reaches its replacement.
2. The system's authority and consistency mechanisms must handle delayed messages after recovery. Reconnection does not automatically make incompatible accepted histories mergeable.
3. Avoid changing membership independently on both sides merely to regain a local majority. That can create two disjoint groups each calling itself authoritative.

36.3.5 TECHNICAL — the engineer's version

1. Treat partition tolerance as a failure-model requirement: communication may be delayed or lost between components. It is not a third optional feature that can be removed from a real network by a configuration slogan. [S153]

2. Consensus membership changes need rules that preserve the relevant quorum intersection. The Raft paper's joint-consensus design requires majorities of both configurations during its transition. [S154]
3. Failure-injection evidence should specify direction, duration, affected paths, clock assumptions and recovery behaviour. Killing a process is not the same experiment as partitioning live participants.

36.3.6 WORDS — remember these

1. **Network partition:** some participants cannot communicate — a communication failure that can separate live components. **Asymmetric failure:** reachability differs by direction or path — a condition in which one observer's connectivity does not describe the whole system. **Membership configuration:** the participants counted by the protocol — a versioned set whose changes require safety-preserving rules.

36.4 CAP with precise definitions

36.4.1 PLAIN — in simple words

1. CAP identifies a limit: under partitioned communication, a shared read/write service cannot guarantee both the specified single-copy current behaviour and eventual completion of every required request at non-failed participants.
2. The conflict comes from missing information. One side cannot always know whether a completed write occurred on the other side while the two cannot communicate.
3. This is not a instruction to choose two attractive letters on a product brochure. It is a reason to specify what the application does when coordination is unavailable.

36.4.2 PLAIN — a picture in your head

1. Dev must answer a question about a board in a room he cannot contact. In one possible history, Mira has changed the board to 1. In another, she changed it to 2.
2. Dev sees the same evidence in both histories. A definite answer cannot be guaranteed correct for both, and waiting forever fails the requirement to eventually answer.
3. **Where the comparison breaks:** the theorem has precise object and execution definitions. An analogy cannot establish every possible trade-off for every kind of service.

36.4.3 PLAIN — a worked example

1. Let replicas A and B be separated. Consider history H1: A completes a write of 1. Consider H2: A completes a write of 2. B receives no messages distinguishing H1 from H2.
2. A later read reaches B. To satisfy linearizable register semantics, its answer must be 1 in H1 and 2 in H2, assuming no intervening writes.
3. But B has the same local evidence in both histories. A deterministic choice cannot be correct in both; waiting for information that may never arrive violates eventual-response availability.
4. This original worked trace illustrates the information conflict discussed by Gilbert and Lynch. It does not claim that every useful distributed service requires this exact register contract. [S153]

36.4.4 PLAIN — what is really happening inside

1. A service can preserve the stronger contract by refusing or delaying operations that cannot safely coordinate. It can instead provide explicitly weaker observations or accept operations under a conflict-resolution policy.
2. Different actions may make different choices. Reading an older product description is not equivalent to promising the same last physical unit twice.
3. Once the partition ends, reconciliation still needs an application meaning. Convergence to one value does not undo commitments already made to customers.

36.4.5 TECHNICAL — the engineer's version

1. In this chapter C means linearizable or atomic read/write semantics, not schema constraints or ACID consistency. A means the formal availability condition, not a measured success percentage. P describes unreliable communication including partitions. [S153]
2. The impossibility concerns guarantees across the allowed executions. It does not say that a healthy connected deployment must always choose between correct responses and availability on each ordinary request.
3. A system may also fail to provide either desired property because of bugs, overload or incomplete design. The theorem is a limit, not a certificate that any chosen implementation provides the remaining letters.

36.4.6 WORDS — remember these

1. **CAP theorem:** a limit on joint guarantees under unreliable communication — the incompatibility of the specified atomic service and availability requirements in the partition-prone model. **Indistinguishable histories:** different realities look the same to an observer — executions with identical local evidence that require different correct outcomes. **Consistency downgrade:** a weaker observation contract is used — a change that must be explicit and acceptable for the operation.

36.5 Quorum reasoning and limits

36.5.1 PLAIN — in simple words

1. A quorum is a sufficiently large set of participants for a particular protocol step. Sets can be chosen so that any two relevant quorums overlap.
2. Overlap means they share at least one participant. It does not, by itself, explain which value that participant stores, which version a reader chooses or how concurrent writes are ordered.
3. Quorum arithmetic is therefore one ingredient of a protocol, not a complete substitute for one.

36.5.2 PLAIN — a picture in your head

1. In a group of five clerks, any two groups of three share at least one clerk. That shared person can connect the groups' information only if the rules require them to retain and respect the relevant evidence.
2. If the clerk forgets an accepted decision, gives two incompatible approvals or answers with an old note, the head count alone does not save the procedure.
3. **Where the comparison breaks:** real protocols specify durable votes, versions, message handling and membership changes. People standing in overlapping circles are only the set-theoretic part.

36.5.3 PLAIN — a worked example

1. With `N=3`, a write set of size `W=2` and a read set of size `R=2` must overlap because `R+W=4 > 3`. The minimum overlap is `R+W-N=1` participant.
2. With `N=5`, two majorities of size 3 also overlap in at least one participant. The lab enumerates such sets and checks the intersection property directly.
3. Now weaken the read to `R=1` while keeping `W=2` among three replicas. A read from the third untouched replica can miss a completed write.
4. Even `R=2,W=2` does not alone prove linearizability. Consider concurrent versions, incomplete writes, repair rules, the selection of the latest value and whether all operations used the same membership.

36.5.4 PLAIN — what is really happening inside

1. Protocols attach meaning to the intersection. A voter may remember an accepted term and log, or a storage replica may keep versioned values and participate in a defined read-repair procedure.
2. "Sloppy" quorum schemes can use substitute nodes outside the usual replica set during failures. The ordinary fixed-set intersection argument cannot simply be copied without examining that change.
3. Wall-clock timestamps are not automatically a safe ordering mechanism. Clock skew and concurrent updates can make the largest displayed timestamp a poor representation of causal or real-time order.

36.5.5 TECHNICAL — the engineer's version

1. For fixed subsets of an N-member universe, `|R ∩ W| >= |R| + |W| - N`. The bound proves intersection only; it does not prove an atomic-register implementation.
2. The original Dynamo design uses version handling, reconciliation and sloppy quorums to pursue its stated availability goals. It should not be summarized as "R+W>N therefore linearizable." [S162]
3. Review membership, write completion, version selection, partial failures, concurrent writes and read repair before asserting a consistency guarantee. The book's set-enumeration tests validate arithmetic, not a complete distributed protocol.

36.5.6 WORDS — remember these

1. **Quorum:** a participant set sufficient for a protocol action — a set meeting defined size or intersection requirements. **Quorum intersection:** relevant sets share participants — a necessary ingredient in many protocols whose meaning depends on retained state and rules. **Sloppy quorum:** substitute reachable nodes may count — a failure-handling scheme that differs from always using one fixed replica set.

36.6 Application-level consequences

36.6.1 PLAIN — in simple words

1. The useful question is not "Which consistency word sounds strongest?" It is "Which incorrect or delayed outcomes can this operation tolerate, and which must be prevented?"
2. The answer may differ between browsing a catalogue, editing a draft, reserving scarce stock and calculating a historical report.

3. A documented refusal can be safer than a confident but unsupported success. A clearly labelled older answer can also be useful when the task genuinely permits it.

36.6.2 PLAIN — a picture in your head

1. A shop can show yesterday's opening-hours brochure while its branch link is down, provided the reader is told its date. It should not call that brochure a confirmed reservation for the final item.
2. The same communication outage therefore leads to different behaviour for different tasks.
3. **Where the comparison breaks:** the acceptable policy belongs to the application owner and users. A technical mechanism does not decide the commercial or safety consequences on their behalf.

36.6.3 PLAIN — a worked example

1. Define three fictional contracts: catalogue descriptions may be up to ten minutes old; a user's draft read must include their last acknowledged edit; a reservation may succeed only through the current authorized stock owner.
2. During a split, a cached catalogue may still be eligible if its age and scope meet the contract. A draft read may need to wait or route elsewhere. A reservation on an isolated stale node must not pretend to succeed authoritatively.
3. The ten-minute allowance is a chosen example policy, not a universal freshness recommendation. Each decision should be tested against the actual harm of stale or unavailable information.
4. Record the mode in the response and operational evidence so later reports can distinguish authoritative outcomes from permitted stale views.

36.6.4 PLAIN — what is really happening inside

1. Consistency requirements propagate through caches, replicas, sessions, transactions and downstream services. A strong database underneath a stale cache does not automatically provide a strong application read.
2. Tests should include ambiguous boundaries: write completed but reply lost, follower behind, ownership changed, cache invalidation delayed and only part of a report available.
3. Review the whole request path, including what the user sees. Correct internal refusal followed by a misleading "confirmed" screen is still an application defect.

36.6.5 TECHNICAL — the engineer's version

1. Express each operation's contract as allowed histories and observable outcomes. Include degraded-mode semantics, retries, deadlines and source-of-authority checks.
2. Combine model reasoning with actual implementation tests. Neither a CAP explanation nor a passing quorum arithmetic test certifies a production application. [S153] [S154]
3. Preserve the difference between a proven impossibility under a formal model, a documented product guarantee and an empirical observation from a bounded experiment.

36.6.6 WORDS — remember these

1. **Operation contract:** the promise made for one action — a specification of valid results, timing, visibility and failure outcomes. **Authoritative outcome:** a result accepted under the valid authority path — a state distinct from a cached guess or tentative local proposal. **Allowed history:** a sequence the contract permits — an operation trace used to evaluate a consistency or correctness claim.

36.97 Practice and worked answers

1. **A write of 1 completes before a read begins, with no later writes. Is a returned 0 linearizable?** No. It cannot fit the required real-time-respecting register history.
2. **Does linearizability require every machine's clock to show the same time?** No. The property concerns invocation-response ordering, not identical wall-clock displays.
3. **Does 99.9% observed deadline success prove formal CAP availability?** No. It describes a finite operational measurement under a defined denominator.
4. **Why cannot an isolated reader distinguish H1's completed write of 1 from H2's completed write of 2?** The relevant messages are absent in both local observations.
5. **What is the minimum overlap of sets of size 3 and 3 in a universe of 5?** At least one member, since 3+3-5=1.
6. **Does that overlap prove linearizability?** No. Version, completion, concurrency, failure and membership rules remain necessary.
7. **May both sides independently replace membership until each has a majority?** Not safely without a protocol preserving the required cross-configuration intersection.
8. **Why can catalogue reads and reservations have different partition behaviour?** They have different correctness and freshness requirements. The distinction must be explicit, not hidden in a generic availability label.

36.98 Common wrong ideas

1. **Wrong:** consistency has one universal meaning. **Right:** name the observation model and scope.
2. **Wrong:** ACID consistency is CAP consistency. **Right:** these terms address different properties.
3. **Wrong:** eventual means within a fixed short delay. **Right:** a deadline needs an additional explicit bound and assumptions.
4. **Wrong:** CAP is always "pick any two features." **Right:** it concerns joint guarantees under unreliable communication and precise definitions.
5. **Wrong:** every error response counts as useful availability. **Right:** responses must satisfy the intended service contract.
6. **Wrong:** quorum intersection alone proves the newest value is returned. **Right:** protocol rules give the intersection its meaning.
7. **Wrong:** reconnecting automatically repairs business promises. **Right:** converged storage may still require application reconciliation.
8. **Wrong:** a strong database guarantees a strong application. **Right:** every layer on the request path must preserve the contract.

36.99 Chapter summary in 20 lines

1. Consistency models define allowed observations across operations.
2. Linearizability respects the real-time order of non-overlapping operations.
3. Overlapping operations may admit more than one valid order.
4. Sequential consistency and causal consistency impose different constraints.
5. Eventual convergence is not a fixed freshness deadline.
6. Transaction isolation and application invariants are separate concepts.
7. CAP availability is a theoretical eventual-completion property.

8. Operational availability requires a defined success predicate and denominator.
9. A network partition can separate participants that remain alive.
10. Timeouts cannot manufacture missing information.
11. Membership changes must preserve protocol safety.
12. CAP exposes an information conflict under precise assumptions.
13. It is not a product ranking or a certificate of implementation quality.
14. Fixed-set quorum arithmetic can prove overlap.
15. Overlap alone does not prove linearizable behaviour.
16. Version selection, partial writes and membership matter.
17. Sloppy quorums change the ordinary fixed-replica analysis.
18. Different operations can justify different degraded modes.
19. Caches and routing must preserve the intended application contract.
20. Distinguish formal reasoning, documented guarantees and measured evidence.

37

CHAPTER

Consensus: Agreeing Despite Failure

37.0 What this chapter gives you

1. Consensus helps participants agree on a decision despite specified failures. In a replicated-log system, the decisions determine which commands occupy the shared history and in what order they may be applied.
2. You will follow a small Raft-style trace through terms, durable voting, log replication, commitment and a leader change. You will distinguish safety from progress and a protocol explanation from a complete implementation.
3. The companion exercises check selected predicates and traces. They are not a new consensus library, a formal proof of Raft or evidence that a production cluster survived arbitrary failures.

37.1 The agreement problem

37.1.1 PLAIN — in simple words

1. Copies are useful only if the system has rules for deciding which changes count. Two replicas independently accepting incompatible commands at the same position cannot both be treated as one coherent history.
2. A replicated state machine applies the same agreed commands in the same order. If the starting state and command behaviour are suitably defined, the replicas can produce matching results.
3. Agreement is not the same as judging whether the command reflects reality. A group can agree to store an incorrect price if the application supplied one.

37.1.2 PLAIN — a picture in your head

1. Three clerks maintain copies of an instruction ledger. Before an instruction becomes official, they follow rules determining that this exact instruction occupies this exact ledger position.
2. Each clerk then carries out official instructions in order. A reservation after a replenishment can differ from the same reservation before it, so order matters.
3. **Where the comparison breaks:** a consensus protocol depends on precise persistent state, messages and failure assumptions. Informal discussion among clerks is not an equivalent algorithm.

37.1.3 PLAIN — a worked example

1. Begin a separate fictional stock state at 5. Command A reserves 3 if enough stock remains. Command B reserves 4 under the same rule.
2. Applying A then B accepts A, leaves 2 and rejects B. Applying B then A accepts B, leaves 1 and rejects A.

3. Both orders can preserve non-negative stock, but they produce different accepted customers. Replicas must not silently choose different orders while claiming one shared result.
4. Consensus can choose the order. The state machine must still implement the reservation rule correctly and retain request identities so a retry does not become a second reservation.

37.1.4 PLAIN — what is really happening inside

1. The log records commands, while the application state is the result of applying the committed prefix. A stored entry need not already be committed, and a committed entry need not already be applied on every follower.
2. Commands should not independently consult uncontrolled local time, randomness or outside services and then expect identical results. Nondeterministic choices need to be controlled or represented in the agreed input under the system's design.
3. External actions remain separate. Agreeing that an email should be sent does not establish that the recipient received it once.

37.1.5 TECHNICAL — the engineer's version

1. Replicated state machines require an agreed command sequence and deterministic or otherwise consistently specified application semantics. Raft decomposes replicated-log consensus into leader election, log replication and safety rules. [S154]
2. Distinguish log agreement from domain validation, authorization and external-effect delivery. Consensus does not replace constraints, idempotency or an outbox protocol.
3. The failure model here is crash and communication failure, not arbitrary malicious participants fabricating votes or violating the algorithm. Byzantine fault tolerance requires different assumptions and mechanisms.

37.1.6 WORDS — remember these

1. **Consensus:** agreeing under a specified failure model — a protocol problem requiring compatible decisions and defined progress conditions. **Replicated state machine:** copies execute one agreed command sequence — an architecture connecting log agreement to matching application state. **Committed prefix:** the log portion accepted as final under the protocol — entries eligible for ordered application rather than merely present on one disk.

