



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

Eight Parts · Fifty-Two Chapters · First Edition

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

Copyright © 2026 KedByte Technologies Private Limited. All rights reserved.

The original text, examples and illustrations in this book are published by KedByte Press. Reproduction or redistribution beyond applicable exceptions requires the publisher’s permission. Third-party names and referenced materials remain the property of their respective owners.

Trademarks. Product and organization names are used for explanation and source attribution. Their appearance does not imply endorsement of this book, its author, publisher or code.

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.

From the Founder & Chairman

A record can begin as a few words in a notebook. Before long, it may become an order in an application, a stock movement, a line in a report and a copy on another machine. Each step seems ordinary. Each step carries a question that deserves a proper answer: what exactly does this information mean, and what are we entitled to conclude from it?

This book starts there. It does not begin by asking the reader to choose a database product or memorize a list of technical terms. It begins with a small shop, two people and a sale. The machinery grows only when a real requirement in that fictional setting gives it a reason to exist.

The aim is to make knowledge transferable. A reader should understand why an index can help before learning its technical name, why two overlapping updates can lose work before choosing a locking rule, and why a backup is useful only when a recovery process can actually use it. Those ideas matter whether the reader is a student, an analyst, a business owner or an engineer.

Every main section follows the same route. First comes the idea in simple words. Then an everyday comparison, with an explicit warning about where the comparison stops being accurate. A worked example makes the quantities and steps visible. An explanation of the internal mechanism replaces the comparison with the actual moving parts. The engineering treatment adds precise vocabulary, implementation boundaries and sources. Finally, the terminology connects the ordinary meaning to its technical counterpart.

Simple language is not an excuse to hide the difficult part. A database accepting a row does not establish that the row describes a real event. A correct sum can answer the wrong question. A message delivered twice need not represent two separate purchases. A successful commit says something about a defined transaction and configuration, not about every external action a person hoped would follow it. Those distinctions are not minor qualifications at the edge of the subject. They are the subject.

The book also keeps the limits of evidence visible. Our shop and its records are invented teaching material. They are not customer data or reports of KedByte production incidents. The included programs exercise small, explicit cases. A local model of a replicated log is not a production consensus service. An exception followed by rollback is not a power-cut experiment. A restored in-memory database is not evidence that an off-site recovery objective has been met.

That discipline is useful beyond programming. When a dashboard offers a percentage, ask which population it describes. When a system claims that information has been deleted, ask which copies were inspected. When a new process appears faster, ask what workload was measured and what was excluded. Clear questions expose gaps that confident language can conceal.

The eight parts form one continuous path: meaning, databases, queries, correct changes, storage, distributed systems, analysis and operation. You may read straight through or return to a chapter when a practical question arises. Stable chapter and section numbers connect the complete edition, the separate volumes and the browser reader.

The final part returns to the original problem with better tools. We assemble a bounded service, trace retries and reports, rehearse a local restore, and separate what the checks demonstrate from what still belongs to a real deployment. The reference chapter and glossary then provide a route back through the ideas.

Understanding is not proved by using the largest vocabulary. It is proved by explaining a mechanism accurately, applying it to a concrete case and knowing where the evidence ends. That is the standard this book asks the reader to practice.

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

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 A — Meaning Before Machines	1
1 The Whole Data System in One Look	2
1.0 What this chapter gives you	2
1.1 One sale, many responsibilities	2
1.2 A record is not the world	5
1.3 Files, databases and the software between them	6
1.4 Questions produce answers, not new certainty	8
1.5 Changes, failures and the word “saved”	9
1.6 The map of the whole book	11
1.97 Check your understanding	12
1.98 Common wrong ideas	13
1.99 Chapter summary in 20 lines	13
2 What a Record Really Means	14
2.0 What this chapter gives you	14
2.1 The claim inside a record	14
2.2 Fields, values and the contract around them	16
2.3 One row means one what?	17
2.4 Identity without guesswork	19
2.5 Source, event time and recording time	21
2.6 Corrections and the limits of evidence	23
2.97 A record-design exercise	24
2.98 Common wrong ideas	25
2.99 Chapter summary in 20 lines	25
3 Numbers, Text, Time and Missing Values	26
3.0 What this chapter gives you	26
3.1 Choose a type before choosing a format	26
3.2 Integers and the units they count	28
3.3 Decimals, fractions and rounding	30
3.4 Text, Unicode and bytes	31
3.5 Dates, instants and durations	33
3.6 Missing is not zero	35
3.97 Practice, then inspect the evidence	37
3.98 Common wrong ideas	38
3.99 Chapter summary in 20 lines	38
4 Measurements, Units and Uncertainty	39
4.0 What this chapter gives you	39
4.1 What exactly was measured?	39
4.2 Units and dimensional checks	41

4.3 Precision versus accuracy	43
4.4 Rounding and significant figures	45
4.5 Sampling and missing observations	47
4.6 Carrying uncertainty into a report	49
4.97 Practice and worked answers	52
4.98 Common wrong ideas	52
4.99 Chapter summary in 20 lines	53
5 Identity, Keys and the Problem of Duplicates	54
5.0 What this chapter gives you	54
5.1 Names and identifiers	54
5.2 Natural and generated keys	56
5.3 Scope and composite identity	58
5.4 Duplicate delivery versus repeated events	61
5.5 Matching imperfect records	63
5.6 Merge, split and identity history	65
5.97 Practice and worked answers	67
5.98 Common wrong ideas	68
5.99 Chapter summary in 20 lines	68
6 From Events to State: Keeping History	70
6.0 What this chapter gives you	70
6.1 An event versus a current value	70
6.2 State transitions	73
6.3 Ordering and causality	76
6.4 Snapshots and projections	78
6.5 Corrections and reversal events	81
6.6 History with retention boundaries	83
6.97 Exercises with worked answers	85
6.98 Common wrong ideas	86
6.99 Chapter summary in 20 lines	87
Part B — From Records to Databases	88
7 Files, Folders and Data Formats	89
7.0 What this chapter gives you	89
7.1 Paths and file identity	89
7.2 Bytes, records and boundaries	92
7.3 CSV and quoting	95
7.4 JSON and nested data	98
7.5 Binary formats and versioning	100
7.6 Import validation and quarantine	103
7.97 Practice and worked answers	106
7.98 Common wrong ideas	107

7.99 Chapter summary in 20 lines	107
8 Why Databases Exist	109
8.0 What this chapter gives you	109
8.1 The shared-file problem	109
8.2 A database and its engine	112
8.3 Client-server and embedded designs	114
8.4 Transactions and queries	118
8.5 Operational responsibilities	121
8.6 When a file is enough	123
8.97 Practice and worked answers	126
8.98 Common wrong ideas	126
8.99 Chapter summary in 20 lines	127
9 Tables, Rows and Relationships	128
9.0 What this chapter gives you	128
9.1 Tables as sets of claims	128
9.2 Columns and domains	131
9.3 One-to-one and one-to-many	134
9.4 Many-to-many relationships	136
9.5 Optional relationships	139
9.6 Modelling the shop	142
9.97 Practice and worked answers	145
9.98 Common wrong ideas	146
9.99 Chapter summary in 20 lines	146
10 Designing a Schema That Matches the Work	148
10.0 What this chapter gives you	148
10.1 Workflows before tables	148
10.2 Row grain and boundaries	151
10.3 Mutable and historical values	154
10.4 Domain types	157
10.5 Naming and documentation	160
10.6 Reviewing the model with users	162
10.97 Practice and worked answers	165
10.98 Common wrong ideas	166
10.99 Chapter summary in 20 lines	166
11 Constraints: Rules the Database Can Enforce	168
11.0 What this chapter gives you	168
11.1 Not-null and check rules	168
11.2 Unique and primary keys	171
11.3 Foreign keys	174
11.4 Cross-row invariants	176
11.5 Deferred enforcement	179

11.6 Error handling and migration	182
11.97 Practice and worked answers	185
11.98 Common wrong ideas	186
11.99 Chapter summary in 20 lines	187
12 SQL From Zero: Asking Your First Questions	188
12.0 What this chapter gives you	188
12.1 Reading SELECT aloud	188
12.2 Filtering and NULL	191
12.3 Explicit ordering	194
12.4 Expressions and aliases	197
12.5 Inserting and updating safely	199
12.6 Parameters and transactions	202
12.97 Practice and worked answers	205
12.98 Common wrong ideas	207
12.99 Chapter summary in 20 lines	207
13 Joins and Normalisation: Putting Facts in the Right Place	209
13.0 What this chapter gives you	209
13.1 Why facts repeat	209
13.2 Joining by meaning	211
13.3 Inner and outer joins	212
13.4 Functional dependencies	214
13.5 Normal forms and trade-offs	216
13.6 Avoiding accidental multiplication	218
13.97 Practice and worked answers	219
13.98 Common wrong ideas	220
13.99 Chapter summary in 20 lines	220
Part C — Finding Answers	222
14 Scanning, Sorting and the Cost of a Question	223
14.0 What this chapter gives you	223
14.1 What a scan does	223
14.2 Row width and data volume	225
14.3 Sorting and memory	226
14.4 External work and spill	227
14.5 Selectivity and filtering	229
14.6 Cost versus elapsed time	230
14.97 Practice and worked answers	232
14.98 Common wrong ideas	232
14.99 Chapter summary in 20 lines	233

15	Indexes: Building a Faster Way to Find Things	234
15.0	What this chapter gives you	234
15.1	An extra access path	234
15.2	Read benefits and write costs	236
15.3	Composite key order	237
15.4	Covering and partial indexes	239
15.5	Selectivity and distribution	240
15.6	When the engine ignores an index	242
15.97	Practice and worked answers	243
15.98	Common wrong ideas	244
15.99	Chapter summary in 20 lines	244
16	B-Trees: Ordered Pages, Fast Lookups	246
16.0	What this chapter gives you	246
16.1	Ordered separators	246
16.2	Pages and fan-out	247
16.3	Search from root to leaf	249
16.4	Inserts and splits	250
16.5	Range scans	252
16.6	Implementation boundaries	253
16.97	Practice and worked answers	255
16.98	Common wrong ideas	255
16.99	Chapter summary in 20 lines	256
17	Hashing, Search Keys and Collisions	257
17.0	What this chapter gives you	257
17.1	A deterministic mapping	257
17.2	Buckets and collisions	258
17.3	Hash tables and equality	260
17.4	Cryptographic and non-cryptographic purposes	261
17.5	Distribution and adversarial inputs	263
17.6	Hashing in storage designs	264
17.97	Practice and worked answers	266
17.98	Common wrong ideas	266
17.99	Chapter summary in 20 lines	267
18	Query Plans and the Optimiser	268
18.0	What this chapter gives you	268
18.1	From SQL to a plan	268
18.2	Statistics and estimates	269
18.3	Join strategies	271
18.4	Access-path choices	272
18.5	Reading EXPLAIN	273
18.6	Bad estimates and safe intervention	274

18.97 Practice and worked answers	276
18.98 Common wrong ideas	276
18.99 Chapter summary in 20 lines	277
19 Aggregation, Windows and Useful Reports	278
19.0 What this chapter gives you	278
19.1 Groups and their grain	278
19.2 Counts, sums and averages	279
19.3 Windows versus groups	281
19.4 Time buckets	282
19.5 Distinctness and duplicates	283
19.6 Reconciliation totals	284
19.97 Practice and worked answers	286
19.98 Common wrong ideas	286
19.99 Chapter summary in 20 lines	286
20 Measuring Performance Without Fooling Yourself	288
20.0 What this chapter gives you	288
20.1 Define the workload	288
20.2 Latency distributions	289
20.3 Throughput and saturation	291
20.4 Warm versus cold state	292
20.5 Fair comparisons	293
20.6 Reporting limitations	294
20.97 Practice and worked answers	295
20.98 Common wrong ideas	296
20.99 Chapter summary in 20 lines	296
Part D — Correct Changes	298
21 Transactions: One Change, All or Nothing	299
21.0 What this chapter gives you	299
21.1 The boundary of a change	299
21.2 Commit and rollback	300
21.3 Atomicity and consistency	302
21.4 Isolation and durability	303
21.5 Savepoints	304
21.6 External effects outside the transaction	306
21.97 Practice and worked answers	307
21.98 Common wrong ideas	307
21.99 Chapter summary in 20 lines	308
22 Two People Change the Same Record	309
22.0 What this chapter gives you	309
22.1 Interleaving actions	309

22.2	Lost updates	310
22.3	Conditional writes	312
22.4	Uniqueness under competition	313
22.5	Read-modify-write traps	314
22.6	A stock-reservation case	315
22.97	Practice and worked answers	317
22.98	Common wrong ideas	317
22.99	Chapter summary in 20 lines	318
23	Isolation Levels and What You Can Observe	319
23.0	What this chapter gives you	319
23.1	Visibility rules	319
23.2	Statement and transaction snapshots	320
23.3	Non-repeatable reads	322
23.4	Write skew	323
23.5	Serialisable execution	324
23.6	Retries and operational cost	325
23.97	Practice and worked answers	327
23.98	Common wrong ideas	327
23.99	Chapter summary in 20 lines	327
24	Locks, Deadlocks and Safe Retries	329
24.0	What this chapter gives you	329
24.1	What is locked	329
24.2	Lock modes and duration	330
24.3	Blocking versus deadlock	331
24.4	Detection and ordering	333
24.5	Bounded retries	334
24.6	Avoiding repeated external effects	335
24.97	Practice and worked answers	336
24.98	Common wrong ideas	337
24.99	Chapter summary in 20 lines	337
25	Versions, Snapshots and Optimistic Concurrency	339
25.0	What this chapter gives you	339
25.1	Old and new row versions	339
25.2	Snapshot visibility	340
25.3	Cleanup and long readers	342
25.4	Version columns	343
25.5	Compare-and-set	344
25.6	Choosing a conflict policy	345
25.97	Practice and worked answers	346
25.98	Common wrong ideas	347
25.99	Chapter summary in 20 lines	347

26 Invariants, Idempotency and the Outbox	349
26.0 What this chapter gives you	349
26.1 Rules across operations	349
26.2 Request identity	350
26.3 Replay detection	352
26.4 Atomic intent recording	353
26.5 Outbox delivery and consumers	354
26.6 End-to-end limits	355
26.97 Practice and worked answers	357
26.98 Common wrong ideas	357
26.99 Chapter summary in 20 lines	358
 Part E — Storage and Recovery	 359
27 Pages, Buffers and the Storage Engine	360
27.0 What this chapter gives you	360
27.1 Logical rows and physical pages	360
27.2 Buffer pools	361
27.3 Row layout	362
27.4 Dirty pages and flushing	364
27.5 Free space and fragmentation	365
27.6 Reading engine-specific evidence	366
27.97 Practice and worked answers	367
27.98 Common wrong ideas	368
27.99 Chapter summary in 20 lines	368
 28 Write-Ahead Logs and Recovery	 369
28.0 What this chapter gives you	369
28.1 Why recovery information exists	369
28.2 Log before data	370
28.3 Log sequence positions	371
28.4 Checkpoints	373
28.5 Redo and rollback concepts	374
28.6 Crash-recovery traces	375
28.97 Practice and worked answers	376
28.98 Common wrong ideas	377
28.99 Chapter summary in 20 lines	377
 29 LSM Trees and Compaction	 379
29.0 What this chapter gives you	379
29.1 Sequential writes	379
29.2 Memory tables and immutable files	380
29.3 Sorted runs	381
29.4 Read amplification	383

29.5 Compaction and tombstones	384
29.6 Workload trade-offs	385
29.97 Practice and worked answers	386
29.98 Common wrong ideas	387
29.99 Chapter summary in 20 lines	387
30 Durability From Application to Device	388
30.0 What this chapter gives you	388
30.1 The acknowledgement chain	388
30.2 Process memory and operating-system caches	389
30.3 Flush requests	391
30.4 Device promises	392
30.5 Failure domains	393
30.6 Testing a durability claim	394
30.97 Practice and worked answers	396
30.98 Common wrong ideas	396
30.99 Chapter summary in 20 lines	396
31 Backups, Restore and Point-in-Time Recovery	398
31.0 What this chapter gives you	398
31.1 Copies with a recovery purpose	398
31.2 Consistent snapshots	399
31.3 Log retention	401
31.4 Recovery objectives	402
31.5 Restore rehearsals	403
31.6 Access and retention of backups	404
31.97 Practice and worked answers	405
31.98 Common wrong ideas	406
31.99 Chapter summary in 20 lines	406
32 Corruption, Reconciliation and Repair	408
32.0 What this chapter gives you	408
32.1 Detecting disagreement	408
32.2 Checksums and their limits	409
32.3 Structural checks	410
32.4 Business reconciliation	412
32.5 Evidence-preserving repair	413
32.6 Incident follow-through	414
32.97 Practice and worked answers	415
32.98 Common wrong ideas	416
32.99 Chapter summary in 20 lines	416
Part F — Data Across Machines	417

33 Replication: More Than One Copy	418
33.0 What this chapter gives you	418
33.1 Why copies exist	418
33.2 Physical and logical approaches	419
33.3 Synchronous and asynchronous paths	421
33.4 Lag and staleness	422
33.5 Read routing	423
33.6 Replication is not a restore plan	424
33.97 Practice and worked answers	426
33.98 Common wrong ideas	426
33.99 Chapter summary in 20 lines	427
34 Leaders, Followers and Failover	428
34.0 What this chapter gives you	428
34.1 Write authority	428
34.2 Failure suspicion	429
34.3 Election and promotion	431
34.4 Fencing stale writers	432
34.5 Lost acknowledgements	433
34.6 Safe return of a former leader	434
34.97 Practice and worked answers	436
34.98 Common wrong ideas	436
34.99 Chapter summary in 20 lines	436
35 Partitioning: Dividing the Work	438
35.0 What this chapter gives you	438
35.1 Partition keys	438
35.2 Range and hash approaches	439
35.3 Hot partitions	440
35.4 Rebalancing	442
35.5 Cross-partition questions	443
35.6 Growth without permanent shortcuts	444
35.97 Practice and worked answers	445
35.98 Common wrong ideas	446
35.99 Chapter summary in 20 lines	446
36 Consistency, Availability and Network Partitions	448
36.0 What this chapter gives you	448
36.1 Name the consistency model	448
36.2 What availability promises	449
36.3 Partitioned communication	451
36.4 CAP with precise definitions	452
36.5 Quorum reasoning and limits	453
36.6 Application-level consequences	454

36.97 Practice and worked answers	456
36.98 Common wrong ideas	456
36.99 Chapter summary in 20 lines	456
37 Consensus: Agreeing Despite Failure	458
37.0 What this chapter gives you	458
37.1 The agreement problem	458
37.2 Logs and terms	459
37.3 Majorities	461
37.4 Leader changes	462
37.5 Safety and progress	463
37.6 Reading a consensus trace	464
37.97 Practice and worked answers	465
37.98 Common wrong ideas	466
37.99 Chapter summary in 20 lines	466
38 Distributed Transactions and Sagas	468
38.0 What this chapter gives you	468
38.1 One transaction across participants	468
38.2 Prepare and commit	469
38.3 In-doubt work	470
38.4 Compensation	472
38.5 Saga state and retries	473
38.6 Choosing an explicit failure policy	474
38.97 Practice and worked answers	475
38.98 Common wrong ideas	476
38.99 Chapter summary in 20 lines	476
39 Caches, Queues and What They Do Not Guarantee	478
39.0 What this chapter gives you	478
39.1 Derived copies	478
39.2 Invalidation and expiry	479
39.3 Queue acknowledgement	481
39.4 Redelivery and order	482
39.5 Backpressure	483
39.6 End-to-end correctness	484
39.97 Practice and worked answers	485
39.98 Common wrong ideas	486
39.99 Chapter summary in 20 lines	486
Part G — From Records to Insight	488

40	Data Pipelines: Capture, Transform and Deliver	489
40.0	What this chapter gives you	489
40.1	Sources and contracts	489
40.2	Batch extraction	490
40.3	Change capture	492
40.4	Transformations	493
40.5	Delivery and checkpoints	494
40.6	Reconciliation and replay	496
40.97	Practice and worked answers	497
40.98	Common wrong ideas	497
40.99	Chapter summary in 20 lines	498
41	Streams, Event Time and Late Arrivals	499
41.0	What this chapter gives you	499
41.1	Continuous records	499
41.2	Event and processing time	500
41.3	Windows	501
41.4	Watermarks	503
41.5	Late corrections	504
41.6	State, checkpoints and replay	505
41.97	Practice and worked answers	506
41.98	Common wrong ideas	507
41.99	Chapter summary in 20 lines	507
42	Warehouses, Lakes and Analytical Tables	509
42.0	What this chapter gives you	509
42.1	Operational versus analytical work	509
42.2	Warehouse modelling	510
42.3	Files and catalogues	512
42.4	Table snapshots	513
42.5	Governance boundaries	514
42.6	Selecting a design by workload	515
42.97	Practice and worked answers	517
42.98	Common wrong ideas	517
42.99	Chapter summary in 20 lines	518
43	Columnar Storage and Compression	519
43.0	What this chapter gives you	519
43.1	Rows versus columns	519
43.2	Encodings	520
43.3	Compression blocks	522
43.4	Predicate pushdown	523
43.5	Statistics and skipping	524
43.6	CPU, memory and input-output trade-offs	525

43.97 Practice and worked answers	527
43.98 Common wrong ideas	527
43.99 Chapter summary in 20 lines	527
44 Full-Text Search and Relevance	529
44.0 What this chapter gives you	529
44.1 Words and documents	529
44.2 Tokenisation	530
44.3 Inverted indexes	531
44.4 Scoring and ranking	533
44.5 Language and spelling	534
44.6 Evaluating search quality	535
44.97 Practice and worked answers	536
44.98 Common wrong ideas	537
44.99 Chapter summary in 20 lines	537
45 Vector Search: Similarity Is Not Truth	539
45.0 What this chapter gives you	539
45.1 Representing items as vectors	539
45.2 Distance and similarity	540
45.3 Exact and approximate search	542
45.4 Index trade-offs	543
45.5 Filtering and evaluation	544
45.6 Retrieval limits	545
45.97 Practice and worked answers	547
45.98 Common wrong ideas	547
45.99 Chapter summary in 20 lines	547
46 Metrics, Bias and Honest Analysis	549
46.0 What this chapter gives you	549
46.1 A metric is a definition	549
46.2 Denominators and populations	550
46.3 Missingness	551
46.4 Selection effects	553
46.5 Association versus cause	554
46.6 Communicating uncertainty	555
46.97 Practice and worked answers	556
46.98 Common wrong ideas	557
46.99 Chapter summary in 20 lines	557
Part H — Running Data Systems	559

47	Permissions, Secrets and Tenant Boundaries	560
47.0	What this chapter gives you	560
47.1	Who may do what	560
47.2	Least privilege	562
47.3	Database and application checks	563
47.4	Tenant separation	564
47.5	Secret handling	566
47.6	Adversarial boundary tests	567
47.97	Practice and worked answers	568
47.98	Common wrong ideas	569
47.99	Chapter summary in 20 lines	569
48	Retention, Deletion and Data Lineage	571
48.0	What this chapter gives you	571
48.1	Purpose and record classes	571
48.2	Retention policies	573
48.3	Deletion across copies	574
48.4	Derived data and lineage	575
48.5	Backups and exceptions	576
48.6	Evidence of completion and remaining limits	578
48.97	Practice and worked answers	579
48.98	Common wrong ideas	580
48.99	Chapter summary in 20 lines	580
49	Schema Changes Without Surprises	581
49.0	What this chapter gives you	581
49.1	Compatibility windows	581
49.2	Expand and contract	582
49.3	Backfills	584
49.4	Validation and cutover	585
49.5	Rollback versus roll-forward	586
49.6	Staged evidence	588
49.97	Practice and worked answers	589
49.98	Common wrong ideas	589
49.99	Chapter summary in 20 lines	590
50	Observability, Capacity and Cost	591
50.0	What this chapter gives you	591
50.1	Signals tied to user outcomes	591
50.2	Logs, metrics and traces	592
50.3	Storage and growth	594
50.4	Saturation and headroom	595
50.5	Cost attribution	596
50.6	Operational reviews	598

50.97 Practice and worked answers	599
50.98 Common wrong ideas	599
50.99 Chapter summary in 20 lines	600
51 Capstone: From One Shop to a Multi-Branch Service	601
51.0 What this chapter gives you	601
51.1 Requirements and boundaries	601
51.2 Schema and workflows	603
51.3 Correctness under retries	604
51.4 Reporting and reconciliation	606
51.5 Recovery and access tests	607
51.6 Review and release evidence	609
51.97 Practice and worked answers	610
51.98 Common wrong ideas	610
51.99 Chapter summary in 20 lines	611
52 The Reference Desk: Glossary and Diagnostic Guides	612
52.0 What this chapter gives you	612
52.1 Plain-to-technical glossary	612
52.2 SQL and data-type reference	613
52.3 Diagnostic decision guides	615
52.4 Design-review questions	617
52.5 Test and recovery templates	619
52.6 Source and version register	620
52.97 Practice and worked answers	622
52.98 Common wrong ideas	622
52.99 Chapter summary in 20 lines	623
Companion Labs	624
A–Z Glossary	627
Sources and Version Register	740



PART A

Meaning Before Machines

Learn what a record says, what it leaves out, and what a system can safely conclude from it.

The Whole Data System in One Look

1.0 What this chapter gives you

By the end of this chapter, you should be able to follow one sale from a person's action to a stored record and then to a report. You should also be able to explain why a correct-looking screen is not enough evidence that the whole operation succeeded.

You will distinguish a record from the event it describes, a database from the program that manages it, and an original record from an answer calculated from it. You will learn where the word "saved" needs a more precise question. No programming installation is needed to read the chapter.

■ The shop we will grow with

Mira runs a fictional stationery shop called Mira's Corner. Dev helps at the counter. At first they can remember most of the day's work. Later they will need several tills, an online shop, deliveries and reports. We will add machinery only when a specific problem creates a reason for it.

All people, orders, stock counts, identifiers and amounts in this case study are invented teaching data. They are not observations of a real KedByte customer or a live commercial system.

Our first order is **O-1042**. A customer buys two notebooks at INR 75.50 each and one pen at INR 20.00. The total is INR 171.00. For now there are no taxes, discounts, refunds or delivery charges in the calculation. Those omissions are deliberate boundaries, not a suggestion that a real checkout can ignore them.

1.1 One sale, many responsibilities

■ 1.1.1 PLAIN — in simple words

1. Selling an item and recording the sale are different actions. A notebook can leave the shelf without the computer learning about it. A computer can also contain a sale that never actually happened. A useful data system connects the work people do with records that other people can interpret and check.
2. For our order, Mira needs answers to several separate questions: What was requested? What price was agreed? What was handed over? What stock should remain? A later payment record may answer another question, but the presence of an order alone does not answer it.
3. The first job is therefore not "put everything in a database." It is to decide which facts the shop needs, what each fact means, and how an action will produce the right record.

■ 1.1.2 PLAIN — a picture in your head

1. Imagine a counter with four trays. One holds requests, one holds checked order slips, one holds stock movements and one holds reports. Moving a slip into the order tray does not magically move a physical notebook or update every other tray.
2. The trays make responsibilities visible. They do not prescribe four separate databases. **Where the comparison stops:** software may store several kinds of records together and change some of them in one controlled operation. The important separation is meaning, not furniture.

■ 1.1.3 PLAIN — a worked example

Order line	Item	Quantity	Unit price	Line amount
1	Notebook	2	INR 75.50	INR 151.00
2	Pen	1	INR 20.00	INR 20.00
Total	Two different items	3 units	Not applicable	INR 171.00

1. First multiply 2 by 75.50 to obtain 151.00. Then multiply 1 by 20.00 to obtain 20.00. Add the two line amounts: $151.00 + 20.00 = 171.00$.
2. If the notebook count starts at 12 and both notebooks are handed over, the expected count becomes 10. If the pen count starts at 25 and one pen is handed over, it becomes 24. These are consequences of the stated example, not claims that every inventory system changes stock at the same business moment.
3. Notice three different counts: one order, two order lines and three physical units. They are all correct. They answer different questions.

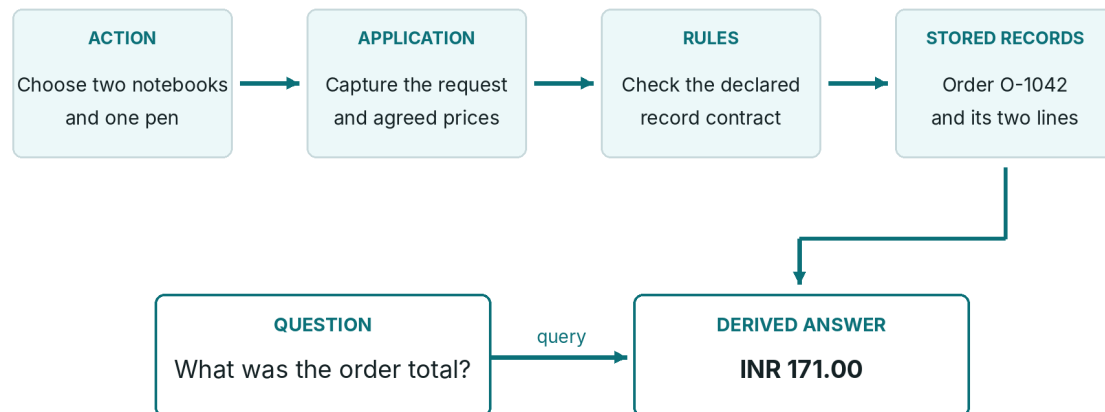
■ 1.1.4 PLAIN — what is really happening inside

1. Our proposed application receives the chosen item identifiers and quantities. It checks them against the shop's rules, determines the agreed prices, and prepares the records that should result. A storage component accepts or rejects the changes it is asked to make. The application then presents an outcome to Dev.
2. Each arrow can introduce a mistake. The wrong item can be selected. A correct item can be paired with an outdated price. A response can disappear after the records have changed. A report can add the wrong field even when the underlying order is correct.
3. A system diagram is valuable because it gives those possible mistakes a location. "The computer is wrong" becomes "the price selected before saving was wrong" or "the report counted lines rather than orders." Those are problems somebody can investigate.