37.2 Logs and terms

37.2.1 PLAIN — in simple words

1. A log position says where an entry sits. A term identifies a period of attempted leadership. The two numbers answer different questions.
2. Terms let participants recognize obsolete messages and authority. Log entries retain the term in which the leader originally accepted them.
3. A participant must remember important voting and log information across restarts. Forgetting a previous vote can allow incompatible decisions that the protocol was designed to prevent.

37.2.2 PLAIN — a picture in your head

1. A ledger has page numbers and a separate appointment number for each supervisor. Page 8 might have been written during appointment 3.

2. A new supervisor does not renumber every old page as though it were newly created. The old appointment numbers help compare histories.
3. **Where the comparison breaks:** terms are protocol counters, not real-time dates or fixed-length time periods. Two machines can temporarily have different beliefs about the current term.

37.2.3 PLAIN — a worked example

1. A log might contain `(index 1, term 1, set stock to 5)` and `(index 2, term 2, reserve 3)`. A later leader in term 3 can retain both entries without changing their term labels.
2. Compare candidate X ending at `(term 4, index 7)` with voter Y ending at `(term 3, index 100)`. Under Raft's freshness comparison, X's higher last term is more up-to-date despite its smaller index.
3. If both end in term 4, the longer log is more up-to-date. A candidate ending at index 6 is then behind a voter ending at index 7.
4. The lab checks this comparison as a predicate. It does not infer full log validity merely from two numbers supplied by an untrusted caller.

37.2.4 PLAIN — what is really happening inside

1. Raft participants keep persistent current-term, vote and log state. Their messages carry enough context to reject obsolete requests and test whether a proposed log continuation matches the preceding entry.
2. A follower does not accept an arbitrary suffix at an arbitrary position. The leader supplies the previous index and term so the follower can check continuity.
3. If a conflicting uncommitted suffix is replaced, the accepted prefix remains protected by the protocol's election and commitment rules. An operator should not generalize that into permission to discard arbitrary business history.

37.2.5 TECHNICAL — the engineer's version

1. Raft's persistent state includes `currentTerm`, `votedFor` and log entries. Required state must be durably updated before the relevant successful response, according to the algorithm's storage assumptions. [S154]
2. Log matching uses index and term; election freshness compares last term first and last index second. Term counters are not wall-clock timestamps. [S154]
3. These small predicates are necessary pieces, not an implementation. Message retries, durable ordering, crash recovery, membership and application integration remain part of a real protocol.

37.2.6 WORDS — remember these

1. **Term:** a protocol generation for leadership attempts — a monotonically advancing counter used to reject obsolete authority and compare log histories. **Log index:** a position in the command sequence — an ordinal distinct from the term in which an entry was created. **Log matching:** matching positions imply compatible preceding history under the protocol — a property enforced through entry terms and append checks.

37.3 Majorities

37.3.1 PLAIN — in simple words

1. A majority is more than half of the configured participants. Two majorities of the same fixed group must overlap.
2. Raft uses that overlap together with voting and log rules. A participant's durable refusal to make incompatible promises gives the overlap its protective meaning.
3. Counting machines without enforcing those rules is not consensus. The set calculation and the protocol must both be correct.

37.3.2 PLAIN — a picture in your head

1. In a group of three clerks, approval from two always includes at least one clerk from any other pair of two. In a group of five, groups of three also overlap.
2. If the shared clerk obeys the required memory and eligibility rules, incompatible proposals cannot simply pass through unnoticed.
3. **Where the comparison breaks:** the protocol does not ask a human to recognize semantic contradictions. It enforces specific rules about terms, votes and logs.

37.3.3 PLAIN — a worked example

1. A three-member group requires two participants for a majority. It can lose one participant and still potentially make progress; with only one reachable participant it cannot obtain a majority.
2. A five-member group requires three. Two can be unavailable while a suitable connected majority remains, but a split of two, two and one has no majority component.
3. Adding a fourth participant to a three-member group raises the majority to three. The simple maximum unavailable count remains one; adding one server does not necessarily add another tolerated failure.
4. These arithmetic statements assume fixed membership and otherwise suitable communication and behaviour. Reachable head count alone does not guarantee immediate election or progress.

37.3.4 PLAIN — what is really happening inside

1. A candidate seeks votes for its term. A voter grants at most one vote per term and only when the candidate satisfies the log condition.
2. A leader replicates entries and tracks the positions acknowledged by followers. Under the required rules, a current-term entry replicated to a majority can advance commitment.
3. The phrase "current-term" matters. An older entry merely appearing on a majority is not, by that fact alone, sufficient for the ordinary Raft commitment rule.

37.3.5 TECHNICAL — the engineer's version

1. For N fixed members, a majority has `floor(N/2)+1` members. The associated intersection supports, but does not replace, Raft's durable-vote and log-freshness rules. [S154]
2. A leader advances commitment by counting replicas only for an entry from its current term. Once such an entry is committed, preceding entries become committed through the log-prefix rules. [S154]
3. Membership transitions cannot be treated as independent edits to local server lists. They must preserve safety across configurations; the paper's joint-consensus method is one explicit design.

37.3.6 WORDS — remember these

1. **Majority:** more than half of the configured group — `floor(N/2)+1` participants for a fixed N-member configuration. **Voting restriction:** a rule limiting acceptable leadership promises — eligibility and one-vote-per-term conditions that help preserve committed history. **Current-term commitment:** commit advancement based on a new-term entry — the Raft rule preventing unsafe conclusions from replica counts of older entries alone.

37.4 Leader changes

37.4.1 PLAIN — in simple words

1. A leader can fail after some followers received an entry and before others did. The next leader must preserve already committed work while resolving incompatible uncommitted endings.
2. Being elected is not permission to replace the past with whatever the new machine happens to prefer. Election eligibility and log repair work together to preserve the accepted history.
3. Clients can still be uncertain about a request whose reply was lost. The new leader needs the application's retained request outcome to answer that question safely.

37.4.2 PLAIN — a picture in your head

1. One clerk disappears while distributing an instruction. Two clerks have the accepted instruction; the third has an older notebook.
2. The replacement procedure must not let the older notebook erase an instruction already made official. Nor should an unapproved private note automatically become official merely because someone finds it later.
3. **Where the comparison breaks:** terms and voting rules—not a majority's informal recollection—determine which history a candidate may lead.

37.4.3 PLAIN — a worked example

1. A, B and C share committed entry 1 from term 1, setting stock to 5. A leads term 2 and appends entry 2, reserving three units under request `R-CONS-1`.
2. A and B durably store entry 2. Under the stated current-term rule and otherwise valid protocol, A commits it and applies the reservation, leaving 2. Its success reply is lost.
3. A becomes unreachable. C still ends at entry 1 and cannot obtain B's vote for a candidate log that is behind B's. B can seek a later term with the more up-to-date log.
4. After B becomes the legitimate term-3 leader, it replicates its prefix and a term-3 no-op entry to a majority. C can then learn and apply the committed reservation in order. The retry of `R-CONS-1` returns the retained result instead of reserving three again.

37.4.4 PLAIN — what is really happening inside

1. The new leader may contain an entry whose commitment status is not yet known locally. Committing a current-term entry helps establish the preceding prefix under the protocol.
2. Followers compare previous index and term when accepting appends. The leader can step back to a matching prefix and send the correct continuation.
3. State-machine application follows commitment in order. A follower must not apply a speculative command merely because it appeared in its local log.

37.4.5 TECHNICAL — the engineer's version

1. Raft's Leader Completeness property connects election restrictions with preservation of committed entries. Its log-repair mechanism can overwrite conflicting uncommitted follower suffixes, not already applied committed history in a correct execution. [S154]
2. A new leader's read-only path needs safeguards against stale authority and incomplete commitment knowledge. Serving a local read simply because a process still believes it is leader is insufficient. [S154]
3. The trace is an educational schedule under stated assumptions. It omits message serialization, durable storage implementation and complete membership management and must not be used as deployable protocol code.

37.4.6 WORDS — remember these

1. **Leader completeness:** later valid leaders retain committed history — a safety property established by the full consensus protocol. **No-op entry:** an agreed command with no domain change — an entry useful for establishing protocol progress without altering the business state. **Speculative suffix:** entries not yet accepted as committed — a local log continuation that may be replaced under valid recovery rules.

37.5 Safety and progress

37.5.1 PLAIN — in simple words

1. Safety means the forbidden thing does not happen, such as two replicas applying different commands at the same committed position. Progress means useful work eventually gets decided under the required conditions.
2. A system can remain safe while temporarily unable to make progress. Stopping without a majority can preserve the accepted-history rule.
3. A system that always returns something may be responsive but unsafe if it invents incompatible decisions. Neither property should be inferred from the other.

37.5.2 PLAIN — a picture in your head

1. A shop can pause reservations when it cannot establish who owns the remaining allocation. Customers wait, but the shop avoids promising the same item twice.
2. Letting every disconnected clerk approve reservations may keep counters busy while breaking the shared-stock rule.
3. **Where the comparison breaks:** the acceptable availability cost is an application decision. Consensus provides mechanisms and limits; it does not decide which service policy users should accept.

37.5.3 PLAIN — a worked example

1. A five-node group splits into `{A,B}` and `{C,D,E}`. The three-node side may form a valid majority if the protocol's other conditions hold. The two-node side cannot independently commit new entries by the same fixed-membership majority rule.
2. If all communication becomes arbitrarily delayed, even a live group may repeatedly fail to settle on a leader in a timely way. Safety need not disappear merely because progress is poor.
3. If an operator changes both sides' membership independently to make each side a local majority, the original intersection argument no longer applies.

4. The lesson is not to force a green availability light by removing the rule that protected the data.

37.5.4 PLAIN — what is really happening inside

1. Progress depends on enough suitable participants communicating for long enough and on operational resources such as storage and processing being available.
2. Timer choices affect election stability and latency. They do not repair an implementation that forgets votes or acknowledges entries before meeting its persistence assumptions.
3. A production consensus library also needs log compaction, snapshot installation, membership changes, authentication, resource bounds and careful crash testing. The short core algorithm is not the whole service.

37.5.5 TECHNICAL — the engineer's version

1. State safety and liveness assumptions separately. Raft's intended safety does not require accurate wall-clock timing, while practical progress depends on suitable timing and communication conditions. [S154]
2. The general impossibility discussion around asynchronous agreement explains why unrestricted failure and timing assumptions cannot be hidden behind a promise of guaranteed immediate progress. [S153]
3. Crash-fault consensus does not tolerate participants arbitrarily forging state or violating the protocol. Secure transport and membership authentication are necessary operational protections but do not transform the algorithm into Byzantine consensus.

37.5.6 WORDS — remember these

1. **Safety:** prohibited outcomes never occur under the model — a property such as preserving a single committed command at each log position. **Liveness:** required progress eventually occurs under the assumptions — a property distinct from avoiding incorrect outcomes. **Byzantine behaviour:** a participant acts arbitrarily or maliciously — a failure model stronger than ordinary crash and omission failures.

37.6 Reading a consensus trace

37.6.1 PLAIN — in simple words

1. A useful trace records enough state to explain why a decision was allowed: configuration, term, log positions, acknowledgements and commitment progress.
2. A line saying "majority reached" is weak evidence without naming which participants counted and what they had actually confirmed.
3. Read the trace as a sequence of justified transitions. Ask what each observer knew at that point rather than using facts discovered only later.

37.6.2 PLAIN — a picture in your head

1. A meeting record should show who was eligible, what proposal was considered and which approvals were valid. A final sentence saying "everyone agreed" hides the evidence.
2. A later replacement must be able to follow the same decision trail without relying on the original coordinator's memory.
3. **Where the comparison breaks:** consensus traces can be incomplete or sampled. Logs alone may not expose the precise persistence point or every message that influenced execution.

37.6.3 PLAIN — a worked example

1. For the Chapter 37 fixture, write a table with columns: step, current configuration, term, leader, each node's last entry, committed index and applied state.
2. At the term-2 majority acknowledgement, record A and B's durable entry-2 confirmations—not merely that their processes are alive. At C's failed candidacy, record the last-term/index comparison that causes B to refuse.
3. At the term-3 no-op commitment, show that the required majority contains the new entry and that application proceeds through the earlier reservation first.
4. The companion checks cover majority arithmetic, freshness comparison and current-term commit eligibility. They do not replay an actual network cluster or certify a full consensus implementation.

37.6.4 PLAIN — what is really happening inside

1. A trace should distinguish observed, inferred and assumed state. "Disk flushed" requires a relevant acknowledgement or instrumentation; it cannot be inferred from a request being sent.
2. Counterexamples are valuable. Try a stale candidate, an obsolete term, too few acknowledgements, an older-term entry and a repeated client request.
3. Any claimed result should identify the software version and failure injection used. A pure predicate test has a different evidentiary scope from a crash-restart experiment on a real cluster.

37.6.5 TECHNICAL — the engineer's version

1. Review invariant checks alongside traces: monotonic terms, one valid vote per term, matching committed prefixes, ordered application and correct membership-aware commit rules. [S154]
2. Independent formal specifications, model checking and implementation fault tests address different risks. None can be replaced by a diagram that simply assumes every desired property.
3. Use established, maintained consensus implementations for real systems and verify their integration boundaries. The code accompanying this book is explicitly bounded educational material, not a recommendation to deploy a hand-built consensus service.

37.6.6 WORDS — remember these

1. **Consensus trace:** a record of protocol-relevant transitions — evidence connecting terms, votes, log positions and application state. **Invariant check:** testing a condition that should always hold — a verification step distinct from testing only one successful output. **Commit eligibility:** evidence satisfying a commitment rule — the conditions permitting a particular log position to become committed.

37.97 Practice and worked answers

1. **Why can two valid reservation orders produce different winners?** The first accepted reservation changes the state seen by the second. Replicas need one agreed command order.
2. **Which Raft log is more up-to-date: last term 4/index 7 or term 3/index 100?** The term-4 log, because last term is compared before index.
3. **How large are majorities for three, four and five members?** Two, three and three respectively.
4. **Can an older-term entry be declared committed merely because a new leader sees it on a majority?** Not under Raft's ordinary replica-counting commitment rule. A current-term entry is needed to advance commitment by that rule.

5. **Why may C fail to win B's vote in the worked trace?** C's log is behind the committed-history-bearing log B already has.
6. **Does an unavailable minority necessarily make the system unsafe?** No. Correctly refusing to commit without the required quorum can preserve safety while reducing progress.
7. **Does consensus alone stop a retried reservation from executing twice?** No. The state machine needs request identity and retained result handling.
8. **What do the companion predicates establish?** Their documented arithmetic and eligibility behaviours for supplied cases. They do not establish a complete networked Raft implementation.

37.98 Common wrong ideas

1. **Wrong:** consensus proves the business data is true. **Right:** it agrees on commands; domain validation remains necessary.
2. **Wrong:** stored, committed and applied are the same state. **Right:** they are different protocol milestones.
3. **Wrong:** a term is a wall-clock interval. **Right:** it is a protocol generation counter.
4. **Wrong:** the longest log always wins. **Right:** Raft compares last term before last index.
5. **Wrong:** any majority copy count proves commitment. **Right:** current-term, membership and protocol rules matter.
6. **Wrong:** a leader can safely answer all local reads indefinitely. **Right:** stale-authority and commitment-knowledge safeguards are required.
7. **Wrong:** crash-fault consensus covers malicious arbitrary participants. **Right:** that is a different failure model.
8. **Wrong:** a short toy algorithm is production consensus. **Right:** persistence, recovery, membership and integration add substantial obligations.

37.99 Chapter summary in 20 lines

1. Consensus helps participants agree under a specified failure model.
2. Replicated state machines apply an agreed command sequence.
3. Agreement does not replace business validation or authorization.
4. Log storage, commitment and application are separate stages.
5. Terms distinguish successive leadership attempts.
6. Log indexes identify positions rather than authority generations.
7. Votes and required log state must survive relevant crashes.
8. Raft compares last term before last index for election freshness.
9. Fixed-group majorities overlap.
10. Durable voting and log rules give that overlap its safety meaning.
11. Current-term entries govern replica-counting commitment advancement.
12. Later valid leaders preserve committed history under the full protocol.
13. Uncommitted follower suffixes may require repair.
14. Client retries still require idempotency state.
15. Read-only paths need protection against stale leadership.
16. Safety and progress are different properties.
17. Progress requires suitable communication, participants and resources.