■ 1.1.5 TECHNICAL — the engineer's version

1. We will call the screen and workflow code the **application**, the accepted stored facts the **database**, and the software responsible for database operations the **database management system**, or DBMS. An application can use an embedded engine inside its process or communicate with a server. Python's SQLite interface is an example of access to an embedded SQL engine. [S20]

A small system, with separate responsibilities



A logical teaching model — not a network or deployment diagram.

Figure 1.1: Figure 1.1. The teaching system: actions become records; questions produce derived answers. The diagram is a logical model, not a deployment design.

2. The toy order has two related structures. An order header identifies the overall order. Its lines identify the individual pricing decisions within that order. The tuple (order_id, line_no) will identify a line in our examples; the same product could legitimately appear on two different lines.
3. Our initial invariant is that the order total equals the sum of quantity multiplied by agreed unit price across its lines. An **invariant** is a rule we intend every accepted state to satisfy. Naming the rule does not enforce it. Later chapters will examine how to enforce it, including when requests overlap.
4. The boundary matters: a database transaction can group its participating database changes. It does not, merely by existing, turn handing over goods, sending a message and charging an external service into the same atomic action. [S01]

■ 1.1.6 WORDS — remember these

1. **Application:** the program people use; technically, the software that implements a workflow and requests operations from other components.
2. **Database:** the organised records; technically, a managed collection of data interpreted under a model and its rules.
3. **DBMS:** the record-management machinery; technically, the engine providing database operations and its documented guarantees.
4. **Invariant:** a rule that must keep holding; technically, a predicate required of accepted states or permitted transitions.

1.2 A record is not the world

■ 1.2.1 PLAIN — in simple words

1. A record is a statement we have preserved. It may describe an observation, a request, an estimate, a plan or a decision. It is not automatically proof that the described event occurred.
2. If Dev types “12 notebooks,” the system can preserve that statement perfectly while there are actually 11 notebooks on the shelf. Perfect storage and accurate observation are different achievements.
3. This is not a reason to distrust every record. It is a reason to ask what evidence supports the particular claim you intend to make from it.

■ 1.2.2 PLAIN — a picture in your head

1. A photograph of a shelf is not the shelf. It captures one view at one time. It may hide an item behind another, and it cannot show what happened after the photograph was taken.
2. A stock record has a similar boundary. **Where the comparison stops:** records can describe things no photograph captures well, such as a reservation or an intended delivery. Some records create an administrative state rather than merely observe a physical scene.

■ 1.2.3 PLAIN — a worked example

1. At 09:00 Dev counts 12 notebooks and records the count. At 09:30 two notebooks are sold under O-1042. At 10:00 a new count finds only nine.
2. The expected count is $12 - 2 = 10$. The observed count is nine. The discrepancy is one notebook. The arithmetic identifies a disagreement; it does not identify the cause.
3. Possible explanations include a counting mistake, an unrecorded sale, damage or an incorrectly recorded starting count. In the story we do not yet know which explanation is true. Changing the computer’s count to nine may reconcile the current number, but it does not establish what happened.

■ 1.2.4 PLAIN — what is really happening inside

1. The system receives values through a boundary: a form, a scanner, a file or another program. It can check some properties of those values. For example, it can require a count to be a non-negative whole number. That rule rejects “twelve-ish” but still accepts an incorrect count of 12.
2. We therefore separate **validation** from **verification against evidence**. Validation asks whether a record obeys declared rules. Verification asks whether the claim is supported by an appropriate observation or other evidence. Our usage is practical; a particular discipline may use these terms more formally.
3. W3C’s data-publication guidance treats quality information and provenance as things that should accompany data, not as properties readers should infer merely from a file being available. [S06]

■ 1.2.5 TECHNICAL — the engineer’s version

1. For the shop, distinguish three predicates. `well_formed(record)` means the record can be interpreted under the format. `valid(record)` means it satisfies the declared domain rules. `supporte`

`d(record, evidence)` means the available evidence supports a stated claim. These are teaching predicates, not functions supplied automatically by a database.

2. An integer stock count may satisfy a type rule and a range rule while failing the third predicate. Conversely, a physically observed count can arrive in a malformed file and fail the first predicate. Different defects need different remedies.
3. In PostgreSQL, a check constraint can enforce an expression over the row being checked. That mechanism does not independently observe a shelf. The difference between an enforceable row rule and an external fact is essential to the design review. [S02]
4. A useful discrepancy record would identify the affected item, the expected quantity, the observed quantity, the relevant time, the source of the observation and the disposition. Recording “investigation pending” is more accurate than inventing a cause to make the record look complete.

■ 1.2.6 WORDS — remember these

1. **Observation:** something someone or something noticed; technically, an acquisition of information under a stated method and context.
2. **Validation:** checking the stated rules; here, testing whether a representation meets declared structural or domain requirements.
3. **Evidence:** what supports a claim; technically, information whose relevance and limitations must be assessed for that claim.
4. **Reconciliation:** comparing related accounts of the same work; technically, identifying and resolving or explicitly retaining discrepancies under a defined procedure.

1.3 Files, databases and the software between them

■ 1.3.1 PLAIN — in simple words

1. A file is a place where a program can keep a sequence of bytes. A spreadsheet file can hold a useful small dataset. A database engine provides a different level of service: programs ask it to manage records according to its rules rather than each inventing their own way to edit the underlying storage.
2. The distinction is not “files are bad, databases are good.” A database itself uses storage representations. The question is who coordinates the work and which guarantees that component supplies.
3. For Mira alone, a carefully maintained file may be enough. Once several tills change the same stock at once, the coordination requirement becomes much more demanding.

■ 1.3.2 PLAIN — a picture in your head

1. A shared notebook lets people read and write. A record office adds an agreed process for accepting forms, identifying records, answering requests and controlling changes. The office represents the service provided around the records.
2. **Where the comparison stops:** a database engine is not a clerk who understands your intentions. It applies the rules and operations you have actually defined. It can execute a mistaken instruction correctly.

■ 1.3.3 PLAIN — a worked example

1. Suppose two tills both read a file saying there are 10 notebooks. Till A sells two and prepares a replacement count of eight. Till B sells one and prepares a replacement count of nine. If A saves eight and B later overwrites it with nine, the final file says nine. The expected count is seven.
2. The failure is not that subtraction stopped working. Both tills subtracted from the same old starting point and one result replaced the other. Putting the same unprotected pattern behind a database connection is not, on its own, a complete correction.
3. We will later write the operation as a controlled change to the current value, with conditions and a concurrency policy. For now, recognising the overlapping sequence is more important than memorising a particular command.

■ 1.3.4 PLAIN — what is really happening inside

1. A request to a database engine is expressed through an interface. The engine interprets the request, checks applicable rules, locates or changes records, and returns a result or an error. The physical arrangement can differ substantially between engines.
2. A CSV file describes a tabular exchange format, not a shared-editing protocol. Its commas and quotation rules do not specify how two tills coordinate updates or how an interrupted operation recovers. RFC 4180 documents common CSV conventions as an informational RFC. [S08]
3. Similarly, JSON provides a way to represent structured values. A JSON object containing an order does not by itself specify authority, concurrency, retention or whether an amount has been paid. Those are contracts above the representation. [S07]

■ 1.3.5 TECHNICAL — the engineer's version

1. Separate four layers in a design: the **data model**, the **operations**, the **engine** and the **storage format**. The model says what entities and relationships mean. Operations specify the permitted questions and changes. The engine implements those operations. The format represents data at a storage or exchange boundary.
2. SQL means **Structured Query Language**. It is a language used by many database engines, not the name of one engine. Product-specific behaviour still matters. SQLite, for example, uses dynamic typing in ordinary tables, and its documented type-affinity rules differ from the rigid assumptions readers sometimes bring from other products. [S21]
3. The foundations laboratory uses Python and an in-memory SQLite database. It demonstrates stated examples only. PostgreSQL references explain PostgreSQL behaviour; running the SQLite laboratory is not a PostgreSQL compatibility test.
4. That separation is a habit worth developing early: name the product, the relevant configuration, the operation and the observed result before generalising.

■ 1.3.6 WORDS — remember these

1. **File format**: how bytes are arranged; technically, the syntax and interpretation rules for a representation.
2. **Interface**: how one component asks another to do something; technically, the exposed operations and their input, output and error contracts.

3. **SQL:** a language for working with data; technically, Structured Query Language and its standardised and product-specific features.
4. **Concurrency:** work that overlaps in time; technically, the execution of operations whose interleaving can affect their observed results.

1.4 Questions produce answers, not new certainty

■ 1.4.1 PLAIN — in simple words

1. A report is an answer calculated from records under a definition. It can be wrong even when every source record was stored correctly. The mistake may be the question, the selection of records or the calculation.
2. “How busy was the shop?” is not yet a precise question. It could mean the number of orders, the number of people served, the number of items sold or the total recorded order value. A single large order makes those measures behave differently.
3. The useful habit is to finish the sentence: “This number counts or measures , *from* , during , *excluding* .”

■ 1.4.2 PLAIN — a picture in your head

1. Think of a recipe made from measured ingredients. The ingredients can be correct while the recipe is wrong for the dish you intended. A spoonful of sugar and a spoonful of salt are both real measurements; swapping their role ruins the result.
2. **Where the comparison stops:** a data calculation often leaves its inputs unchanged. Several different reports can legitimately be made from the same records. The issue is whether each one matches its stated definition.

■ 1.4.3 PLAIN — a worked example

1. Our second fictional order, O-1043, contains one notebook at INR 75.50 and two pens at INR 20.00 each. Its total is $75.50 + 40.00 = \text{INR } 115.50$.
2. Across O-1042 and O-1043, we have two orders, four order lines and six physical units. Their total recorded order value is $171.00 + 115.50 = \text{INR } 286.50$. The average recorded value per order is $286.50 / 2 = \text{INR } 143.25$.
3. Dividing by four would produce INR 71.625 per line, which is a different measure. None of these calculations establishes recognised revenue, settled receipts or profit. We have not introduced the records or definitions needed for those claims.

■ 1.4.4 PLAIN — what is really happening inside

1. A report chooses records, combines related records when necessary, performs calculations and formats the result. Each step needs a defined meaning. “Today” requires a time boundary. “Completed” requires a business status. “Customer” requires an identity policy.
2. A dashboard may store a previously calculated answer so it can show it quickly. In that case the displayed value also has an “as of” boundary. The presence of a current-looking screen does not tell you when its underlying calculation was last refreshed.

3. For our teaching system, every derived report will have a name, an input definition, an inclusion rule and a calculation. Where a value is cached or refreshed periodically, the report will also state that timing policy.

■ 1.4.5 TECHNICAL — the engineer's version

1. A **query** is a specified request for data or a calculation over it. A **projection**, in the broader application-design sense used here, is a derived representation built for a particular use. Do not confuse that usage with relational algebra's narrower projection operator, which selects attributes.
2. The cardinality of an input is its number of rows. A join can change that cardinality. A line-level dataset is not an order-level dataset simply because every line carries an order identifier. The case study's four rows require two distinct order identifiers to answer the order-count question.
3. Ordering is also part of the contract. PostgreSQL does not promise a particular output order unless sorting is requested. A query that happens to return O-1042 before O-1043 in a small demonstration does not establish a permanent ordering rule. [S03]
4. A report specification should distinguish its logical definition from the plan used to execute it. We will defer optimisation until the question itself is stable.

■ 1.4.6 WORDS — remember these

1. **Query:** a precise question; technically, an operation expressing a requested result over data.
2. **Derived data:** an answer made from other data; technically, a representation produced by a transformation or calculation over inputs.
3. **Grain:** what one record stands for; technically, the observation or business unit represented by each row.
4. **Cardinality:** how many; here, the number of rows in a dataset or intermediate result.

1.5 Changes, failures and the word "saved"

■ 1.5.1 PLAIN — in simple words

1. "Saved" can mean several things. The screen may have accepted your typing. The application may have sent a request. The database may have accepted the change. The information may also have reached storage expected to survive a particular failure.
2. Those are different milestones. A useful system tells the user which outcome it knows rather than using one reassuring word for all of them.
3. A timeout is especially important. It means an expected response did not arrive within the allowed time. From that fact alone, the caller cannot always know whether the change happened.

■ 1.5.2 PLAIN — a picture in your head

1. You post a signed form to an office. The office records it, then sends you a receipt. The receipt is lost on the way back. You have no receipt, but the office may already have accepted the form.

2. **Where the comparison stops:** software can attach a stable request identifier and expose a status lookup. A well-designed retry protocol can use those records to avoid treating every resend as a brand-new request. The analogy describes uncertainty, not an unavoidable absence of solutions.

■ 1.5.3 PLAIN — a worked example

1. Imagine a controlled database operation that creates O-1042 and records its associated stock reduction. If an error interrupts the operation between those changes, our intended outcome is either that all participating changes are accepted or that none are.
2. Now consider a different sequence: the database accepts the complete operation, but the network response to Dev is lost. Dev sees a timeout. Retrying as a completely new order could record the sale twice. Immediately claiming that no order exists could also be wrong.
3. The next safe question is about the original operation's identity and status. A useful request identifier makes that question possible; its existence alone still does not prove that duplicate handling was implemented correctly.

■ 1.5.4 PLAIN — what is really happening inside

1. A transaction groups participating database changes into a controlled unit. In the model used by PostgreSQL's transaction tutorial, a successful commit accepts the unit, while rollback cancels the transaction's changes. [S01]
2. That solves a specific problem. It does not by itself explain when a response reaches a screen, how a notification is delivered, or how an already-completed transaction should be reversed as a business matter.
3. Storage promises also have conditions. A database may allow a mode that acknowledges a transaction before the corresponding recovery information has been durably written. PostgreSQL documents that an asynchronous commit can therefore be lost after a crash even though it was acknowledged. [S04]

■ 1.5.5 TECHNICAL — the engineer's version

1. Keep four questions separate. **Atomicity:** which changes take effect together? **Isolation:** how can overlapping operations observe or affect one another? **Durability:** which accepted changes survive which failures under which configuration? **Knowledge at the caller:** what has the caller actually learned from the messages it received?
2. These questions interact, but none is a replacement for the others. A local rollback demonstration is evidence about that demonstration. It is not proof of power-loss durability, multi-machine failover, or correctness under concurrent requests.
3. Our laboratory deliberately creates a table in memory, makes a temporary change and rolls it back. It verifies that the demonstrated rollback restores the queried row count. Because the database is in memory, the laboratory makes no claim that its data survives process exit or a machine failure.
4. Later chapters will add explicit request identity, bounded retries and external-effect handling. The important lesson now is to put the guarantee next to its boundary.

■ 1.5.6 WORDS — remember these

1. **Transaction:** a controlled unit of database work; technically, operations managed together under an engine's atomicity and related semantics.
2. **Commit:** accepting the transaction; technically, the transaction-completion operation under the engine's documented configuration.
3. **Rollback:** cancelling uncommitted work; technically, discarding the transaction changes within the requested rollback scope.
4. **Durability:** surviving specified failures; technically, the persistence guarantee for accepted changes under stated assumptions.

1.6 The map of the whole book

■ 1.6.1 PLAIN — in simple words

1. The rest of the book expands this one shop without treating scale as magic. We first learn what records mean. Then we organise them, ask questions, control changes, survive failures and distribute work. Finally we build useful analysis and operate the system responsibly.
2. This order matters. Making an ambiguous record travel faster does not make it clearer. Copying an incorrect calculation to more machines does not make it more correct.

■ 1.6.2 PLAIN — a picture in your head

1. Think of learning to run a workshop. Before buying faster machines, you learn what you are making, how parts fit, how to measure the result and what makes it safe to use.
2. **Where the comparison stops:** data systems can be reorganised without moving physical walls, but changes still have compatibility and recovery costs. "Software can change" does not mean "software can change without consequences."

■ 1.6.3 PLAIN — a worked example

1. For O-1042, the early chapters ask whether the quantity means units or boxes. The database chapters ask where the order and its lines belong. The query chapters ask how to find the order quickly. The transaction chapters ask what happens when a request repeats.
2. The recovery chapters ask how the order can be restored after a failure. The distributed chapters ask which copy a reader is seeing. The analytical chapters ask what the order contributes to a report. The operational chapters ask who may inspect it and what happens when it is no longer needed.
3. It is the same order throughout. New machinery earns its place by answering a new question.

■ 1.6.4 PLAIN — what is really happening inside

1. Each major section will use the same six reading blocks. First comes the idea in ordinary words. Then comes a mental picture, a worked example and a closer account of the mechanism. The technical block adds formal names, implementation details and limits. The final block connects new terms to the explanation you have already read.

2. You can read the plain blocks first and return for the technical blocks. You should not have to invent missing definitions to make that route work.

■ 1.6.5 TECHNICAL — the engineer's version

1. The planned sequence has eight parts and 52 chapters. Parts A and B establish semantics and relational foundations. Parts C, D and E develop query execution, concurrency and storage. Parts F, G and H address distribution, analysis and operations.
2. Claims will be labelled by their status: a definition used in the book, a rule selected for the fictional case study, a documented product behaviour, or an observed result from a specified experiment. Those categories are not interchangeable.
3. This complete edition contains all fifty-two chapters. The companion-lab guide distinguishes executable exercises from conceptual discussions. A chapter's presence does not mean that every system it describes has been implemented, simulated or independently reviewed. Consult the recorded test scope before making a stronger claim.

■ 1.6.6 WORDS — remember these

1. **Model:** a deliberately limited representation; technically, a set of concepts and rules chosen to answer particular questions.
2. **Contract:** what participants can rely on; technically, specified inputs, outputs, invariants, errors and relevant guarantees.
3. **Implementation detail:** how a particular system works; technically, behaviour that must not be assumed to apply to every conforming or competing system.
4. **Reproducible example:** an example another reader can check; technically, one supplied with sufficient inputs, procedure and environment information to repeat its stated observation.

1.97 Check your understanding

Question 1. A report shows four orders because it counted the four lines in O-1042 and O-1043. Which definition was lost?

Answer. It lost the row grain. Each row represented an order line, not a complete order. There are two distinct orders, four lines and six units in the example.

Question 2. A stock count passes a non-negative-integer rule. Has the number of notebooks on the shelf been verified?

Answer. No. The representation passed the stated rule. A suitable observation or reconciliation is still needed to support the physical claim.

Question 3. Dev sees a timeout after sending O-1042. What can he conclude?

Answer. He did not receive the expected response in time. The example does not give enough evidence to conclude either that the order was accepted or that it was not. He needs an outcome check tied to the original operation.

Question 4. The laboratory demonstrates rollback in memory. Does that demonstrate crash durability?

Answer. No. The experiment tests the stated rollback result in its own environment. It does not exercise durable storage or a power failure.

1.98 Common wrong ideas

“A stored record must be true.” Storage preserves representations. It does not independently establish their correspondence with the world.

“A database removes the need to design rules.” The engine enforces the rules and operations actually supplied, subject to its documented behaviour.

“A correct total means the whole process is correct.” A total can be arithmetically correct while using the wrong records, units or business definition.

“A timeout means nothing happened.” The response can be lost after an operation succeeds.

“A successful toy test proves production readiness.” A test supplies evidence only for the claim, inputs, environment and failure conditions it actually exercised.

1.99 Chapter summary in 20 lines

1. A data system connects actions, records, rules and questions.
2. O-1042 contains two notebooks and one pen.
3. Its recorded order value is INR 171.00 under our stated assumptions.
4. One order, two lines and three units are different counts.
5. A record is a preserved statement, not automatic proof of an event.
6. Valid structure and accurate observation are different properties.
7. A discrepancy identifies disagreement before it identifies a cause.
8. An application implements a workflow and requests data operations.
9. A DBMS manages database operations under documented rules.
10. A format describes a representation, not an entire operational contract.
11. Overlapping changes need a concurrency policy.
12. A report needs an explicit definition of what it measures.
13. Derived answers inherit the assumptions and limits of their inputs.
14. Output order should be requested when it matters.
15. The word “saved” hides several possible milestones.
16. A timeout alone may leave the outcome unknown.
17. A transaction groups participating database changes.
18. Durability claims require stated failure and configuration boundaries.
19. An experiment demonstrates only what it actually exercises.
20. The next chapter makes the meaning of each record explicit.

What a Record Really Means

2.0 What this chapter gives you

This chapter teaches you to look at a record and ask the questions that a screen full of neat columns can hide. What is being claimed? What does one row stand for? Which thing does an identifier refer to? Who supplied the information, and which moment does its time describe?

You will turn an ambiguous line of text into an explicit teaching record. You will also learn why keeping more fields is not the same as keeping better evidence, and why a correction needs a meaning as well as a new value.

The running example remains Mira's Corner. O-1042 is the order for two notebooks and one pen, totalling INR 171.00 under the simplified pricing rules established in Chapter 1. We will not quietly change those assumptions halfway through an example.

2.1 The claim inside a record

■ 2.1.1 PLAIN — in simple words

1. Consider this line: 1042, notebook, 2, 75.50, 09:30. It looks informative, but important parts of its meaning are missing. Is 1042 an order, a customer or a delivery? Does two mean two notebooks or two boxes? Is 75.50 the price of one item or the amount for the whole line? Which day does 09:30 belong to?
2. A record becomes useful when the reader can recover the intended claim without guessing. A useful sentence might be: "Order O-1042, line 1, records an agreed quantity of two individual notebooks at INR 75.50 per notebook."
3. That sentence is already doing design work. It separates the order, the line, the item, the quantity, the unit and the price basis.

■ 2.1.2 PLAIN — a picture in your head

1. Imagine receiving a parcel with only the number "2" on its label. It might mean two items, a second attempt, a shelf number or a delivery priority. A better label does not change the parcel; it tells the next person how to treat it.
2. **Where the comparison stops:** a data field's meaning can depend on rules stored somewhere else, such as a schema or data dictionary. You do not need to repeat a long explanation in every row, but the explanation must be available and connected to the row's version.

■ 2.1.3 PLAIN — a worked example

1. We can make the first line of O-1042 explicit:

Field	Example value	Meaning in this book
order_id	O-1042	The shop's identifier for this order
line_no	1	This line's number within the order
product_id	P-NOTE	The notebook product in our catalogue
quantity	2	Two individual saleable units
unit_price_minor	7550	Agreed price per unit in paise
currency	INR	The currency attached to the amount

- For this case study, one rupee is represented by 100 paise. The line amount is $2 \times 7550 = 15100$ paise, displayed as INR 151.00. Storing 7550 without stating "paise per unit" would not preserve the full claim.
- The record says what was agreed for the order line. It does not, on its own, say that two notebooks were handed over or that a payment completed. Those require their own definitions and evidence.

■ 2.1.4 PLAIN — what is really happening inside

- The producer and consumer of a record share a contract. That contract says how to read the fields, which values are allowed and what business meaning the combination has.
- A field name can help, but names are not a complete contract. `amount` is vague. `unit_price_min` or is better, but it still needs a currency, a definition of the minor unit and a rule about when the price was agreed.
- The data dictionary is where we gather these meanings. W3C's data guidance distinguishes structural metadata from descriptive information that helps people understand and use data. In our book, the dictionary supplies both where appropriate. [S06]

■ 2.1.5 TECHNICAL — the engineer's version

- Model the example as a predicate: `AgreedOrderLine(order_id, line_no, product_id, quantity, unit_price_minor, currency)`. Read that as a statement with named arguments, not as runnable code.
- The predicate's interpretation contains facts the type system alone cannot express. For example, `quantity` counts saleable individual units in this exercise. A later product sold by mass would need a different quantity-and-unit policy; reusing the integer field without revisiting its meaning would be a modelling error.
- Separate the **syntactic contract** from the **semantic contract**. Syntax tells a parser where a field begins and ends. Semantics tells an application what the value means and which operations are meaningful. A JSON number can parse correctly while being interpreted in the wrong unit. [S07]
- For each field, document its domain, unit or scale, missing-value policy, source, mutability and version boundary. These are proposed design requirements for our case study, not properties acquired merely by choosing JSON or SQL.

■ 2.1.6 WORDS — remember these

- Field:** one named part of a record; technically, an attribute position with an agreed interpretation.

2. **Domain:** the allowed kind of value; technically, the set of permitted values and associated meaning for an attribute.
3. **Syntax:** the shape of a representation; technically, the grammar used to recognise valid constructions.
4. **Semantics:** what a representation means; technically, the interpretation and permitted consequences assigned to it.

2.2 Fields, values and the contract around them

■ 2.2.1 PLAIN — in simple words

1. A label and a value travel together, but they are not the whole story. The same text can mean different things under different rules. A date written as 03/04/2026 may be read as 3 April or 4 March. Neither reader has necessarily made an arithmetic mistake; they may be using different conventions.
2. A safe exchange tells the receiver which convention applies. When that information is missing, “pick the interpretation that looks likely” is not a reliable general rule.
3. The goal is to make the contract explicit before a guess turns into a stored fact.

■ 2.2.2 PLAIN — a picture in your head

1. Picture a board game with pieces but no instructions. You can see the pieces clearly and still have no idea whether moving diagonally is allowed. A schema describes much of the board and the pieces; the full application contract also describes the permitted play.
2. **Where the comparison stops:** a schema may itself encode some rules, while others live in application logic or procedures. There is no universal split that makes every important rule belong in exactly one place.

■ 2.2.3 PLAIN — a worked example

1. Here is the first order line as a JSON object. JSON is a text format for exchanging structured values. The braces enclose an object; each quoted name is followed by its value. The structure below is an original teaching example, not a complete production message specification. [S07]

```
{
  "record_type": "agreed_order_line",
  "schema_version": 1,
  "order_id": "0-1042",
  "line_no": 1,
  "product_id": "P-NOTE",
  "quantity": 2,
  "unit_price_minor": 7550,
  "currency": "INR"
}
```

2. Suppose another program sends the same field names but uses 75.50 to mean rupees. It has not followed our contract. Silently accepting the value and treating it as paise would produce a different price.
3. The receiver should reject or explicitly transform that message under a documented input contract. It should not decide the unit by whether a number contains a decimal point.

■ 2.2.4 PLAIN — what is really happening inside

1. A receiver first recognises the representation, then checks its structure and values. In our example, it expects a supported record type and schema version. It checks required fields and rejects values outside the contract.
2. A version number is useful only when it points to an actual definition. Writing `schema_version: 1` does not make an undocumented format self-explanatory. Nor should a receiver automatically accept version 2 because it understood version 1.
3. When a producer changes a field's meaning, readers need a compatibility plan. Changing an amount from paise to rupees under the same name and version would break that agreement even if every message still parsed.

■ 2.2.5 TECHNICAL — the engineer's version

1. Define four outcomes for an incoming teaching record: accepted without transformation; accepted through a documented transformation; rejected with a precise reason; or held for review because its meaning is unresolved. The review path must not quietly feed uncertain values into ordinary totals.
2. For quantity, our contract requires a positive integer, not just something a language can coerce to one. The included Python validator deliberately rejects booleans as quantities, even though Python's Boolean type has an integer relationship. This is our validator's domain policy, demonstrated by a test. [S22]
3. Unknown fields also need a policy. An extensible analytics feed may preserve them. A tightly bounded command may reject them. Our small validator rejects unexpected field names so a misspelling such as `unit_prcie_minor` does not silently disappear.
4. The contract should also define how invalid inputs are reported. "Bad data" is not as useful as "quantity must be a positive integer." The latter helps the sender repair the actual defect without exposing unrelated records.

■ 2.2.6 WORDS — remember these

1. **Schema:** the declared structure; technically, a definition of fields, types, relationships and applicable structural rules.
2. **Schema version:** which contract applies; technically, an identifier for a defined revision of the schema and its interpretation.
3. **Parser:** a shape reader; technically, software that recognises a representation according to its grammar.
4. **Coercion:** automatic conversion; technically, a transformation between types that may preserve or change intended meaning.

2.3 One row means one what?

■ 2.3.1 PLAIN — in simple words

1. Before counting rows, finish this sentence: "One row in this dataset represents one ____." This is the row's **grain**.

2. An order row and an order-line row are not interchangeable. A product row describes a catalogue item. A stock-count row describes an observation of an item at a location and time. All can mention the same product without representing the same kind of fact.
3. When different grains are mixed, a table can look tidy while its totals become misleading.

■ 2.3.2 PLAIN — a picture in your head

1. Imagine counting a classroom from two lists. The attendance list has one line per pupil. The timetable has one line per lesson attended by a pupil. Counting timetable lines does not count pupils unless you deliberately remove the repetition.
2. **Where the comparison stops:** repetition is not always a defect. The timetable genuinely needs several records for the same pupil because each record stands for a different lesson. The design question is whether repetition matches the intended grain.

■ 2.3.3 PLAIN — a worked example

1. Our order-line dataset contains these four rows:

Order	Line	Product	Units	Line amount, paise
O-1042	1	P-NOTE	2	15100
O-1042	2	P-PEN	1	2000
O-1043	1	P-NOTE	1	7550
O-1043	2	P-PEN	2	4000

2. There are four rows because the grain is one agreed order line. There are two orders because only O-1042 and O-1043 appear as order identifiers. There are six units because $2 + 1 + 1 + 2 = 6$.
3. Now imagine that each row also repeats its complete order total. Adding that repeated column would count each two-line order twice: $17100 + 17100 + 11550 + 11550 = 57300$ paise. The correct combined order value is 28650 paise.
4. The stored numbers need not be individually false for the calculation to be wrong. The error comes from aggregating at the wrong grain.

■ 2.3.4 PLAIN — what is really happening inside

1. A query works with the rows it receives. If a combination operation produces several rows for one order, a later sum sees several rows. The engine does not automatically know that a repeated order total should count only once.
2. The repair is not a universal instruction to remove duplicates. Two different orders can legitimately have the same total. Deduplicating the amount itself could remove a real order from the calculation.
3. Instead, decide which identity defines the things being counted and at which grain each amount is meaningful. We will use separate order-level and line-level structures when the relational model is built in later chapters.

■ 2.3.5 TECHNICAL — the engineer's version

1. The sample line identity is (order_id, line_no). The quantity and agreed unit price describe that line. An order total describes the order, so its natural aggregation boundary is different.
2. A **functional dependency** says that one set of attributes determines another under the data model. For our case study, the order-line identifier determines the values for that accepted line version. It does not imply that product identity alone determines an agreed historical price.
3. A product's current catalogue price may change tomorrow. Replacing yesterday's agreed line price with today's catalogue price would change the meaning of the historical order. Whether to preserve a copied price, reference a versioned price record or use another design is a later modelling decision; the requirement is to retain the correct historical meaning.
4. A basic test can expose the grain mistake: compute totals from the four line amounts and compare them with totals from the two order headers. Agreement is a useful check in this fixture, not a guarantee that every other report is correct.

■ 2.3.6 WORDS — remember these

1. **Grain:** what one row represents; technically, the unit of observation or business fact represented at the dataset's chosen level.
2. **Aggregation:** combining several records into an answer; technically, a calculation over a defined group of inputs.
3. **Functional dependency:** one part determines another; technically, a constraint under which equal determining attributes imply equal dependent attributes.
4. **Historical value:** a value tied to an earlier event or agreement; technically, information whose interpretation must not be replaced by an unrelated current-state value.

2.4 Identity without guesswork

■ 2.4.1 PLAIN — in simple words

1. An identifier lets us refer to a particular thing without redescribing it every time. It answers "which one?" It does not automatically answer "is this genuine?" or "is this the same real-world person?"
2. Two orders can contain the same items and amounts and still be different orders. One order can be delivered twice as a message and still represent only one order. Matching visible values does not settle the distinction.
3. We need to define the thing being identified and the scope within which its identifier is unique.

■ 2.4.2 PLAIN — a picture in your head

1. A seat number is useful inside a particular theatre and performance. "Seat 12" alone may not identify where you should sit. You may also need the row, venue and performance.
2. **Where the comparison stops:** a software identifier can be designed to have a broader scope. Even then, a broad identifier does not authenticate the person presenting it or prove that two separately created records describe different real-world entities.

■ 2.4.3 PLAIN — a worked example

1. At Mira's single shop, 0-1042 identifies our first order. Later a second branch starts its own sequence and also creates 0-1042. If both identifiers were only promised to be unique within their own branch, the collision is a failure of the combined interpretation, not necessarily a broken local sequence.
2. One solution is to identify the order by the pair (branch_id, order_id). Another is to introduce an identifier allocated across the whole system. We will compare designs later; for now, either requires an explicit scope.
3. For the current one-shop fixture, line 1 of O-1042 and line 1 of O-1043 are different lines. line_no alone is not their full identity.

■ 2.4.4 PLAIN — what is really happening inside

1. The system can enforce a uniqueness rule over a chosen set of stored values. That rule is valuable, but its meaning depends on the columns chosen.
2. If every incoming message receives a fresh database identifier, the database may contain perfectly unique rows for the same repeated business event. The uniqueness rule succeeded at row identity while the business deduplication rule was never supplied.
3. Conversely, collapsing all records with the same product, quantity and amount could merge two genuine purchases. Duplicate detection requires a definition of "same event," not simply a desire for fewer rows.

■ 2.4.5 TECHNICAL — the engineer's version

1. Distinguish **row identity**, **business identity** and **delivery identity**. Row identity selects a stored record. Business identity selects the real or administrative occurrence represented. Delivery identity selects a particular transmission or attempt. Our design may intentionally map several deliveries to one business event.
2. In PostgreSQL, a primary key can span several columns and requires unique, non-null values for the key. That supports the composite line identity used in our model. It does not infer whether two different keys refer to the same underlying purchase. [S02]
3. A deduplication decision also needs a conflict path. Two deliveries claiming the same business identifier but carrying different quantities should not automatically be treated as identical replays. The system must apply a documented rule or retain the conflict for review.
4. For now, the safe mental model is that an identifier is a reference under a contract. Its format, scope, allocation and lifecycle all belong in that contract.

■ 2.4.6 WORDS — remember these

1. **Identifier:** a reference to a particular thing; technically, a value or tuple interpreted within a defined namespace and lifecycle.
2. **Composite key:** an identifier made from several parts; technically, a key consisting of more than one attribute.

3. **Namespace:** where a name has its meaning; technically, the scope within which identifiers are interpreted and uniqueness is defined.
4. **Deduplication:** recognising repeated representations of the same thing; technically, a policy-driven process whose identity assumptions and conflict handling must be explicit.

2.5 Source, event time and recording time

■ 2.5.1 PLAIN — in simple words

1. A record has a history. Somebody or something produced it, possibly from earlier records. Knowing that history can help you assess what it supports and how to reproduce it.
2. Time is part of that context. The time an event happened may differ from the time its record reached the system. An offline till can record a sale at the counter and upload it later.
3. Calling both moments simply time makes later questions unnecessarily difficult.

■ 2.5.2 PLAIN — a picture in your head

1. A letter has a date written inside and a date stamped when it reaches an office. The first describes what the sender says about timing; the second describes an observed step in delivery. You may need both to answer a question fairly.
2. **Where the comparison stops:** neither timestamp guarantees a perfectly accurate clock. The date inside may also describe a plan rather than an occurrence. The field's meaning and the clock's limitations still matter.

■ 2.5.3 PLAIN — a worked example

1. For our fictional order, assume the till records an occurrence time of 2026-09-01T09:30:00+05:30. The receiving system records arrival at 2026-09-01T09:37:00+05:30.
2. Under the example's shared, accurate-clock assumption, the difference is seven minutes. A report asking when the order occurred uses the first timestamp. A report asking how long the feed was delayed compares the two.
3. A third report might ask what the central office knew at 09:35. At that point the order had occurred in the story but had not yet arrived in the central dataset. Rebuilding that historical view requires more than the occurrence time alone.

■ 2.5.4 PLAIN — what is really happening inside

1. The receiver can preserve an event-time field supplied by the source and create a separate recording-time field itself. It can also record the source identifier and import batch so a suspicious record can be traced.
2. This is a practical form of **provenance**: information about how a record or result came to exist. W3C's PROV model distinguishes entities, activities and agents, and represents relationships such as derivation. Our small shop metadata borrows that distinction without claiming to implement the complete PROV standard. [S05]
3. A source label is a starting point for investigation, not a certificate of truth. A wrongly configured till can send consistently wrong times with a perfectly consistent source identifier.

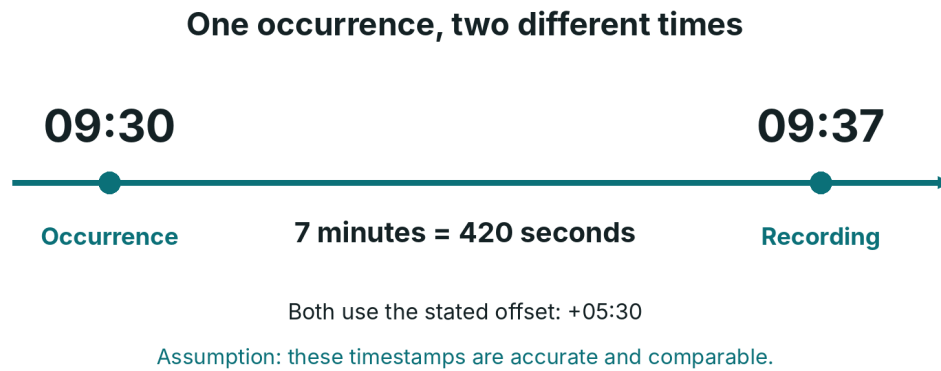


Figure 2.2: Figure 2.1. One occurrence, two relevant times. The timeline assumes the clocks are accurate and comparable; the record itself does not prove that assumption.

■ 2.5.5 TECHNICAL — the engineer’s version

1. Use explicit field semantics such as `occurred_at`, `recorded_at` and `source_system`. Their exact names are a case-study convention. `recorded_at` should state which receiving boundary it describes: initial receipt, validation completion or database commit are not necessarily the same moment.
2. Distinguishing event time from recording time introduces the need for multiple temporal perspectives. Two timestamp columns alone do not implement a full bitemporal database or preserve every revision needed for historical queries.
3. The numeric offset in our timestamp identifies a relationship to UTC for that representation. RFC 3339 defines a widely used Internet timestamp syntax with such offsets. It does not make a source clock accurate, and a numeric offset is not the same thing as a named time zone with rules. [S09]
4. For a derived report, provenance should connect the output to its input set and transformation version. An output label such as “sales report” is not enough to reconstruct which records were included or which rule produced the result.

■ 2.5.6 WORDS — remember these

1. **Event time:** when the described occurrence happened; technically, the timestamp assigned to the event under its source’s semantics.
2. **Recording time:** when a defined system boundary recorded the information; technically, a timestamp whose acquisition point must be specified.
3. **Provenance:** the history behind a record; technically, information about entities, activities, agents and derivations involved in producing it.
4. **Lineage:** how one result came from other data; technically, the dependency path between inputs, transformations and outputs.

2.6 Corrections and the limits of evidence

■ 2.6.1 PLAIN — in simple words

1. Suppose a quantity was entered incorrectly. Replacing the value can make the current record more useful, but it may erase the explanation for why yesterday's report differed from today's.
2. A correction should say what claim changed and why the change was accepted. It should not pretend that the corrected record was always what the system knew.
3. This does not mean every system must retain every old value forever. Keeping history has a purpose and a lifecycle. The requirement is to choose the policy deliberately and describe its limits honestly.

■ 2.6.2 PLAIN — a picture in your head

1. On a paper form, a correction might cross out one value, write the replacement and add a note. The visible change helps a later reader understand the discrepancy.
2. **Where the comparison stops:** digital records can have access controls, version histories and multiple derived copies. Preserving an old value in one place does not ensure that every report or export has been corrected, and indefinite preservation can itself be inappropriate.

■ 2.6.3 PLAIN — a worked example

1. This is a separate correction exercise, not a change to the canonical O-1042 fixture. Imagine a draft order D-2001 with quantity three, later verified as quantity two. At a unit price of 7550 paise, the draft line amount was 22650 paise and the corrected amount is 15100 paise.
2. The difference is 7550 paise. If a report already included the draft amount, simply changing the current row does not tell us whether that report refreshed. We need to identify which outputs depend on the changed record.
3. A useful correction note might record the affected draft, old and new quantities, the reason "entry corrected against the retained source slip," the approving role, and the time of the decision. The note describes evidence used in this hypothetical example; it does not prove the source slip was itself accurate.

■ 2.6.4 PLAIN — what is really happening inside

1. The system applies a chosen correction policy. It might update a draft in place while keeping an authorised revision history. It might append a correcting event. It might prevent direct editing after a particular business transition and require a separate reversal workflow.
2. Which policy is appropriate depends on what the record means. Correcting a spelling mistake in a draft label is not necessarily the same operation as reversing an accepted commercial obligation.
3. Derived outputs need their own response: refresh, correction notice, versioned replacement or an explicit statement that a previous export cannot be recalled. A deletion from the current screen does not tell us what happened to every retained copy.

■ 2.6.5 TECHNICAL — the engineer's version

1. Separate **current-state correctness**, **historical explainability** and **retention policy**. A design may improve one while damaging another. For example, overwriting a value simplifies the current row but can destroy the information needed to explain an earlier output.
2. The case-study correction model will record the identity of the subject, the scope of the change, the asserted reason, the permitted actor and the affected revision. It will avoid treating an unverified reason code as established causation.
3. Provenance relationships can connect a corrected output with its predecessor and the activity that produced it. The PROV model provides concepts for such relationships; it does not dictate our business authorisation or retention policy. [S05]
4. The acceptance test must match the claim. A test that verifies the latest row says two does not also prove that historical reports are reproducible, permissions are correct or all external copies have changed. Those require separate observations.

■ 2.6.6 WORDS — remember these

1. **Revision:** a particular state after a change; technically, an identifiable version of a record or artefact.
2. **Correction:** a change to repair a stated defect; technically, an authorised transition whose scope, evidence and consequences should be recorded.
3. **Audit trail:** information used to explain actions; technically, a record of relevant operations and context, with its own integrity and retention requirements.
4. **Retention:** how long and why information is kept; technically, a policy governing continued storage and eventual disposition for defined record classes.

2.97 A record-design exercise

You receive this export from a fictional third-party till:

```
id,item,qty,amount,time
1042,Notebook,2,151,09:30
1042,Pen,1,20,09:30
```

Task. Write down what can be read directly and what remains unknown. Do not fill the gaps from the familiar numbers in our running story; in this exercise the sender has not supplied that context.

Worked answer. The file contains two data rows with the same `id`, different item labels, quantities two and one, amounts 151 and 20, and matching time strings. That describes the representation, not its full business interpretation.

We do not yet know what `id` identifies, whether the amount is per unit or per line, which currency and scale apply, what units the quantities count, which date and offset accompany the time, or whether the records describe requests, accepted orders or completed handovers. We also lack a source record identifier and a duplicate-delivery policy.

A suitable repair is to obtain the sender's contract, then transform the records under that contract while retaining appropriate import provenance and rejected-input evidence. It is not to guess INR because the values happen to resemble our previous example.

Second task. Design a field dictionary for an agreed order line.

Worked answer. State the line grain and composite identity first. Define product reference, quantity unit, agreed unit price, currency, schema version and missing-field rules. Define source and temporal fields according to the actual workflow. A field is not required merely because it is fashionable; it is required when the stated use needs it.

2.98 Common wrong ideas

“Clear field names eliminate the need for a contract.” Names help, but they do not fully define units, timing, scope, missingness or permitted changes.

“A unique database ID proves there are no duplicate business events.” Separate stored rows can refer to the same event.

“Repeated values should always be removed.” Repetition can be legitimate at the chosen grain. Two different orders can have the same total.

“A timestamp tells us everything about time.” It needs a defined meaning and a clock context. Occurrence, receipt and commit are different boundaries.

“An audit record proves the recorded explanation.” It preserves an asserted explanation and relevant context. The explanation still needs suitable evidence.

2.99 Chapter summary in 20 lines

1. A useful record makes an interpretable claim.
2. Values without units and context can be ambiguous.
3. Field names support a contract but do not replace it.
4. Syntax describes form; semantics describes meaning.
5. A data dictionary connects values to their intended interpretation.
6. A schema version must identify a real definition.
7. Parsing successfully does not prove domain validity.
8. A transformation should be explicit rather than silently guessed.
9. The row grain states what one record represents.
10. Counting lines is not the same as counting orders.
11. Aggregating repeated order totals can multiply the answer.
12. Historical prices must retain their historical meaning.
13. Identifiers operate within defined scopes.
14. Row identity and business-event identity can differ.
15. A repeated delivery and a repeated purchase require different treatment.
16. Event time and recording time answer different questions.
17. Provenance helps explain where information came from.
18. Provenance does not automatically certify truth.
19. Corrections need meaning, authority and an explicit history policy.
20. The next chapter explains the values that fields can carry.

Numbers, Text, Time and Missing Values

3.0 What this chapter gives you

A database cannot make a field meaningful if the application has chosen the wrong kind of value. This chapter explains the everyday decisions hidden inside “just store it”: whether something is a count or a label, whether a fraction must be exact, what a character means, which kind of time we are recording, and how to preserve the difference between zero and not known.

You will calculate integer limits, inspect a floating-point surprise, compare two representations of the same visible letter, read an offset timestamp and reason about SQL NULL. The included laboratory repeats the numerical and textual examples with synthetic inputs in an isolated environment.

Code blocks are optional on the first reading. Each one is accompanied by an explanation of what it demonstrates. A printed result is not a promise that every language or database implements the same behaviour.

3.1 Choose a type before choosing a format

■ 3.1.1 PLAIN — in simple words

1. A **type** describes the kind of value a field can carry and the operations that make sense for it. A number of notebooks is a count. An order identifier is a label. Both may contain digits, but adding one to them has different meanings.
2. For the count, adding one can mean another notebook. For the label 0017, converting it to the number 17 may throw away a distinction the sending system needs. A field should not become numeric merely because its current examples contain only digits.
3. Start by asking what the value represents. Then choose the representation and the rules that preserve that meaning.

■ 3.1.2 PLAIN — a picture in your head

1. Think of drawers marked “quantities,” “names” and “dates.” Putting a name in the quantity drawer does not make arithmetic on the name useful. Labels help prevent inappropriate operations before they happen.
2. **Where the comparison stops:** programming languages differ in how strongly they enforce types and how readily they convert values. The drawer analogy describes the purpose of a type, not a guarantee that every language locks the drawers.

■ 3.1.3 PLAIN — a worked example

Value	Meaning in an example	Suitable starting model
12	Counted notebooks on a shelf	Non-negative integer count
0017	Identifier supplied as a four-character label	Text with a documented identifier rule
7550	Unit price in paise for our fixture	Integer amount plus currency and scale
2026-09-01	A calendar date	Date, not a made-up midnight instant
09:30 at offset +05:30	A local clock reading with an offset	Needs a date to identify the example's instant
Not yet counted	Absence of an observation	Missing-value state, not the number zero

1. The same visible digits can participate in different models. A product named “100” is still a product label. A genuine measured quantity may require fractions rather than a whole-number type. The meaning decides the direction of the design.

■ 3.1.4 PLAIN — what is really happening inside

1. A program reads bytes and interprets them as values. It may convert text to a number, validate the range and store a representation chosen by its engine. Each conversion is a place where information can be preserved, rejected or lost.
2. An import that turns 0017 into 17 cannot reconstruct the original four-character spelling from 17 alone without another rule. If the field's contract treats the spelling as significant, the conversion has lost information.
3. Conversely, storing every value as text postpones rather than solves the problem. A later calculation still needs to know whether “12” is a quantity, whether “12.0” is allowed and whether an empty string means missing or invalid.

■ 3.1.5 TECHNICAL — the engineer's version

1. Separate the **domain type** from the **storage type**. `OrderId`, `StockCount` and `UnitPriceInPaise` are domain concepts. `text` and `integer` are possible storage types. Several domain concepts can share a storage type while having different valid operations.
2. For our fixture, `StockCount` permits zero, while an agreed order line's quantity must be positive. Both are integer-valued, but their domains differ. A type alone may therefore need a constraint. PostgreSQL documents column and table constraints as a way to express additional restrictions. [S02]
3. Conversions should have explicit failure behaviour. Reject a quantity of two, distinguish an unsupported format from an out-of-range value, and do not truncate a fractional quantity to make it fit a whole-unit contract.
4. SQLite's ordinary dynamic typing is a product-specific reminder not to infer all runtime behaviour from a declared type name. The laboratory uses explicit validation for its teaching contract rather than claiming that every engine will enforce identical types. [S21]

■ 3.1.6 WORDS — remember these

1. **Type:** the kind of value; technically, a classification that governs representation and permitted or defined operations.
2. **Domain type:** the business meaning of the value; technically, a model-specific type whose rules may exceed the underlying storage type.
3. **Storage type:** the engine's representation category; technically, a supported type or storage class with documented operations and limits.
4. **Lossy conversion:** a change that forgets information; technically, a mapping from which the original value cannot always be reconstructed.

3.2 Integers and the units they count

■ 3.2.1 PLAIN — in simple words

1. An integer is a whole number: zero, one, two, and their negative counterparts. Integers are useful for counting indivisible units and for representing amounts in a deliberately chosen smallest unit.
2. Whole does not mean unlimited. A fixed-size representation has only so many available patterns. You must know both what each step counts and how large the value can become.
3. For our order prices, one step counts one paisa. For notebook stock, one step counts one notebook. They should not be added to each other merely because both are integers.

■ 3.2.2 PLAIN — a picture in your head

1. Imagine a mechanical counter with three decimal wheels. It has room for 000 through 999. A fourth digit cannot appear without a design change or some defined response to exceeding the limit.
2. **Where the comparison stops:** computers use binary patterns and different signed representations. Some languages can allocate more space for larger integers; some fixed-size operations reject an excessive value, and some have other specified overflow behaviour. Always name the actual environment.

■ 3.2.3 PLAIN — a worked example

1. A bit has two possible values. With eight bits there are $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$ patterns. An unsigned interpretation can use them for 0 through 255: 256 distinct integers, not 255.
2. A 32-bit signed two's-complement interpretation has the range -2,147,483,648 through 2,147,483,647. PostgreSQL's four-byte integer type has that range and rejects values outside its range. [S10]
3. If a positive price amount uses that maximum as a count of paise, it represents INR 21,474,836.47. The calculation is $2,147,483,647 / 100$. Changing the unit changes the represented amount without changing the bit limit.
4. For O-1042, the calculation remains comfortably within this example range:

```
2 × 7550 + 1 × 2000 = 17100 paise
17100 paise / 100 = INR 171.00
```

■ 3.2.4 PLAIN — what is really happening inside

1. The representation stores a pattern interpreted under a rule. “Signed” means the chosen interpretation includes negative values. The unit comes from the surrounding contract, not from a special “money” property in each bit.
2. Multiplication and addition can produce results larger than their inputs. Choosing a type that holds one unit price is not enough if the same type must also hold the total for many items or a long reporting period.
3. The review question is therefore about the maximum intermediate result as well as the maximum stored input. In a toy fixture, the answer may be simple. In a real workload, it needs a stated bound and behaviour when that bound is exceeded.

■ 3.2.5 TECHNICAL — the engineer’s version

1. For an unsigned n -bit representation, the range is 0 through $2^n - 1$. For an n -bit signed two’s-complement representation, it is $-2^{(n-1)}$ through $2^{(n-1)} - 1$. These expressions count finite bit patterns; they do not claim that all language integer types use a fixed n .
2. Python’s integers in the laboratory can grow beyond a fixed 32-bit range, subject to available resources. That means a passing arithmetic example in Python does not test whether a PostgreSQL integer column would accept the same enormous result. Our explicit range demonstration calculates the bound rather than claiming to exercise a PostgreSQL server. [S22]
3. The unit should be carried by naming, documentation and, where practical, domain-specific types. Converting a minor-unit amount to display text should not change the stored value. Combining currencies is outside the fixture’s permitted operations; an amount tagged INR is not interchangeable with the same integer tagged another currency.
4. Integer minor units are not a universal answer to all price calculations. Proportions and allocations can produce fractions of the chosen unit. The next section introduces the rounding policy needed at that boundary.

■ 3.2.6 WORDS — remember these

1. **Integer:** a whole number; technically, a value in the set of signed whole numbers, represented with implementation-specific limits.
2. **Range:** the smallest through largest supported values; technically, the domain bounds of a representation or declared type.
3. **Overflow:** exceeding a representation’s capacity; technically, an operation whose mathematical result lies outside the representable range.
4. **Scale:** what one stored step represents; here, the fixed relationship between an integer amount and its displayed unit.

3.3 Decimals, fractions and rounding

■ 3.3.1 PLAIN — in simple words

1. Some fractions fit a number system neatly and others do not. In decimal notation, one half is 0.5, but one third continues as 0.333... without ending. Binary arithmetic has a different set of neatly fitting fractions.
2. This is why a computer may show a tiny surprise when adding familiar decimals with a binary floating-point type. The machine is operating on nearby representable values, not on an infinitely precise version of every number you typed.
3. The solution is not “never use floating point.” It is to choose a representation and an error or rounding policy suitable for the task.

■ 3.3.2 PLAIN — a picture in your head

1. A ruler marked only in whole millimetres cannot directly express every possible distance. A point halfway between two marks needs an approximation or a more detailed measuring method.
2. **Where the comparison stops:** floating-point spacing changes with magnitude, unlike the evenly spaced marks on that ruler. The analogy explains limited representation, not the exact layout of floating-point numbers.

■ 3.3.3 PLAIN — a worked example

1. In the Python environment used by the supplied laboratory:

```
| print(0.1 + 0.2)
```

2. The displayed result is:

```
| 0.30000000000000004
```

3. Python’s floating-point tutorial explains why common decimal fractions are approximated in binary and why display formatting can hide or reveal that approximation. This is a representation issue, not evidence that addition is randomly unreliable. [S12]
4. Now construct decimal values directly from decimal text:

```
| from decimal import Decimal  
| print(Decimal("0.1") + Decimal("0.2"))
```

5. The result in the supplied example is 0.3. Constructing the values from strings is deliberate: it avoids first introducing a binary floating-point approximation. Decimal arithmetic still has a precision context and rounding rules; it does not make every fraction finitely representable. [S13]

■ 3.3.4 PLAIN — what is really happening inside

1. The chosen numeric representation determines which values are exact and how operations produce rounded results. Formatting determines what part of the result is shown. These are different layers.
2. Displaying a total with two decimal places does not prove that the underlying calculation used an appropriate representation or rounding policy. It can merely hide a discrepancy.

3. Consider dividing 100 paise equally among three recipients. Each mathematical share is 33 and one third paise. Under our whole-paise output rule, three displayed shares of 33 paise add to only 99 paise. One paise remains to allocate. An explicit rule might give the extra paise to the first recipient in a defined ordering, producing 34, 33 and 33. That is a selected allocation policy, not an arithmetic law.

■ 3.3.5 TECHNICAL — the engineer's version

1. Distinguish **representation error**, **measurement error** and **business rounding**. Representation error comes from storing or computing with an approximation. Measurement error concerns how closely a measured value reflects the relevant quantity. Business rounding is a deliberate policy for mapping an amount to an allowed unit. Treating all three as “rounding” conceals different causes.
2. PostgreSQL provides exact numeric/decimal types as well as floating-point types. Declared precision and scale affect what is stored. A field's type choice should be tied to its required calculations and boundaries rather than copied from an unrelated system. [S10]
3. A robust calculation specification names the input units, intermediate precision, rounding rule, rounding stage and reconciliation rule. Rounding each line and then summing is not always the same operation as summing exact intermediates and rounding once.
4. The laboratory's `allocate_minor_units(100, 3)` example uses integer division and remainder to produce [34, 33, 33]. It verifies that the shares sum to 100 and differ by at most one. The rule selects recipients by list position; it does not claim to be the correct policy for every allocation problem.

■ 3.3.6 WORDS — remember these

1. **Floating point**: a number representation with a movable scale; technically, a finite significand and exponent representation with defined rounding behaviour.
2. **Decimal arithmetic**: arithmetic based on decimal representation; technically, operations over decimal values under a precision and rounding context.
3. **Quantisation**: fitting values to allowed steps; technically, mapping values to a discrete set of representable or permitted outputs.
4. **Rounding policy**: the chosen way to handle non-exact results; technically, the rule and stage used to select an allowed output value.

3.4 Text, Unicode and bytes

■ 3.4.1 PLAIN — in simple words

1. A computer stores a representation of text, not tiny printed letters. A text system needs an agreement about which abstract characters are represented and how to turn that sequence into bytes.
2. Unicode provides a shared character framework. UTF-8 is one way to encode Unicode text as bytes. They are related concepts, not two names for the same layer. A Unicode scalar value takes from one to four bytes in UTF-8. [S14]

3. The number of bytes is therefore not generally the number of visible characters. That distinction matters when storing names, limiting input length or cutting a string into pieces.

■ 3.4.2 PLAIN — a picture in your head

1. Imagine a catalogue assigning a number to each symbol, and a packing rule for sending those numbers in small envelopes. The catalogue is one layer; the packing rule is another. Some symbols need more envelope space than others.
2. **Where the comparison stops:** a visible character can be formed from several Unicode code points. The reader's "one character" is not always one catalogue entry, and rendering may combine or shape entries according to the writing system.

■ 3.4.3 PLAIN — a worked example

1. The Latin letter A is one code point, U+0041. The rupee sign ₹ is one code point, U+20B9. In UTF-8 their byte sequences differ in length:

Text	Code point	UTF-8 bytes in hexadecimal	Byte count
A	U+0041	41	1
₹	U+20B9	E2 82 B9	3

2. The supplied laboratory checks these sequences directly. Hexadecimal, or base 16, is just a compact way to write the byte values; it does not change the text.
3. Now compare the visible letter "é." It can be represented by U+00E9, or by U+0065 followed by U+0301, a combining acute accent. The first sequence has one code point and the second has two. Unicode defines canonical equivalence and normalisation forms for cases such as this. [S15]

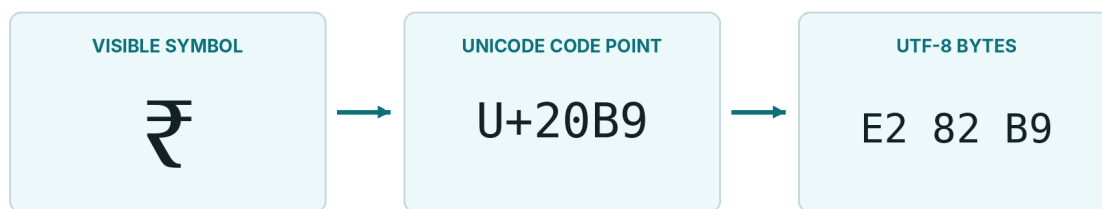
■ 3.4.4 PLAIN — what is really happening inside

1. A decoder turns valid encoded bytes into a text representation. A renderer draws the resulting text using fonts and shaping rules. These are not the same operation: being able to decode a character does not guarantee that a particular font draws it well.
2. A program comparing code-point sequences can find our two forms of "é" unequal even when they look alike. Applying an appropriate normalisation policy can make canonically equivalent forms comparable. It does not translate a name, decide whether two people are the same, or make all similar-looking characters identical.
3. When a user asks for "20 characters," the product should decide what that limit is for. A storage byte limit, a code-point count and a user-visible editing limit are different requirements. Unicode's text-segmentation guidance defines grapheme-cluster boundaries relevant to user-perceived characters. [S16]

■ 3.4.5 TECHNICAL — the engineer's version

1. The laboratory uses Python strings and UTF-8 encoding. `len("₹")` is 1 in that environment, while `len("₹".encode("utf-8"))` is 3. Neither result alone defines a universal user-interface character-count policy.

A symbol is not the same thing as its bytes



In this example: one code point, three bytes.

Encoding converts a character sequence into a byte sequence.

Figure 3.3: Figure 3.1. The rupee sign passes through three different descriptions: visible symbol, Unicode code point and UTF-8 byte sequence.

2. For the two acute-e forms, Python reports code-point counts of one and two. Their UTF-8 lengths are two and three bytes. Normalising each with NFC makes the two resulting strings equal in the demonstrated case. The equality before normalisation is false; the equality after it is true. [S15]
3. Normalisation is not the same operation as case folding, language-sensitive sorting or identity matching. A system that needs all of them should specify their order and purpose rather than hide them behind a vague “clean text” step.
4. At an import boundary, invalid encoding needs a deliberate policy. Silent replacement can make a file displayable while losing the original distinction that an identifier depended on. The laboratory uses strict decoding behaviour for its bounded example and verifies that an invalid UTF-8 byte sequence is rejected.