18. Membership changes need their own safety-preserving procedure.
19. Trace evidence must distinguish observations from assumptions.
20. Educational predicates are not a production consensus implementation.

38

CHAPTER

Distributed Transactions and Sagas

38.0 What this chapter gives you

1. A local transaction can protect changes inside one database boundary. A workflow touching several independent resources needs an additional agreement or recovery design.
2. You will distinguish two-phase commit from a sequence of local transactions with compensation, understand prepared and in-doubt work, and design explicit policies for retries and irreversible effects.
3. The examples use fictional reservations and delivery arrangements. The companion models are state-transition exercises, not an implementation of a distributed transaction manager or permission to run prepared transactions on a live server.

38.1 One transaction across participants

38.1.1 PLAIN — in simple words

1. Suppose an operation must change two databases as one logical decision. A successful commit in the first database does not force the second database to commit.
2. Network failures create intermediate situations: one participant may be ready, another may refuse, and the coordinator may disappear before everyone learns the decision.
3. The design must say whether it needs one coordinated commit decision or whether intermediate completed steps are allowed and later corrected through business actions.

38.1.2 PLAIN — a picture in your head

1. Mira needs both a stock reservation and a delivery slot. Two separate clerks control those resources.
2. Asking one clerk and then the other can leave a stock reservation without a delivery slot. The workflow needs a rule for that partial outcome.
3. **Where the comparison breaks:** a database can sometimes hold prepared transactional state under a formal protocol. A delivery company may offer only ordinary requests and cancellations, not participation in the same transaction mechanism.

38.1.3 PLAIN — a worked example

1. A fictional order needs three units from database A and one delivery slot from database B. Initially both are available.
2. A local transaction in A commits the stock reservation. Before B commits, the connection fails. Rolling back the caller's remaining code does not reverse A's completed transaction.

3. A single Python `try` block around both calls therefore does not provide distributed atomicity. The program can catch an exception while the first resource has already changed.
4. Before choosing a mechanism, ask whether both resources support coordinated prepare/commit, whether temporary partial states are acceptable and what a valid compensation would mean.

■ 38.1.4 PLAIN — what is really happening inside

1. Independent resource managers own their local locks, logs and recovery. A coordinator must connect their decisions through a protocol if one global atomic decision is required.
2. Atomic commitment and isolation are distinct. Agreement that all participants commit does not automatically give arbitrary external readers a single synchronized multi-database snapshot.
3. A simpler architecture may keep tightly coupled data within one transactional owner. Distribution should follow requirements, not create a cross-resource invariant without a plan to enforce it.

■ 38.1.5 TECHNICAL — the engineer's version

1. A distributed transaction coordinates multiple transactional resources. Two-phase commit is an atomic-commit protocol; it is not identical to replicated-log consensus or to a complete distributed isolation model. [S155] [S156]
2. PostgreSQL prepared transactions are intended for external transaction managers. Ordinary application sessions should not create long-lived prepared work without a reliable mechanism to resolve it. [S156]
3. Review resource capability, coordinator durability, identity, locks, failure detection and recovery ownership. A local connection library cannot promise guarantees unsupported by the participants.

■ 38.1.6 WORDS — remember these

1. **Participant:** a resource involved in a coordinated transaction — a resource manager responsible for its local prepared and final state. **Coordinator:** the component managing a shared decision — a protocol participant that gathers votes and records or distributes the transaction outcome. **Atomic commitment:** one compatible commit-or-abort decision — a property distinct from simultaneous visibility or complete transaction isolation.

38.2 Prepare and commit

■ 38.2.1 PLAIN — in simple words

1. Two-phase commit first asks whether each participant can promise to complete its part. A participant that says yes prepares enough state to honor the eventual valid decision.
2. If the required participants prepare successfully, the coordinator can record the commit decision and tell them to commit. A refusal can lead to an abort decision instead.
3. Preparing is not the same as finishing. The participant may hold resources while waiting for the final decision.

■ 38.2.2 PLAIN — a picture in your head

1. The stock clerk and delivery clerk each set aside the necessary resources and say, "I am ready to follow the final decision." The coordinator then announces the agreed outcome.
2. Once a clerk has made that promise, independently releasing the resource while others commit can break the all-or-nothing agreement.

3. **Where the comparison breaks:** a real prepared promise must survive the relevant crashes and be tied to a durable transaction identity. Verbal readiness alone is not enough.

38.2.3 PLAIN — a worked example

1. Give the synthetic transaction identity `TX-AB-1`. A and B begin in an initial state. Each successfully records a prepared state for that same transaction.
2. The coordinator durably records `COMMIT` under the protocol. A receives the decision and commits. B's decision message is delayed.
3. B remains prepared until it obtains the valid decision; it must not invent `ABORT` merely because the reply took longer than expected.
4. When B later receives the commit decision, it completes. Repeated delivery of that same decision should be recognized as the same transaction outcome, not applied as a new business operation.

38.2.4 PLAIN — what is really happening inside

1. The prepare phase establishes the participants' ability to follow the eventual decision. The decision and recovery records must remain available after failures under the implementation's assumptions.
2. Participants can learn the same final outcome at different times. Protocol atomicity is about compatible decisions, not all disks changing at the same physical instant.
3. The coordinator cannot safely forget a transaction merely because it sent the decision once. It needs a completion and recovery policy for participants that were disconnected.

38.2.5 TECHNICAL — the engineer's version

1. PostgreSQL `PREPARE TRANSACTION` detaches the prepared transaction from the session and stores state needed for later `COMMIT PREPARED` or `ROLLBACK PREPARED`. The transaction remains a resource-management obligation. [S156]
2. Required coordinator and participant records must be durable before the protocol makes dependent promises. The precise recovery algorithm belongs to the transaction manager, not to an ad hoc timeout handler.
3. The book's state model accepts only valid transitions and repeated matching decisions. It does not persist coordinator logs or implement network recovery, so it is not a two-phase-commit service.

38.2.6 WORDS — remember these

1. **Prepare:** promise readiness for a later decision — a state in which a participant retains the resources and recovery information needed by the commit protocol. **Two-phase commit:** prepare followed by a final decision — an atomic-commit protocol across participating transactional resources. **Decision record:** retained evidence of commit or abort — information needed to resolve participants consistently after interruption.

38.3 In-doubt work

38.3.1 PLAIN — in simple words

1. A participant is in doubt when it has made a prepared promise but does not know the final decision. It may be unable to release resources safely on its own.
2. This is one cost of coordinated atomic commitment. A failed coordinator or missing decision path can leave otherwise healthy resources waiting.

3. An operator needs a documented resolution procedure. Guessing a decision to clear a lock can create a contradiction with participants that already completed the opposite outcome.

38.3.2 PLAIN — a picture in your head

1. The delivery clerk has promised to hold a slot until the coordinator decides. The phone line then fails. The stock clerk may already have been told to commit.
2. Releasing the slot because "surely the coordinator gave up" can leave a confirmed order without the promised delivery resource.
3. **Where the comparison breaks:** real protocols may replicate coordination state or use specialized recovery procedures. The simple story explains the uncertainty, not every implementation's availability properties.

38.3.3 PLAIN — a worked example

1. Continue `TX-AB-1`: A is committed, B is prepared and the coordinator is temporarily unreachable. B's local timeout fires.
2. The timeout establishes that B has not learned the outcome within its waiting policy. It does not prove the global decision was abort.
3. A valid recovery path must obtain authoritative decision evidence. If such evidence is unavailable, the system records the unresolved state and follows its approved incident procedure instead of inventing a successful resolution.
4. A test confirming that B remains prepared in this model demonstrates the waiting limitation. It does not mean the missing recovery service has been implemented.

38.3.4 PLAIN — what is really happening inside

1. Prepared work can hold locks and prevent cleanup. Its age and resource footprint need monitoring, especially if a transaction manager fails repeatedly.
2. The recovery procedure must bind an action to the exact transaction and evidence. A similarly named request or an old log from another environment is not enough.
3. Prevention includes bounded operational handling, reliable coordinator state and ownership for abandoned work. Disabling a warning does not remove the retained locks or the uncertainty.

38.3.5 TECHNICAL — the engineer's version

1. Two-phase commit can block in failure cases when a prepared participant cannot determine the decision safely. Replicating the coordinator may improve availability but does not erase the underlying protocol obligations. [S155] [S156]
2. PostgreSQL warns that long-lived prepared transactions retain locks and interfere with vacuum and transaction-ID management. Without a responsible external transaction manager, the feature should remain disabled. [S156]
3. Operational resolution is not a substitute for protocol evidence. Record transaction identity, participant states, coordinator decision evidence and any explicitly authorized exceptional disposition.

38.3.6 WORDS — remember these

1. **In-doubt transaction:** prepared work lacks a known final decision — a state requiring authoritative recovery information before safe resolution. **Blocking:** progress waits for unavailable coor-

dination — a limitation distinct from a participant process being physically stopped. **Heuristic resolution:** an exceptional locally chosen disposition — a decision outside normal coordinated evidence that can create inconsistency and requires explicit treatment.

38.4 Compensation

38.4.1 PLAIN — in simple words

1. Compensation is a new business action intended to address a previously completed action. It is not time travel and not the same as rolling back an uncommitted local transaction.
2. Cancelling a reservation can release stock. It does not erase the fact that the reservation existed, and it must not undo unrelated work performed meanwhile.
3. Some effects cannot be fully undone. A parcel already delivered or a message already read may require a different response rather than pretending the original event disappeared.

38.4.2 PLAIN — a picture in your head

1. Mira reserves three notebooks for a delivery that later becomes impossible. She records a cancellation and releases those three notebooks back to availability.
2. She does not reset the entire shelf count to what it was yesterday, because other customers may have bought or reserved items since then.
3. **Where the comparison breaks:** the compensation must follow the real business rules and authority. An arithmetic inverse is not automatically a valid or permitted reversal.

38.4.3 PLAIN — a worked example

1. Start a separate stock fixture at 5. Saga reservation S1 takes 3, leaving 2. Another legitimate reservation S2 takes 1, leaving 1.
2. If S1 is cancelled, releasing S1's three units leaves 4. Resetting stock to S1's old value of 5 would incorrectly erase S2's reservation.
3. The cancellation must be tied to S1 and applied once under its own idempotency rules. A repeated cancellation message must not add another three units and raise the quantity to 7.
4. This fixture demonstrates why compensation is a state-aware new operation. It does not model actual warehouse movement or establish that every reservation may legally or commercially be cancelled.

38.4.4 PLAIN — what is really happening inside

1. A compensating operation checks the current state and the prior operation's identity. It records what it reversed or adjusted and the outcome of that attempt.
2. Other transactions may observe intermediate saga states. Reports must distinguish pending, completed, compensating and compensated work instead of forcing everything into a final-success column.
3. Compensation itself can fail or time out. Its retry and escalation policy must be as deliberate as the original action's policy.

38.4.5 TECHNICAL — the engineer's version

1. The original saga formulation decomposes long-lived work into transactions that can interleave with other work and pairs appropriate steps with compensating transactions. [S157]

2. Compensation preserves application meaning rather than necessarily restoring identical prior bytes. Its correctness depends on concurrent work, allowed transitions and business policy.
3. Use stable step identities, conditional transitions and retained outcomes. A reversal amount or released quantity must be derived from the specific original operation, not from untrusted repeated input alone.

38.4.6 WORDS — remember these

1. **Compensation:** a new action addressing an earlier completed action — an application-defined correction distinct from local transaction rollback. **Saga:** a workflow of local transactions with recovery behaviour — a sequence whose partial completion is handled through continuation or compensation. **Semantic reversal:** restoring an intended business condition — an operation whose meaning is not necessarily identical to restoring old storage bytes.

38.5 Saga state and retries

38.5.1 PLAIN — in simple words

1. A saga needs a durable record of its progress. After a process restart, the system should know which steps completed, which are pending and which outcomes remain unknown.
2. Each retry must refer to the same intended step unless a new operation is explicitly being requested. Otherwise the recovery process can create duplicates.
3. A final label such as "cancelled" should be assigned only after the required cancellation conditions are established, not merely because a cancellation message was queued.

38.5.2 PLAIN — a picture in your head

1. Mira uses a checklist for each delivery: reserve stock, arrange delivery, confirm dispatch. If the delivery step fails, the checklist records whether stock still needs to be released.
2. A new worker can resume from the recorded state rather than starting every task again from memory.
3. **Where the comparison breaks:** durable workflow state must be connected atomically to local changes and outgoing intents where required. A handwritten checkmark after an external call has the same lost-reply ambiguity as other distributed actions.

38.5.3 PLAIN — a worked example

1. Define fictional saga states `NEW`, `STOCK_RESERVED`, `DELIVERY_CONFIRMED`, `COMPENSATING`, `COMPENSATED` and `REVIEW_REQUIRED`.
2. After the stock step commits, the delivery request times out. The correct next state may be `REVIEW_REQUIRED` or a pending status query because the slot could already have been confirmed remotely.
3. Blindly releasing stock and reporting cancellation while the remote delivery remains confirmed can create a new inconsistency. First reconcile the delivery request using its stable identity and supported cancellation/status contract.
4. If delivery is definitively refused, transition to compensation, release the specific stock reservation once, then record the compensated outcome. The state names are example policy, not a universal workflow standard.

38.5.4 PLAIN — what is really happening inside

1. Local state transitions can use version checks or row locks. Outbox records can connect a committed local transition with a later message without pretending that message delivery is part of the same local transaction.
2. Incoming messages need duplicate and conflict handling. A late success may arrive after a timeout initiated review; the workflow must reconcile it with the current state rather than ignoring it blindly.
3. Keep retries bounded operationally and retain enough evidence for manual review. A permanently failing compensation must not remain an invisible infinite loop.

38.5.5 TECHNICAL — the engineer's version

1. A saga is a state machine with durable step identities, transition guards and explicit recovery policy. Orchestration centralizes coordination; choreography distributes reactions, but neither removes the need for observable workflow state.
2. The transactional outbox and idempotent consumer patterns address local intent and replay boundaries. They do not establish a globally atomic saga or guaranteed exactly-once external effects. [S125] [S126]
3. Persist unknown outcomes distinctly from confirmed failures. Recovery must use authoritative status or supported idempotency mechanisms before choosing a potentially conflicting compensation.

38.5.6 WORDS — remember these

1. **Workflow state:** the recorded stage and evidence of a process — durable information used to resume or reconcile a multi-step operation. **Orchestration:** a coordinator directs workflow steps — an arrangement that centralizes sequencing and recovery decisions. **Choreography:** participants react to events — an arrangement distributing workflow reactions while still requiring explicit correctness and observability.

38.6 Choosing an explicit failure policy

38.6.1 PLAIN — in simple words

1. The mechanism should follow the requirement. Some work belongs in one local transaction; some needs coordinated atomic commitment; some can tolerate visible intermediate states with carefully designed compensation.
2. "Use microservices" does not answer whether a half-completed order is acceptable. "Use transactions" does not make an external partner support a prepare protocol.
3. The chosen policy must say what happens when a step fails, a reply is lost, a participant is unavailable or compensation cannot finish.

38.6.2 PLAIN — a picture in your head

1. Before opening a second counter, Mira writes down what a customer can be promised at each stage. A tentative reservation is not called a dispatched order.
2. Staff know which interruptions can be retried, which need review and which require cancelling a specific prior commitment.

3. **Where the comparison breaks:** written policy is necessary but not sufficient. The actual implementation must enforce it under concurrent requests and failures.

38.6.3 PLAIN — a worked example

1. Compare three designs for the fictional stock-and-delivery workflow. Design A keeps both resources in one local database transaction. Design B uses participants capable of managed two-phase commit. Design C records local steps and defined compensations.
2. A may be simplest when the resources truly share one owner and database. B preserves a coordinated decision but introduces prepared-state and recovery obligations. C permits intermediate committed states and needs an explicit semantic recovery policy.
3. None is universally best. A delivery partner exposing only a request API may rule out B; a rule forbidding visible partial completion may rule out a naive C.
4. Evaluate the allowed states and failure traces before selecting the architecture, then test the implemented design rather than the diagram alone.

38.6.4 PLAIN — what is really happening inside

1. Write down the invariant, the transaction boundary and every external effect. Mark which resources can atomically participate and which can only acknowledge separate actions.
2. For each interruption point, identify known committed state, unknown outcomes, recovery evidence and permitted next actions. Include compensation failures and stale responses.
3. Review the user-visible labels against those states. A screen that says "complete" while compensation remains pending conceals the very uncertainty the protocol needs to manage.

38.6.5 TECHNICAL — the engineer's version

1. Compare designs by their supported invariants and failure semantics, not by treating two-phase commit and sagas as interchangeable syntax. Prepared atomic commitment and compensating workflows solve different versions of the problem. [S156] [S157]
2. The companion state models test valid prepare/decision transitions and idempotent compensation arithmetic. No external transaction manager, delivery provider or production distributed workflow is executed.
3. A release review requires implementation-level evidence for concurrency, durable recovery, authorization and external-effect handling. Local educational tests are useful but narrower evidence.

38.6.6 WORDS — remember these

1. **Failure policy:** the permitted response to an interruption — explicit rules for retry, waiting, reconciliation, compensation or escalation. **Intermediate state:** a visible stage before final completion — a condition that a saga-based application must interpret correctly. **Recovery ownership:** who is responsible for unresolved work — an operational assignment ensuring pending and in-doubt operations are not abandoned.

38.97 Practice and worked answers

1. **Do two successful local transaction APIs automatically form one distributed transaction?** No. Their decisions remain independent without a coordinating protocol.
2. **A is committed and B is prepared. Can B abort solely because its timeout fired?** Not safely under the illustrated coordinated decision. It needs authoritative outcome evidence.

3. **Why monitor old prepared transactions?** They retain resources and can interfere with cleanup; they require a responsible resolution mechanism.
4. **Stock starts at 5; S1 reserves 3; S2 reserves 1; S1 is compensated. What remains?** Four units. Restoring the old value of 5 would erase S2's effect.
5. **What should a repeated S1 compensation do?** Return or recognize the already recorded compensation, not release the quantity again.
6. **A delivery request times out. Is cancellation automatically the next valid step?** No. The remote request may have succeeded; reconcile its outcome and supported cancellation contract first.
7. **Does a saga restore exactly the same historical bytes?** Not necessarily. Compensation is a new semantic action and preserves the fact of prior work.
8. **What does the educational model leave unimplemented?** Durable distributed coordination, real network recovery, provider integration, authentication and production concurrency handling.

38.98 Common wrong ideas

1. **Wrong:** one exception handler makes remote calls atomic. **Right:** an earlier resource may already have committed.
2. **Wrong:** prepared means finished. **Right:** prepared work awaits a valid final decision and can retain locks.
3. **Wrong:** a timeout proves global abort. **Right:** it may leave a participant or caller in doubt.
4. **Wrong:** two-phase commit is the same as consensus. **Right:** atomic commitment and replicated agreement have different roles and assumptions.
5. **Wrong:** compensation is ordinary rollback. **Right:** it is a new action after earlier work has committed.
6. **Wrong:** restoring an old total is always a valid compensation. **Right:** it can erase legitimate intervening work.
7. **Wrong:** sagas remove the need for idempotency. **Right:** original and compensating steps both need replay handling.
8. **Wrong:** every failure should trigger automatic cancellation. **Right:** unknown outcomes and irreversible effects require explicit policy.

38.99 Chapter summary in 20 lines

1. Local transaction guarantees stop at their resource boundary.
2. Independent resources need an explicit coordination or recovery design.
3. Atomic commitment is distinct from global isolation and simultaneous visibility.
4. Two-phase commit separates preparation from the final decision.
5. Prepared participants retain a promise and relevant recovery state.
6. A durable decision must be recoverable after interruption.
7. Participants can learn the same outcome at different times.
8. In-doubt work cannot be resolved safely by guessing.
9. Prepared transactions require monitoring and recovery ownership.
10. Compensation is a new business action, not time travel.
11. Correct compensation preserves unrelated intervening work.
12. Repeated compensation needs stable identity and duplicate handling.

13. Sagas allow intermediate committed states under defined policy.
14. Workflow state must distinguish pending, failed and unknown outcomes.
15. Lost external replies require reconciliation rather than blind repetition.
16. Outboxes connect local intent to later delivery without global atomicity.
17. Compensation can fail and needs bounded operational handling.
18. Architecture should follow invariants and participant capabilities.
19. User-visible labels must match the actual workflow state.
20. Educational transition tests do not certify a distributed transaction service.

39

CHAPTER

Caches, Queues and What They Do Not Guarantee

39.0 What this chapter gives you

1. Caches reduce repeated work by keeping derived copies. Queues separate the arrival of work from its processing. Both can improve a system while also introducing new states that must be interpreted correctly.
2. You will trace stale cache fills, expiry, message acknowledgement, redelivery, ordering and back-pressure. You will learn why fast retrieval or reliable transport is not the same as end-to-end business correctness.
3. The companion exercises use deterministic cache and delivery models together with the local outbox/inbox example. They do not operate Redis or RabbitMQ servers or prove production messaging guarantees.

39.1 Derived copies

39.1.1 PLAIN — in simple words

1. A cache stores an answer or copy so the system can avoid repeating expensive work. It is useful only when the cached result is still suitable for the next request.
2. Suitability includes more than age. The result must belong to the right tenant, user permissions, query parameters and interpretation of the data.
3. A cache hit means a matching cache entry was found. It does not prove that the underlying information is current, authorized or correct.

39.1.2 PLAIN — a picture in your head

1. Mira keeps a short list of frequently requested product descriptions beside the till. Reading the list is quicker than opening the full catalogue each time.
2. The list must identify which catalogue edition it came from. A customer's private discount or another branch's restricted information must not be served simply because the product name matches.
3. **Where the comparison breaks:** digital cache keys can include structured identity and version information. Their correctness depends on exactly which inputs the implementation includes, not on human recognition of the right paper.

39.1.3 PLAIN — a worked example

1. Suppose a cached report key contains only `daily-sales`. Tenant A requests a report, and the cache stores A's result under that key.
2. Tenant B then requests its own report and receives the same entry. The cache can be working exactly as implemented while the application violates the tenant boundary.
3. A safer key specification includes the authorized tenant scope, report definition, date range, currency and relevant permission or data-version conditions. The exact fields follow the report contract.
4. Adding a tenant identifier supplied freely by an untrusted caller does not solve authorization. The scope must be derived from or checked against trusted access context.

39.1.4 PLAIN — what is really happening inside

1. Cache-aside logic checks the cache, loads from the source on a miss and stores the result for later use. Each step can race with updates or invalidation.
2. Cached data remains a derived view. The authoritative source and the policy for unavailable or stale entries must remain explicit.
3. Cache failures can increase load on the source suddenly. A system sized only for a perfect hit rate may overload when the cache is cold or unavailable.

39.1.5 TECHNICAL — the engineer's version

1. A cache key is part of the semantic and security contract. Include every input that changes the eligible result, or use a design that validates those differences before reuse.
2. Cache-aside is a loading and invalidation pattern, not an automatic consistency guarantee. Its benefit depends on hit rate, access cost and tolerated staleness. [S161]
3. The authoritative path must retain validation and authorization. A cache should not become an accidental alternate API that bypasses those checks.

39.1.6 WORDS — remember these

1. **Cache:** a retained result or copy used to avoid repeated work — a derived store with explicit validity, scope and eviction rules. **Cache hit:** a lookup finds an eligible entry — an event that establishes presence, not automatically freshness or correctness. **Cache-aside:** the application loads missing data — a pattern in which code checks a cache and populates it from another source.

39.2 Invalidation and expiry

39.2.1 PLAIN — in simple words

1. Invalidation tells the system not to reuse an old entry. Expiry removes eligibility after a specified interval. Neither should be confused with proof that every returned value is current.
2. A delayed source read can arrive after an invalidation and refill the cache with an old value. The order of asynchronous events matters.
3. A lifetime measured from cache insertion also does not establish the age of the source information. An already stale value can be freshly inserted.

39.2.2 PLAIN — a picture in your head

1. Dev copies a price from an old catalogue. While he is walking back to the till, Mira updates the catalogue and removes the old quick-reference card.
2. Dev arrives afterwards and places his old copy into the now-empty slot. The invalidation happened, but a stale refill undid its intended effect.
3. **Where the comparison breaks:** a correct implementation can use generation checks or ordering protocols. The walking story identifies the race; it does not supply a complete cache-coherence algorithm.

39.2.3 PLAIN — a worked example

1. Begin a separate cache fixture with source value 10 and cache generation 4. Reader R starts loading value 10 under generation 4.
2. The source changes to 11, and invalidation advances the cache generation to 5. R's delayed response then tries to store value 10.
3. A generation-checked fill rejects that insertion because its captured generation 4 no longer matches 5. A naive unconditional fill would store the stale value.
4. This prevents the specific stale-refill race in the model. It does not guarantee that R's own already obtained response is fresh; the caller's read contract may require a retry or authoritative check as well.

39.2.4 PLAIN — what is really happening inside

1. The cache needs a rule linking in-flight loads with invalidation. A placeholder or generation can let a delayed result detect that its right to populate the cache has expired.
2. Lost invalidation connections create another uncertainty. A client that cannot know which entries changed must follow its documented flush, reconnect and freshness policy.
3. Expiry bounds cache residence under its timer assumptions. It is useful but is not equivalent to linearizability, authorization freshness or a guaranteed bound on the age of an already stale source replica.

39.2.5 TECHNICAL — the engineer's version

1. Redis's client-side caching documentation explicitly describes invalidation arriving before a delayed GET response and a placeholder-based way to avoid retaining that obsolete response. It also discusses flushing on loss of the invalidation connection. [S160]
2. The book's generation model is an original simplification: invalidation increments a local generation, and a fill succeeds only for the captured current generation. It models one cache boundary, not Redis's full protocol.
3. A TTL, invalidation signal or generation check must be evaluated against the promised read semantics. Preventing one cache race does not prove a complete distributed freshness guarantee.

39.2.6 WORDS — remember these

1. **Invalidation:** mark a cached result unusable — a signal or state transition preventing further eligible reuse. **Time to live:** an entry's configured remaining lifetime — an expiry policy that does not by itself prove source-data age. **Stale refill:** an old in-flight result repopulates a cache — a race occurring after an update or invalidation unless the fill protocol prevents it.

39.3 Queue acknowledgement

39.3.1 PLAIN — in simple words

1. A queue holds work for later processing. Sending a message, the broker accepting it, a worker receiving it and the business action completing are different events.
2. An acknowledgement covers a particular boundary. A publisher confirmation does not automatically mean that a consumer completed the task.
3. A worker should acknowledge according to the intended processing contract. Acknowledging before a required durable change can lose work if the worker then fails.

39.3.2 PLAIN — a picture in your head

1. Mira hands a delivery request to a dispatch desk. The desk stamps "received." That stamp does not mean the courier has delivered the parcel.
2. The courier's later completion report answers a different question. Confusing the two can turn "accepted for processing" into a false "delivered" message.
3. **Where the comparison breaks:** brokers have precise acknowledgement modes, durability settings and queue types. A stamp is not a substitute for their documented guarantees.

39.3.3 PLAIN — a worked example

1. Message M1 asks a consumer to create a local fulfillment task. In unsafe schedule A, the consumer acknowledges M1 first and crashes before committing the task. The broker may now consider the delivery handled even though no task exists.
2. In schedule B, the consumer commits its inbox record and local task together, then acknowledges. If it crashes after commit but before acknowledgement, M1 may be delivered again.
3. The unique inbox identity lets the consumer recognize that repeat and avoid creating a second task. The two schedules trade lost work for replay that the application must handle correctly.
4. The local task is not a physical delivery. Its existence does not prove that a courier acted or a customer received anything.

39.3.4 PLAIN — what is really happening inside

1. The publisher and consumer interact with different broker boundaries. A lost publisher confirmation can lead to repeated publication; a lost consumer acknowledgement can lead to repeated delivery.
2. Durable queue configuration and message persistence need to match the required failure model. A successfully written client socket is not a broker durability acknowledgement.
3. The application must decide what "processed" means before acknowledging. It may mean a durable local task, not completion of every later external effect.

39.3.5 TECHNICAL — the engineer's version

1. RabbitMQ documents publisher confirms and consumer acknowledgements as separate, unaware mechanisms. Confirms cover publishing to the broker; consumer acknowledgements cover broker-to-consumer delivery processing. [S158]
2. Consumer delivery tags are scoped to their channel and are not stable business request identities. Use an application message identity for idempotent processing across redelivery and reconnects. [S158]

3. An inbox record and local effect can share one database transaction. The broker acknowledgement remains outside that local transaction, so the consumer must tolerate the corresponding replay window.

39.3.6 WORDS — remember these

1. **Publisher confirm:** the broker acknowledges publication — evidence about the publisher-to-broker boundary rather than consumer completion. **Consumer acknowledgement:** a worker reports a delivery handled — a protocol action whose timing must match the application's processing contract. **Inbox record:** retained evidence of an accepted incoming message — a local identity and outcome record used to recognize duplicates and conflicts.

39.4 Redelivery and order

39.4.1 PLAIN — in simple words

1. Reliable messaging often includes the possibility of another delivery after failure or uncertainty. A repeated message is not necessarily a second business event.
2. Ordering also has a scope. Publishing order, delivery order and completion order can differ, especially with several workers or retries.
3. An application that depends on sequence must enforce or validate that sequence rather than assume that the word "queue" guarantees every kind of ordering.

39.4.2 PLAIN — a picture in your head

1. Two workers take tasks numbered 1 and 2. Worker 2 finishes quickly while worker 1 is delayed. The desk handed them out in order, but completion happened in the opposite order.
2. If task 1 is later returned for retry, it can appear again after other work. The workers need identities and state rules, not only a line to stand in.
3. **Where the comparison breaks:** brokers expose specific ordering and consumer features. The analogy shows why those features must be checked, not that every queue behaves identically.

39.4.3 PLAIN — a worked example

1. A synthetic entity receives event E1, "create reservation," followed by E2, "cancel reservation." Worker A takes E1 and pauses; worker B takes E2 and runs first.
2. A consumer that treats "reservation missing" as "cancellation completed forever" can later create the reservation when E1 resumes. The final state contradicts the intended event order.
3. Possible designs include per-entity ordered processing, a state machine that retains pending cancellation evidence, or version checks that reject obsolete changes. The appropriate choice depends on the contract.
4. A separate duplicate test delivers the same message identity twice. It should produce one local effect, while two distinct valid event identities remain two events even if their payloads look similar.

39.4.4 PLAIN — what is really happening inside

1. Concurrency can preserve enqueue order while changing completion order. Redelivery can reintroduce older work after newer work has already run.

2. Product features such as single-active-consumer modes or ordered streams have documented limits. They do not automatically order all database actions or external effects performed by an application.
3. A message identity reused with a different semantic payload is a conflict to investigate, not a duplicate to ignore silently. The identity contract must protect meaning as well as count.

39.4.5 TECHNICAL — the engineer's version

1. RabbitMQ documents ordering within particular publishing and queue conditions and explains how multiple consumers, priorities and redelivery affect effective order. [S159]
2. Its acknowledgement documentation states that unacknowledged deliveries can be requeued after channel or connection closure. Consumers must be designed for redelivery. [S158]
3. Sequence guarantees should be scoped by entity, partition, channel or stream as appropriate. Application state transitions and external effects must still preserve the required causality.

39.4.6 WORDS — remember these

1. **Redelivery:** a message is delivered again — a transport event that must be distinguished from a new business event. **Completion order:** the sequence in which effects finish — an order that may differ from publishing or delivery order. **Per-entity ordering:** related changes follow one entity's sequence — a scoped requirement rather than a claim of global ordering across all work.