■ 3.4.6 WORDS — remember these

1. **Code point:** an assigned position in the Unicode code space; technically, a numeric value used in the character model, not a guarantee of one visible glyph.
2. **Encoding:** the mapping into bytes; technically, a defined representation of a character sequence for storage or interchange.
3. **Grapheme cluster:** a unit closer to what a reader perceives as a character; technically, a text-segmentation unit defined by applicable Unicode rules and possible tailoring.
4. **Normalisation:** a specified way to standardise equivalent text sequences; technically, conversion to a Unicode normalisation form, not general identity verification.

3.5 Dates, instants and durations

■ 3.5.1 PLAIN — in simple words

1. Not every value involving time is the same kind of thing. A birthday is a calendar date. A completed sale has an occurrence instant. “The shop opens at 09:00” is a local schedule. “The upload took seven minutes” is a duration.

2. Treating all four as an arbitrary text field creates ambiguity. Treating all four as midnight in a time zone can create a different ambiguity: a birthday was not necessarily meant to identify an instant at all.
3. Choose the temporal meaning before choosing the storage representation.

■ 3.5.2 PLAIN — a picture in your head

1. A calendar, a wall clock and a stopwatch answer different questions. A calendar tells you which date. A wall clock gives a local time of day. A stopwatch measures elapsed time.
2. **Where the comparison stops:** computers can combine these concepts and convert between representations, but a conversion cannot recover context that was never supplied. A bare 09:30 does not identify one instant in history.

■ 3.5.3 PLAIN — a worked example

1. Read this timestamp from left to right:

| 2026-09-01T09:30:00+05:30

2. It states the date 1 September 2026, local clock time 09:30:00, and a numeric offset five hours and thirty minutes ahead of UTC. For this representation, subtracting that offset gives:

| 2026-09-01T04:00:00Z

3. The two strings describe the same instant. The Z form uses UTC. The syntax is within the timestamp format described by RFC 3339. [S09]
4. Now compare occurrence at 09:30 and recording at 09:37 with the same date and offset. Under the example's comparable-clock assumption, the interval is 420 seconds. The laboratory verifies the conversion and subtraction for these inputs.

■ 3.5.4 PLAIN — what is really happening inside

1. The program interprets the date, clock fields and offset, then can compare the resulting instants. A named time zone adds rules associated with a place; a numeric offset only describes an offset for the representation in front of you.
2. For a historical event, an instant representation may be sufficient for ordering and duration calculations, with additional source context retained when needed. For a recurring local opening time, "09:00 in the shop's local time zone" is a different requirement from "the same UTC time forever."
3. Two source clocks can disagree. Extra decimal places on a timestamp do not establish that the clock was accurate to that precision. A latency calculation between independent systems therefore needs assumptions about the clocks or another measurement method.

■ 3.5.5 TECHNICAL — the engineer's version

1. PostgreSQL distinguishes date, timestamp without time zone and timestamp with time zone. For the latter, input is normalised to an instant and output is displayed using the session's time-zone setting; the original named zone is not retained in that value. The type's name should not be read as a promise to preserve all original zone context. [S11]

2. Use a date for a date-only concept. Use an instant-capable representation for an occurrence when an actual instant is required. Preserve a named zone separately when the application needs that original context or future local scheduling rules. A local recurring schedule is not fully represented by one past offset.
3. Duration arithmetic also has a domain boundary. “Add 24 elapsed hours” and “same local time tomorrow” need not be interchangeable in every zone and calendar situation. Our 420-second example avoids those complications by using two explicit offsets on the same date.
4. The Python laboratory checks the stated examples using its standard datetime facilities. It is not a complete RFC 3339 conformance test, a time-zone database certification or a clock-synchronisation experiment.

■ 3.5.6 WORDS — remember these

1. **Date:** a calendar position; technically, a calendar day without necessarily specifying a time of day or an instant.
2. **Instant:** one point on a timeline; technically, a temporal position distinguishable from its local display representation.
3. **UTC offset:** the difference from UTC in a representation; technically, a signed hours-and-minutes offset, not a complete named-zone rule set.
4. **Duration:** elapsed amount of time; technically, a measured or specified interval whose units and arithmetic semantics must be stated.

3.6 Missing is not zero

■ 3.6.1 PLAIN — in simple words

1. Zero is a value. “We have not counted yet” is a lack of a value. Replacing the second with the first changes the claim from “not known” to “known to be none.”
2. An empty text value, a field that was omitted, an explicit missing marker and the word “unknown” are also different representations. A system needs a policy for them rather than assuming they all mean the same thing.
3. The policy should match the purpose. Sometimes a missing value should reject a record. Sometimes it should be preserved so the record remains honest about what is not known.

■ 3.6.2 PLAIN — a picture in your head

1. Imagine four envelopes labelled “refund amount.” One contains a slip saying zero. One contains a slip saying 100. One is empty. One has a note saying the envelope was never requested.
2. You cannot add the empty envelope as zero without making an extra assumption. **Where the comparison stops:** SQL NULL is a specific database marker with defined expression behaviour. It does not automatically preserve every possible reason for absence. A separate status may be needed when the reason matters.

■ 3.6.3 PLAIN — a worked example

1. Suppose four fictional observations contain values 100, 200, missing and 0. The sum of the known values is 300. There are four rows, but only three known values. The mean of the known values is $300 / 3 = 100$.
2. Replacing missing with zero would produce an average of $300 / 4 = 75$. That is not another spelling of the same answer. It is a result under a different assumption.
3. In the laboratory's SQLite example, the query is:

```
SELECT
    COUNT(*) AS row_count,
    COUNT(value) AS known_count,
    AVG(value) AS mean_known
FROM observations;
```

4. For the supplied fixture, the result is (4, 3, 100.0). The same distinction between counting rows and counting non-null values is documented for PostgreSQL's aggregate functions. [S19]

■ 3.6.4 PLAIN — what is really happening inside

1. In ordinary SQL comparisons, a NULL operand generally produces an unknown result rather than a true equality or inequality. Asking `value = NULL` is not the way to find missing values; use `value IS NULL`. PostgreSQL documents this distinction explicitly. [S17]
2. Think of an unknown sealed-box quantity. You cannot say that it equals five, but you also cannot conclude that it does not equal five. The comparison is unknown. A filter keeps rows whose condition is true, so those unknown comparison results do not behave like true matches.
3. This also explains why changing `value > 0` to `NOT (value > 0)` does not automatically include missing values. Negating unknown still does not establish a known true condition. The missing case needs to be handled according to the question being asked. [S18]

■ 3.6.5 TECHNICAL — the engineer's version

1. The relevant three-valued logic has true, false and unknown. Here are selected operations; "unknown" is shown as a word rather than confused with a stored text value.

Expression	Result
NOT unknown	unknown
true AND unknown	unknown
false AND unknown	false
true OR unknown	true
false OR unknown	unknown

2. These operations are consistent with PostgreSQL's documented logical tables. They do not imply that evaluation order is a safe substitute for controlling side effects in expressions. [S18]
3. A check constraint and a missing-value rule are also distinct. In PostgreSQL, `CHECK (quantity > 0)` alone does not reject NULL: a check is satisfied by true or null. Add `NOT NULL` when absence is forbidden. [S02]

4. JSON has a `null` literal, while an absent object member is structurally different. Neither representation carries a universal business explanation for missingness. Our line validator requires all its fields; it does not use `null` as a substitute for an agreed quantity. [S07]
5. For reporting, publish the denominator. “Mean among three known observations, one missing” is more informative than showing 100 with no population or missingness context. The arithmetic cannot decide whether the missing case should be excluded, estimated or treated as an error; that is part of the analytical contract.

■ 3.6.6 WORDS — remember these

1. **NULL:** a specific SQL missing/unknown marker; technically, a marker with defined behaviour in comparisons, logic, constraints and aggregates.
2. **Empty string:** text containing no characters; technically, a zero-length string, distinct from SQL `NULL` in the engines discussed here.
3. **Denominator:** what the division is over; technically, the quantity that defines the base of a ratio or average.
4. **Missingness policy:** what absence means and how it is handled; technically, the domain rules for rejecting, preserving, classifying or analysing unavailable values.

3.97 Practice, then inspect the evidence

Exercise 1: the identifier. A source contract defines 0017 as a four-character order label. An import turns it into the integer 17. What was lost?

Worked answer. The original representation’s leading zeros were lost. Whether that changes business identity depends on the source contract, but this contract explicitly required the four-character label. The receiver should not perform that conversion silently.

Exercise 2: the allocation. Split 100 paise among three recipients using the book’s rule: integer base shares, then give one extra paisa to the earliest recipients until the remainder is exhausted.

Worked answer. Integer division gives a base share of 33 and a remainder of one. The shares are 34, 33 and 33. They sum to 100. A different recipient ordering may allocate the extra paisa differently, so the ordering must be part of the policy.

Exercise 3: the timestamp. Do 2026-09-01T09:30:00+05:30 and 2026-09-01T04:00:00Z identify different instants?

Worked answer. No. The first is five and a half hours ahead of UTC in its local representation. Subtracting the offset yields the second.

Exercise 4: the average. Four rows contain 100, 200, `NULL` and 0. Explain both 100 and 75 without calling them equivalent.

Worked answer. The mean among three known values is 100. Replacing `NULL` with zero produces a mean of 75 over four values. The latter requires an additional assumption that the missing value should count as zero. The source rows alone do not justify it.

Exercise 5: the text. Two strings look like “é,” but one has one code point and the other two. Does this prove a corruption?

Worked answer. No. They may be the canonically equivalent forms demonstrated in the chapter. Inspect their code points and apply the relevant text policy rather than judging identity only by appearance.

3.98 Common wrong ideas

“Digits mean the field should be numeric.” Identifiers can contain digits while remaining labels with significant spelling and scope.

“Integers cannot have numerical limits.” Fixed-size representations have finite ranges. A language with growing integers does not remove a database column’s limit.

“Showing two decimals makes a calculation correct.” Display formatting does not define representation, intermediate precision or rounding policy.

“Decimal arithmetic makes every fraction exact.” Some fractions still require rounding under a finite precision context.

“One visible character is one byte.” Text has distinct byte, code-point and user-perceived segmentation layers.

“Store every time as UTC and all temporal questions disappear.” Dates, instants, local schedules and original zone context remain different requirements.

“Unknown means zero or false.” Missingness and three-valued logic must be handled according to the actual domain and query.

3.99 Chapter summary in 20 lines

1. Choose a value’s meaning before choosing its representation.
2. An identifier containing digits is not automatically a quantity.
3. Domain types can be more specific than storage types.
4. A conversion can lose information while appearing convenient.
5. Integers count whole steps under a stated unit.
6. Fixed-size integer representations have finite ranges.
7. Intermediate calculations need range analysis too.
8. Minor-unit amounts still need currency and scale context.
9. Binary floating point cannot represent every familiar decimal exactly.
10. Decimal arithmetic also has precision and rounding boundaries.
11. Formatting is not a substitute for a calculation policy.
12. Allocations need a rule for indivisible remainders.
13. Unicode and UTF-8 describe different layers of text representation.
14. Bytes, code points and grapheme clusters are different counting units.
15. Normalisation is not general identity verification.
16. A calendar date is not the same concept as an instant.
17. A numeric UTC offset is not a complete named-zone rule set.
18. Missing, zero and empty text must not be silently equated.
19. SQL NULL changes comparison, logical and aggregate behaviour.
20. Clear types preserve meaning before a system ever becomes large.

Measurements, Units and Uncertainty

4.0 What this chapter gives you

1. You will turn “we measured it” into a statement that names the object, quantity, unit, method and time.
2. You will convert units without accidentally changing the quantity, and use units to catch impossible calculations.
3. You will distinguish a repeatable instrument from an accurate result and a detailed display from genuine knowledge.
4. You will work through an average, a sample standard deviation and the uncertainty of an average without skipping the meaning of the arithmetic.
5. You will explain why missing observations and timeouts can change what a report is allowed to claim.
6. You will carry uncertainty through a calculation and state the assumptions behind the resulting interval.

■ A new fixture, not a change to the orders

Mira’s Corner now weighs parcels and records how long work takes. These are new synthetic measurement exercises. They do not change O-1042, O-1043 or the notebook discrepancy introduced earlier. The shop still has no evidence establishing the cause of that discrepancy.

Our main measurement fixture is **M-BOX-01**: five repeated indications for the same sealed test box, in grams: **998, 1000, 1001, 999, 1002**. These are repeated measurements of one box, not the masses of five different boxes. An instrument comparison, a gross-minus-tare exercise and a request-timing exercise will be introduced separately and labelled when used.

4.1 What exactly was measured?

■ 4.1.1 PLAIN — in simple words

1. A number does not tell you what happened until you know what it measures. “The parcel is 1,000” could describe grams, a price, a product code or an item count.
2. Even “the parcel weighs 1,000 grams” leaves a question: does that include the box, tape and packing material, or only the goods inside?
3. Decide what you are trying to measure before deciding where to store the answer. Otherwise two careful people can report different numbers while measuring different things.
4. Keep the difference between a direct reading and a calculated result. Subtracting the packaging mass produces a new result from two inputs; it is not another direct reading of the same display.

■ 4.1.2 PLAIN — a picture in your head

1. Imagine two people measuring a journey. One starts a stopwatch when the driver leaves home. The other starts when the vehicle joins the main road. Their stopwatches can both work perfectly and still disagree.
2. The missing piece is an agreed starting line. A useful measurement definition supplies both the starting and finishing lines, even when the quantity is not time.
3. **Where this comparison breaks:** some quantities, such as a parcel's mass, are not journeys. Their equivalent boundaries concern the object, its condition and the procedure. Agreement about boundaries does not remove instrument error.

■ 4.1.3 PLAIN — a worked example

1. In a separate teaching exercise, the scale indicates **1,052 g** for a packed parcel and **52 g** for its empty packaging under the stated procedure.
2. Gross mass includes the packaging. Tare mass is the packaging mass being subtracted. The estimated net mass is therefore $1,052 - 52 = 1,000$ g.
3. A report that compares 1,052 g from one branch with 1,000 g from another without checking gross versus net has mixed definitions, not discovered a heavier product.
4. The input record below makes one observation's meaning visible. Its identifiers are invented, and the method code points to our own teaching procedure, not a recognised certification.

Field	Example	What it prevents you from guessing
observation_id	M-GROSS-01	Which observation is being discussed
object_id	PARCEL-TEST-01	Which physical test object was measured
quantity_kind	gross_mass	Whether packaging is included
indicated_value	1052	What the instrument displayed
unit	g	The size of each numerical unit
method_version	PACKED-01	Which measurement procedure was followed
recorded_at	2026-09-02T10:00:00+05:30	When this example was recorded

5. This table is not a complete laboratory record. A real procedure might also require instrument identity, environmental conditions, calibration information and a separate measurement time.

■ 4.1.4 PLAIN — what is really happening inside

1. A measurement process connects an object to an instrument, an instrument indication to a recorded value, and that value to a documented interpretation.
2. Software often sees only the last part. It cannot recover an omitted packaging rule by looking harder at the integer 1052.
3. The international metrology vocabulary calls the quantity intended to be measured the **measurand**. Naming it includes enough information about the quantity and object to distinguish the intended measurement. [S26]
4. For the shop, we propose a method register. Every observation refers to a supported method version. The register explains boundaries, units, permitted instruments and what to do when a reading is unavailable.

5. A changed method receives a new version. That lets a later report separate a genuine trend from a change in how the shop measured the work.

■ 4.1.5 TECHNICAL — the engineer's version

1. Distinguish the physical quantity, the instrument indication, the stored representation and the estimator used in a report. A database column may contain any one of these; the column name alone is insufficient evidence.
2. A **measurement model** specifies how inputs produce a result. In our gross-minus-tare exercise, write $\text{net} = \text{gross} - \text{tare}$. This equation also exposes which inputs need compatible units and uncertainty information.
3. The observation's **grain** is one execution of the stated procedure on one object. The record is not automatically one object forever: repeated observations legitimately share the same object identifier.
4. Keep raw input where the purpose and retention policy justify it, but distinguish it from interpreted data. For example, the received string "1052" and the interpreted value 1052 g are not the same representation or the same claim.
5. A practical schema separates a value from its status. `status = not_measured` with no value is different from `status = measured` with value zero. A type-valid number accompanied by the wrong method version should not silently enter a comparable series.
6. Provenance can describe the source entities, the activity that transformed them and the agent associated with that activity. W3C PROV provides a vocabulary for such relationships; our small method register is not a claim to implement all of PROV. [S05]
7. The honest version: more metadata does not repair a badly defined measurand. Start with a question that people can execute consistently, then preserve the information required to interpret the answer.

■ 4.1.6 WORDS — remember these

1. **Measurand:** the quantity you intend to measure; technically, the quantity defined as the target of a measurement, with the necessary object and condition information.
2. **Indication:** what the instrument shows; technically, the quantity value supplied by a measuring instrument or system.
3. **Gross mass:** goods plus the included packaging; technically, mass under a definition that includes the stated container and packing materials.
4. **Tare mass:** the packaging amount taken away; technically, the measured or assigned container mass used in a net-mass calculation.

4.2 Units and dimensional checks

■ 4.2.1 PLAIN — in simple words

1. A unit says how big one step in a number is. One kilogram and one gram are different steps for describing the same kind of quantity.
2. Converting units changes the number, not the object. A mass written as 1.25 kg is written as 1,250 g after conversion.

3. Units are also a way to check reasoning. Adding a mass to a duration does not become sensible just because a spreadsheet accepts both as numbers.
4. Two values can share a physical unit and still mean different things. A gram of packaging and a gram of goods have compatible dimensions, but whether they should be added depends on the question.

■ 4.2.2 PLAIN — a picture in your head

1. Think of the same distance drawn on two maps. One map uses centimetres on the page for kilometres outside. Another uses a different scale. The road has not moved.
2. A conversion factor is the rule connecting the scales. Forget that rule and identical-looking numbers describe different distances.
3. **Where this comparison breaks:** many unit conversions are simple multiplication, but not all are. Temperature scales can include an offset, and a currency exchange rate is not a fixed physical conversion factor.

■ 4.2.3 PLAIN — a worked example

1. Convert the parcel mass: $1.25 \text{ kg} \times 1,000 \text{ g/kg} = 1,250 \text{ g}$. The kilograms cancel in the written calculation, leaving grams.
2. Now convert an area. A rectangular label is 20 cm by 30 cm, so its area is 600 cm^2 .
3. One centimetre is 0.01 m. Therefore one square centimetre is $0.01 \text{ m} \times 0.01 \text{ m} = 0.0001 \text{ m}^2$.
4. The label area is consequently $600 \times 0.0001 = 0.06 \text{ m}^2$. Dividing by 100 instead of 10,000 would give the wrong area.
5. Finally, a synthetic request takes 400 ms. Since 1 ms is 0.001 s, its duration is 0.4 s . Recording the number 400 under a column labelled seconds creates a thousandfold error.

Calculation	Resulting unit	Interpretation
grams + grams	grams	A sum of compatible masses
grams / seconds	g/s	A mass flow rate, if the model supports it
requests / seconds	requests/s	A throughput measure under a stated request definition
centimetres × centimetres	cm ²	An area, not a length
order IDs + order IDs	None defined here	Arithmetic on these labels has no business meaning

■ 4.2.4 PLAIN — what is really happening inside

1. The computer usually stores a number separately from the unit's meaning. It will happily calculate $400 + 1$ even when the first number means milliseconds and the second means kilograms.
2. A safe boundary checks a quantity's kind and unit before combining it with another quantity. It converts accepted inputs into a chosen internal unit, while preserving enough metadata to explain that conversion.
3. SI prefixes specify decimal multipliers. For example, kilo is 10^3 , milli is 10^{-3} and micro is 10^{-6} . Their letter case matters: m and M are not interchangeable prefix symbols. [S27]
4. The shop's quantity field for O-1042 still counts individual notebooks or pens. It must not quietly become a mass field when the shop starts weighing parcels. Those are different contracts.

■ 4.2.5 TECHNICAL — the engineer's version

1. A **quantity equation** expresses a relationship independently of a chosen unit system. $\text{area} = \text{length} \times \text{width}$ remains meaningful whether lengths are expressed in metres or centimetres.
2. A **numerical value equation** may include conversion factors tied to the units actually used. $\text{area_m2} = \text{length_cm} * \text{width_cm} / 10000$ is one such implementation for our rectangular-label model.
3. Dimensional analysis can reject inconsistent operations, but it is not a complete semantic checker. Counts of orders and counts of lines are both dimensionless physical counts; their business grains remain different.
4. Consider a result labelled “average request duration.” Dividing total duration by request count gives a time per request. Dividing by successful-request count instead gives a different denominator unless only successful durations were included.
5. For a linear unit conversion $y = c \times x$, an uncertainty expressed in the same measurement sense scales by $\text{abs}(c)$ as well. Converting 1,000 g with standard uncertainty 2 g produces 1.000 kg with standard uncertainty 0.002 kg; it does not improve the measurement. [S31]
6. Store conversion rules deliberately. A measurement service might allow g and kg as input for mass, convert both to grams, and reject ms in that field. Rejecting the wrong dimension is better than applying a guessed conversion.
7. Standard, convention or implementation detail: the SI prefix multipliers are standardised; our decision to store parcel mass in grams is a design convention; the exact validation code is our implementation.

■ 4.2.6 WORDS — remember these

1. **Unit:** the agreed size of a numerical step; technically, a reference quantity used to express values of quantities of the same kind.
2. **Conversion factor:** the multiplier between compatible units; technically, the ratio used to transform numerical values while preserving the represented quantity.
3. **Dimension:** the physical kind of a quantity; technically, its expression in powers of the chosen base quantities, such as length or time.
4. **Dimensional analysis:** checking a calculation through its units; technically, checking compatibility of dimensions across an equation while recognising that semantics require further checks.

4.3 Precision versus accuracy

■ 4.3.1 PLAIN — in simple words

1. A scale can give nearly the same reading every time and still be consistently off target. Repeatability and correctness are not the same achievement.
2. A display with three decimal places can also be misleading. The extra digits show how finely the instrument reports a result, not how close the result is to the quantity being measured.
3. In everyday words, precision concerns how closely repeated measurements agree under stated conditions. Accuracy concerns closeness to the true value, which is often not directly available to us. [S28] [S29]
4. That is why a comparison needs a suitable reference and information about that reference's own uncertainty. One unverified scale is not automatically the truth against which another should be judged.

■ 4.3.2 PLAIN — a picture in your head

1. Imagine throwing paper balls into a bin. A tight cluster on the floor beside the bin is repeatable but displaced. A scattered set around the bin has a different problem.
2. The cluster's spread suggests one kind of improvement; its displacement suggests another. "Throw more times" does not necessarily move a consistently displaced cluster into the bin.
3. **Where this comparison breaks:** a bin gives a visible target. In measurement the true quantity is generally not known exactly. The reference procedure and its uncertainty must be part of the comparison.

■ 4.3.3 PLAIN — a worked example

1. For a separate synthetic instrument check, assign a reference value of **1,000 g**, with reference uncertainty set aside temporarily so we can inspect the arithmetic.
2. Scale A reads **1008, 1008, 1009, 1007, 1008 g**. Its average is **1,008 g**, which is 8 g above that assigned reference.
3. Scale B produces our M-BOX-01 readings: **998, 1000, 1001, 999, 1002 g**. Their sum is 5,000 g and their average is **1,000 g**.
4. Scale A's readings are more tightly grouped, while Scale B's average is closer to the assigned reference in this small exercise. Neither conclusion alone establishes a complete instrument specification.
5. For M-BOX-01, subtract the mean from each reading: **-2, 0, 1, -1, 2 g**. Square those differences and add: $4 + 0 + 1 + 1 + 4 = 10 \text{ g}^2$.
6. Divide by $n - 1 = 4$, then take the square root. The sample standard deviation is $\text{sqrt}(10/4) \approx 1.581 \text{ g}$.
7. This spread describes the five readings under the stated conditions. It is not a certificate that the box's mass lies within 1.581 g of 1,000 g.

■ 4.3.4 PLAIN — what is really happening inside

1. Repeated observations can vary because of repositioning, environmental changes, instrument behaviour and other effects. Some effects change between readings; others are shared by all of them.
2. Averaging may reduce the effect of independent, zero-centred variation. It does not automatically remove a shared offset, such as a scale that consistently reads too high.
3. A calibration establishes a relationship between indications and reference values with associated uncertainties. Adjusting the instrument changes it. These are connected activities, but not synonyms. [S30]
4. When storing an average, also retain the sample count, relevant spread and method information where justified. Without them, another reader cannot tell whether "1,000 g" came from one reading or a carefully defined repeated-measurement procedure.

■ 4.3.5 TECHNICAL — the engineer's version

1. For observations $x_1 \dots x_n$, the sample mean is $\bar{x} = \text{sum}(x_i) / n$. The sample variance is $s^2 = \text{sum}((x_i - \bar{x})^2) / (n - 1)$, requiring at least two observations.
2. Under an independent, identically distributed model with a finite common variance, this sample variance estimates the variance of individual observations. Independence and stable conditions are assumptions to investigate, not consequences of collecting many rows.

3. Under the corresponding independent-observation model, the estimated standard uncertainty of the mean is $s / \sqrt{n}$. For M-BOX-01 it is $1.58113883 / \sqrt{5} \approx 0.70710678$ g. [S32]
4. Do not divide an individual reading's uncertainty by $\sqrt{n}$ merely because there are n rows somewhere in the database. The reduction applies to the particular average under its statistical model, not to every future box.
5. If all readings share an uncertain offset, write each as $x_i = m + b + e_i$: the target mass m , shared offset b and varying component e_i . Averaging reduces the independent part of the e_i terms, while the shared b remains.
6. The VIM treats measurement accuracy as a qualitative concept, not a quantity assigned a numerical value. A statement such as "accuracy = 99.9%" needs an independently defined metric; it is not the formal definition of measurement accuracy. [S28]
7. **Resolution** concerns the smallest change that can be distinguished in the stated context. Quantisation to a whole gram can hide smaller changes, while an instrument displaying tenths of a gram can still have an error larger than one gram. [S33]
8. The honest version: five neat values do not establish long-term stability, independence, freedom from bias or performance across an instrument's entire range.

■ 4.3.6 WORDS — remember these

1. **Precision:** repeated readings agreeing closely; technically, closeness among replicate indications or measured values under specified conditions.
2. **Accuracy:** closeness to what is actually being measured; technically, closeness of agreement between a measured value and a true value of the measurand, not a numerical quantity in VIM usage.
3. **Sample standard deviation:** the observed spread of a sample; technically, the square root of the sum of squared deviations from its mean divided by $n - 1$.
4. **Resolution:** the smallest distinguishable change; technically, the change in the measured quantity that causes a perceptible change in the corresponding indication under the stated conditions.
5. **Calibration:** comparing indications with suitable references; technically, establishing the specified relationship between reference values and indications with their associated uncertainties.

4.4 Rounding and significant figures

■ 4.4.1 PLAIN — in simple words

1. Rounding shortens a representation. It does not improve a measurement and can discard information needed by a later calculation.
2. A rounding rule needs both a target precision and a way to handle values exactly halfway between two choices.
3. Do not let the display format silently become the stored evidence. A report may display one decimal place while the controlled calculation retains more digits.
4. Extra digits should not imply extra confidence. A computer can calculate a very long decimal from uncertain inputs; the input uncertainty does not disappear inside the arithmetic.

■ 4.4.2 PLAIN — a picture in your head

1. Think of cutting a length of ribbon to the nearest marked centimetre. Once you cut it, the discarded part cannot be recovered from the new length alone.
2. Rounding at every intermediate step is like trimming the ribbon repeatedly before knowing the final use. The accumulated change may matter.
3. **Where this comparison breaks:** software can preserve both the original value and its rounded display. The loss occurs when a rounded value replaces the information required for later reasoning.

■ 4.4.3 PLAIN — a worked example

1. Use exact decimal inputs for this exercise, not binary floating-point approximations. Round each to two decimal places.

Exact input	Half-even	Half-up
2.345	2.34	2.35
2.355	2.36	2.36
1.245	1.24	1.25

2. In half-even, a halfway case chooses the neighbour with an even final retained digit. In half-up, a halfway case is rounded away from zero. The examples above are positive. [S13]
3. Take two amounts of **1.245**. Rounding each half-up first gives $1.25 + 1.25 = 2.50$.
4. Adding first gives $1.245 + 1.245 = 2.490$; rounding that sum gives **2.49**. The difference is 0.01, entirely explained by the location of rounding.
5. A business or measurement procedure must choose which operation it intends. “Both used rounding” does not make the two pipelines equivalent.

```
from decimal import Decimal, ROUND_HALF_EVEN, ROUND_HALF_UP

value = Decimal("2.345")
step = Decimal("0.01")
print(value.quantize(step, rounding=ROUND_HALF_EVEN)) # 2.34
print(value.quantize(step, rounding=ROUND_HALF_UP))   # 2.35
```

■ 4.4.4 PLAIN — what is really happening inside

1. Quantisation maps many possible inputs to a smaller set of reported values. Several nearby masses may all appear as 1,000 g on a whole-gram display.
2. A calculation cannot determine which of those original values occurred from the rounded reading alone. It needs more evidence or an explicitly stated model.
3. Significant figures are a reporting convention for indicating the digits being retained. The spelling 1000 by itself does not establish the uncertainty, measurement method or number of reliable digits.
4. Prefer an explicit result with its unit and uncertainty statement where those distinctions matter. A value written as 1.000 kg without further information is still not a complete measurement report.

■ 4.4.5 TECHNICAL — the engineer's version

1. In the Python decimal model, quantize produces a value with the exponent of the supplied operand, subject to the decimal context. Constructing Decimal from the string "2.345" avoids importing a prior binary floating-point approximation. [S13]

2. The precision of a decimal arithmetic context is not the physical precision of an instrument. One controls numerical representation and arithmetic; the other concerns repeated measurement behaviour.
3. Write the pipeline explicitly: raw observation → checked unit → calculation → final rounding → display. Record any mandatory intermediate rounding as part of the calculation's definition.
4. For an ideal rounding-to-nearest device with step q , the rounding error lies within a half-step interval under that model. Modelling it as uniformly distributed gives standard uncertainty $q / \sqrt{12}$, because the half-width is $q/2$. [S34]
5. With $q = 1$ g, that component is about **0.288675 g**. This is an assumed quantisation model, not a measured total uncertainty for every one-gram scale.
6. Do not add that component again if the same effect has already been included in another uncertainty estimate. A longer uncertainty budget is not automatically a better one; correlated or duplicated components can misstate the result.
7. For this book, retain sufficient intermediate digits and round the final uncertainty and result together. The exact reporting rule belongs to the stated procedure; a blanket “always keep two decimals” is not a substitute.

■ 4.4.6 WORDS — remember these

1. **Rounding:** reporting a nearby value on a chosen scale; technically, mapping a value to a specified representable set under a tie-breaking rule.
2. **Half-even:** choosing the even final digit at a tie; technically, rounding halfway cases to the nearest value whose retained least significant digit is even.
3. **Quantisation:** replacing a continuous range with steps; technically, mapping input values to discrete output levels.
4. **Significant figures:** the digits deliberately retained in reporting; technically, a numerical presentation convention that does not alone specify a full uncertainty model.

4.5 Sampling and missing observations

■ 4.5.1 PLAIN — in simple words

1. The records you can see may not represent everything you want to describe. Busy periods, broken devices and failed requests can be missing for reasons connected to the result.
2. A report of successful operations is not automatically a report of all attempted operations. The excluded cases may be exactly the cases that most need attention.
3. More rows can make an answer more repeatable while leaving its selection problem untouched. A million readings from the wrong population still describe the wrong population.
4. Always name what was eligible, what was observed and what was excluded before interpreting the average.

■ 4.5.2 PLAIN — a picture in your head

1. Suppose Mira asks only customers who completed a purchase whether checkout was easy. People who abandoned a long queue are absent from the answers.
2. The survey may accurately describe the people who answered. The problem arises when the headline claims to describe everyone who tried to buy something.

3. **Where this comparison breaks:** missing telemetry is not always a voluntary decision like leaving a queue. It can arise from an instrument fault, a filtering rule or a storage failure. The cause must be investigated rather than assumed.

■ 4.5.3 PLAIN — a worked example

1. Introduce a separate synthetic timing fixture: **20 requests** are attempted. Twelve complete successfully, with completed-request durations summing to **24 seconds**. Eight reach a **10-second timeout** without an observed completion.
2. The average among successful completions is $24 / 12 = 2$ **seconds**. That is a legitimate descriptive result for those twelve observations.
3. It is not the average completion time of all twenty requests. The eight missing completion times cannot be replaced by zero.
4. If each timed-out request eventually completes, and the timing definition confirms it had not completed within those ten seconds, its completion duration is at least ten seconds. Under those assumptions, the all-request average completion duration is at least $(24 + 8 \times 10) / 20 = 5.2$ **seconds**.
5. Some requests might never complete. In that case a finite mean completion duration for all twenty is not established at all.
6. The report should therefore separate **20 attempts, 12 observed successes, 8 timeouts and 2 seconds mean among successes**, rather than compressing them into “average request time: 2 seconds.”

■ 4.5.4 PLAIN — what is really happening inside

1. A collection pipeline decides which events become records. A sensor may sample once a minute. An application may log only successful responses. A report may remove rows with missing values.
2. Each choice changes the set of observations available to the calculation. A database cannot infer all the missing events simply because it can count the rows that arrived.
3. Distinguish a missing value from a missing row. A row marked `timed_out` preserves evidence of an attempt. A pipeline that drops the whole attempt hides even the denominator.
4. In our shop’s reports, inclusion and exclusion counts will travel with the result. That is a design rule for this case study, not a claim that every analytical product supplies those counts automatically.

■ 4.5.5 TECHNICAL — the engineer’s version

1. Separate the **target population**, the **sampling frame**, the selected sample and the actually observed sample. The first is what the question concerns; the others describe how evidence was acquired.
2. Missing observations can depend on the measurement itself. Slow operations are especially likely to be absent from a dataset containing only successes observed before a deadline. Treating that dataset as a random sample of all durations is an additional assumption, and a poor one for this fixture.
3. The unknown completion durations in the example are **right-censored** at ten seconds under the stated observation protocol: we have a lower boundary, not an exact completion value. The toy lower-bound calculation does not fit a survival model.

4. A client's observed waiting time is a different metric. If it deliberately stops waiting at ten seconds, its recorded wait can be ten seconds even though server-side completion is later or never observed. State which clock and endpoint the metric describes.
5. Weighting also changes a question. Giving every branch equal weight estimates an average of branch results; weighting by each branch's request count estimates a request-level result under compatible definitions. Neither choice is universally correct.
6. SQL aggregate functions commonly distinguish missing values from observed zero. Our earlier SQLite fixture illustrates the practical consequence: omitting an unknown from a mean and treating it as zero give different answers. Reuse that lesson rather than hiding the exclusion behind a clean chart. [S19] [S24]
7. A useful report contract contains numerator, denominator, time window, time basis, missing-value treatment and exclusions. Later Chapter 46 will develop selection bias and analytical interpretation further.

■ 4.5.6 WORDS — remember these

1. **Target population:** everything the question is meant to describe; technically, the defined set of units about which an inference or descriptive claim is intended.
2. **Sampling frame:** the set from which observations can be selected; technically, the operational list or mechanism used to reach candidate sample units.
3. **Selection bias:** the selection process distorting the intended picture; technically, systematic differences caused by how observations enter the analysed sample relative to the target population.
4. **Censored observation:** a boundary rather than an exact value; technically, partial information restricting a value to a range, such as an unobserved completion time known to exceed a deadline.

4.6 Carrying uncertainty into a report

■ 4.6.1 PLAIN — in simple words

1. An uncertainty statement describes the spread of values reasonably associated with a measurement under the available information. It is not an admission that the measurement was careless. [S35]
2. Different sources of uncertainty need to be considered together. Repeating a reading may reduce one component while leaving another unchanged.
3. The meaning of "plus or minus" must be stated. A standard uncertainty, an expanded uncertainty and a guaranteed physical limit are different statements.
4. An honest report contains both a result and the boundary of what the evidence supports. A narrow interval produced from unjustified assumptions is not stronger evidence than a wider, well-supported one.

■ 4.6.2 PLAIN — a picture in your head

1. Imagine locating a shop on a map using two clues. One clue has a small uncertainty about the street position; another has an uncertainty about where the whole map was aligned.
2. Repeating the first clue does not automatically fix the map's alignment. Some uncertainty is local, and some is shared.

ONE BOX. FIVE READINGS. TWO DIFFERENT QUESTIONS.

M-BOX-01 · repeated indications in grams · all values are synthetic

998 g	1000 g	1001 g	999 g	1002 g
Mean of these indications				
1000.0 g				
Sample standard deviation				
about 1.5811 g				
Standard uncertainty of the mean				
about 0.7071 g				

The mean does not remove calibration uncertainty or prove accuracy.

The stated uncertainty model assumes independent repeat readings.

It does not treat these values as five different boxes.

Figure 4.4: Figure 4.1. Keep the object, procedure, readings and uncertainty model attached to the result. The arithmetic is only one stage of the measurement record.

3. **Where this comparison breaks:** uncertainties are not simply distances that should always be added. Their combination depends on the mathematical model and on how the contributing effects are related.

■ 4.6.3 PLAIN — a worked example

1. Return to M-BOX-01. The mean indication is **1,000 g**. Under the repeated-independent-observation model, the standard uncertainty of the mean from those readings is **0.70710678 g**.
2. For this calculation, assume a separate calibration contribution with **standard uncertainty 1.5 g**, uncorrelated with the first component. Assume any required correction has already been applied and that the two components cover the effects included in this toy budget.
3. Square, add, then take the square root:

```
u_repeatability = 0.70710678 g
u_calibration   = 1.5 g

u_combined = sqrt(0.70710678^2 + 1.5^2)
            = sqrt(0.5 + 2.25)
            = sqrt(2.75)
            = 1.65831240 g
```

4. If the reporting procedure uses coverage factor **k = 2**, the expanded uncertainty is **U = 2 × 1.65831240 = 3.31662479 g**.
5. One rounded report is **mass estimate 1,000.0 g; expanded uncertainty 3.3 g; k = 2**, with the observation procedure and the assumptions stated. This is an illustrative uncertainty calculation, not a real calibration certificate.
6. The association of **k = 2** with approximately 95% coverage needs an appropriate approximately normal model and a reliable uncertainty estimate. The multiplier alone does not establish that coverage. [S36]

■ 4.6.4 PLAIN — what is really happening inside

1. The report is applying a measurement model to input estimates and their uncertainty information. That information can come from statistical analysis or from other justified evidence.
2. A Type A evaluation uses a statistical analysis of observations. A Type B evaluation uses other available information, such as a calibration report or a specified distribution. These labels describe evaluation methods, not “good” versus “bad” uncertainty. [S32] [S34]
3. The model must account for shared effects. When subtracting a tare reading from a gross reading made on the same scale, some offset effects may be shared and may partly cancel. They should not automatically be treated as independent.
4. The software should keep enough of the uncertainty budget to explain the published result. Storing only `plus_minus = 3.3` discards the unit, coverage factor, components and assumptions that make the number meaningful.

■ 4.6.5 TECHNICAL — the engineer’s version

1. For a result $y = f(x_1, \dots, x_n)$, first-order uncertainty propagation uses sensitivities of f to the input values, their variances and their covariances. A sensitivity is how much the result changes for a small change in an input. [S31]
2. For an uncorrelated sum $y = a_1x_1 + a_2x_2$, the combined variance is $a_1^2u_1^2 + a_2^2u_2^2$. Taking the square root gives the combined standard uncertainty. This is not the same operation as adding worst-case bounds.
3. For $\text{net} = \text{gross} - \text{tare}$, the general two-input expression is $u_{\text{net}}^2 = u_{\text{gross}}^2 + u_{\text{tare}}^2 - 2\text{cov}(\text{gross}, \text{tare})$. Ignoring the covariance assumes something about the relationship between the readings.
4. In a separate illustrative budget, standard uncertainties of 3 g and 4 g produce 5 g for the difference only when the covariance term is zero. A shared uncertain offset does not satisfy that independence assumption merely because two rows were recorded.
5. Strong nonlinearity, awkward distributions, dependent inputs or small-sample effects can require more careful methods than a first-order approximation. This chapter introduces the model; it does not certify every use of its shortcut.
6. A machine-readable report could include `estimate`, `unit`, `standard_uncertainty`, `coverage_factor`, `expanded_uncertainty`, `method_version`, `input_observation_ids` and `assumptions`. Include only fields justified by the actual method, rather than filling unknown quantities with invented zeros.
7. The final quality test is interpretability: can another reader reproduce the arithmetic and distinguish what was observed from what was assumed? A long decimal without that distinction is merely a precise-looking string.

■ 4.6.6 WORDS — remember these

1. **Standard uncertainty:** uncertainty expressed like a standard deviation; technically, measurement uncertainty expressed as a standard deviation under the stated model.
2. **Combined standard uncertainty:** the uncertainty after input effects are combined; technically, the standard uncertainty of the result obtained from its input uncertainty model, including relevant dependence.
3. **Expanded uncertainty:** a stated multiple of standard uncertainty; technically, $U = k \times u_{\text{combined}}$, accompanied by the coverage factor and justified interpretation.

4. **Covariance:** how two quantities vary together; technically, a measure of joint variation that contributes to propagation when input estimates are dependent.
5. **Uncertainty budget:** the explanation behind the uncertainty figure; technically, a statement of contributing uncertainty components, their evaluation and their combination in the measurement model.

4.97 Practice and worked answers

■ First predict the result

1. An export contains $\text{mass} = 1.25$ without a unit. Can you safely compare it with a record containing $\text{mass_g} = 1250$?
2. A rectangular label is 25 cm by 40 cm. Calculate its area in square metres and show why a single division by 100 is insufficient.
3. Use M-BOX-01 to calculate the mean, sample standard deviation and standard uncertainty of the mean under the stated independent-observation model.
4. Two exact decimal inputs are 1.245 and 1.245. Compare half-up rounding to two decimal places before and after addition.
5. In the request-timing fixture, what does the 2-second average establish, and what does it leave unknown?
6. Combine independent standard uncertainties of 0.70710678 g and 1.5 g, then apply $k = 2$. Name one assumption that can invalidate a simple interpretation of the resulting interval.

■ Worked answers

1. No. The first record's unit and quantity definition are missing. It might be kilograms, grams or something else. A plausible match is not permission to rewrite the source. Obtain the contract or quarantine the ambiguous record.
2. The area is $25 \times 40 = 1,000 \text{ cm}^2$. Each square centimetre is 0.0001 m^2 , so the area is 0.1 m^2 . Both length dimensions change scale; therefore the factor is squared.
3. The mean is 1,000 g. Squared deviations sum to 10 g^2 , so the sample variance is $10/4 = 2.5 \text{ g}^2$ and $s \approx 1.58113883 \text{ g}$. Dividing by $\sqrt{5}$ gives about 0.70710678 g for the average's repeatability component, not its complete uncertainty.
4. Before addition: $1.25 + 1.25 = 2.50$. After addition: 2.490 rounds to 2.49. The procedure must specify where rounding occurs.
5. It describes the twelve observed successful completions. It does not establish the mean completion time of all twenty attempts. Under the stated eventual-completion assumptions the latter has a lower bound of 5.2 seconds; without eventual completion a finite all-request mean is not established.
6. The combined value is $\sqrt{2.75} \approx 1.65831240 \text{ g}$, and the expanded value is approximately 3.31662479 g. Unmodelled shared error, double-counted components or an unjustified distributional assumption can invalidate the intended interpretation.

4.98 Common wrong ideas

1. **Wrong:** every extra decimal place is extra accuracy. **Right:** displayed resolution and closeness to the measured quantity are different properties.
2. **Wrong:** a stable instrument must be correct. **Right:** it can repeat a shared offset consistently.

3. **Wrong:** an average of five readings is the same thing as a typical value for five products. **Right:** repeated measurements of one object and measurements across different objects answer different questions.
4. **Wrong:** units are decorative labels added at the end. **Right:** they determine interpretation and can expose invalid calculations before a report is produced.
5. **Wrong:** rounding each input and rounding the final result are interchangeable. **Right:** the order can change the answer.
6. **Wrong:** missing means zero. **Right:** replacing absence with zero introduces a substantive assumption.
7. **Wrong:** a low mean among successes proves the service is fast for everyone. **Right:** excluded timeouts can change the conclusion.
8. **Wrong:** all uncertainty components should be added as independent squares. **Right:** the model must account for dependence and avoid counting the same effect twice.
9. **Wrong:** $k = 2$ always guarantees 95% coverage. **Right:** that interpretation depends on the model and reliability of the uncertainty estimate.
10. **Wrong:** a report is complete once its arithmetic is correct. **Right:** it also needs the quantity definition, population, method, time boundary and limits of inference.

4.99 Chapter summary in 20 lines

1. Name the quantity intended to be measured before choosing a database field.
2. The measurand includes the object and conditions needed to distinguish the intended question.
3. Gross mass, tare mass and net mass have different definitions.
4. A recorded indication is not automatically a verified physical truth.
5. Units travel with values and determine meaningful calculations.
6. Converting a squared unit requires squaring its conversion factor.
7. Compatible dimensions do not guarantee compatible business meanings.
8. Precision concerns agreement among repeated observations under stated conditions.
9. Accuracy concerns closeness to the true value and is not itself a numerical percentage in VIM usage.
10. A detailed display does not prove a small measurement error.
11. The M-BOX-01 readings have mean 1,000 g and sample standard deviation about 1.581 g.
12. Under the independent-observation model, their mean's repeatability uncertainty is about 0.707 g.
13. Averaging does not automatically remove shared offsets.
14. Rounding needs a scale, a tie rule and a defined place in the calculation.
15. Preserve missing observations rather than silently converting them to zero.
16. Report denominators and exclusions before generalising an average.
17. Uncertainty components need a stated model and a check for dependence.
18. The example's independent components combine to about 1.658 g.
19. Multiplying by $k = 2$ gives about 3.317 g, with coverage interpretation conditional on assumptions.
20. A useful measurement report lets the next reader separate observations, calculations and assumptions.

Identity, Keys and the Problem of Duplicates

5.0 What this chapter gives you

1. You will distinguish a thing, its record, its display name and the identifier used to refer to it.
2. You will choose a key by examining its scope and lifetime, not merely whether its sample values look different.
3. You will explain why generating an identifier and enforcing uniqueness are separate jobs.
4. You will separate a repeated delivery of one event from a second genuine event with similar contents.
5. You will design a reviewable matching process that can preserve an uncertain outcome instead of pretending every pair is a match or a non-match.
6. You will preserve enough history to explain and reverse a mistaken identity merge without quietly rewriting unrelated records.

■ Keep the meanings separate

This chapter concerns **record identity**: which product, order, source record or event a reference names. It is not a complete treatment of proving who a person is or deciding what they may access. Those are distinct problems. The existence of a database key is not authentication, and knowing that key is not authorisation.

The established shop fixtures remain unchanged. O-1042 still has two order lines; its first line is identified by (O-1042, 1). Multi-branch and matching examples below are new synthetic exercises. They do not assert that existing customers have been merged or that the shop has already deployed a multi-tenant service.

5.1 Names and identifiers

■ 5.1.1 PLAIN — in simple words

1. A name helps a person recognise something. An identifier helps a system refer to the intended thing under an agreed rule. Sometimes one value does both jobs, but the jobs remain different.
2. Two products can both be called “Blue Notebook.” A product can also change its display name without becoming a different product in the shop’s catalogue.
3. The system therefore needs a rule for what stays the same through a change. A product identifier might remain stable while its label, description or supplier changes.
4. That stability is a design decision. Software cannot discover it merely by noticing that two strings look similar.

■ 5.1.2 PLAIN — a picture in your head

1. Imagine a room of people wearing name badges. Two badges say “Dev.” Calling out the name does not identify one person unambiguously.
2. Now give each visitor a numbered ticket for this event. The ticket can identify a particular admission record even when several visitors share a name.
3. **Where this comparison breaks:** the ticket identifies an admission in its issuing system, not necessarily one person for life. A person can have two tickets, and a stolen ticket does not prove the holder’s identity.

■ 5.1.3 PLAIN — a worked example

1. In our original catalogue, **P-NOTE** identifies the notebook product used by O-1042. Its agreed unit price in that order is 7,550 paise.
2. Suppose a new display label is proposed: “Ruled Notebook.” Under this teaching policy the product identity remains P-NOTE. The historical order still points to the same product and retains its own agreed price.
3. Now introduce a distinct catalogue item, **P-NOTE-A5**, whose display label also contains “Notebook.” Similar words do not justify replacing P-NOTE with P-NOTE-A5 in old orders.
4. Finally, the supplier’s catalogue might call our P-NOTE **NB-17**. That creates a mapping between identifier systems; it does not make NB-17 universally meaningful.

Reference	What it names in this example	What it does not establish
P-NOTE	One product in our catalogue	A physical individual notebook
O-1042	One order in our shop fixture	Payment settlement or delivery by itself
O-1042, line 1	One agreed order line	Every line containing the same product
NB-17 under supplier SUP-A	A supplier’s product reference	The meaning of NB-17 at another supplier

■ 5.1.4 PLAIN — what is really happening inside

1. A reference points into an identifier space. The receiving system looks up that reference and interprets the returned record under the correct contract.
2. When a human-readable label changes, a stable identifier can keep references connected. When an identifier changes, the system needs a deliberate migration or mapping policy.
3. A record ID should also have a lifecycle rule. Can it be reused after deletion? Can it be imported from another system? Can it survive a merge? Those choices affect old links, delayed messages and historical reports.
4. For our teaching catalogue, retired product identifiers will not be reassigned to different products. That is an explicit policy for the story, not a universal property of the word “identifier.”

■ 5.1.5 TECHNICAL — the engineer’s version

1. Distinguish an **entity** from its representations. A product can have a current catalogue record, an export row and historical order-line references. These records do not all have the same grain or identifier.

2. A **key** identifies a record under the model's uniqueness rules. A primary key is the selected identifying key of a table; other candidate keys can be enforced independently. PostgreSQL primary keys require uniqueness and non-null values. [S02]
3. A displayed name is not a candidate key unless the domain actually guarantees uniqueness and the schema enforces the intended rule. A current dataset containing no duplicate names is only an observation about that dataset.
4. Consider mutability. An email address may change, be shared or later be reassigned. Whether it is suitable for a particular key depends on the domain contract and lifecycle; a convenient login field is not automatically a stable entity identity.
5. Store external identifiers with their authority or namespace. A useful mapping grain is one external identifier under one issuing namespace, linked to an internal record under a documented validity policy.
6. The honest version: adding a generated key does not solve every identity problem. It guarantees neither that only one record represents each real-world entity nor that an existing reference was attached to the correct entity.

■ 5.1.6 WORDS — remember these

1. **Entity:** the thing the model is about; technically, a distinguishable object, concept or occurrence represented within a domain model.
2. **Identifier:** a value used to refer to something; technically, a designation interpreted within an issuing namespace and its lifecycle rules.
3. **Display name:** the label shown to a person; technically, a presentation attribute that need not be unique or stable.
4. **Primary key:** the table's chosen identifying key; technically, a selected set of columns whose combined values uniquely identify each row and cannot be null under the relevant database rules.

5.2 Natural and generated keys

■ 5.2.1 PLAIN — in simple words

1. Some identifiers already exist in the work being modelled. Others are created by the new system solely to name its records.
2. An existing catalogue code can be useful when its issuing organisation maintains it consistently. A generated number can be useful when ordinary attributes are mutable or ambiguous.
3. Neither option is automatically correct. Ask who issues the value, whether it can change, where it must be unique and what happens when records arrive from elsewhere.
4. A machine that generates new numbers is not the same thing as a rule that refuses duplicate stored numbers. Good designs examine both.

■ 5.2.2 PLAIN — a picture in your head

1. Imagine organising books using the number already printed by the publisher, or attaching your library's own accession label to each copy.
2. The publisher's number and the library label can both be useful because they name things at different levels. One edition can have many physical copies.

3. **Where this comparison breaks:** database key terminology depends on the model. A value that looks “natural” in one system may have been generated by another. The important question is what it identifies and what rules govern it.

■ 5.2.3 PLAIN — a worked example

1. In a new supplier-import exercise, SUP-A supplies product code 017, while SUP-B also supplies product code 017.
2. Treating 017 alone as a globally unique product key incorrectly collides two issuing systems.
3. The pair (SUP-A, 017) differs from (SUP-B, 017). An internal product record can have a separate generated ID while preserving either mapping as evidence.
4. A second import of (SUP-A, 017) might be an update to the same supplier reference, not a new product. Its meaning depends on that supplier’s versioning and reuse policy.
5. A generated internal ID therefore solves the naming of the receiving row, while the supplier mapping solves a different problem: recognising which source reference the row represents.

```
-- PostgreSQL 17 illustration; not an executed PostgreSQL lab.
CREATE TABLE supplier_reference_example (
    reference_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    supplier_id text NOT NULL,
    supplier_code text NOT NULL,
    UNIQUE (supplier_id, supplier_code)
);
```

6. Read the two rules separately: the primary key identifies the receiving row, while the unique pair prevents two current rows from claiming the same supplier reference in this deliberately narrow model.

■ 5.2.4 PLAIN — what is really happening inside

1. A sequence-like generator allocates values. A uniqueness constraint checks whether the relevant values conflict with accepted rows. The generator and the constraint can be configured or bypassed differently.
2. PostgreSQL documents that an identity column is not, by itself, a uniqueness guarantee. The underlying sequence can be reset, and explicit values can be inserted under permitted rules. A primary-key or unique constraint is still needed for uniqueness. [S37]
3. Another approach uses a large identifier format designed for distributed generation, such as a UUID. This reduces the need for every issuer to ask one shared counter, but it still needs correct generation and handling. [S38]
4. Choosing a large identifier does not remove the need to reject reused IDs, preserve namespaces or distinguish duplicate source records from duplicate real-world entities.

■ 5.2.5 TECHNICAL — the engineer’s version

1. A **natural key** is derived from domain attributes or identifiers already meaningful in the modelled work. A **surrogate key** is introduced to identify a record without carrying the same business meaning. These are modelling categories, not quality rankings.
2. A natural key may be long, composite or mutable. A surrogate key can provide a compact reference, but it can also permit duplicate business records unless relevant alternate uniqueness rules remain enforced.

3. In UUID version 4, 128 total bits include fixed version and variant fields, leaving **122 random bits** in the specified layout. The collision model therefore has 2^{122} possible random values, not 2^{128} . [S38]
4. Under independent uniform generation, the approximate probability of at least one collision among n IDs is $n*(n-1)/(2*N)$ when this probability is small and N is the number of possible values. This is the birthday approximation, derived by counting possible pairs.
5. With $n = 1,000,000,000$ and $N = 2^{122}$, the approximation is about 9.4×10^{-20} . This is an arithmetic illustration under the stated random model, not an observed failure rate of a deployed system.
6. Broken randomness, copied state, accidental identifier reuse and incorrect imports are outside that ideal collision calculation. Extremely small random-collision probability does not prove correct implementation.
7. Different UUID versions encode different information. A time-ordered identifier is not automatically a globally authoritative event order or an authentication token. Interpret the actual version rather than assuming every UUID has the same meaning. [S38]
8. Query patterns and storage costs matter too, but only after identity requirements are defined. Later index and storage chapters will examine the costs of different key layouts without turning them into universal “always use this type” rules.

■ 5.2.6 WORDS — remember these

1. **Natural key:** a key drawn from the work’s own attributes; technically, an identifying set of domain attributes rather than a record-only identifier introduced by the receiving model.
2. **Surrogate key:** an extra identifier created for the record; technically, a key whose principal role is row identity rather than existing business meaning.
3. **Unique constraint:** a rule against repeated key values; technically, database enforcement that the relevant value combination cannot appear in conflicting rows under its null and comparison rules.
4. **UUID:** a standardised large identifier; technically, a 128-bit identifier whose version and variant determine the interpretation and generation rules.

5.3 Scope and composite identity

■ 5.3.1 PLAIN — in simple words

1. “Number 17” is incomplete when two branches each issue their own number 17. The missing information is the scope in which the number was issued.
2. A composite identity uses several fields together. The combination identifies the intended record even when none of the fields is unique alone.
3. An order-line number is a familiar example. Line 1 appears in many orders. The order identifier plus the line number names the intended line.
4. Do not lose the scope while exporting, joining or constructing an address. Keeping it in the database but dropping it in the next message reintroduces the ambiguity.

■ 5.3.2 PLAIN — a picture in your head

1. Two apartment buildings can both contain flat 17. A courier needs the building and the flat, not just the number painted on the door.

THE SAME NUMBER CAN NAME DIFFERENT RECORDS

Receipt identity in this example includes the branch.

Branch BR-A

Local receipt number: 17

Amount: 500 paise

Branch BR-B

Local receipt number: 17

Amount: 900 paise

BR-A / 17 is not BR-B / 17

Comparing only 17 discards the scope that makes each key meaningful.

Similar content is not enough to establish duplicate identity.

Figure 5.5: Figure 5.1. The same local number can be valid in two namespaces. Keep the full identifier together through lookups, messages and joins.

2. The full address works because its pieces belong together. Sorting the pieces independently and pairing them again can create an address that was never supplied.
3. **Where this comparison breaks:** a database scope need not be a physical place. It might be a tenant, issuing authority, dataset version or parent record.

■ 5.3.3 PLAIN — a worked example

1. Our established order lines include **(O-1042, 1)** and **(O-1043, 1)**. They share a line number but not an identity.
2. Now use a separate branch-local receipt example: **(BR-A, 17)** and **(BR-B, 17)**. Both are legitimate under the policy that each branch numbers its own receipts.
3. A lookup using only `local_receipt_no = 17` may find two rows. Selecting the first one is not a valid resolution rule.
4. A lookup using both fields expresses the complete key: `branch_id = 'BR-A' AND local_receipt_no = 17`.
5. The same issue appears in joins. Joining receipt details to headers only by local receipt number can pair one branch's details with another branch's header.

Branch	Local receipt	Amount in this separate fixture
BR-A	17	500 paise
BR-B	17	900 paise

6. The two amounts are invented for this exercise and do not replace either canonical order total. The example exists to reveal the missing key component.

■ 5.3.4 PLAIN — what is really happening inside

1. A database can enforce a rule over a column combination. The combination is checked as a whole; the individual columns need not each be globally unique. [S02]

2. The application must then carry the same combination through its workflows. A user-interface link that drops the branch component undermines the model even when the database schema is correct.
3. The record's parent relationship may also be enforced by a foreign key. That checks the relevant stored relationship; it does not decide whether the current caller is allowed to read the parent or child.
4. The design therefore has two independent questions: **which record is this?** and **may this caller perform this action on it?** A complete key answers only the first.

■ 5.3.5 TECHNICAL — the engineer's version

1. A composite primary key such as (branch_id, local_receipt_no) identifies a row by the ordered set of its component meanings. Referencing tables should preserve the relevant scope in their foreign-key relationship.
2. If a local number is unique only within a branch, enforcing a foreign key on the number alone does not express the intended relationship. Where a database rejects that incomplete reference, the error is useful feedback about the model.
3. The order-line grain remains (order_id, line_no) in our original single-shop fixture. A future multi-tenant design might need a tenant component or a separately guaranteed global order ID. That is a future schema decision, not a silent reinterpretation of existing examples.
4. Avoid ambiguous string concatenation. The pairs ('AB', 'C') and ('A', 'BC') both become ABC when joined without boundaries. Use a structured representation or a documented escaping and length rule.
5. Nullability is also part of identity. A complete identifying key cannot rely on an absent parent component. Different databases have distinct null and uniqueness semantics, so state the actual constraints rather than relying on a label such as "unique." [S02]
6. A scope check is not a substitute for authorisation. In a tenant-scoped service, the server must connect the requested scope to the authenticated caller's allowed actions. Accepting a user-supplied tenant ID without that connection does not enforce isolation.
7. Preserve key components when generating reports and cache entries. A cache keyed only by a branch-local number can return a perfectly valid record belonging to the wrong branch.
8. The honest version: "the ID is unique" is an incomplete claim until you state **where, for what kind of record, over which lifetime, and under which enforcement rule.**

■ 5.3.6 WORDS — remember these

1. **Namespace:** the context in which an identifier has meaning; technically, an issuing or interpretive scope within which names are assigned and resolved.
2. **Composite key:** several fields identifying a row together; technically, a key consisting of more than one attribute.
3. **Foreign key:** a checked reference to another stored key; technically, a constraint requiring referencing values to correspond to an eligible referenced key, subject to the database's rules.
4. **Tenant scope:** the customer boundary in a shared service; technically, the domain boundary used to associate records and permitted operations with a tenant, requiring access control beyond identification.