39.5 Backpressure

39.5.1 PLAIN — in simple words

1. A queue can absorb a temporary burst, but it cannot make a permanently overloaded worker fast enough. If work arrives faster than it leaves, the backlog grows.
2. Backpressure makes that pressure visible upstream through limits, slowing, rejection or controlled admission. It prevents an unbounded pile from silently consuming all available memory or disk.
3. Retry traffic also counts as work. A failing dependency can trigger enough retries to worsen the overload that caused the failures.

39.5.2 PLAIN — a picture in your head

1. A dispatch desk receives 1,000 parcels each minute but can process only 600. A larger room delays the moment it fills; it does not close the 400-parcel gap.
2. The desk needs more sustainable capacity, less admitted work or a clear policy for excess demand. Hiding the queue behind a door changes none of those facts.
3. **Where the comparison breaks:** digital work varies in size and cost, and some can be safely combined or discarded under a stated policy. A parcel count alone may not represent resource demand.

39.5.3 PLAIN — a worked example

1. A synthetic queue receives 1,000 messages per second while consumers finish 600. The net growth is 400 messages per second.
2. At 1,000 bytes per message, payload alone grows by 400,000 bytes per second, or 24 MB per minute using decimal units. Broker metadata, replication and indexing add further storage.

3. After ten minutes, the simple backlog is 240,000 messages and 240 MB of payload, assuming fixed rates and no other losses or processing changes.
4. A queue length limit can bound storage, but its overflow policy must be explicit. Silently dropping required reservation work is not a valid capacity solution.

■ 39.5.4 PLAIN — what is really happening inside

1. Prefetch or in-flight limits cap how much unacknowledged work a consumer receives. Too much can create memory pressure and long recovery delays; too little can leave capacity idle.
2. Admission control should consider deadlines and downstream health. Work that cannot possibly finish before its useful deadline may need an explicit rejection rather than a hidden hour-long wait.
3. Monitor queue age as well as length. Ten expensive messages can be more urgent than a thousand tiny ones, and the oldest pending request may reveal starvation hidden by average throughput.

■ 39.5.5 TECHNICAL — the engineer's version

1. In a simple stable-rate model, backlog growth is arrival rate minus service completion rate when that difference is positive. Real workloads require distributions, cost variation and retry accounting.
2. RabbitMQ prefetch limits constrain outstanding unacknowledged deliveries under their documented scope; they are not a complete admission-control or business-deadline policy. [S158]
3. Combine bounded queues, retry budgets, idempotency and observability. Record any overflow, dead-letter or expiration disposition so the application can distinguish delayed work from permanently unprocessed work.

■ 39.5.6 WORDS — remember these

1. **Backpressure:** downstream limits affect upstream admission — a mechanism preventing work from accumulating without bound. **Prefetch:** how much unacknowledged work a consumer may receive — a broker/client flow-control setting with a defined scope. **Queue age:** how long work has waited — a latency and starvation signal distinct from the number of queued messages.

39.6 End-to-end correctness

■ 39.6.1 PLAIN — in simple words

1. A database, cache and queue each provide limited guarantees. The application must connect them into a valid story from request to user-visible outcome.
2. A message can be durably queued while a cache shows an old state. A consumer can finish a local task while the external action remains pending. These states need honest labels.
3. The final question is not whether every component is individually "reliable," but whether the complete operation preserves its intended meaning across every boundary.

■ 39.6.2 PLAIN — a picture in your head

1. A parcel journey includes acceptance at the desk, loading onto a vehicle, arrival at a branch and delivery to the customer. Each stamp is evidence for one stage.
2. The tracking screen should not label the first stamp "delivered." Nor should a missing final stamp automatically trigger shipment of a second identical parcel.

3. **Where the comparison breaks:** the system needs explicit identities, state transitions and reconciliation, not merely a longer list of status words.

39.6.3 PLAIN — a worked example

1. A local reservation and outbox message commit together. The publisher sends the message; the consumer atomically records its inbox identity and creates one local task.
2. The publisher then fails before marking the outbox row delivered. On retry, the same message is published again. The consumer recognizes its identity and returns the prior result without a second local task.
3. A cache invalidation can still be delayed. The status endpoint must use an eligible authoritative or sufficiently fresh path rather than interpreting a stale cache miss as proof that the reservation never existed.
4. The companion test records two delivery attempts and one local task. It does not claim exactly-once email, physical dispatch or execution at an uncontrolled external provider.

39.6.4 PLAIN — what is really happening inside

1. Define each boundary: request acceptance, transaction commit, publication, consumer commit, external acknowledgement and user-visible confirmation.
2. Test failures between boundaries rather than only the uninterrupted happy path. Unknown outcomes, repeated messages and stale reads are normal design cases.
3. Keep a reconciliation path that can connect the request, outbox, inbox and resulting task without exposing unnecessary private data. Evidence should explain progress, not merely generate large logs.

39.6.5 TECHNICAL — the engineer's version

1. The outbox pattern records local intent atomically with business state, while idempotent consumers limit repeated local effects. Their scope must remain explicit. [S125] [S126]
2. Cache coherence, broker acknowledgement and database atomicity are independent mechanisms. None can be promoted into an end-to-end guarantee without checking how the application composes them. [S158] [S160]
3. The next part follows data into analytical and search systems. Those derived systems inherit the same obligations: identity, progress boundaries, replay handling and honest interpretation of incomplete results.

39.6.6 WORDS — remember these

1. **End-to-end correctness:** the complete operation preserves its contract — a property spanning components rather than inferred from any single successful API call. **Boundary evidence:** proof about a particular stage — an acknowledgement or record whose scope must not be exaggerated. **Dead-letter disposition:** work is moved aside after unsuccessful processing — a recorded outcome requiring review rather than automatic proof that the business task finished.

39.97 Practice and worked answers

1. **Why is a cache key containing only a report name dangerous?** Different tenants, parameters or permissions can produce different eligible results. Reusing one entry can leak or misrepresent data.

2. **A delayed value loaded under generation 4 arrives after invalidation to 5. Should it populate the model cache?** No. Its generation is obsolete.
3. **Does rejecting that fill prove the original caller received current data?** No. The caller's read result needs its own freshness policy.
4. **What does a publisher confirm prove about consumer completion?** Nothing by itself. It covers a separate publisher-to-broker boundary.
5. **Why commit an inbox record and local effect before acknowledging?** A later redelivery can then be recognized without losing the local task or duplicating it.
6. **Can FIFO delivery guarantee FIFO completion with two workers?** No. Processing duration and retries can change completion order.
7. **What is the payload backlog after ten minutes at a 400-message/s deficit and 1,000 bytes/message?** 240,000 messages and 240 MB, excluding overhead, under the fixed-rate assumptions.
8. **What does the repeated outbox delivery test establish?** Two attempts produce one local consumer task under the tested inbox contract. It does not establish exactly-once external effects.

39.98 Common wrong ideas

1. **Wrong:** a cache hit proves freshness. **Right:** it proves an entry was found under the implemented eligibility rule.
2. **Wrong:** TTL alone gives linearizable reads. **Right:** expiry is not a complete coherence or source-freshness protocol.
3. **Wrong:** invalidation cannot be undone by a delayed read. **Right:** stale refill is a real ordering hazard.
4. **Wrong:** broker acceptance means the task finished. **Right:** publication, delivery and processing have separate acknowledgements.
5. **Wrong:** a repeated message is always a new event. **Right:** message identity and semantic intent determine replay handling.
6. **Wrong:** queue order means every effect completes in order. **Right:** parallel consumers and retries can change completion order.
7. **Wrong:** a large queue solves permanent overload. **Right:** it only postpones resource exhaustion when arrival exceeds service.
8. **Wrong:** reliable components automatically compose into exactly-once business behaviour. **Right:** the complete boundary and recovery design must establish the intended outcome.

39.99 Chapter summary in 20 lines

1. Caches retain derived results to avoid repeated work.
2. Cache validity includes scope, permissions, parameters and freshness.
3. A hit is not proof that the source information is current.
4. Invalidation and expiry are different mechanisms.
5. Delayed reads can create stale refills.
6. Generation or ordering checks can prevent specific refill races.
7. Losing an invalidation channel requires explicit recovery policy.
8. Queue publication, delivery and processing are separate stages.
9. Publisher confirms do not establish consumer completion.

10. Consumer acknowledgements must match the intended durable processing boundary.
11. Inbox identity and local effects can share one transaction.
12. Redelivery must be distinguished from a new business event.
13. Publishing, delivery and completion order can differ.
14. Per-entity sequence rules require explicit enforcement.
15. Backlogs grow when arrival exceeds sustainable completion.
16. Prefetch and admission limits bound different kinds of outstanding work.
17. Queue age and overflow dispositions must remain observable.
18. Outbox retries can coexist with one idempotent local consumer effect.
19. External actions and cache freshness remain separate obligations.
20. End-to-end correctness requires evidence across the whole request path.

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

Allowed history

a sequence the contract permits — an operation trace used to evaluate a consistency or correctness claim.

Read in context: 36.6.6

Apply position

how far changes affect the readable replica — the prefix already replayed into the destination's visible state.

Read in context: 33.3.6

Asymmetric failure

reachability differs by direction or path — a condition in which one observer's connectivity does not describe the whole system.

Read in context: 36.3.6

Asynchronous replication

remote progress may follow acknowledgement — a configuration that does not wait for a selected remote replication milestone before reporting success.

Read in context: 33.3.6

Atomic commitment

one compatible commit-or-abort decision — a property distinct from simultaneous visibility or complete transaction isolation.

Read in context: 38.1.6

Authoritative outcome

a result accepted under the valid authority path — a state distinct from a cached guess or tentative local proposal.

Read in context: 36.6.6

B

Backpressure

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

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

Read in context: 39.5.6

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: 38.3.6

Boundary evidence

proof about a particular stage — an acknowledgement or record whose scope must not be exaggerated.

Read in context: 39.6.6

Byzantine behaviour

a participant acts arbitrarily or maliciously — a failure model stronger than ordinary crash and omission failures.

Read in context: 37.5.6

C

Cache

a retained result or copy used to avoid repeated work — a derived store with explicit validity, scope and eviction rules.

Read in context: 39.1.6

Cache hit

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

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

Read in context: 39.1.6

Cache-aside

the application loads missing data — a pattern in which code checks a cache and populates it from another source.

Read in context: 39.1.6

Candidate eligibility

the conditions a replacement must satisfy — requirements covering authority, history and readiness rather than mere reachability.

Read in context: 34.3.6

CAP theorem

a limit on joint guarantees under unreliable communication — the incompatibility of the specified atomic service and availability requirements in the partition-prone model.

Read in context: 36.4.6

Catch-up rate

how quickly a backlog shrinks — destination progress minus continuing source generation over the measured interval.

Read in context: 33.4.6

Catch-up suffix

changes after an initial copy boundary — the remaining ordered work needed before a destination matches the selected source state.

Read in context: 35.4.6

Causal consistency

effects follow their relevant causes — a model preserving causal dependencies while allowing flexibility for concurrent unrelated operations.

Read in context: 36.1.6

Causal token

evidence of a required prior position — a scoped marker used to ensure that a read includes specified earlier work.

Read in context: 33.5.6

Choreography

participants react to events — an arrangement distributing workflow reactions while still requiring explicit correctness and observability.

Read in context: 38.5.6

Commit eligibility

evidence satisfying a commitment rule — the conditions permitting a particular log position to become committed.

Read in context: 37.6.6

Committed prefix

the log portion accepted as final under the protocol — entries eligible for ordered application rather than merely present on one disk.

Read in context: 37.1.6

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: 38.4.6

Completion order

the sequence in which effects finish — an order that may differ from publishing or delivery order.

Read in context: 39.4.6

Consensus

agreeing under a specified failure model — a protocol problem requiring compatible decisions and defined progress conditions.

Read in context: 37.1.6

Consensus trace

a record of protocol-relevant transitions — evidence connecting terms, votes, log positions and application state.

Read in context: 37.6.6

Consistency downgrade

a weaker observation contract is used — a change that must be explicit and acceptable for the operation.

Read in context: 36.4.6

Consistent hashing

a scheme limiting remapping when owners change — a family of placement techniques whose exact balance and movement properties depend on the design.

Read in context: 35.6.6

Consumer acknowledgement

a worker reports a delivery handled — a protocol action whose timing must match the application's processing contract.

Read in context: 39.3.6

Control plane

the machinery choosing and configuring behaviour — coordination mechanisms such as role assignment and membership management.

Read in context: 34.1.6

Coordinator

the component managing a shared decision — a protocol participant that gathers votes and records or distributes the transaction outcome.

Read in context: 38.1.6

Current-term commitment

commit advancement based on a new-term entry — the Raft rule preventing unsafe conclusions from replica counts of older entries alone.

Read in context: 37.3.6

Cutover

the point a new path becomes authoritative — a transition with explicit prerequisites and postconditions.

switching the authoritative path — a controlled transition from old to new representation or service behaviour.

Read in context: 35.4.6

D

Data plane

where requests produce actual effects — the operational path that must enforce the control plane's decisions.

Read in context: 34.1.6

Dead-letter disposition

work is moved aside after unsuccessful processing — a recorded outcome requiring review rather than automatic proof that the business task finished.

Read in context: 39.6.6

Decision record

retained evidence of commit or abort — information needed to resolve participants consistently after interruption.

Read in context: 38.2.6

Degraded mode

a documented reduced service — an operational state with explicit limits rather than an undisclosed weakening of guarantees.

Read in context: 36.2.6

Degraded redundancy

fewer healthy copies than intended — an operational state that may persist after the service resumes.

Read in context: 34.3.6

Divergent history

copies contain incompatible continuations — a situation requiring authority and recovery analysis rather than simple row copying.

Read in context: 34.6.6

E

Election

selecting a coordinator under protocol rules — an authority decision with explicit membership and eligibility conditions.

Read in context: 34.3.6

End-to-end correctness

the complete operation preserves its contract — a property spanning components rather than inferred from any single successful API call.

Read in context: 39.6.6

Epoch

a generation of authority or configuration — a value distinguishing successive valid ownership periods under a defined protocol.

Read in context: 34.4.6

F

Failover runbook

the documented role-transition procedure — steps, prerequisites, checks and failure dispositions for transferring service authority.

Read in context: 34.6.6

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: 33.1.6

Failure policy

the permitted response to an interruption — explicit rules for retry, waiting, reconciliation, compensation or escalation.

Read in context: 38.6.6

Failure suspicion

evidence that a participant may be unavailable — an observation requiring protocol handling rather than certain knowledge of a crash.

Read in context: 34.2.6

False suspicion

a healthy participant is treated as failed — a classification possible under delays, partitions or process pauses.

Read in context: 34.2.6

Fencing

preventing obsolete authority from acting — enforcement at a protected resource or infrastructure boundary that rejects stale owners.

Read in context: 34.4.6

G

Global invariant

a rule spanning several owners — a correctness condition not established by independent local checks alone.

Read in context: 35.5.6

H

Hash partition

a piece selected through a repeatable mapping — placement based on a hash-derived value rather than original key order.

Read in context: 35.2.6

Heartbeat

a repeated liveness signal — periodic communication used to detect absence or delay under a configured policy.

Read in context: 34.2.6

Heuristic resolution

an exceptional locally chosen disposition — a decision outside normal coordinated evidence that can create inconsistency and requires explicit treatment.

Read in context: 38.3.6

Historical recovery point

a selected earlier usable state — a state reconstructible from compatible retained recovery material.

Read in context: 33.6.6

Hot partition

one piece receives disproportionate work — a localized throughput, latency or resource bottleneck.

Read in context: 35.3.6

I

In-doubt transaction

prepared work lacks a known final decision — a state requiring authoritative recovery information before safe resolution.

Read in context: 38.3.6

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: 39.3.6

Indirection

use a directory instead of embedding a location — a mapping layer separating stable identity from changing placement.

Read in context: 35.6.6

Indistinguishable histories

different realities look the same to an observer — executions with identical local evidence that require different correct outcomes.

Read in context: 36.4.6

Intermediate state

a visible stage before final completion — a condition that a saga-based application must interpret correctly.

Read in context: 38.6.6

Invalidation

mark a cached result unusable — a signal or state transition preventing further eligible reuse.

Read in context: 39.2.6

Invariant check

testing a condition that should always hold — a verification step distinct from testing only one successful output.

Read in context: 37.6.6

L

Layout version

the identity of a placement map — metadata distinguishing successive assignments of logical partitions to owners.

Read in context: 35.2.6

Leader

the current coordinator for specified work — a role granted under a protocol with a defined authority scope.

Read in context: 34.1.6

Leader completeness

later valid leaders retain committed history — a safety property established by the full consensus protocol.

Read in context: 37.4.6

Linearizability

operations appear to take effect in one real-time-respecting order — a correctness condition placing each operation within its invocation-response interval.

Read in context: 36.1.6

Liveness

required progress eventually occurs under the assumptions — a property distinct from avoiding incorrect outcomes.

Read in context: 37.5.6

Locality

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

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

Read in context: 35.1.6

Log index

a position in the command sequence — an ordinal distinct from the term in which an entry was created.

Read in context: 37.2.6

Log matching

matching positions imply compatible preceding history under the protocol — a property enforced through entry terms and append checks.

Read in context: 37.2.6

Logical replication

following changes to data objects — replication based on record identity and logical operations rather than identical physical placement.

Read in context: 33.2.6

M

Majority

more than half of the configured group — `floor(N/2)+1` participants for a fixed N-member configuration.

Read in context: 37.3.6

Membership configuration

the participants counted by the protocol — a versioned set whose changes require safety-preserving rules.

Read in context: 36.3.6

Monotonic reads

not going backward in observed progress — a session constraint preventing later reads from using an older relevant state.

Read in context: 33.5.6

N

Network partition

some participants cannot communicate — a communication failure that can separate live components.

Read in context: 36.3.6

No-op entry

an agreed command with no domain change — an entry useful for establishing protocol progress without altering the business state.

Read in context: 37.4.6

O

Operation contract

the promise made for one action — a specification of valid results, timing, visibility and failure outcomes.

Read in context: 36.6.6

Operation reconciliation

checking a prior request's authoritative result — resolving uncertainty through stable identity and retained outcome evidence.

Read in context: 34.5.6

Orchestration

a coordinator directs workflow steps — an arrangement that centralizes sequencing and recovery decisions.

Read in context: 38.5.6

P

Partial aggregation

summarize locally before combining — retaining sufficient intermediate state for the intended final calculation.

Read in context: 35.5.6

Participant

a resource involved in a coordinated transaction — a resource manager responsible for its local prepared and final state.

Read in context: 38.1.6

Partition key

the value deciding placement — a field or expression mapping a record into a defined partition.

Read in context: 35.1.6

Per-entity ordering

related changes follow one entity's sequence — a scoped requirement rather than a claim of global ordering across all work.

Read in context: 39.4.6

Physical replication

following the storage engine's changes — replication tied to physical representation and recovery information.

Read in context: 33.2.6

Placement invariant

a rule about where records belong — a condition linking keys, layout versions and valid owners.

Read in context: 35.6.6

Prefetch

how much unacknowledged work a consumer may receive — a broker/client flow-control setting with a defined scope.

Read in context: 39.5.6

Prepare

promise readiness for a later decision — a state in which a participant retains the resources and recovery information needed by the commit protocol.

Read in context: 38.2.6

Promotion

making a replica an active authority — a role transition requiring a supported procedure and protection against competing writers.

Read in context: 33.6.6

Publisher confirm

the broker acknowledges publication — evidence about the publisher-to-broker boundary rather than consumer completion.

Read in context: 39.3.6

Q

Queue age

how long work has waited — a latency and starvation signal distinct from the number of queued messages.

Read in context: 39.5.6

Quorum

a participant set sufficient for a protocol action — a set meeting defined size or intersection requirements.

Read in context: 36.5.6

Quorum intersection

relevant sets share participants — a necessary ingredient in many protocols whose meaning depends on retained state and rules.

Read in context: 36.5.6

R

Range partition

a piece owning an ordered interval — a partition defined by lower and upper key bounds.

Read in context: 35.2.6

Read scaling

sharing question-answering work — distributing eligible read operations across resources without assuming identical freshness.

Read in context: 33.1.6

Read-your-writes

seeing your own completed changes — a session guarantee that later eligible reads include the session's prior acknowledged writes.

Read in context: 33.5.6

Rebalancing

changing where pieces are served — controlled movement of data or ownership to meet capacity and workload needs.

Read in context: 35.4.6

Recovery ownership

who is responsible for unresolved work — an operational assignment ensuring pending and in-doubt operations are not abandoned.

Read in context: 38.6.6

Recovery rehearsal

trying the recovery procedure in isolation — a test of restoration steps, evidence and operational assumptions before an incident.

Read in context: 33.6.6

Redelivery

a message is delivered again — a transport event that must be distinguished from a new business event.

Read in context: 39.4.6

Reinitialization

rebuilding a participant from valid source state — a controlled procedure restoring a compatible replica and its role.

Read in context: 34.6.6

Replica

another maintained copy — a data instance updated through a defined replication mechanism.

Read in context: 33.1.6

Replicated state machine

copies execute one agreed command sequence — an architecture connecting log agreement to matching application state.

Read in context: 37.1.6

Replication identity

the handle for locating a changed row — the key or supported old-row representation used to apply updates and deletions.

Read in context: 33.2.6

Replication lag

one replication stage is behind another — a difference measured using compatible positions, bytes or carefully interpreted timing evidence.

Read in context: 33.4.6

Retry horizon

how long old requests remain recognizable — the interval over which a system retains the evidence needed by its idempotency contract.

Read in context: 34.5.6

S

Safety

prohibited outcomes never occur under the model — a property such as preserving a single committed command at each log position.

Read in context: 37.5.6

Saga

a workflow of local transactions with recovery behaviour — a sequence whose partial completion is handled through continuation or compensation.

Read in context: 38.4.6

Scatter-gather

ask several pieces and combine their replies — a distributed query pattern with completeness and coordination requirements.

Read in context: 35.5.6

Semantic reversal

restoring an intended business condition — an operation whose meaning is not necessarily identical to restoring old storage bytes.

Read in context: 38.4.6

Sequencer

evidence describing a lock generation — a token that a recipient validates before accepting a protected operation.

Read in context: 34.4.6

Sequential consistency

one shared order respects each client's own order — a model that does not additionally require all cross-client real-time precedence.

Read in context: 36.1.6

Service-level indicator

a measured service outcome — a defined quantity such as the fraction of eligible requests meeting a correctness and latency condition.

a measured service property — a defined observation of availability, latency, correctness or another relevant outcome.

Read in context: 36.2.6

Shard

an independently placed portion of data — commonly a partition hosted on a separate storage node or database, with product-specific semantics.

Read in context: 35.1.6

Single-key contention

many operations compete on one identity — a workload limitation that ordinary redistribution of unrelated keys cannot remove.

Read in context: 35.3.6

Skew

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

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

Read in context: 35.3.6

Sloppy quorum

substitute reachable nodes may count — a failure-handling scheme that differs from always using one fixed replica set.

Read in context: 36.5.6

Speculative suffix

entries not yet accepted as committed — a local log continuation that may be replaced under valid recovery rules.

Read in context: 37.4.6

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: 33.4.6

Stale refill

an old in-flight result repopulates a cache — a race occurring after an update or invalidation unless the fill protocol prevents it.

Read in context: 39.2.6

Synchronous acknowledgement

success waits for specified remote progress — a policy whose exact receipt, flush or apply condition must be named.

Read in context: 33.3.6

T

Term

a protocol generation for leadership attempts — a monotonically advancing counter used to reject obsolete authority and compare log histories.

Read in context: 37.2.6

Theoretical availability

required requests eventually complete — a liveness property under a stated distributed-system model.

Read in context: 36.2.6

Time to live

an entry's configured remaining lifetime — an expiry policy that does not by itself prove source-data age.

Read in context: 39.2.6

Two-phase commit

prepare followed by a final decision — an atomic-commit protocol across participating transactional resources.

Read in context: 38.2.6

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: 34.5.6

V

Voting restriction

a rule limiting acceptable leadership promises — eligibility and one-vote-per-term conditions that help preserve committed history.

Read in context: 37.3.6

W

Workflow state

the recorded stage and evidence of a process — durable information used to resume or reconcile a multi-step operation.

Read in context: 38.5.6

Sources and Version Register

These primary sources support adjacent technical claims. The synthetic shop, arithmetic fixtures and teaching models are original examples. Versioned references are not automatically descriptions of a newer release.

[S01] PostgreSQL 17 documentation: Transactions

PostgreSQL Global Development Group

Atomic changes, commit and rollback; product-specific tutorial.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/tutorial-transactions.html

[S02] PostgreSQL 17 documentation: Constraints

PostgreSQL Global Development Group

CHECK, NOT NULL, primary-key and foreign-key behaviour.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/ddl-constraints.html

[S03] PostgreSQL 17 documentation: Sorting Rows (ORDER BY)

PostgreSQL Global Development Group

No guaranteed result order without explicit ordering.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/queries-order.html

[S04] PostgreSQL 17 documentation: Asynchronous Commit

PostgreSQL Global Development Group

Acknowledgement and durability depend on configuration.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/wal-async-commit.html

[S05] PROV-DM: The PROV Data Model (2013)

W3C; editors Luc Moreau and Paolo Missier

Entities, activities, agents and derivation in provenance.

Consulted: 23 September 2026

https://www.w3.org/TR/prov-dm/

[S06] Data on the Web Best Practices (2017)

W3C

Metadata, provenance, quality, versioning and persistent identifiers.

Consulted: 23 September 2026

https://www.w3.org/TR/dwbp/

[S07] RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format (2017)

T. Bray, editor; IETF

JSON value types, interoperable numbers, names and encoding.

Consulted: 23 September 2026

https://www.rfc-editor.org/info/rfc8259/

[S08] RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files (2005)

Y. Shafranovich; IETF

Informational RFC describing common CSV conventions; not a universal spreadsheet schema.

Consulted: 23 September 2026

https://www.rfc-editor.org/info/rfc4180/

[S09] RFC 3339: Date and Time on the Internet: Timestamps (2002)

G. Klyne and C. Newman; IETF

Timestamp syntax and numeric UTC offsets.

Consulted: 23 September 2026

https://www.rfc-editor.org/info/rfc3339/

[S10] PostgreSQL 17 documentation: Numeric Types

PostgreSQL Global Development Group

Integer ranges, exact numeric types and floating-point types.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/datatype-numeric.html

[S11] PostgreSQL 17 documentation: Date/Time Types

PostgreSQL Global Development Group

Date and timestamp semantics; time-zone handling.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/datatype-datetime.html

[S12] Python 3.12 tutorial: Floating-Point Arithmetic, Issues and Limitations

Python Software Foundation

Binary fractions and displayed floating-point results.

Consulted: 23 September 2026

https://docs.python.org/3.12/tutorial/floatingpoint.html

[S13] Python 3.12 library reference: decimal

Python Software Foundation

Decimal construction, precision, quantisation and rounding contexts.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/decimal.html

[S14] FAQ: UTF-8, UTF-16, UTF-32 and BOM

Unicode Consortium

Unicode encoding forms and UTF-8 byte lengths.

Consulted: 23 September 2026

https://www.unicode.org/faq/utf_bom.html

[S15] Unicode Standard Annex #15: Unicode Normalization Forms

Unicode Consortium

Canonical equivalence, NFC and the limits of normalisation.

Consulted: 23 September 2026

https://www.unicode.org/reports/tr15/

[S16] Unicode Standard Annex #29: Unicode Text Segmentation

Unicode Consortium

Grapheme clusters and user-perceived character boundaries.

Consulted: 23 September 2026

https://www.unicode.org/reports/tr29/

[S17] PostgreSQL 17 documentation: Comparison Functions and Operators

PostgreSQL Global Development Group

NULL comparisons and IS NULL.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/functions-comparison.html

[S18] PostgreSQL 17 documentation: Logical Operators

PostgreSQL Global Development Group

Three-valued Boolean logic.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/functions-logical.html

[S19] PostgreSQL 17 documentation: Aggregate Functions

PostgreSQL Global Development Group

COUNT and AVG treatment of non-null inputs.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/functions-aggregate.html

[S20] Python 3.12 library reference: sqlite3

Python Software Foundation

Embedded SQLite connections and bound parameters.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/sqlite3.html

[S21] Datatypes in SQLite

SQLite project

Dynamic typing, storage classes and differences from other engines.

Consulted: 23 September 2026

https://www.sqlite.org/datatype3.html

[S22] Python 3.12 library reference: Built-in Types

Python Software Foundation

Integer precision and the bool subclass relationship.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/stdtypes.html

[S23] Python 3.12 library reference: datetime

Python Software Foundation

Aware datetimes, offsets and conversion to UTC.

Consulted: 23 September 2026

https://docs.python.org/3.12/library/datetime.html

[S24] SQL Language Expressions

SQLite project

NULL and logical operations in the supplied SQLite lab.

Consulted: 23 September 2026

https://www.sqlite.org/lang_expr.html

[S25] Transaction

SQLite project

Explicit BEGIN, COMMIT and ROLLBACK in the supplied lab.

Consulted: 23 September 2026

https://www.sqlite.org/lang_transaction.html

[S26] VIM3, 2.3: Measurand

JCGM / BIPM

Defining the quantity intended to be measured.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.3.html

[S27] SI prefixes

BIPM

Decimal factors and prefix symbols; conversions in the text are original calculations.

Consulted: 23 September 2026

https://www.bipm.org/en/measurement-units/si-prefixes

[S28] VIM3, 2.13: Measurement accuracy

JCGM / BIPM

Accuracy is distinguished from precision.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.13.html

[S29] VIM3, 2.15: Measurement precision

JCGM / BIPM

Agreement among repeated indications under specified conditions.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.15.html

[S30] VIM3, 2.39: Calibration

JCGM / BIPM

Calibration is not the same operation as adjustment or verification.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.39.html

[S31] Combining uncertainty components

NIST

Propagation of uncertainty, sensitivities and covariances.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/combination.html

[S32] Type A evaluation of standard uncertainty

NIST

Standard uncertainty of a mean for independent observations.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/typea.html

[S33] VIM3, 4.14: Resolution

JCGM / BIPM

A perceptible response to a change in the measured quantity.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/4.14.html

[S34] Type B evaluation of standard uncertainty

NIST

Uncertainty evaluated from other information and assumed distributions.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/typeb.html

[S35] VIM3, 2.26: Measurement uncertainty

JCGM / BIPM

Dispersion of values attributed to a measurand using available information.

Consulted: 23 September 2026

https://jcgm.bipm.org/vim/en/2.26.html

[S36] Expanded uncertainty and coverage factors

NIST

U = k times combined standard uncertainty; coverage depends on assumptions.

Consulted: 23 September 2026

https://physics.nist.gov/cuu/Uncertainty/coverage.html

[S37] PostgreSQL 17 documentation: Identity Columns

PostgreSQL Global Development Group

Generated values do not by themselves enforce uniqueness.

Consulted: 23 September 2026

https://www.postgresql.org/docs/17/ddl-identity-columns.html

[S38] RFC 9562: Universally Unique IDentifiers (UUIDs), 2024

IETF; K. Davis, B. Peabody and P. Leach

UUID version 4 has 122 random bits; collision estimate is an original conditional calculation.

Consulted: 23 September 2026

https://www.rfc-editor.org/rfc/rfc9562.html

[S39] CloudEvents specification, version 1.0.2

Cloud Native Computing Foundation; CloudEvents project

Event source and id uniqueness; the wire specversion value is 1.0.

Consulted: 23 September 2026

https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/spec.md

[S40] Event Sourcing pattern

Microsoft Azure Architecture Center

Architecture context: event history, projections, snapshots, replay and trade-offs. The shop state machines are original teaching designs, not a Microsoft reference implementation.