5.4 Duplicate delivery versus repeated events

■ 5.4.1 PLAIN — in simple words

1. Receiving a message twice does not necessarily mean the underlying action happened twice. A sender may repeat delivery because the first acknowledgement was lost.
2. The reverse also matters: two real actions can have identical-looking contents. A customer can buy one pen now and another pen a minute later.
3. The system needs to distinguish the identity of an event from its contents and from the identity of one delivery attempt.
4. “Delete every repeated-looking row” is therefore not a safe general duplicate policy. It can remove genuine work or preserve a harmful replay, depending on what it compares.

■ 5.4.2 PLAIN — a picture in your head

1. Mira sends a photocopy of one signed dispatch note because the recipient says the first copy did not arrive. Two envelopes now carry evidence about one dispatch.
2. On another day she sends two separate dispatch notes for two separate parcels containing the same product. Similar descriptions now concern two events.
3. **Where this comparison breaks:** software can preserve an explicit source-and-event identifier and compare a replay against recorded content. A paper description may not contain those machine-checkable fields.

■ 5.4.3 PLAIN — a worked example

1. Introduce four synthetic messages from source `/miras-corner/till-a`. None is a new statement about the already established stock discrepancy.

Delivery	Event ID	Meaningful payload	Our proposed treatment
D-A	E-700	One pen issued	First accepted delivery of E-700
D-B	E-700	One pen issued	Replay of E-700; do not apply again
D-C	E-701	One pen issued	Distinct event; apply under its own rules
D-D	E-700	Three pens issued	Identifier conflict; investigate

2. D-A and D-B are two deliveries describing one event. D-C describes a second event even though its quantity matches.
3. D-D reuses E-700 with different meaningful content. Silently treating it as an ordinary replay would hide an inconsistency; applying it would reuse one identity for conflicting claims.
4. Our toy consumer will therefore distinguish **accepted**, **duplicate**, and **conflict**. The conflict is rejected rather than resolved by choosing whichever value arrived last.
5. The standard CloudEvents model similarly defines event uniqueness through the combination of source and id. Its specification permits a resent duplicate to retain that identity. [S39]

■ 5.4.4 PLAIN — what is really happening inside

1. The receiver checks whether it has already accepted the source-and-ID pair within the relevant retention and processing boundary.
2. If the pair is new, it validates the event and applies the intended update. If the pair is already present with the same agreed content, it returns the previous logical outcome without repeating the update.
3. If the pair has conflicting content, the receiver needs an explicit failure path. A producer bug, import mistake or hostile input must not become an unnoticed change to the original event.
4. Recording “seen” and applying the update are themselves related changes. In a persistent consumer they need appropriate coordination. Otherwise a failure between them can either lose an update or apply it twice.
5. The companion lab keeps these operations in a single in-memory routine. That illustrates the contract; it does not test a durable multi-process consumer.

■ 5.4.5 TECHNICAL — the engineer’s version

1. An **idempotent** operation produces the same intended state effect when repeated under its declared identity and conditions. Idempotency is a property of an operation’s semantics, not a magic attribute attached to a request header.
2. Distinguish four possible identifiers: the business object ID, the command/request ID, the accepted event ID and the delivery-attempt ID. A retry can preserve some while changing others. Their relationship must be documented.
3. CloudEvents 1.0.2 requires the source and id combination to identify distinct events. Its wire-level specversion value remains 1.0; the patch release number is not written as 1.0.2 in that attribute. [S39]
4. A uniqueness rule on (source, event_id) prevents storing two rows under the same identity, but does not alone prove that the first update and its deduplication record were committed together.
5. Compare meaningful content under a stable schema. Raw JSON text can differ in insignificant spacing or member order. Conversely, a comparison that drops a meaningful field can falsely classify conflicting events as equivalent. JSON representation and application semantics are separate boundaries. [S07]
6. A hash can accelerate a comparison or bind a retained representation, but it must be interpreted with its algorithm and canonicalisation rules. The lab compares a fixed tuple of validated domain fields rather than presenting an arbitrary JSON hash as semantic truth.
7. A deduplication record also has a lifetime. If it expires before a legitimate retry arrives, the consumer may treat the retry as new unless another durable business rule prevents the repeated effect. The guarantee must state its retention and recovery assumptions.
8. “Exactly once” is therefore incomplete without an effect boundary: once in a table, once in a projection, once in an email service or once in the physical world are different claims.

■ 5.4.6 WORDS — remember these

1. **Event identity:** the name of one accepted occurrence or change; technically, the identifier combination that distinguishes an event under the producer’s contract.
2. **Delivery attempt:** one effort to transport a message; technically, a transmission attempt that may repeat the same logical event.

3. **Idempotency:** repeating the operation without repeating its intended effect; technically, stability of the declared effect under repeated application within the operation's identity and failure model.
4. **Deduplication:** recognising a repeated identity; technically, detecting and handling duplicate deliveries or records under an explicitly defined equivalence and retention policy.
5. **Identity conflict:** one ID attached to incompatible content; technically, a violation of the mapping from an identifier to the immutable or otherwise controlled claim it is supposed to name.

5.5 Matching imperfect records

■ 5.5.1 PLAIN — in simple words

1. Sometimes two systems describe the same thing without sharing an identifier. Matching then becomes a reasoning problem rather than a simple key lookup.
2. Similar spelling is evidence, not certainty. Two supplier records can look alike and describe different products; one product can also appear under several spellings.
3. A useful matching process can say **same**, **different**, or **not enough evidence**. Forcing every uncertain pair into one of the first two categories hides the uncertainty instead of solving it.
4. The consequences matter. An incorrect suggestion is different from an automatic merge that changes orders, permissions or historical reports.

■ 5.5.2 PLAIN — a picture in your head

1. Imagine sorting two boxes of old catalogue cards. Some cards share an exact issuing code; others share only an abbreviated name and a partial description.
2. You can place uncertain pairs beside each other for review without gluing them into one card. That preserves the option to inspect more evidence.
3. **Where this comparison breaks:** a software merge may redirect thousands of relationships instantly. A convenient-looking shortcut can have effects far beyond the two rows visible on screen.

■ 5.5.3 PLAIN — a worked example

1. In an independent synthetic matching exercise, consider records **SRC-A-17**, labelled "Blue ruled notebook," and **SRC-B-88**, labelled "Blue notebook." Their names suggest a candidate pair, not a completed identity decision.
2. A checked common manufacturer code would be different evidence from name similarity, but even that requires understanding the code's issuing scope and reuse rules.
3. Suppose a labelled evaluation set contains **10 genuinely matching pairs** and **90 genuinely different pairs**. A candidate rule accepts 8 of the matches and incorrectly accepts 3 different pairs.
4. It has therefore accepted **11 pairs**, of which 8 are true matches. Its precision on this set is $8/11 \approx 72.7\%$.
5. It found **8 of the 10 true matches**, so its recall is **80%**. Its false-positive rate among the different pairs is $3/90 \approx 3.33\%$.
6. These are different denominators. None of the figures means there is a 72.7% probability that any particular new pair is the same product.

Label in the fixture	Accepted as a match	Not accepted as a match
Same product	8	2

Label in the fixture	Accepted as a match	Not accepted as a match
Different products	3	87

7. The matrix is invented to teach the arithmetic. It is not a performance claim for a real KedByte matching service.

■ 5.5.4 PLAIN — what is really happening inside

1. A matching workflow first prepares comparable features, then proposes candidates, evaluates evidence and applies a decision policy.
2. Normalising spaces or case can make comparison useful, but it also changes a representation. Preserve the original source value when needed to explain the decision, subject to the purpose and retention policy.
3. Unicode normalisation can make certain equivalent text sequences comparable. It does not prove that two named entities are the same, or that visually similar characters are safe to treat as identical. [S15]
4. A human-review queue should show why a pair was proposed and what remains uncertain. “The system says 0.92” is not enough when the score’s meaning and validation are unexplained.
5. For our teaching design, a candidate match has no authority to rewrite source records. Only an explicitly recorded decision can create a mapping, and that mapping remains reviewable.

■ 5.5.5 TECHNICAL — the engineer’s version

1. **Entity resolution** attempts to identify records that refer to the same underlying entity. It is different from deduplicating identical source event IDs: one compares uncertain referents, while the other can enforce a known producer identity contract.
2. Candidate generation, sometimes called blocking, reduces the number of pairs examined. It can improve computational efficiency while excluding some true matches. Evaluate that stage rather than measuring only the final scorer.
3. Precision is $\text{true_positives} / (\text{true_positives} + \text{false_positives})$. Recall is $\text{true_positives} / (\text{true_positives} + \text{false_negatives})$. Define how zero denominators are reported instead of quietly inventing a perfect score.
4. Performance measured on a labelled sample depends on the sample’s construction, label quality and relationship to the deployment population. A hand-selected easy test set does not establish production reliability.
5. A threshold selects a trade-off between missed matches and false matches. The appropriate policy depends on consequences, review capacity and available evidence. A score is not automatically a calibrated probability.
6. Separate matching from permissions. Linking two product records should not accidentally merge ownership rules. Linking person records is even more consequential and requires a separately designed access and decision process; this toy exercise does not implement one.
7. Preserve provenance for each accepted mapping: source record IDs, rule version, evidence references, decision outcome and the applicable time or version boundary. W3C PROV’s distinctions between entities, activities and agents help describe this reasoning without confusing a source fact with a derived decision. [S05]

8. The honest version: an algorithm that compares strings can be perfectly implemented and still make wrong identity decisions. Correct computation and justified inference are separate requirements.

■ 5.5.6 WORDS — remember these

1. **Entity resolution:** deciding which records concern the same thing; technically, inference over records that may lack a shared reliable identifier.
2. **Candidate match:** a pair worth investigating; technically, a proposed association that has not yet satisfied the final decision policy.
3. **Precision, matching:** the share of accepted matches that are correct; technically, true positives divided by all positive predictions, not measurement precision from Chapter 4.
4. **Recall:** the share of true matches found; technically, true positives divided by the actual positive cases under the evaluation labels.
5. **False positive:** a match accepted when the entities differ; technically, a positive decision for a negatively labelled case in the stated evaluation.

5.6 Merge, split and identity history

■ 5.6.1 PLAIN — in simple words

1. A merge says that records should now be treated as referring to one entity under a decision policy. It does not make every original value true or erase why the records differed.
2. A split reverses or revises that association. To do it well, the system needs to know which source records and relationships belonged to the disputed decision.
3. “Delete the duplicate row” is often too crude. It can discard the evidence needed to repair a wrong merge or explain an old report.
4. At the same time, keeping every detail forever is not the solution. Retain the information justified by the purpose and define what becomes impossible when particular evidence is removed.

■ 5.6.2 PLAIN — a picture in your head

1. Imagine attaching a removable note to two catalogue folders: “For current purchasing, treat these as one product group; decision M-01.” The folders still show where their original information came from.
2. If the decision proves wrong, you can remove the association and identify the work performed while it was active.
3. **Where this comparison breaks:** undoing an association does not automatically undo purchases, exports or messages already produced from it. Those downstream effects need their own review and correction.

■ 5.6.3 PLAIN — a worked example

1. Use a separate synthetic identity-history fixture with three source records: **SRC-A-17**, **SRC-B-88** and **SRC-B-89**. No real customer information is present.
2. Decision **M-01** associates SRC-A-17 and SRC-B-88 with canonical product group **CAN-10**. SRC-B-89 remains in **CAN-11**.

3. A later review finds that SRC-B-88 describes a different pack size. Decision **M-02** supersedes the earlier association for that source record and places it in **CAN-12**.
4. The original source IDs remain intact. A report can now distinguish “grouping under M-01” from “grouping under M-02” rather than quietly pretending the earlier interpretation never existed.
5. The corrected grouping does not automatically recalculate every previously exported report. The system must identify which outputs depended on M-01 and decide what correction or notification is appropriate.

Decision version	SRC-A-17	SRC-B-88	SRC-B-89
M-01	CAN-10	CAN-10	CAN-11
M-02	CAN-10	CAN-12	CAN-11

6. A small reversible mapping is therefore more informative than overwriting SRC-B-88’s identity and losing its source history.

■ 5.6.4 PLAIN — what is really happening inside

1. The source record and the current identity interpretation are separate layers. A mapping connects them under a decision version.
2. Updating the mapping changes what a current query resolves, but it does not change the original observation’s meaning. A historical query may intentionally use the older mapping version.
3. A safe change checks dependent records, invalidates or rebuilds affected derived views, and records the decision’s basis. Where the evidence is insufficient, the association can remain unresolved.
4. The process must also prevent impossible mappings, such as a chain that loops back to itself or one source assigned to two current canonical records when the domain permits only one.
5. These are proposed invariants for our exercise. The presence of a `canonical_id` column does not enforce them without the necessary constraints and change procedure.

■ 5.6.5 TECHNICAL — the engineer’s version

1. One modelling option is a versioned **crosswalk** from (`source_system`, `source_record_id`) to an internal canonical ID, accompanied by a decision identifier and status. It preserves both source identity and the receiving system’s interpretation.
2. Specify whether one source may map to several canonical entities. A bundled product might require a decomposition relationship rather than an ordinary identity mapping. Do not force a many-part relationship into a false “same thing” assertion.
3. Distinguish **effective time** from **recording time** when needed. A decision can be recorded today while correcting a grouping used for an earlier period. Adding two timestamp columns alone does not implement a complete bitemporal system.
4. Review reference redirection. A merged ID might become an alias rather than vanish; a split can make an old alias ambiguous. Returning an explicit unresolved or retired state can be safer than guessing a destination.
5. Rebuildable projections should depend on named mapping versions or retain sufficient lineage to determine which outputs need reconsideration. A current dashboard and an archived decision report need not use the same grouping policy.

6. Retention changes the evidence available for reversal. If original attributes or mapping history are removed, a later repair may no longer be reproducible. Document that limitation rather than promising perfect reversibility forever.
7. The companion lab demonstrates two immutable mapping snapshots and verifies that changing the current grouping does not mutate the earlier snapshot. It does not implement a complete production merge workflow, cascading database migration or person-identity system.
8. The lesson is architectural: identity decisions deserve explicit state, provenance and failure handling. They are not harmless string replacements.

■ 5.6.6 WORDS — remember these

1. **Canonical ID:** the receiving system's chosen reference; technically, an identifier used for the entity representation treated as the current reference under a stated mapping policy.
2. **Crosswalk:** a mapping between identifier systems; technically, a relation connecting scoped source identifiers with identifiers in another model.
3. **Merge decision:** an explicit association of records; technically, a versioned decision to treat selected representations as referring to one entity under stated evidence and rules.
4. **Split:** revising an association into distinct entities; technically, a correction of mapping relationships whose downstream effects may require separate repair.
5. **Identity history:** the record of how associations changed; technically, retained versions or events describing identifier mappings and their applicable decision boundaries.

5.97 Practice and worked answers

■ First predict the result

1. Two catalogue rows both have display name "Notebook." Does that prove one is a duplicate? What evidence is missing?
2. SUP-A and SUP-B both issue code 017. Propose a source-reference key and explain why a generated receiving-row ID solves a different problem.
3. A receipt join uses only local receipt number 17 and omits branch. Describe a possible wrong result using the two-branch fixture.
4. E-700 arrives twice with the same validated content; E-701 arrives once with identical business fields. How many logical events should our proposed consumer apply?
5. Recalculate precision, recall and false-positive rate for 8 true matches accepted, 2 true matches missed, 3 different pairs accepted and 87 different pairs rejected.
6. After M-02 separates SRC-B-88 from CAN-10, which statements can be made about the current grouping, the old grouping and reports previously exported under M-01?

■ Worked answers

1. No. A shared display name does not establish shared identity. Check product definitions, issuing identifiers, pack sizes, source provenance and the domain's equivalence rule. Keep an uncertain result when evidence is insufficient.

2. Use (supplier_id, supplier_code) under the stated supplier-code lifecycle. The generated receiving ID identifies the receiving row; the scoped pair recognises the source reference. Neither alone proves that two supplier references describe the same product.
3. BR-A's detail could be paired with BR-B's header, or a report could multiply the joined rows. Include the branch scope in the relationship and verify the resulting grain.
4. Apply E-700 once and E-701 once: two logical events. The second delivery of E-700 is a replay, while E-701 has a distinct producer identity. A changed payload under E-700 is a conflict, not a third ordinary event.
5. Precision is $8/(8+3) \approx 72.7\%$; recall is $8/(8+2) = 80\%$; false-positive rate is $3/(3+87) \approx 3.33\%$. Each uses a different denominator. These are fixture-level evaluation results, not individual-match probabilities.
6. The current mapping places SRC-B-88 in CAN-12. The retained M-01 snapshot still records the former association. Previously exported reports require a separate dependency and correction review; they are not automatically repaired by changing the current mapping.

5.98 Common wrong ideas

1. **Wrong:** equal names mean the same entity. **Right:** names can be shared, changed or incomplete.
2. **Wrong:** a generated key prevents duplicate business records. **Right:** it can give two duplicate business records two different IDs.
3. **Wrong:** a UUID cannot collide. **Right:** random-collision models assign a small probability under assumptions; implementation and reuse failures remain separate risks.
4. **Wrong:** an identity column is automatically unique. **Right:** generation and uniqueness enforcement are separate in PostgreSQL's documented model.
5. **Wrong:** an ID needs no context. **Right:** scope and lifecycle determine where its uniqueness claim applies.
6. **Wrong:** knowing the full key grants access. **Right:** identification and authorisation are separate decisions.
7. **Wrong:** identical contents always indicate a replay. **Right:** two real events can have identical business fields and distinct identities.
8. **Wrong:** a matching score is automatically a probability. **Right:** its interpretation requires calibration and validation appropriate to the intended population.
9. **Wrong:** merging records makes the source facts agree. **Right:** a merge is a decision about interpretation; contradictory source claims can remain.
10. **Wrong:** undoing a merge repairs every consequence. **Right:** downstream reports, exports and actions need their own correction process.

5.99 Chapter summary in 20 lines

1. An entity, its record, its display name and its identifier are different things.
2. Stable references depend on an explicit identity and lifecycle policy.
3. A primary key identifies a row within the relevant table's rules.
4. Natural keys come from domain attributes; surrogate keys are introduced for record identity.
5. Generated keys do not replace relevant business uniqueness constraints.
6. PostgreSQL identity columns need separate primary-key or unique enforcement for uniqueness.
7. UUID version 4 has 122 random bits within its 128-bit layout.

8. Collision arithmetic is conditional on correct independent uniform generation.
9. A namespace supplies the context required to interpret a local identifier.
10. A composite key keeps all necessary identifying components together.
11. Dropping scope in a join, cache key or message can select the wrong record.
12. Identification never substitutes for authorisation.
13. One event can have several delivery attempts.
14. Two distinct events can carry the same business contents.
15. Our consumer distinguishes accepted events, same-content duplicates and identity conflicts.
16. Deduplication guarantees depend on coordinated updates and retention boundaries.
17. Entity resolution evaluates uncertain references rather than merely comparing keys.
18. Matching precision and recall describe different denominators and are not measurement precision.
19. Versioned mappings preserve the distinction between source records and identity decisions.
20. A merge or split needs provenance, explicit invariants and review of downstream effects.

From Events to State: Keeping History

6.0 What this chapter gives you

1. You will distinguish a record of something happening from a value describing the current position.
2. You will reconstruct a notebook stock figure without treating a physical count as a movement of goods.
3. You will read a state-transition rule and explain why some requests must be rejected.
4. You will separate recorded order, physical time, delivery order and causal order.
5. You will understand what a snapshot saves, what a projection computes and what neither proves.
6. You will correct an earlier record without pretending that an external action has been undone.
7. You will describe the limits of reconstruction when records, interpretation rules or supporting evidence are no longer available.

The stock stream in this chapter is a synthetic continuation of Mira's Corner. The order 0-1042 still contains two notebooks and one pen, for INR 171.00. No price or quantity from that order is changed. A later count and an explicitly authorised stock adjustment are new teaching events. The separate order-state and reversal exercises use different identifiers so their histories cannot be mistaken for the original order.

6.1 An event versus a current value

■ 6.1.1 PLAIN — in simple words

1. A current value answers a question such as “how many notebooks does our stock record say we have now?” An event answers a different question: “what did we record as happening?”
2. “Ten notebooks” is a position. “Two notebooks were issued against this order” is a change that helps explain a position.
3. The current position alone does not tell us how we arrived there. Ten could result from twelve minus two, eight plus two, or a much longer history.
4. Keeping a history can make an answer explainable. It does not automatically make every entry in that history true. A mistaken count remains a mistaken count even when stored carefully. [S05]
5. We must also distinguish an observation from a movement. Looking at nine notebooks on a shelf does not itself remove one notebook from a record that expected ten.
6. A useful system can show both: “expected stock: ten” and “last observed count: nine”. It can then show the unresolved difference without inventing a reason for it.

■ 6.1.2 PLAIN — a picture in your head

1. Imagine a water tank with a level indicator and a notebook beside it. The indicator says how much water appears to be there. The notebook records deliveries, withdrawals and inspections.
2. A delivery entry explains why the expected level rose. A withdrawal explains why it fell. A person reading the indicator makes an observation, not a delivery or withdrawal.
3. If the indicator and the notebook disagree, writing “leak” in the book is not a diagnosis. A leak is one possible explanation, but so are a bad reading and a missed entry.
4. **Where this comparison breaks:** goods are often counted as separate units, while water is measured continuously. Our notebook example also assumes that “one notebook” means one item of the same product and stock location, not a pack, a reservation or an item in another branch.
5. The analogy helps separate movement from observation. It does not tell us which stock policy the shop should adopt after a discrepancy.

■ 6.1.3 PLAIN — a worked example

1. The stream identifier is STOCK-P-NOTE. It describes the expected physical stock of product P-NOTE at the one shop location in this example. It is not an “available to promise” quantity with reservations subtracted.
2. Here are the first three entries. Sequence means accepted position in this one stream.

Sequence	Recorded event	Effect on expected stock	Position afterwards
1	Opening balance: 12 notebooks	Establish 12	12
2	Goods issued: 2, for O-1042 line 1	Subtract 2	10
3	Physical count observed: 9	No movement	10

3. After sequence 3, the latest observed count is nine, the expected position is ten, and the discrepancy is $9 - 10 = -1$ notebook.
4. We know that the two figures differ. We do not know from these three entries whether an item was lost, miscounted, issued without an entry, or assigned to the wrong location.
5. In this teaching continuation, Mira reviews the discrepancy and explicitly authorises an adjustment of minus one. That decision becomes sequence 4 and refers to observation 3.
6. The expected position becomes nine because of the authorised adjustment, not because the software silently treated every observation as a movement.
7. A later receipt of five notebooks, with reference R-500, becomes sequence 5. The expected position becomes fourteen.

opening balance	12
issue for O-1042 line 1	-2
count observation of 9	0 (no movement)
authorised adjustment linked to count 3	-1
receipt R-500	+5
	--
expected position after sequence 5	14

A COUNT OBSERVATION DOES NOT MOVE GOODS

STOCK-P-NOTE · one location · expected physical stock

1	Opening balance: 12	12
2	Issue 2 for O-1042 line 1	10
3	Observe 9: no movement	10
4	Authorise adjustment of -1	9
5	Receive 5 under R-500	14

Sequence 4 is an explicit new decision linked to observation 3.

The discrepancy does not, by itself, identify a cause.

Figure 6.6: Figure 6.1. Expected stock changes only under the declared movement and adjustment rules; the observation is separate evidence.

■ 6.1.4 PLAIN — what is really happening inside

1. The system reads the event type before deciding how an entry affects the position. A number without its meaning is not enough.
2. An opening entry creates the initial position. An issue reduces it. A receipt increases it. A count observation stores separate evidence. An authorised adjustment applies an explicit change.
3. The calculation starts from a stated boundary and applies the appropriate rule to each accepted entry. The answer depends on both the entries and those rules.
4. The program also tracks the last sequence processed. “Fourteen notebooks through sequence 5” is a stronger description than an unexplained “fourteen”. It tells us the boundary of included history.
5. A person may still dispute the history. Linking the adjustment to the observation allows a reviewer to follow the decision; it does not prove that the observation was accurate or that every required approval occurred outside this teaching example.
6. The honest version: a record can preserve what the system accepted and why it produced an answer. It cannot travel back in time and directly inspect the shelf.

■ 6.1.5 TECHNICAL — the engineer’s version

1. A **state** is the information a model holds at a particular logical boundary. An **event** records an occurrence relevant to that model. A transition function can be written `next_state = apply(previous_state, event)`.
2. A projection folds a sequence of events into a derived representation. For an ordered sequence `e1 ... en`, the state is `apply(...apply(apply(initial, e1), e2)..., en)`. That notation is repeated application, not a requirement to hold the entire history in memory. [S40]
3. **Event sourcing** is a particular architectural choice in which an event history is the authoritative record used to reconstruct application state. A system having an audit table is not, for that reason

alone, event sourced. The relationship between the event store and any current-state tables must be specified. [S40]

4. Our stock projection has an explicit schema: expected quantity, latest observation and its sequence, processed sequence, and stream identifier. The teaching implementation rejects unknown event types instead of guessing that every numeric payload is a delta.
5. Opening balance and subsequent movements are different operations. Replaying an opening-balance event in the middle of an established stream would replace rather than explain the position; our lab rejects that malformed sequence.
6. An invariant in this bounded model is that expected physical stock must not become negative. That is a chosen rule for this fixture, not a universal database rule or a claim that every inventory application must reject negative book stock.
7. Retain provenance for important derived statements: input identifiers, transformation identity and the boundary of included input. W3C PROV provides a vocabulary for entities, activities and derivations, rather than an automatic guarantee of truth. [S05]
8. The lab tests the transition rules in memory. It does not implement a durable event store, multi-process concurrency, crash recovery or production access control. Those require additional mechanisms introduced later in the book.

■ 6.1.6 WORDS — remember these

1. **State:** the position the model currently holds — a representation at a defined logical boundary.
2. **Event:** a record of something relevant happening — an occurrence interpreted under a stated event schema.
3. **Projection:** an answer built from other records — a derived representation produced by applying transformation rules to source data.
4. **Event sourcing:** keeping the accepted event history as the principal record — an architecture that reconstructs state from authoritative events.
5. **Observation:** something recorded as seen or measured — evidence that need not itself authorise or cause a state-changing business action.
6. **Stock adjustment:** an explicit change to the expected stock record — a domain operation distinct from a goods movement or a count observation.

6.2 State transitions

■ 6.2.1 PLAIN — in simple words

1. Not every requested change makes sense from every starting point. A draft order can be confirmed. An order already fulfilled cannot simply become an untouched draft again.
2. A state-transition rule says which change is allowed, from which starting state, and what the resulting state will be.
3. A request is not the same as an accepted event. “Please cancel” describes an intention. “Cancellation accepted” says that the system checked a request and accepted a change.
4. A rejection is also useful information, but it must not be mistaken for the successful action the requester wanted.

- Two people can act on the same old view. The system needs to notice that one person's accepted change may make the other person's request stale.
- These rules are choices made for the work being modelled. A database cannot invent the shop's cancellation policy merely by knowing what a table is. [S02]

■ 6.2.2 PLAIN — a picture in your head

- Picture a parcel moving through labelled trays: "preparing", "ready to send", and "handed to carrier". Moving it between trays is allowed only through specified steps.
- A clerk who still sees an old list saying "preparing" must not erase the carrier handover just by placing a cancellation slip in the first tray.
- The clerk needs the current position and a rule for what a cancellation means at that position. It might be a rejection or a different process, such as a return request.
- Where this comparison breaks:** physical trays encourage us to imagine one visible location. In software, several screens can display copies with different ages. A page that looks current is not proof that no accepted change has happened since it loaded.
- Nor does a tray enforce the business rule by itself. We still need an authority that checks and records the move.

■ 6.2.3 PLAIN — a worked example

- Use a separate fictional order, P-3001. These rules do not assert anything new about the fulfilment status of 0-1042.
- We choose four states: DRAFT, CONFIRMED, FULFILLED and CANCELLED.

Current state	Request	Result in this teaching policy
DRAFT	Confirm	CONFIRMED
DRAFT	Cancel	CANCELLED
CONFIRMED	Fulfil	FULFILLED
CONFIRMED	Cancel	CANCELLED
FULFILLED	Cancel	Reject; a different returns process would be needed
CANCELLED	Fulfil	Reject

- Mira and Dev both load P-3001 at version 2, in state CONFIRMED. Mira requests fulfilment and Dev requests cancellation.
- Suppose the system accepts Mira's request first. It records fulfilment and advances the version from 2 to 3.
- Dev's request says that it was based on version 2. The system rejects it as stale rather than silently overwriting version 3.
- On refresh, the current state is FULFILLED. Under our chosen policy, cancellation is not an allowed transition from that state.
- Retrying does not mean repeatedly forcing the same desired result. It means reading the new state and re-evaluating whether the requested action is still valid.

■ 6.2.4 PLAIN — what is really happening inside

- The request includes an identity for the object being changed and the version on which the requester based the decision.

2. The system checks the current version and the transition rule together with accepting the new state. Separating the check from the acceptance leaves a gap in which another change could win.
3. A rejected request must not increase the version as though a successful transition occurred. Nor should a failed attempt send a success notification.
4. We can preserve a record of the failed attempt where appropriate, but it belongs to a different kind of record from the accepted order event.
5. The version number is not a timestamp. Version 3 means a position after version 2 in this object's accepted history; it does not mean three seconds have passed.
6. The honest version: checking versions in a single Python function demonstrates the idea. It does not make a shared production system safe unless the storage operation enforces the check under real concurrent access.