Consulted: 23 September 2026

https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

[S41] Time, Clocks, and the Ordering of Events in a Distributed System (1978)

Leslie Lamport

Communications of the ACM 21(7), 558–565; happened-before and logical clocks. Counter exercises are original illustrations.

Consulted: 23 September 2026

https://lamport.azurewebsites.net/pubs/time-clocks.pdf

[S42] Python 3.13 library reference: pathlib

Python Software Foundation

Pure paths versus filesystem operations; path components and lexical interpretation.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/pathlib.html

[S43] Python 3.13 library reference: io

Python Software Foundation

Text/byte streams, encoding, newline handling and input boundaries.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/io.html

[S44] Python 3.13 library reference: csv

Python Software Foundation

CSV dialects, reader and writer behaviour; strict mode is not domain validation.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/csv.html

[S45] Python 3.13 library reference: json

Python Software Foundation

Object pairs hooks, numeric constants, supported types and parser resource cautions.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/json.html

[S46] Python 3.13 library reference: struct

Python Software Foundation

Explicit byte order, widths and binary packing; the KDBF envelope is original teaching material.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/struct.html

[S47] Apache Parquet documentation: Overview

Apache Software Foundation

Column-oriented file-format purpose; no Parquet performance measurements are claimed.

Consulted: 24 September 2026

https://parquet.apache.org/docs/overview/

[S48] Apache Avro 1.12.0 specification

Apache Software Foundation

Schema-based representation and writer/reader schema resolution; not implemented by the toy envelope.

Consulted: 24 September 2026

https://avro.apache.org/docs/1.12.0/specification/

[S49] Model for Tabular Data and Metadata on the Web

W3C

Representations, semantic values, cells and annotations in tabular data.

Consulted: 24 September 2026

https://www.w3.org/TR/tabular-data-model/

[S50] SQLite Is Serverless

SQLite project

Embedded, in-process engine versus a separately running database server.

Consulted: 24 September 2026

https://www.sqlite.org/serverless.html

[S51] Appropriate Uses For SQLite

SQLite project

Embedded-use scope, local access and situations favouring client-server databases.

Consulted: 24 September 2026

https://www.sqlite.org/whentouse.html

[S52] PostgreSQL 17 documentation: Architectural Fundamentals

PostgreSQL Global Development Group

Database client/server architecture and separation from application code.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/tutorial-arch.html

[S53] Atomic Commit In SQLite

SQLite project

Journaling and filesystem/device assumptions. This lab does not inject power failures.

Consulted: 24 September 2026

https://www.sqlite.org/atomiccommit.html

[S54] Isolation In SQLite

SQLite project

Writer serialization and transaction visibility; local lock demonstration is separately bounded.

Consulted: 24 September 2026

https://www.sqlite.org/isolation.html

[S55] SQLite Online Backup API

SQLite project

Engine-supported copying into a destination database; no production recovery-time claim.

Consulted: 24 September 2026

https://www.sqlite.org/backup.html

[S56] PostgreSQL 17 documentation: Table Basics

PostgreSQL Global Development Group

Tables, columns and declared data types.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/ddl-basics.html

[S57] PostgreSQL 17 documentation: Select Lists

PostgreSQL Global Development Group

Projection, output columns and duplicate elimination.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/queries-select-lists.html

[S58] PostgreSQL 17 documentation: Table Expressions

PostgreSQL Global Development Group

Joins, outer joins, qualification and the effect of later filtering.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/queries-table-expressions.html

[S59] SQLite Foreign Key Support

SQLite project

Per-connection enforcement, parent/child references and nullable child keys.

Consulted: 24 September 2026

https://www.sqlite.org/foreignkeys.html

[S60] STRICT Tables

SQLite project

SQLite 3.37.0 minimum, strict storage types and permitted lossless coercion.

Consulted: 24 September 2026

https://www.sqlite.org/stricttables.html

[S61] CREATE TABLE

SQLite project

CHECK, NOT NULL, uniqueness and primary-key implementation details.

Consulted: 24 September 2026

https://www.sqlite.org/lang_createtable.html

[S62] Python 3.13 library reference: pickle

Python Software Foundation

Do not unpickle untrusted data; serialization is not automatically safe execution.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/pickle.html

[S63] CSV Injection

OWASP Foundation community

Spreadsheet formula interpretation is separate from CSV quotation; no universal sanitizer is claimed.

Consulted: 24 September 2026

https://community.owasp.org/attacks/CSV_Injection

[S64] Python 3.13 library reference: os

Python Software Foundation

Filesystem operations, replacement semantics and system-dependent boundaries.

Consulted: 24 September 2026

https://docs.python.org/3.13/library/os.html

[S65] PostgreSQL 17 documentation: Using EXPLAIN

PostgreSQL Global Development Group

Plan costs and actual execution with EXPLAIN ANALYZE.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/using-explain.html

[S66] PostgreSQL 17 documentation: Transaction Isolation

PostgreSQL Global Development Group

Engine-specific isolation guarantees and transaction retry requirements.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/transaction-iso.html

[S67] PostgreSQL 17 documentation: SQL Dump

PostgreSQL Global Development Group

Logical backup and restoration; cited context, not executed in the SQLite lab.

Consulted: 24 September 2026

https://www.postgresql.org/docs/17/backup-dump.html

[S68] PostgreSQL 17 documentation: CREATE DOMAIN

PostgreSQL Global Development Group

Reusable constrained base types and domain-nullability caveats; the SQL illustration is not executed in the SQLite lab.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-createdomain.html

[S69] PostgreSQL 17 documentation: Schemas

PostgreSQL Global Development Group

Schema namespaces, qualified names and name-resolution context; distinct from the general modelling meaning of schema.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-schemas.html

[S70] PostgreSQL 17 documentation: COMMENT

PostgreSQL Global Development Group

Object comments as descriptive metadata, not enforced business rules or a place for secrets.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-comment.html

[S71] PostgreSQL 17 documentation: The Information Schema

PostgreSQL Global Development Group

Standard-oriented metadata inspection and its distinction from business meaning.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/information-schema.html

[S72] PostgreSQL 17 documentation: Lexical Structure

PostgreSQL Global Development Group

Identifiers, text literals and quoting; naming conventions in the book are editorial choices.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-syntax-lexical.html

[S73] PostgreSQL 17 documentation: SET CONSTRAINTS

PostgreSQL Global Development Group

Eligible constraint classes and checking-mode changes; PostgreSQL is documented, not executed.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-set-constraints.html

[S74] PostgreSQL 17 documentation: CREATE TABLE

PostgreSQL Global Development Group

Keys, references, match semantics and deferrability; product-specific rather than a cross-engine promise.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-createtable.html

[S75] PRAGMA Statements

SQLite project

Metadata interfaces and separate scopes of foreign_keys, integrity_check and foreign_key_check.

Consulted: 25 September 2026

https://www.sqlite.org/pragma.html

[S76] The ON CONFLICT Clause

SQLite project

Default ABORT statement rollback and the distinction between replacement and an ordinary update.

Consulted: 25 September 2026

https://www.sqlite.org/lang_conflict.html

[S77] Result and Error Codes

SQLite project

Machine-readable error categories; selected extended constraint codes observed by the lab.

Consulted: 25 September 2026

https://www.sqlite.org/rescode.html

[S78] PostgreSQL 17 documentation: Modifying Tables

PostgreSQL Global Development Group

Constraint-change and existing-data considerations; no production or PostgreSQL migration is run.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-alter.html

[S79] PostgreSQL 17 documentation: Querying a Table

PostgreSQL Global Development Group

SELECT fundamentals, source columns, expressions and result questions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/tutorial-select.html

[S80] SELECT

SQLite project

Logical result semantics, output expressions, filtering, duplicate elimination and ordering.

Consulted: 25 September 2026

https://www.sqlite.org/lang_select.html

[S81] PostgreSQL 17 documentation: LIMIT and OFFSET

PostgreSQL Global Development Group

Output limits and the need for predictable ordering; not a performance bound.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/queries-limit.html

[S82] PostgreSQL 17 documentation: Conditional Expressions

PostgreSQL Global Development Group

CASE and COALESCE as explicit conditional choices; the narrative fixtures are original.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/functions-conditional.html

[S83] INSERT

SQLite project

Explicit target columns and value insertion in isolated draft examples.

Consulted: 25 September 2026

https://www.sqlite.org/lang_insert.html

[S84] UPDATE

SQLite project

Assignment and predicate scope; unqualified update demonstrated only in a disposable transaction.

Consulted: 25 September 2026

https://www.sqlite.org/lang_update.html

[S85] DELETE

SQLite project

Selected-row deletion semantics; not a claim of erasure across backups or external copies.

Consulted: 25 September 2026

https://www.sqlite.org/lang_delete.html

[S86] Python 3.13 library reference: sqlite3

Python Software Foundation

Driver binding, cursors, result counts, connection closing and explicit transaction configuration.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/sqlite3.html

[S87] Built-In Scalar SQL Functions

SQLite project

COALESCE behaviour, including the difference between NULL and empty text.

Consulted: 25 September 2026

https://www.sqlite.org/lang_corefunc.html

[S88] Database System Concepts, seventh edition: Chapter 7 author slides, Normalization

Abraham Silberschatz, Henry F. Korth and S. Sudarshan

Functional dependencies, lossless decomposition, dependency preservation, BCNF and third normal form. The shop exercises are original.

Consulted: 25 September 2026

https://www.db-book.com/slides-dir/PDF-dir/ch7.pdf

[S89] Query Planning

SQLite project

Scans, ordered lookup, compound and covering indexes, and sorting choices.

Consulted: 25 September 2026

https://www.sqlite.org/queryplanner.html

[S90] EXPLAIN QUERY PLAN

SQLite project

Human-readable SCAN, SEARCH and temporary-tree descriptions; output format is not a stable application interface.

Consulted: 25 September 2026

https://www.sqlite.org/eqp.html

[S91] PostgreSQL 17: Multicolumn Indexes

PostgreSQL Global Development Group

Column order and constraints on efficient access in the explicitly cited version.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-multicolumn.html

[S92] PostgreSQL 17: Index-Only Scans and Covering Indexes

PostgreSQL Global Development Group

INCLUDE payloads and visibility checks; covering does not guarantee zero heap visits.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-index-only-scans.html

[S93] Partial Indexes

SQLite project

Predicate-restricted indexes and eligibility based on implication.

Consulted: 25 September 2026

https://www.sqlite.org/partialindex.html

[S94] PostgreSQL 17: Statistics Used by the Planner

PostgreSQL Global Development Group

Sampling, distribution estimates and extended statistics.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/planner-stats.html

[S95] PostgreSQL 17: Resource Consumption

PostgreSQL Global Development Group

Operation-level memory allowances and spill; not a benchmark of this manuscript.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/runtime-config-resource.html

[S96] PostgreSQL 17 tutorial: Window Functions

PostgreSQL Global Development Group

Window partitions, ordering and preservation of detail rows.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/tutorial-window.html

[S97] Window Functions

SQLite project

Window frames, peers and aggregate-window behaviour in the educational SQL exercises.

Consulted: 25 September 2026

https://www.sqlite.org/windowfunctions.html

[S98] The SQLite Query Optimizer Overview

SQLite project

Access-path transformations, joins and implementation-specific planning decisions.

Consulted: 25 September 2026

https://www.sqlite.org/optoverview.html

[S99] Python 3.13: hashlib

Python Software Foundation

Digest interfaces; the collision and comparison exercises are original.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/hashlib.html

[S100] FIPS 180-4: Secure Hash Standard, August 2015

NIST

Standardized SHA-2 digests, distinguished from authentication and encryption.

Consulted: 25 September 2026

https://csrc.nist.gov/pubs/fips/180-4/upd1/final

[S101] PostgreSQL 17: Hash Indexes

PostgreSQL Global Development Group

Equality-oriented, lossy hash access and collision rechecking.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/hash-index.html

[S102] Python 3.13: time

Python Software Foundation

Monotonic performance timers and their units, not an assertion of nanosecond accuracy.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/time.html

[S103] Monitoring Distributed Systems

Rob Ewaschuk; Google SRE

Latency, traffic, errors and saturation as monitoring context; workload examples are invented.

Consulted: 25 September 2026

https://sre.google/sre-book/monitoring-distributed-systems/

[S104] PostgreSQL 17: Index Types

PostgreSQL Global Development Group

Index methods support different operators; a method name is not a universal performance promise.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-types.html

[S105] PostgreSQL 17: B-Tree Indexes

PostgreSQL Global Development Group

B-tree structure and implementation notes. The hand-worked B+ tree is deliberately not a PostgreSQL implementation.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/btree.html

[S106] PostgreSQL 17: Indexes and ORDER BY

PostgreSQL Global Development Group

Ordered access and explicit query ordering.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/indexes-ordering.html

[S107] PostgreSQL 17: ANALYZE

PostgreSQL Global Development Group

Updating sampled planner statistics and operational scope.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-analyze.html

[S108] Python 3.13: Data Model

Python Software Foundation

Hash/equality contracts and process-randomized string and byte hashes.

Consulted: 25 September 2026

https://docs.python.org/3.13/reference/datamodel.html

[S109] Python 3.13: bisect

Python Software Foundation

Ordered-list insertion positions; insertion cost and concurrent-mutation limitations.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/bisect.html

[S110] PostgreSQL 17: Window Functions

PostgreSQL Global Development Group

Ranking, offsets, frames and frame-sensitive last_value behaviour.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/functions-window.html

[S111] PostgreSQL 17: The Path of a Query

PostgreSQL Global Development Group

Parsing, rewriting, planning and execution stages.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/query-path.html

[S112] PostgreSQL 17: Planner/Optimizer

PostgreSQL Global Development Group

Plan search, nested-loop, merge and hash joins; optimality is bounded by search and estimation.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/planner-optimizer.html

[S113] PostgreSQL 17: EXPLAIN

PostgreSQL Global Development Group

Diagnostic options, actual execution with ANALYZE, instrumentation boundaries and non-text formats.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-explain.html

[S114] PostgreSQL 17: Query Planning

PostgreSQL Global Development Group

Planner cost and method settings; experimentation is distinct from production tuning.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/runtime-config-query.html

[S115] PostgreSQL 17: Row Estimation Examples

PostgreSQL Global Development Group

Selectivity estimation and its assumptions; the shop arithmetic is original.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/row-estimation-examples.html

[S116] PostgreSQL 17: WITH Queries (Common Table Expressions)

PostgreSQL Global Development Group

Named query stages and materialisation/folding behaviour in the explicitly cited version.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/queries-with.html

[S117] PostgreSQL 17 tutorial: Transactions

PostgreSQL Global Development Group

Transaction blocks, commit, rollback and the scope of database changes.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/tutorial-transactions.html

[S118] PostgreSQL 17: Explicit Locking

PostgreSQL Global Development Group

Lock modes, row/table distinctions, deadlocks and advisory-lock lifetimes.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/explicit-locking.html

[S119] PostgreSQL 17: Concurrency Control Introduction

PostgreSQL Global Development Group

Multiversion visibility and distinction from application history.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/mvcc-intro.html

[S120] PostgreSQL 17: Serialization Failure Handling

PostgreSQL Global Development Group

Whole-transaction retries, SQLSTATE 40001 and careful classification of other failures.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/mvcc-serialization-failure-handling.html

[S121] PostgreSQL 17: SAVEPOINT

PostgreSQL Global Development Group

Savepoint scope within a transaction.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-savepoint.html

[S122] PostgreSQL 17: Sequence Manipulation Functions

PostgreSQL Global Development Group

Sequence allocation and non-rollback behaviour; gaps are not missing-business-event proof.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/functions-sequence.html

[S123] PostgreSQL 17: Routine Vacuuming

PostgreSQL Global Development Group

Version cleanup, visibility maps, space reuse and transaction-identifier maintenance.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/routine-vacuuming.html

[S124] PostgreSQL 17: System Columns

PostgreSQL Global Development Group

Version-related metadata and the limits of physical tuple identifiers.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-system-columns.html

[S125] Transactional outbox pattern

Amazon Web Services

Atomic recording of business changes and delivery intent; duplicate delivery still requires handling.