■ 6.2.5 TECHNICAL — the engineer's version

1. A finite state machine specifies a set of states and permitted transitions. A command is evaluated against a state; a successful command may produce one or more domain events. Keep command identity, delivery identity and event identity distinct. [S39] [S40]
2. Optimistic concurrency attaches an expected version to a write. The storage boundary must condition acceptance on the expected version still matching. In a relational system, related checks and updates belong in an appropriate transaction with suitable constraints or locking. A transaction boundary alone does not choose the isolation behaviour needed for every multi-row invariant. [S01] [S02]
3. The lab's `transition_order(state, version, expected_version, action)` is a pure educational check. It returns a new state and version or raises a validation error. It never contacts a payment provider or changes the actual shop.
4. A repeated request after a lost reply creates a second problem: the client may not know that its first request succeeded. Request-level idempotency can let it retrieve the earlier outcome, but only within the stated identity and retention policy. Expected-version checking by itself is not a complete idempotency design. [S40]
5. Domain validation is different from authorisation. "Cancellation is allowed from CONFIRMED" does not mean any caller may cancel any confirmed order. The production operation must also check the caller's authority at the appropriate boundary.
6. An accepted transition can have effects outside one database. An order-state transaction does not atomically recall a parcel already handed to a courier. Separate external effects, retries and compensations must be modelled explicitly. [S40]
7. Test forbidden edges as well as allowed ones. Include unknown states, unknown actions, mismatched versions and a rejected action that leaves the original state unchanged. These are assertions about this chosen transition graph, not exhaustive proof of a complete commerce system.

■ 6.2.6 WORDS — remember these

1. **Command:** a request to do something — an intention evaluated against current state and applicable rules.
2. **State transition:** an allowed move between positions — a specified change from one model state to another.

3. **Expected version:** the history position a request relies on — a concurrency precondition used to detect stale writes.
4. **Stale request:** a request based on an older position — an operation whose assumed version no longer matches the accepted state.
5. **Invariant:** a rule that must remain true — a property the chosen model preserves across accepted operations.
6. **Authorisation:** permission to perform an action — an access decision separate from whether the action is a valid domain transition.

6.3 Ordering and causality

■ 6.3.1 PLAIN — in simple words

1. “Which came first?” can ask several different questions. Which event happened first? Which was written first? Which message arrived first? Which action depended on another?
2. These orders can disagree. A slow connection can deliver an earlier event after a later one.
3. Two devices can also have clocks that disagree. A printed time is a useful observation, but it is not automatically a trustworthy instruction about causal order.
4. If one action sends a message and another action receives that same message, the send comes before that receive in the communication relationship. The receive cannot be the cause of the send it receives. [S41]
5. An accepted sequence number can define the replay order for one stock stream. It does not automatically tell us the order of every action in every shop.
6. Before sorting a history, name the question the sort is supposed to answer. Sorting by the easiest available column may produce a tidy but misleading story.

■ 6.3.2 PLAIN — a picture in your head

1. Imagine two clerks posting letters. Each writes the time shown on their own desk clock. One clock is five minutes fast.
2. A reply depends on the original letter arriving, even if the timestamps make the reply look earlier. The communication relationship is evidence that the clock labels alone are not the whole story.
3. Now consider two unrelated letters sent to different people. Sorting them by envelope colour gives an order, but tells us nothing about whether one caused the other.
4. **Where this comparison breaks:** software can exchange enormous numbers of messages with retries, relays and duplicates. Establishing that a receive corresponds to a particular send requires identifiers and protocol evidence, not a human guess from similar wording.
5. The analogy separates an ordering convention from a causal claim. It does not provide a universal clock.

■ 6.3.3 PLAIN — a worked example

1. Our stock projection has processed sequences 1 and 2 and holds expected stock ten. It next receives sequence 4, the authorised adjustment.
2. The strict replay rule in this lab requires sequence 3 next. It rejects sequence 4 as a gap rather than pretending that the preceding history is complete.

3. A real ingestion service might buffer 4 until 3 arrives or fetch the missing entry. Our small lab does neither: rejection makes the missing prerequisite visible.
4. Once 3 is available, applying 3, 4 and 5 in sequence gives expected positions ten, nine and fourteen.
5. Here is a separate logical-clock exercise. A logical counter is a number used to preserve communication order; it is not a time of day.

```

Process A starts with counter 0.
A performs an event: counter becomes 1.
A sends a message: counter becomes 2; message carries 2.

Process B already has counter 5.
B receives that message: counter becomes  $\max(5, 2) + 1 = 6$ .
B performs its next event: counter becomes 7.

```

6. The send is labelled 2 and its receive 6, preserving their order. Those four steps between 2 and 6 are not four seconds, and they do not count four intervening events in the whole system.
7. An unrelated process could also have an event labelled 6. Equal logical values do not prove the events happened at the same physical instant.

■ 6.3.4 PLAIN — what is really happening inside

1. The stock stream uses one explicit sequence for interpretation. An arrival with a missing predecessor is held outside the accepted projection boundary in our lab.
2. An event timestamp can still be stored for reporting. It must not silently replace the sequence rule when clocks disagree or old events arrive late.
3. A logical clock advances locally and incorporates values carried by messages. That gives it information about communication dependencies without requiring perfectly synchronised wall clocks. [S41]
4. The clock provides a useful guarantee in one direction: if an event causally precedes another under the model, the first receives a smaller logical value. A smaller value alone does not establish a causal link. [S41]
5. A system may break ties using a process identifier to produce a deterministic total order. That tie-break is a chosen order, not a newly discovered fact about the physical world.
6. The honest version: the identifiers and clocks only describe the events the system models. A phone call or a human action outside the recorded communication can matter without appearing in that model.

■ 6.3.5 TECHNICAL — the engineer's version

1. Lamport's happened-before relation includes local process order, message send before its receive, and transitive consequences. It is a partial order: some distinct events are not ordered by it. Logical clocks can satisfy $a \rightarrow b$ implies $C(a) < C(b)$; the reverse implication is not generally valid. [S41]
2. The ordinary scalar-clock update is local increment for an event and, at a receive, $C = \max(C, \text{received_C}) + 1$. A deterministic process-ID tie-break can extend clock order to a total order without proving additional causal relationships. [S41]
3. Distinguish **event time**, **recording time**, **delivery time** and **stream sequence**. Chapter 3 introduced representations of instants; this chapter explains why a correctly formatted timestamp still does not define every required processing order. [S09] [S40]
4. Our projector requires an exact next sequence within an exact stream. It rejects duplicates, gaps and unexpected stream identifiers. Duplicate transport delivery is handled before this strict fold; silently applying it twice would corrupt state. [S39] [S40]

5. Ingesting several streams does not create a globally agreed sequence just because each stream is internally ordered. Cross-stream invariants need a separate coordination model, addressed in the later parts on transactions and distributed systems.
6. Integer deltas can commute in an unconstrained arithmetic sum, but admission rules may not. Starting at zero, receiving one and then issuing one passes a non-negative-stock rule; issuing before receiving fails it. Final arithmetic equality does not erase the intermediate rule violation.
7. The lab demonstrates that difference with a small sequence and checks that a gap raises an error. It does not simulate network consensus or promise a global real-time order.

■ 6.3.6 WORDS — remember these

1. **Stream sequence:** a record's position in one accepted history — a scoped ordering value used for interpretation or concurrency checks.
2. **Causal order:** one event preceding another through a dependency — the order induced by the model's local actions and communications.
3. **Logical clock:** a counter that helps preserve event order — a timestamping mechanism that need not represent physical time.
4. **Partial order:** an order that does not compare every pair — a relation under which some distinct events remain unordered.
5. **Sequence gap:** an expected predecessor is missing — a discontinuity that prevents this strict projector from claiming a complete prefix.
6. **Commutative operation:** an operation whose result is unchanged by swapping order — a property that does not automatically extend to validation rules or external effects.

6.4 Snapshots and projections

■ 6.4.1 PLAIN — in simple words

1. Re-reading a long history from the very beginning can cost time. A snapshot is a saved position at a known point in that history.
2. To continue, we load the snapshot and read only the later events. That works only when the snapshot belongs to the correct stream and interpretation rules.
3. A snapshot without its boundary is like an answer without the question. We need to know which entries it already includes so we do not skip or repeat them.
4. A projection is the answer produced for a particular purpose. The same accepted events can support a stock balance, a daily movement report and a list of unresolved count discrepancies.
5. Those answers need not update at the same instant. A report can lag behind the accepted history, so it should make its processed boundary visible where that matters. [S40]
6. A snapshot and a backup are not interchangeable words. A snapshot used to accelerate one calculation may be stored on the same machine and fail with the rest of that machine.

■ 6.4.2 PLAIN — a picture in your head

1. Imagine reading a long account book and writing a subtotal at the bottom of each page. Tomorrow you can begin from a checked subtotal rather than adding every old line again.

2. “Subtotal after page 12” is useful. “Subtotal: 10” with no page number is dangerous because you do not know where to resume.
3. A second clerk might make a different summary from the same pages, such as the number of deliveries rather than the number of items. Both are projections of the same records for different questions.
4. **Where this comparison breaks:** a software interpretation can change between versions. A subtotal made by a buggy program can be reproduced exactly and still be wrong. Its existence is not a certificate of correctness.
5. Nor is a subtotal a replacement for the original lines when a reviewer needs the details the subtotal discarded.

■ 6.4.3 PLAIN — a worked example

1. Save a snapshot of STOCK-P-NOTE immediately after sequence 2. It says expected quantity ten, no count observation yet, and last processed sequence two.

```
{
  "stream": "STOCK-P-NOTE",
  "projection_version": "stock-v1",
  "last_sequence": 2,
  "expected_quantity": 10,
  "last_observation": null
}
```

2. Resume with sequence 3. The expected position stays ten and the latest observation becomes nine.
3. Apply sequence 4. The authorised adjustment reduces expected stock to nine. Apply sequence 5 and it rises to fourteen.
4. Replaying all five entries from the beginning also gives fourteen, with the same last observation and sequence boundary. The companion lab compares the full resulting state, not just one coincidentally equal number.
5. Now deliberately label the snapshot STOCK-P-PEN while keeping notebook events. The lab rejects it. A plausible number in a snapshot for the wrong product is not a valid starting point.
6. Deliberately change the projection version to an unknown value. The lab rejects that too; it does not assume an unfamiliar interpretation is compatible.

■ 6.4.4 PLAIN — what is really happening inside

1. The snapshot contains both the derived state and metadata that says how to use it. Together they define a starting point for replay.
2. The projector accepts only events beyond that boundary, in the required order. Reapplying sequence 2 would issue the same two notebooks twice if the duplicate were not rejected.
3. A snapshot made while new entries are arriving must correspond to one coherent point. Copying the balance after event 5 but the checkpoint after event 2 would produce a mixed record that no valid replay created.
4. A report’s checkpoint similarly needs to agree with the result it describes. A crash between updating a report and updating its checkpoint can cause a retry problem unless those writes are coordinated. [S40]
5. Rebuilding a projection should calculate data, not casually repeat physical actions. Reading an old “goods issued” event must not send a second instruction to issue the goods again.

6. The honest version: successful replay proves agreement between two calculations under the tested rules and inputs. It does not prove that the input history includes every real-world event.

■ 6.4.5 TECHNICAL — the engineer’s version

1. A snapshot is an optimisation associated with a stream boundary and a projection schema or version. It is not necessarily the authoritative history and can often be discarded and rebuilt when the required source history is retained. That rebuilding condition matters. [S40]
2. Deterministic replay requires that interpretation not depend on uncontrolled current values. Calling “today’s price”, the current exchange rate or a random generator inside the fold can change a past result. Store the relevant historical input or define a versioned, reproducible rule where appropriate. [S05] [S40]
3. Our stock-v1 projection uses integers, explicit event types and a stated sequence. A snapshot serialises expected quantity, observation information and last sequence. The lab validates matching stream and version before continuing.
4. A materialised projection may be updated asynchronously. A checkpoint such as “processed through sequence 5” is a statement about incorporated input, not a wall-clock guarantee that every source anywhere has been observed. [S40]
5. Persisting a projection and its consumer position requires an appropriate atomicity or idempotency strategy. In a single database, both can sometimes be committed in one transaction; across external systems that assumption must not be made without evidence. [S01] [S25] [S40]
6. Side effects deserve their own boundary. Sending an email, dispatching a parcel or charging a payment cannot safely be treated as an ordinary pure replay step. Recording delivery intent and handling retries are later topics; this chapter’s fold deliberately performs none of those actions. [S40]
7. Test full replay against snapshot replay, wrong-stream snapshots, unknown versions and a missing first suffix event. Those checks expose specific errors. They do not replace tests of persistent storage, concurrent snapshot creation or recovery after process failure.

■ 6.4.6 WORDS — remember these

1. **Snapshot:** a saved position at a known boundary — a serialised state with enough metadata to resume a compatible interpretation.
2. **Checkpoint:** how far processing has reached — a recorded boundary of incorporated source input.
3. **Deterministic replay:** the same retained inputs and rules give the same state — reconstruction without uncontrolled dependencies on current external values.
4. **Materialised projection:** an answer stored for later reading — a persisted derived view rather than a query result recomputed on every request.
5. **Projection version:** which interpretation produced the answer — an identifier for the relevant transformation or state representation.
6. **Side effect:** an action beyond calculating the returned value — an external change that must not accidentally be repeated during reconstruction.

6.5 Corrections and reversal events

■ 6.5.1 PLAIN — in simple words

1. Keeping history does not mean pretending mistakes never happen. It means deciding how a mistake will be corrected and how the correction will be explained.
2. Overwriting an old entry can remove evidence of what the system previously believed. Adding a correction can preserve both the earlier record and the later repair.
3. A reversal often applies an opposite effect to an earlier entry. That arithmetic can repair a recorded position, but it does not necessarily undo anything outside the record.
4. If a parcel left the shop, writing an opposite quantity does not teleport it back. The physical return and the record of that return are separate matters.
5. A correction needs a target. “Subtract one” without saying which mistake it repairs can make today’s total look right while leaving tomorrow’s reviewer unable to explain it.
6. Different kinds of records need different correction policies. We should not impose a complicated event system on every small spreadsheet, or erase important history merely because editing a cell is easy. [S40]

■ 6.5.2 PLAIN — a picture in your head

1. Imagine a teacher marking an exercise. The first mark was written beside the wrong answer. A later note says which mark was mistaken and gives the corrected result.
2. Crossing out the first mark without a trace makes it harder to explain why the student saw a different score yesterday. Keeping both marks without indicating which one now applies is also confusing.
3. The correction therefore has two jobs: make the current answer right and preserve an understandable relationship to the earlier answer.
4. **Where this comparison breaks:** reversing an accounting entry or correcting an inventory issue can affect several related records, permissions and external processes. A note in the margin alone does not enforce any of those relationships.
5. The analogy is about explanation, not a universal instruction to retain every detail forever.

■ 6.5.3 PLAIN — a worked example

1. Use a separate teaching stream, R-EX-01. Do not apply these entries to 0-1042 or to the five-event notebook stream above.
2. This stream opens at twelve notebooks. In the scenario, two notebooks were actually issued, but the operator mistakenly recorded an issue of three.

Sequence	Entry	Recorded effect	Expected position
1	Opening balance	Establish 12	12
2	Erroneous issue record	-3	9
3	Reversal referring to entry 2	+3	12
4	Correct replacement issue	-2	10

3. The combined correction is $+3 - 2 = +1$. It takes the mistaken recorded position from nine to ten.
4. The physical issue remains two. We are repairing its representation, not asserting that three notebooks were physically returned and then two issued again.

5. A report of accepted entries can show all four events. A report of the corrected issue should explain the reversal relationship so it does not count two unrelated real-world issues.
6. A second reversal of entry 2 would add another three and corrupt the result. Our teaching lab rejects a second reversal of the same target.
7. A reversal referring to an unknown or non-reversible entry is also rejected. The word “reversal” is not permission to add any convenient number.

■ 6.5.4 PLAIN — what is really happening inside

1. The system locates the target event and checks that the chosen correction policy allows it to be reversed.
2. It derives the reversal’s effect from that target instead of trusting a caller’s arbitrary positive quantity. In this example, reversing minus three produces plus three.
3. It records the link between the reversal and the original. A later reader can inspect why the extra positive entry exists.
4. It tracks whether the target was already reversed. Otherwise, a retry could apply the repair twice.
5. A replacement entry expresses the corrected fact. Reversal and replacement may need to be accepted together when a temporarily reversed-only position would violate the intended operation.
6. The honest version: our arithmetic example explains a correction model. Production acceptance must also address concurrent requests, transaction boundaries, permissions and the availability of evidence supporting the correction. [S01] [S40]

■ 6.5.5 TECHNICAL — the engineer’s version

1. A compensating event changes subsequent interpretation without mutating the accepted earlier event. Compensation is domain-specific: not every effect has a simple mathematical inverse, and an inverse in a ledger does not imply reversal of an external action. [S40]
2. The lab’s reversal fixture permits reversal only of a prior issue in the same fixture and only once. It computes the inverse of the recorded delta and links to the target sequence. It deliberately does not expose a general “reverse anything” API.
3. The pure implementation can check these rules sequentially. In persistent storage, “not already reversed” must be enforced under concurrent access, for example through an appropriate unique relationship and transaction design. A preliminary application query alone leaves a race. [S01] [S02]
4. Preserve the distinction between event validity at acceptance and later interpretation. An entry can be a faithful record of what the system accepted even after a later correction establishes that its business content was wrong.
5. Queries need a declared perspective. “What did we record by yesterday?” and “What is our corrected account of yesterday’s activity using information available now?” can return different answers without either being a database error. Source and derivation boundaries must be explicit. [S05]
6. A schema change can also affect how old events are read. A versioned reader or an explicit transformation may be needed; it should not silently reinterpret an old quantity’s unit or turn an optional field into a fabricated historical fact. [S05] [S40]
7. Do not assume a sum is a sufficient audit. A wrongly duplicated issue and an unrelated positive adjustment can cancel numerically. Reconciliation should inspect identities, links and expected relationships as well as totals.

■ 6.5.6 WORDS — remember these

1. **Reversal:** an opposite recorded effect tied to an earlier entry — a constrained correction operation whose scope must be defined.
2. **Compensation:** a later action that addresses an earlier effect — a domain-specific response that need not restore the exact original world.
3. **Replacement entry:** the newly accepted corrected representation — a record distinguished from the earlier entry it supersedes or repairs.
4. **Correction target:** the particular entry being repaired — an explicit reference needed to validate and explain a correction.
5. **Reconciliation:** checking whether related records agree as expected — a comparison of identities, relationships and totals rather than a single balancing sum.

6.6 History with retention boundaries

■ 6.6.1 PLAIN — in simple words

1. “We keep history” needs a boundary. Which records are kept, for how long, for which purpose, and under whose control?
2. An append-only rule means ordinary changes add records rather than editing earlier ones. It does not, by itself, mean that every detail must be retained forever.
3. A history can contain unnecessary personal details or sensitive text. Adding more copies can increase the work of protecting and eventually removing those details.
4. Reconstruction depends on what remains available. If old source events are removed, a retained snapshot may preserve a position while losing the ability to explain every earlier movement.
5. That can be an intentional choice. The honest report says what was retained and what can no longer be reconstructed, rather than promising both complete deletion and unlimited recovery from the same removed evidence.
6. This chapter describes engineering boundaries, not a legal retention schedule. Applicable obligations and permissions must be established separately for the actual system and jurisdiction. [S40]

■ 6.6.2 PLAIN — a picture in your head

1. Imagine an archive that retains annual totals but disposes of some old working slips under its approved policy. A total can still be read, but the archivist cannot honestly promise to produce every disposed slip.
2. A catalogue saying a slip once existed is not the slip itself. A checksum saying a retained file has not changed is not proof that the file originally contained a correct observation.
3. The archive might keep different categories for different periods. That requires an inventory and a process, not a single switch labelled “history”.
4. **Where this comparison breaks:** digital records can be copied into logs, exports, backups, caches and analytical systems. Removing a main entry does not automatically remove every copy or every derived identifier.
5. The analogy is useful only if we remember that the archive has many rooms, not one cupboard.

■ 6.6.3 PLAIN — a worked example

1. Suppose the teaching stock system retains a checked snapshot through sequence 5: expected quantity fourteen, latest observation nine at sequence 3, with interpretation `stock-v1`.
2. Under a hypothetical retention decision, the detailed events 1–5 are no longer available to this reader. The snapshot is still present and later events remain readable.
3. The reader can continue a compatible calculation from that boundary, provided the snapshot and its metadata are trustworthy enough for the intended use.
4. The reader cannot replay the missing prefix from scratch or inspect the original wording of the authorised adjustment. That evidence is absent from the available package.
5. A remaining event identifier or hash may help compare a future recovered copy. It cannot recreate the deleted content by itself.
6. Write the limitation plainly: “Current state continues from snapshot through sequence 5; the underlying prefix is not available here. Full prefix reconstruction was not performed.”
7. This is a hypothetical retention exercise. No source files, user records or published materials were deleted to demonstrate it.

■ 6.6.4 PLAIN — what is really happening inside

1. The system separates data categories and their purposes. A minimal operational event need not repeat every detail of the customer profile that happened to be on screen when the action occurred.
2. It records what a projection actually needs. Where an event refers to another record, we must decide what happens if that referenced detail changes or is no longer retained.
3. A replay that depends on deleted external content may fail or produce a less detailed result. The system should not silently fill the gap with a current profile or a guessed value.
4. Backups and exported copies need their own handling. A restored old copy can reintroduce information that a later process removed unless restoration accounts for subsequent actions.
5. A retained audit statement can say that a deletion or expiry action was recorded. It should not overclaim that every possible copy on every system has been examined unless that was actually established.
6. The honest version: recordkeeping, reproducibility and minimisation can pull in different directions. Good design makes the trade-off explicit and tests the promises it actually makes. [S05] [S40]

■ 6.6.5 TECHNICAL — the engineer’s version

1. Event-store immutability is an application or storage property with a defined enforcement boundary. It is not an automatic retention policy and does not imply immunity to privileged changes, storage loss or deletion. [S40]
2. Avoid putting unnecessary sensitive payloads into long-lived events. One design keeps an event’s minimal operational fact separate from mutable or separately governed attributes. That separation creates reference-resolution and replay questions that must be designed, not ignored. [S40]
3. A reconstruction claim should identify input availability, schema versions, transformation versions, snapshots and the scope of the resulting output. A provenance model can describe those relationships, while verification remains an additional activity. [S05]
4. If a projection was rebuilt from a retained snapshot rather than the full history, record that distinction. Equal current totals do not prove equality of all earlier event histories.

5. A hash-based integrity check detects disagreement with a trusted reference value under its stated assumptions. It does not establish that the source was true, restore missing content, or prove that no unexamined copies remain.
6. The lab tests a snapshot continuation and its limits as data structures. It does not implement retention enforcement, encryption-key destruction, backup expiry or a legal erasure workflow. Do not treat its successful tests as evidence that those unimplemented capabilities exist.
7. Operationally, inventory the stores, define category-specific decisions, record authorised changes, test restoration and disclose gaps. The later chapters on retention and operations will develop that workflow; it is not complete merely because an event log exists.

■ 6.6.6 WORDS — remember these

1. **Append-only:** ordinary updates add rather than replace — a mutation policy whose enforcement and exceptions must be specified.
2. **Retention boundary:** which history remains available — a stated limit on kept records, detail, time or access.
3. **Reconstruction boundary:** how far an answer can be rebuilt — the scope supported by retained inputs, snapshots and interpretation rules.
4. **Data minimisation:** avoiding unnecessary stored detail — collecting and retaining only the information justified for the defined purpose.
5. **Integrity check:** a check that retained content agrees with a reference — evidence of consistency, not automatic evidence of real-world truth.
6. **Restoration:** bringing back a stored copy — a recovery operation that must account for the copy's age and subsequent required changes.

6.97 Exercises with worked answers

■ Exercise 1 | A count is not a movement

The stream opens at 12, records an issue of 2, and then records a count observation of 9. What are expected stock, latest observed count and discrepancy?

Worked answer. Expected stock is $12 - 2 = 10$. Latest observation is 9. Observed minus expected is $9 - 10 = -1$. No adjustment has yet been authorised, so changing the expected balance to 9 would add a decision not present in these three entries. The discrepancy does not identify its cause.

■ Exercise 2 | Continue from a snapshot

A compatible STOCK-P-NOTE snapshot through sequence 2 holds expected quantity 10. Sequence 3 observes 9, sequence 4 authorises an adjustment of -1 linked to 3, and sequence 5 receives 5. What is the result, and what must be checked before replay?

Worked answer. The observation leaves expected quantity at 10. Adjustment gives 9. Receipt gives 14. Check the stream identifier, projection version, snapshot state and last sequence. Require the suffix to begin at 3 and continue without gaps under this lab's strict rule. Record that the final position includes sequence 5.

■ Exercise 3 | Two requests, one old version

An order is CONFIRMED at version 2. Fulfilment is accepted first, producing FULFILLED at version 3. A cancellation request based on version 2 then arrives. Should the system blindly retry the state change?

Worked answer. No. Its precondition is stale. After reloading version 3, cancellation is forbidden by the teaching transition graph. A separate returns process could exist, but it is not implemented here and cannot be invented by relabelling the fulfilled order as cancelled.

■ Exercise 4 | Read the logical clock

A process has counter 5 and receives a message carrying counter 8. What is the new counter? Does an unrelated event with a smaller counter necessarily cause this event?

Worked answer. $\max(5, 8) + 1 = 9$. No. Logical-clock order preserves causal precedence in one direction; a smaller scalar value alone does not establish a causal relationship. The counter is not a physical duration. [S41]

■ Exercise 5 | Repair a mistaken issue

In separate stream R-EX-01, the opening quantity is 12. An issue was incorrectly recorded as 3 when the scenario says 2 were actually issued. Show a reversal and replacement, and explain why a second reversal should fail.

Worked answer. Record -3, giving 9. Reverse that target with +3, returning the recorded position to 12. Record the correct replacement -2, giving 10. A second reversal would add another 3 despite referring to the same already repaired entry, so the fixture rejects it. These entries repair the record; they do not describe goods physically travelling back and forth.

■ Exercise 6 | State what missing history prevents

You retain a compatible snapshot through sequence 5 and all later events, but not events 1–5. Can you claim to have reconstructed the entire history from its original source events?

Worked answer. No. You can report continuation from the retained snapshot under stated assumptions. You cannot inspect or replay the missing prefix. The snapshot's metadata and any retained verification help characterise its basis; they do not replace the absent source details.

6.98 Common wrong ideas

1. **Wrong:** a current balance explains its own history. **Right:** several different histories can produce the same balance.
2. **Wrong:** observing a stock count is the same as authorising an adjustment. **Right:** evidence and state-changing decisions are distinct operations in this model.
3. **Wrong:** a cancellation request proves that cancellation occurred. **Right:** a request may fail validation, authorisation or its version precondition.
4. **Wrong:** sorting timestamps discovers every causal relationship. **Right:** clocks, delivery and dependencies require separate interpretation.
5. **Wrong:** a smaller logical counter proves causation. **Right:** causal precedence implies increasing counters, but the converse does not generally hold. [S41]
6. **Wrong:** a snapshot is useful without knowing where it belongs. **Right:** stream identity, interpretation version and sequence boundary are essential.

7. **Wrong:** replaying old events should repeat their physical effects. **Right:** rebuilding a projection should not resend parcels or repeat charges.
8. **Wrong:** a reversal automatically undoes the outside world. **Right:** an opposite recorded effect and an external compensation are different things.
9. **Wrong:** append-only means every payload must remain forever. **Right:** mutation policy and retention policy answer different questions.
10. **Wrong:** passing this in-memory lab proves a production event platform is reliable. **Right:** it checks bounded rules and fixtures, not unimplemented storage, security, concurrency or recovery.

6.99 Chapter summary in 20 lines

1. A state describes a position; an event records an occurrence relevant to a model.
2. A retained event is not automatically a true account of the physical world.
3. Our notebook stream starts at twelve and issues two for O-1042, giving ten.
4. Observing nine records a discrepancy; it does not itself authorise a movement.
5. An explicit minus-one adjustment and a receipt of five produce fourteen.
6. A projection applies declared interpretation rules to a defined input history.
7. Event sourcing makes an event history authoritative; an audit table alone does not establish that architecture.
8. A command requests a change and can be rejected rather than becoming an accepted event.
9. State-transition rules identify permitted moves from particular starting states.
10. An expected version helps detect requests based on stale state.
11. Physical time, recording time, delivery order and stream sequence are different concepts.
12. Logical clocks can preserve causal order without being clocks of physical time.
13. Smaller logical values do not, by themselves, prove causation.
14. A strict projector must not quietly claim completeness across a sequence gap.
15. A snapshot needs a stream, interpretation version and included-history boundary.
16. Rebuilding a projection must not casually repeat external side effects.
17. A reversal needs a valid target and a rule preventing duplicate application.
18. Correcting a record does not necessarily undo a real-world action.
19. Retention decisions can limit which earlier details remain reconstructible.
20. State the evidence boundary honestly, then move to Chapter 7: files, folders and data formats.