Consulted: 25 September 2026

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html

[S126] Making retries safe with idempotent APIs

Amazon Web Services Builders Library

Caller request identity, repeated intent and semantic-equivalence considerations.

Consulted: 25 September 2026

https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

[S127] Savepoints

SQLite project

ROLLBACK TO and RELEASE, including the difference between inner release and outer commit.

Consulted: 25 September 2026

https://www.sqlite.org/lang_savepoint.html

[S128] PostgreSQL 17 documentation: Database Page Layout

PostgreSQL Global Development Group

Page headers, item identifiers, tuple layout and implementation boundaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/storage-page-layout.html

[S129] PostgreSQL 17 documentation: Write-Ahead Logging

PostgreSQL Global Development Group

WAL-before-data ordering, redo and grouped synchronization.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-intro.html

[S130] PostgreSQL 17 documentation: Reliability

PostgreSQL Global Development Group

Storage-cache assumptions, partial page writes and reliability limits.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-reliability.html

[S131] PostgreSQL 17 documentation: Continuous Archiving and Point-in-Time Recovery

PostgreSQL Global Development Group

Base backups, continuous WAL coverage, recovery targets and timelines.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/continuous-archiving.html

[S132] Write-Ahead Logging

SQLite project

WAL frames, checkpointing, concurrency limits and the disclosed 2026 WAL-reset bug; not an endorsement of the historical lab runtime.

Consulted: 25 September 2026

https://www.sqlite.org/wal.html

[S133] Database File Format

SQLite project

Page sizes, header fields, overflow and journal/WAL representation.

Consulted: 25 September 2026

https://www.sqlite.org/fileformat.html

[S134] Compaction

RocksDB project

Sorted runs, leveled/tiered strategies and read/write/space trade-offs.

Consulted: 25 September 2026

https://github.com/facebook/rocksdb/wiki/Compaction

[S135] PostgreSQL 17 documentation: WAL Configuration

PostgreSQL Global Development Group

Checkpoints, redo positions and resource trade-offs.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-configuration.html

[S136] PostgreSQL 17 documentation: Asynchronous Commit

PostgreSQL Global Development Group

Acknowledgement before WAL flush and the distinction from disabling fsync.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/wal-async-commit.html

[S137] PostgreSQL 17 documentation: Data Checksums

PostgreSQL Global Development Group

Detection scope and limitations of page checksums.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/checksums.html

[S138] PostgreSQL 17 documentation: pg_verifybackup

PostgreSQL Global Development Group

Manifest verification and the explicit need for test restores.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/app-pgverifybackup.html

[S139] How To Corrupt An SQLite Database File

SQLite project

Operational corruption hazards, copying and journal-handling cautions.

Consulted: 25 September 2026

https://www.sqlite.org/howtocorrupt.html

[S140] RocksDB Overview

RocksDB project

Memtables, log files, sorted tables and the library boundary.

Consulted: 25 September 2026

https://github.com/facebook/rocksdb/wiki/RocksDB-Overview

[S141] fsync(2)

Linux man-pages project

File synchronization, directory metadata and error reporting; Linux-specific.

Consulted: 25 September 2026

https://man7.org/linux/man-pages/man2/fsync.2.html

[S142] write(2)

Linux man-pages project

Partial writes and the difference between write success and persistence.

Consulted: 25 September 2026

https://man7.org/linux/man-pages/man2/write.2.html

[S143] PostgreSQL 17 documentation: TOAST

PostgreSQL Global Development Group

Large-value compression/out-of-line storage and physical row boundaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/storage-toast.html

[S144] PostgreSQL 17 documentation: The Cumulative Statistics System

PostgreSQL Global Development Group

I/O statistics, cache boundaries, update timing and monitoring scope.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/monitoring-stats.html

[S145] VACUUM

SQLite project

File rebuilding, space reclamation and operational constraints.

Consulted: 25 September 2026

https://www.sqlite.org/lang_vacuum.html

[S146] PostgreSQL 17 documentation: amcheck

PostgreSQL Global Development Group

Table/index structural verification and limitations.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/amcheck.html

[S147] PostgreSQL 17 documentation: Log-Shipping Standby Servers

PostgreSQL Global Development Group

Physical streaming, lag, slots and synchronous standby acknowledgement boundaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/warm-standby.html

[S148] PostgreSQL 17 documentation: Logical Replication

PostgreSQL Global Development Group

Publication/subscription, initial synchronization, identities and conflicts.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/logical-replication.html

[S149] PostgreSQL 17 documentation: Write Ahead Log settings

PostgreSQL Global Development Group

synchronous_commit modes, local/remote flush and replay distinctions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/runtime-config-wal.html

[S150] PostgreSQL 17 documentation: Failover

PostgreSQL Global Development Group

Promotion, external failure detection and preventing two active primaries.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/warm-standby-failover.html

[S151] PostgreSQL 17 documentation: pg_rewind

PostgreSQL Global Development Group

Divergent histories, prerequisites and recovery/reconfiguration cautions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/app-pgrewind.html

[S152] PostgreSQL 17 documentation: Table Partitioning

PostgreSQL Global Development Group

Range/list/hash table partitions, pruning and implementation restrictions; not automatic multi-host sharding.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-partitioning.html

[S153] Perspectives on the CAP Theorem

Seth Gilbert and Nancy A. Lynch

Authors' retrospective explaining atomic read/write consistency, availability, unreliable communication and the proof sketch.

Consulted: 25 September 2026

https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf

[S154] In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout

Raft terms, durable voting, log matching, current-term commitment, membership and client-read safeguards.

Consulted: 25 September 2026

https://raft.github.io/raft.pdf

[S155] PostgreSQL 17 documentation: Two-Phase Transactions

PostgreSQL Global Development Group

Prepared-transaction state and the external transaction-manager boundary.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/two-phase.html

[S156] PostgreSQL 17 documentation: PREPARE TRANSACTION

PostgreSQL Global Development Group

Prepared state, retained locks, completion responsibilities and operational cautions.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-prepare-transaction.html

[S157] Sagas (1987)

Hector Garcia-Molina and Kenneth Salem

Original paper on long-lived transactions, interleaving and application-defined compensation.

Consulted: 25 September 2026

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

[S158] Consumer Acknowledgements and Publisher Confirms

RabbitMQ project

Separate confirmation boundaries, delivery tags, redelivery and bounded unacknowledged work; unversioned documentation.

Consulted: 25 September 2026

https://www.rabbitmq.com/docs/confirms

[S159] Queues

RabbitMQ project

Ordering scope, multiple consumers, redelivery and queue properties; unversioned documentation.

Consulted: 25 September 2026

https://www.rabbitmq.com/docs/queues

[S160] Client-side caching reference

Redis

Invalidation races, connection loss and cache-resource bounds; unversioned documentation.

Consulted: 25 September 2026

https://redis.io/docs/latest/develop/reference/client-side-caching/

[S161] Cache-Aside pattern

Microsoft

Loading/invalidation trade-offs and consistency limits of derived caches.

Consulted: 25 September 2026

https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

[S162] Dynamo: Amazon's Highly Available Key-value Store (2007)

Giuseppe DeCandia and co-authors

Original Dynamo design: consistent hashing, versions, sloppy quorums and reconciliation; not a statement about current DynamoDB.

Consulted: 25 September 2026

https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[S163] The Chubby lock service for loosely-coupled distributed systems (2006)

Mike Burrows

Lock generations and recipient-validated sequencers for rejecting obsolete operations.

Consulted: 25 September 2026

https://storage.googleapis.com/gweb-research2023-media/pubtools/4444.pdf

[S164] PostgreSQL 17 documentation: Logical Replication Restrictions

PostgreSQL Global Development Group

DDL, sequence and object coverage boundaries and subscriber compatibility.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/logical-replication-restrictions.html

[S165] PostgreSQL connector documentation, stable page showing version 3.6 when consulted

Debezium project

Initial snapshot/change-stream boundary, offsets and connector failure behaviour; no connector deployment is performed.

Consulted: 25 September 2026

https://debezium.io/documentation/reference/stable/connectors/postgresql.html

[S166] PostgreSQL 17 documentation: Logical Decoding Concepts

PostgreSQL Global Development Group

Slots, retained log positions and possible change redelivery following a crash.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/logicaldecoding-explanation.html

[S167] Apache Beam Programming Guide: windows, watermarks, triggers and state

Apache Software Foundation

Event-time membership, late-data policy, triggers and accumulation modes; the small window model is original, not a Beam runtime.

Consulted: 25 September 2026

https://beam.apache.org/documentation/programming-guide/

[S168] Apache Iceberg table specification

Apache Software Foundation

Field IDs, snapshots, manifests, atomic metadata replacement and snapshot retention. Discussion is limited to these concepts; not an implementation of the entire evolving specification.

Consulted: 25 September 2026

https://iceberg.apache.org/spec/

[S169] PostgreSQL 17 documentation: Materialized Views

PostgreSQL Global Development Group

Stored query results and refresh/freshness trade-offs.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/rules-materializedviews.html

[S170] Apache Parquet: File Format

Apache Software Foundation

File metadata, row groups and column chunks; companion byte-layout model does not produce Parquet files.

Consulted: 25 September 2026

https://parquet.apache.org/docs/file-format/

[S171] Apache Parquet: Encodings

Apache Software Foundation

Plain, dictionary, run-length and delta encoding concepts; actual Parquet bitstream rules are distinct from the toy codec.

Consulted: 25 September 2026

https://parquet.apache.org/docs/file-format/data-pages/encodings/

[S172] Apache Parquet: Page Index

Apache Software Foundation

Optional statistics and offsets for conservative page skipping.

Consulted: 25 September 2026

https://parquet.apache.org/docs/file-format/pageindex/

[S173] SQLite FTS5 Extension

SQLite project

Tokenizers, phrase queries, external-content responsibilities and negative BM25 scores. Only basic available FTS5 facilities are exercised.

Consulted: 25 September 2026

https://www.sqlite.org/fts5.html

[S174] PostgreSQL 17 documentation: Full Text Search Introduction

PostgreSQL Global Development Group

Documents, lexemes and full-text query representation.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/textsearch-intro.html

[S175] PostgreSQL 17 documentation: Controlling Text Search

PostgreSQL Global Development Group

Parsing, normalization, query construction, ranking and safe display limitations.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/textsearch-controls.html

[S176] MetricType and distances

Faiss project

Squared Euclidean distance, inner product, cosine and normalization. The lab uses direct Python arithmetic, not Faiss.

Consulted: 25 September 2026

https://github.com/facebookresearch/faiss/wiki/MetricType-and-distances

[S177] Faiss indexes

Faiss project

Exhaustive versus approximate indexes, vector-memory estimates and index trade-offs.

Consulted: 25 September 2026

https://github.com/facebookresearch/faiss/wiki/Faiss-indexes

[S178] Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

Yu. A. Malkov and D. A. Yashunin

Original HNSW research, first submitted 2016; graph navigation concept, not a claim about every present product or workload.

Consulted: 25 September 2026

https://arxiv.org/abs/1603.09320

[S179] e-Handbook of Statistical Methods: Autocorrelation

NIST / SEMATECH

Dependence between observations across lags; repeated observations are not automatically independent.

Consulted: 25 September 2026

https://www.itl.nist.gov/div898/handbook/eda/section3/eda35c.htm

[S180] e-Handbook of Statistical Methods: Scatter Plot

NIST / SEMATECH

Association can be inspected graphically but does not establish cause. All business examples in the chapter are synthetic.

Consulted: 25 September 2026

https://www.itl.nist.gov/div898/handbook/eda/section3/scatterp.htm

[S181] Introduction to Information Retrieval: Evaluation of unranked retrieval sets

Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze

Authors-hosted textbook definitions of precision and recall; all local query judgements are synthetic.

Consulted: 25 September 2026

https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-unranked-retrieval-sets-1.html

[S182] Introduction to Information Retrieval: Okapi BM25, a non-binary model

Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze

Term-frequency saturation, document-length normalization and development-set tuning.

Consulted: 25 September 2026

https://nlp.stanford.edu/IR-book/html/htmledition/okapi-bm25-a-non-binary-model-1.html

[S183] e-Handbook of Statistical Methods: Confidence intervals for a proportion

NIST / SEMATECH

Wilson interval formula, binomial assumptions and small-sample cautions. Numerical examples in the book are original.

Consulted: 25 September 2026

https://www.itl.nist.gov/div898/handbook/prc/section2/prc241.htm

[S184] PostgreSQL 17 documentation: Preferred Index Types for Text Search

PostgreSQL Global Development Group

GIN inverted indexes and GiST signatures/rechecking; PostgreSQL examples are documented, not executed.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/textsearch-indexes.html

[S185] Authorization Cheat Sheet

OWASP Foundation

Default-deny authorization, permission checks and tests; not authentication implementation.

Consulted: 25 September 2026

https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

[S186] PostgreSQL 17: Privileges

PostgreSQL Global Development Group

Role privileges and object ownership.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-priv.html

[S187] PostgreSQL 17: Row Security Policies

PostgreSQL Global Development Group

Row-policy semantics, default denial and privileged bypass; not executed in the SQLite labs.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/ddl-rowsecurity.html

[S188] Multi-Tenant Security Cheat Sheet

OWASP Foundation

Tenant-aware authorization and isolation across storage and caches.

Consulted: 25 September 2026

https://cheatsheetseries.owasp.org/cheatsheets/Multi_Tenant_Security_Cheat_Sheet.html

[S189] Secrets Management Cheat Sheet

OWASP Foundation

Secret inventory, lifecycle, rotation and avoiding leakage.

Consulted: 25 September 2026

https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

[S190] PostgreSQL 17: Routine Vacuuming

PostgreSQL Global Development Group

Dead tuples, reuse of storage and cleanup; not proof of media sanitization.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/routine-vacuuming.html

[S191] PRAGMA secure_delete

SQLite project

Secure-delete configuration and limitations, including virtual tables and separate copies.

Consulted: 25 September 2026

https://www.sqlite.org/pragma.html#pragma_secure_delete

[S192] SP 800-88 Revision 2: Guidelines for Media Sanitization (September 2025)

NIST

Publication landing record and sanitization scope. Not a claim that a device was sanitized or the full publication independently audited.

Consulted: 25 September 2026

https://csrc.nist.gov/pubs/sp/800/88/r2/final

[S193] Object Model

OpenLineage project

Datasets, jobs, runs and metadata used to describe lineage.

Consulted: 25 September 2026

https://openlineage.io/docs/spec/object-model/

[S194] PostgreSQL 17: ALTER TABLE

PostgreSQL Global Development Group

Schema changes, constraint validation and lock behavior in the cited version.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-altertable.html

[S195] PostgreSQL 17: CREATE INDEX

PostgreSQL Global Development Group

Concurrent index creation restrictions and operational caveats.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/sql-createindex.html

[S196] ALTER TABLE

SQLite project

Supported alterations and table-rebuild procedure; version-sensitive documentation.

Consulted: 25 September 2026

https://www.sqlite.org/lang_altertable.html

[S197] Service Level Objectives

Google SRE

Indicators, objectives and measurement boundaries.

Consulted: 25 September 2026

https://sre.google/sre-book/service-level-objectives/

[S198] Signals

OpenTelemetry project

Roles of traces, metrics, logs and related telemetry signals.

Consulted: 25 September 2026

https://opentelemetry.io/docs/concepts/signals/

[S199] Handling Overload

Google SRE

Load, saturation and overload-control considerations.

Consulted: 25 September 2026

https://sre.google/sre-book/handling-overload/

[S200] PostgreSQL 17: The Cumulative Statistics System

PostgreSQL Global Development Group

Engine-specific statistics and observation context.

Consulted: 25 September 2026

https://www.postgresql.org/docs/17/monitoring-stats.html

[S201] Python 3.13: unittest

Python Software Foundation

Test discovery, assertions, subtests and skipped-test semantics.

Consulted: 25 September 2026

https://docs.python.org/3.13/library/unittest.html