PART B

From Records to Databases

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

Files, Folders and Data Formats

7.0 What this chapter gives you

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

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

7.1 Paths and file identity

■ 7.1.1 PLAIN — in simple words

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

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

■ 7.1.2 PLAIN — a picture in your head

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

■ 7.1.3 PLAIN — a worked example

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

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

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

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

■ 7.1.4 PLAIN — what is really happening inside

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

■ 7.1.5 TECHNICAL — the engineer's version

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

```
from pathlib import PurePosixPath

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

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

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

■ 7.1.6 WORDS — remember these

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

7.2 Bytes, records and boundaries

■ 7.2.1 PLAIN — in simple words

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

■ 7.2.2 PLAIN — a picture in your head

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

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

■ 7.2.3 PLAIN — a worked example

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

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

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

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

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

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

■ 7.2.4 PLAIN — what is really happening inside

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

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

■ 7.2.5 TECHNICAL — the engineer's version

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

■ 7.2.6 WORDS — remember these

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

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

7.3 CSV and quoting

■ 7.3.1 PLAIN — in simple words

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

■ 7.3.2 PLAIN — a picture in your head

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

■ 7.3.3 PLAIN — a worked example

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

```
| product_id,description
```

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

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

```
import csv
import io

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

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

■ 7.3.4 PLAIN — what is really happening inside

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

■ 7.3.5 TECHNICAL — the engineer's version

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

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

■ 7.3.6 WORDS — remember these

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

7.4 JSON and nested data

■ 7.4.1 PLAIN — in simple words

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

■ 7.4.2 PLAIN — a picture in your head

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

■ 7.4.3 PLAIN — a worked example

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

```
{
  "record_type": "agreed_order_line",
  "schema_version": 1,
  "order_id": "0-1042",
  "line_no": 1,
  "product_id": "P-NOTE",
  "quantity": 2,
  "unit_price_minor": 7550,
  "currency": "INR"
}
```


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

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

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

■ 7.4.4 PLAIN — what is really happening inside

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

■ 7.4.5 TECHNICAL — the engineer's version

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

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



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

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

■ 7.4.6 WORDS — remember these

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

7.5 Binary formats and versioning

■ 7.5.1 PLAIN — in simple words

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

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

■ 7.5.2 PLAIN — a picture in your head

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

■ 7.5.3 PLAIN — a worked example

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

```
import struct

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

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

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

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

■ 7.5.4 PLAIN — what is really happening inside

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

■ 7.5.5 TECHNICAL — the engineer's version

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

■ 7.5.6 WORDS — remember these

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

7.6 Import validation and quarantine

■ 7.6.1 PLAIN — in simple words

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

■ 7.6.2 PLAIN — a picture in your head

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

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

■ 7.6.3 PLAIN — a worked example

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

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

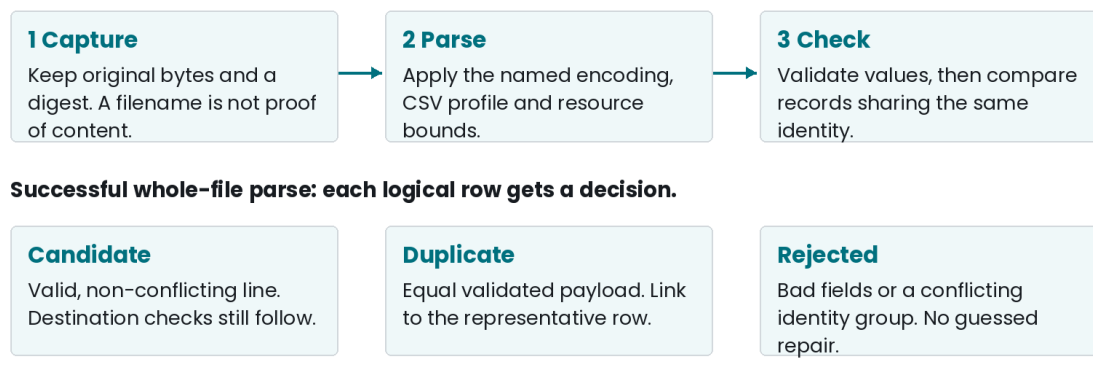
2. The accounting identity is eight data rows = four candidates + three rejected rows + one duplicate. The header is not counted as a data row.
3. The four candidates still describe two orders, four lines and six units. Their total remains 28,650 paise; the duplicate contributes no additional candidate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

■ 7.6.4 PLAIN — what is really happening inside

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

AN IMPORT IS A SERIES OF SEPARATE DECISIONS

A candidate is not yet a committed record. All inputs here are synthetic.



File-level syntax or decoding failure: zero candidates.

Retain the captured bytes and the failure reason. Do not publish a valid-looking prefix.

Later publication is a separate transaction; rejected rows remain disclosed.

Figure 7.7: Figure 7.1 — An import keeps captured bytes, parses under a named profile, checks values and identity, and separates candidates, duplicate deliveries and rejected material. File-level syntax failure publishes no candidates.

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

■ 7.6.5 TECHNICAL — the engineer's version

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

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

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

■ 7.6.6 WORDS — remember these

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

7.97 Practice and worked answers

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

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

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

7.98 Common wrong ideas

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

7.99 Chapter summary in 20 lines

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

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

Why Databases Exist

8.0 What this chapter gives you

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

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

8.1 The shared-file problem

■ 8.1.1 PLAIN — in simple words

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

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

■ 8.1.2 PLAIN — a picture in your head

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

■ 8.1.3 PLAIN — a worked example

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

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

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

■ 8.1.4 PLAIN — what is really happening inside

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

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

■ 8.1.5 TECHNICAL — the engineer’s version

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

■ 8.1.6 WORDS — remember these

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

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

8.2 A database and its engine

■ 8.2.1 PLAIN — in simple words

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

■ 8.2.2 PLAIN — a picture in your head

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

■ 8.2.3 PLAIN — a worked example

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

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

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

■ 8.2.4 PLAIN — what is really happening inside

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

■ 8.2.5 TECHNICAL — the engineer’s version

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

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

■ 8.2.6 WORDS — remember these

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

8.3 Client-server and embedded designs

■ 8.3.1 PLAIN — in simple words

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

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

■ 8.3.2 PLAIN — a picture in your head

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

■ 8.3.3 PLAIN — a worked example

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

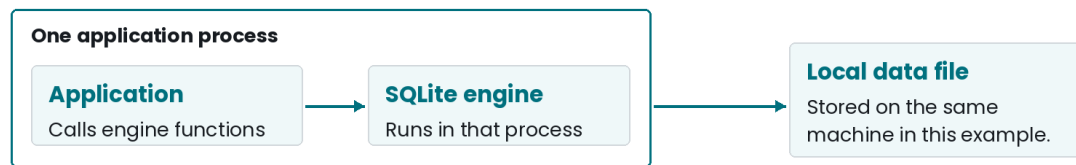
Design	Request path	Who directly opens database storage?
A: one local desktop application	User → application → embedded SQLite	The local application’s engine
B: one application server serving browsers	Browser → application service → embedded SQLite	The application service’s engine
C: application service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

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

WHERE THE DATABASE ENGINE RUNS

Engine location and application location are different architectural choices.

EMBEDDED EXAMPLE



No separate database-server process is required here.

CLIENT-SERVER EXAMPLE



A browser normally talks to the application service, not directly to these database files.

Figure 8.8: Figure 8.1 — Embedded SQLite executes in the application process; a client-server design sends requests to a separate database process. Neither picture determines the business rules or proves a performance claim.

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

■ 8.3.4 PLAIN — what is really happening inside

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

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

■ 8.3.5 TECHNICAL — the engineer's version

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

■ 8.3.6 WORDS — remember these

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

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

8.4 Transactions and queries

■ 8.4.1 PLAIN — in simple words

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

■ 8.4.2 PLAIN — a picture in your head

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

■ 8.4.3 PLAIN — a worked example

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

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

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

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

■ 8.4.4 PLAIN — what is really happening inside

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

■ 8.4.5 TECHNICAL — the engineer's version

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

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

■ 8.4.6 WORDS — remember these

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

8.5 Operational responsibilities

■ 8.5.1 PLAIN — in simple words

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

■ 8.5.2 PLAIN — a picture in your head

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

■ 8.5.3 PLAIN — a worked example

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

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

■ 8.5.4 PLAIN — what is really happening inside

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

■ 8.5.5 TECHNICAL — the engineer's version

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

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

■ 8.5.6 WORDS — remember these

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

8.6 When a file is enough

■ 8.6.1 PLAIN — in simple words

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

■ 8.6.2 PLAIN — a picture in your head

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

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

■ 8.6.3 PLAIN — a worked example

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

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

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

■ 8.6.4 PLAIN — what is really happening inside

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

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

■ 8.6.5 TECHNICAL — the engineer's version

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

■ 8.6.6 WORDS — remember these

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

8.97 Practice and worked answers

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

8.98 Common wrong ideas

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

8.99 Chapter summary in 20 lines

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

Tables, Rows and Relationships

9.0 What this chapter gives you

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

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

9.1 Tables as sets of claims

■ 9.1.1 PLAIN — in simple words

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

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

■ 9.1.2 PLAIN — a picture in your head

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

■ 9.1.3 PLAIN — a worked example

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

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

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

■ 9.1.4 PLAIN — what is really happening inside

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

■ 9.1.5 TECHNICAL — the engineer’s version

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

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

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

■ 9.1.6 WORDS — remember these

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

9.2 Columns and domains

■ 9.2.1 PLAIN — in simple words

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

■ 9.2.2 PLAIN — a picture in your head

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

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

■ 9.2.3 PLAIN — a worked example

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

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

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

■ 9.2.4 PLAIN — what is really happening inside

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

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

■ 9.2.5 TECHNICAL — the engineer’s version

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

■ 9.2.6 WORDS — remember these

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

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

■ 9.3.1 PLAIN — in simple words

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

■ 9.3.2 PLAIN — a picture in your head

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

■ 9.3.3 PLAIN — a worked example

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

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

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

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

■ 9.3.4 PLAIN — what is really happening inside

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

■ 9.3.5 TECHNICAL — the engineer's version

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

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

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

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

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

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

■ 9.3.6 WORDS — remember these

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

9.4 Many-to-many relationships

■ 9.4.1 PLAIN — in simple words

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

■ 9.4.2 PLAIN — a picture in your head

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

■ 9.4.3 PLAIN — a worked example

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

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

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

■ 9.4.4 PLAIN — what is really happening inside

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

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

■ 9.4.5 TECHNICAL — the engineer's version

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

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

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

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

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

■ 9.4.6 WORDS — remember these

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

9.5 Optional relationships

■ 9.5.1 PLAIN — in simple words

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

■ 9.5.2 PLAIN — a picture in your head

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

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

■ 9.5.3 PLAIN — a worked example

1. These are the newly introduced relationship annotations.

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

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

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

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

■ 9.5.4 PLAIN — what is really happening inside

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

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

■ 9.5.5 TECHNICAL — the engineer's version

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

■ 9.5.6 WORDS — remember these

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

9.6 Modelling the shop

■ 9.6.1 PLAIN — in simple words

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

■ 9.6.2 PLAIN — a picture in your head

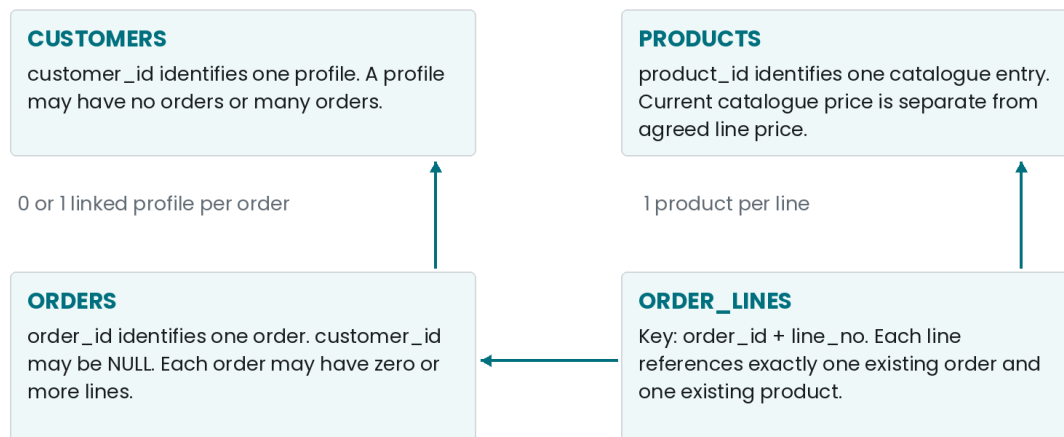
1. Picture a small records desk with an order envelope, its line cards, a catalogue index and an optional pointer to a customer profile.
2. The order envelope says which order it is and which currency applies. Each line card says which product, how many units and the agreed price for that line.
3. The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

■ 9.6.3 PLAIN — a worked example

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

FOUR CORE TABLES. THREE DIFFERENT RELATIONSHIPS.

Arrows below point from a child reference to the required or optional parent.



Foreign keys constrain child references. They do not require a parent to have children.

Agreed quantities and prices live on the line. Optional profile linkage does not erase an order.

Tags and preferences are separate teaching extensions, not hidden columns in these four tables.

Figure 9.9: Figure 9.1 — Core shop relationships: each line names one existing order and one existing product; an order optionally names one customer. Parent rows may have zero children unless another rule establishes a minimum.

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

■ 9.6.4 PLAIN — what is really happening inside

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

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

■ 9.6.5 TECHNICAL — the engineer's version

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

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

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

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

■ 9.6.6 WORDS — remember these

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

9.97 Practice and worked answers

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

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

9.98 Common wrong ideas

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

9.99 Chapter summary in 20 lines

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

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

Designing a Schema That Matches the Work

10.0 What this chapter gives you

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

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

10.1 Workflows before tables

■ 10.1.1 PLAIN — in simple words

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

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

■ 10.1.2 PLAIN — a picture in your head

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

■ 10.1.3 PLAIN — a worked example

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

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

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

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

■ 10.1.4 PLAIN — what is really happening inside

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

■ 10.1.5 TECHNICAL — the engineer’s version

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

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

■ 10.1.6 WORDS — remember these

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

10.2 Row grain and boundaries

■ 10.2.1 PLAIN — in simple words

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

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

■ 10.2.2 PLAIN — a picture in your head

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

■ 10.2.3 PLAIN — a worked example

1. Compare three ways of counting the unchanged fixture.

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

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

■ 10.2.4 PLAIN — what is really happening inside

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

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

■ 10.2.5 TECHNICAL — the engineer's version

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

■ 10.2.6 WORDS — remember these

1. **Identity boundary:** the scope inside which a label identifies something — the domain over which a key or composite key is interpreted and uniqueness is enforced.
2. **Grain contract:** a precise statement of what one row represents — documentation tying a record's unit, key, multiplicity and temporal meaning together.
3. **Participation minimum:** the least number of required related records — a cardinality condition such as "every completed order has at least one line".

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

10.3 Mutable and historical values

■ 10.3.1 PLAIN — in simple words

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

■ 10.3.2 PLAIN — a picture in your head

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

THREE VALUES, THREE DIFFERENT MEANINGS

Equal numbers today need not describe the same kind of fact.

CURRENT CATALOGUE

P-NOTE: 8,250 paise per unit after the separate price change. Answers: what is the catalogue price now?

EDITABLE DRAFT

D-SQL-01: three notebooks at 7,550 paise; one pen has no quotation yet. This is unfinished, new teaching work.

AGREED ORDER LINES

O-1042: 17,100 paise. O-1043: 11,550 paise. Their recorded notebook unit price remains 7,550 paise.

Historical combined amount: 28,650 paise.

Hypothetical amount at the changed catalogue prices: 30,750 paise.

Do not label that repricing as historical revenue. Draft finalisation and complete historical-write protection are not implemented here.

Figure 10.10: Figure 10 — Current catalogue, editable draft and agreed order lines answer different questions. The draft is new teaching data; the historical combined total remains 28,650 paise.

■ 10.3.3 PLAIN — a worked example

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

■ 10.3.4 PLAIN — what is really happening inside

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

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

■ 10.3.5 TECHNICAL — the engineer's version

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

■ 10.3.6 WORDS — remember these

1. **Current attribute:** what a record says about the present state — a value whose intended temporal interpretation follows the entity's current representation.
2. **Historical snapshot attribute:** a value copied to describe an earlier event or agreement — a stored attribute with event-specific temporal semantics, not automatic immutability.

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

10.4 Domain types

■ 10.4.1 PLAIN — in simple words

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

■ 10.4.2 PLAIN — a picture in your head

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

■ 10.4.3 PLAIN — a worked example

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

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

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

■ 10.4.4 PLAIN — what is really happening inside

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

■ 10.4.5 TECHNICAL — the engineer's version

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

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

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

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

■ 10.4.6 WORDS — remember these

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

10.5 Naming and documentation

■ 10.5.1 PLAIN — in simple words

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

■ 10.5.2 PLAIN — a picture in your head

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

■ 10.5.3 PLAIN — a worked example

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

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

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

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

■ 10.5.4 PLAIN — what is really happening inside

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

■ 10.5.5 TECHNICAL — the engineer's version

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

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

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

■ 10.5.6 WORDS — remember these

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

10.6 Reviewing the model with users

■ 10.6.1 PLAIN — in simple words

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

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

■ 10.6.2 PLAIN — a picture in your head

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

■ 10.6.3 PLAIN — a worked example

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

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

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

■ 10.6.4 PLAIN — what is really happening inside

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

■ 10.6.5 TECHNICAL — the engineer’s version

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

■ 10.6.6 WORDS — remember these

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

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

10.97 Practice and worked answers

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

Worked answers

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

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

10.98 Common wrong ideas

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

10.99 Chapter summary in 20 lines

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

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

Constraints: Rules the Database Can Enforce

11.0 What this chapter gives you

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

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

11.1 Not-null and check rules

■ 11.1.1 PLAIN — in simple words

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

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

■ 11.1.2 PLAIN — a picture in your head

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

■ 11.1.3 PLAIN — a worked example

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

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

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

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

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

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

■ 11.1.4 PLAIN — what is really happening inside

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

■ 11.1.5 TECHNICAL — the engineer's version

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

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

■ 11.1.6 WORDS — remember these

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

11.2 Unique and primary keys

■ 11.2.1 PLAIN — in simple words

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

■ 11.2.2 PLAIN — a picture in your head

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

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

■ 11.2.3 PLAIN — a worked example

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

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

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

■ 11.2.4 PLAIN — what is really happening inside

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

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

■ 11.2.5 TECHNICAL — the engineer’s version

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

■ 11.2.6 WORDS — remember these

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

11.3 Foreign keys

■ 11.3.1 PLAIN — in simple words

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

■ 11.3.2 PLAIN — a picture in your head

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

■ 11.3.3 PLAIN — a worked example

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

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

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

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

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

■ 11.3.4 PLAIN — what is really happening inside

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

■ 11.3.5 TECHNICAL — the engineer's version

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

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

■ 11.3.6 WORDS — remember these

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

11.4 Cross-row invariants

■ 11.4.1 PLAIN — in simple words

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

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

■ 11.4.2 PLAIN — a picture in your head

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

■ 11.4.3 PLAIN — a worked example

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

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

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

PASSING ONE RULE DOES NOT PROVE ANOTHER

CAP-DEMO is an isolated allocation example, not the earlier stock discrepancy.

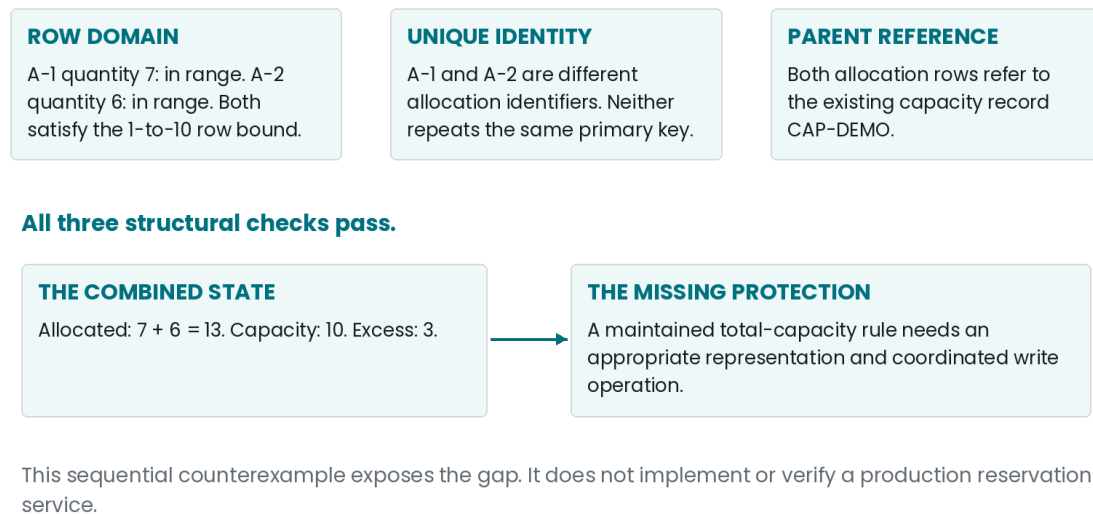


Figure 11.11: Figure 11 — Different rules protect different properties. Row checks, unique identities and parent references can all pass while the combined allocation of 13 exceeds capacity 10.

■ 11.4.4 PLAIN — what is really happening inside

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

■ 11.4.5 TECHNICAL — the engineer’s version

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

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

■ 11.4.6 WORDS — remember these

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

11.5 Deferred enforcement

■ 11.5.1 PLAIN — in simple words

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

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

■ 11.5.2 PLAIN — a picture in your head

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

■ 11.5.3 PLAIN — a worked example

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

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

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

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

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

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

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

■ 11.5.4 PLAIN — what is really happening inside

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

■ 11.5.5 TECHNICAL — the engineer’s version

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

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

■ 11.5.6 WORDS — remember these

1. **Immediate constraint:** a rule checked at its ordinary early boundary — an invariant enforced without the supported transaction-level postponement.
2. **Deferred constraint:** a rule whose check is postponed to a declared later boundary — an eligible constraint allowed to be temporarily unsatisfied within a transaction but required by successful final checking.
3. **Commit-time failure:** rejection when trying to complete a transaction — an error arising at the commit boundary rather than necessarily during an earlier statement.
4. **Transaction ownership:** responsibility for starting and ending a transaction — a contract defining which component may commit, roll back or leave the connection active.
5. **Repair branch:** a deliberate resolution of incomplete transactional work — an explicitly permitted continuation that restores the failed condition before a later commit.

11.6 Error handling and migration

■ 11.6.1 PLAIN — in simple words

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

■ 11.6.2 PLAIN — a picture in your head

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

■ 11.6.3 PLAIN — a worked example

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

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

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

■ 11.6.4 PLAIN — what is really happening inside

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

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

■ 11.6.5 TECHNICAL — the engineer's version

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

■ 11.6.6 WORDS — remember these

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

11.97 Practice and worked answers

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

Worked answers

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

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

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

11.98 Common wrong ideas

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

11.99 Chapter summary in 20 lines

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

SQL From Zero: Asking Your First Questions

12.0 What this chapter gives you

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

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

12.1 Reading `SELECT` aloud

■ 12.1.1 PLAIN — in simple words

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

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

■ 12.1.2 PLAIN — a picture in your head

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

■ 12.1.3 PLAIN — a worked example

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

```
from contextlib import closing
from schema_sql_lab import canonical_shop

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

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

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

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

■ 12.1.4 PLAIN — what is really happening inside

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

■ 12.1.5 TECHNICAL — the engineer's version

1. `SELECT` is declarative: its clauses specify result semantics, while the engine chooses an execution strategy. The simplified conceptual sequence used in this chapter is an explanatory model, not a promise about physical operator order. [S79] [S80]
2. In the basic single-table case, identify the source, apply the row predicate, evaluate output expressions, apply requested duplicate elimination, order the result and apply the row limit. Later features introduce additional stages and interactions.
3. An output column can be a source column, a literal or an expression. For example, `quantity * unit_price_minor` is an expression with the current line's agreed quantity and price as operands.
4. Keep identifier and literal syntax distinct. `order_id` names a column; `'0-1042'` is a text value. SQL text literals use single quotes in these examples. Do not teach double-quoted identifiers as a portable replacement for text literal syntax. [S72]
5. The example's Python connection is a new in-process SQLite database. PostgreSQL tutorial syntax is useful comparative documentation, but a PostgreSQL server, roles and connection lifecycle are not exercised here.
6. The Python helper closes the database after the block using `contextlib.closing`. A connection's transaction context-manager behaviour and actually closing that connection are not the same operation; the distinction is explicit in the driver documentation. [S86]
7. Basic read-only `SELECT` statements here have no intended mutation effects. Do not generalise that every possible `SELECT` in every engine is incapable of side effects; functions and product-specific features require their own contracts.

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

■ 12.1.6 WORDS — remember these

1. **SELECT statement:** a description of an answer to read — a SQL query specifying output expressions and the operations needed to define a result.
2. **Select list:** the values requested for each output row — the column references or expressions following SELECT.
3. **Text literal:** text written as a value in a statement — a quoted string such as '0-1042', distinct from a column identifier.
4. **Cursor:** the driver’s interface to a statement and its results — an object through which rows and relevant execution information are retrieved.
5. **Declarative query:** a request that describes the required result — a statement whose logical meaning does not prescribe one physical execution plan.

12.2 Filtering and NULL

■ 12.2.1 PLAIN — in simple words

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

■ 12.2.2 PLAIN — a picture in your head

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

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

■ 12.2.3 PLAIN — a worked example

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

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

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

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

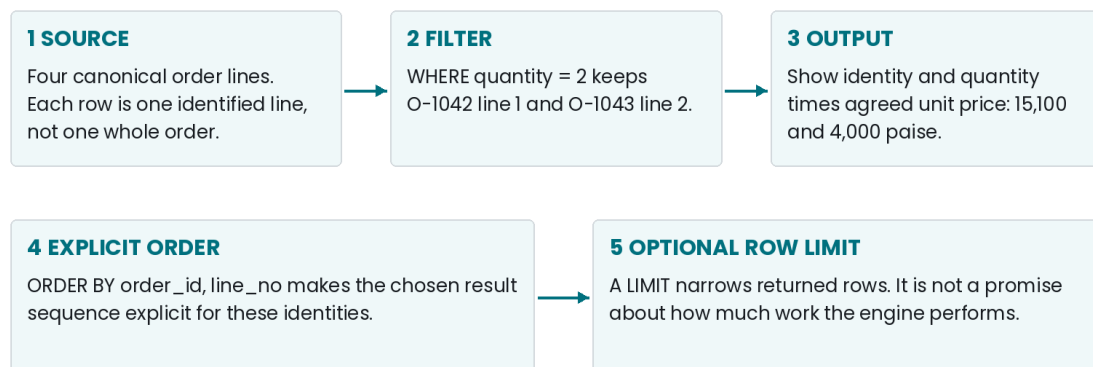
4. The answer is O-1042. Replacing IS NULL with = NULL returns no rows in the lab, not the desired missing-link result.
5. Now compare two other predicates. `customer_id <> 'C-001'` returns no rows: O-1043 equals C-001, while O-1042's comparison is unknown. Adding `OR customer_id IS NULL` returns O-1042.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

■ 12.2.4 PLAIN — what is really happening inside

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

READ A QUERY AS A PRECISE QUESTION

Conceptual result reasoning, not a required physical execution order.



Without an added limit: 2 lines, 4 units, 19,100 paise.

The query reads recorded claims. It does not prove agreement, authenticate a caller or mutate the original orders.

Indexes and query plans may produce the same answer by different physical routes.

Figure 12.12: Figure 12 — A conceptual path through the quantity-two query: source, filter, output, explicit order and an optional result limit. These are result semantics, not a required physical execution plan.

■ 12.2.5 TECHNICAL — the engineer's version

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

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

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

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

■ 12.2.6 WORDS — remember these

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

12.3 Explicit ordering

■ 12.3.1 PLAIN — in simple words

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

■ 12.3.2 PLAIN — a picture in your head

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

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

■ 12.3.3 PLAIN — a worked example

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

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

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

■ 12.3.4 PLAIN — what is really happening inside

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

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

■ 12.3.5 TECHNICAL — the engineer’s version

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

■ 12.3.6 WORDS — remember these

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

12.4 Expressions and aliases

■ 12.4.1 PLAIN — in simple words

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

■ 12.4.2 PLAIN — a picture in your head

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

■ 12.4.3 PLAIN — a worked example

1. Label every canonical line and calculate its amount:

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

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

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

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

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

■ 12.4.4 PLAIN — what is really happening inside

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

■ 12.4.5 TECHNICAL — the engineer's version

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

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

■ 12.4.6 WORDS — remember these

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

12.5 Inserting and updating safely

■ 12.5.1 PLAIN — in simple words

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

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

■ 12.5.2 PLAIN — a picture in your head

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

■ 12.5.3 PLAIN — a worked example

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

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

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

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

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

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

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

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

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

■ 12.5.4 PLAIN — what is really happening inside

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

■ 12.5.5 TECHNICAL — the engineer's version

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

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

■ 12.5.6 WORDS — remember these

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

12.6 Parameters and transactions

■ 12.6.1 PLAIN — in simple words

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

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

■ 12.6.2 PLAIN — a picture in your head

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

■ 12.6.3 PLAIN — a worked example

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

```
rows = db.execute(
    "SELECT order_id, line_no, product_id, quantity, unit_price_minor "
    "FROM order_lines WHERE order_id = ? ORDER BY line_no",
    ("0-1042",),
).fetchall()
```

2. The comma in the one-item tuple matters in Python. It makes a sequence containing one bound value rather than just a parenthesised string.
3. The lab also supplies the hostile-looking text 0-1042' OR 1=1 -- as the parameter. It returns no rows because there is no order with that literal identifier. It does not turn the comparison into “return every order”.
4. Now use a fresh draft workspace and ask for a guarded change from three notebooks to two:

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

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

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

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

■ 12.6.4 PLAIN — what is really happening inside

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

■ 12.6.5 TECHNICAL — the engineer's version

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

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

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

■ 12.6.6 WORDS — remember these

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

12.97 Practice and worked answers

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

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

Worked answers

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

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

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

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

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

12.98 Common wrong ideas

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

12.99 Chapter summary in 20 lines

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

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

Joins and Normalisation: Putting Facts in the Right Place

13.0 What this chapter gives you

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

13.1 Why facts repeat

■ 13.1.1 PLAIN — in simple words

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

■ 13.1.2 PLAIN — a picture in your head

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

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

■ 13.1.3 PLAIN — a worked example

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

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

■ 13.1.4 PLAIN — what is really happening inside

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

■ 13.1.5 TECHNICAL — the engineer's version

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

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

■ 13.1.6 WORDS — remember these

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

13.2 Joining by meaning

■ 13.2.1 PLAIN — in simple words

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

■ 13.2.2 PLAIN — a picture in your head

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

■ 13.2.3 PLAIN — a worked example

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

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


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

■ 13.2.4 PLAIN — what is really happening inside

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

■ 13.2.5 TECHNICAL — the engineer's version

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

■ 13.2.6 WORDS — remember these

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

13.3 Inner and outer joins

■ 13.3.1 PLAIN — in simple words

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

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

■ 13.3.2 PLAIN — a picture in your head

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

■ 13.3.3 PLAIN — a worked example

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

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

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

■ 13.3.4 PLAIN — what is really happening inside

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

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

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

■ 13.3.5 TECHNICAL — the engineer’s version

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

■ 13.3.6 WORDS — remember these

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

13.4 Functional dependencies

■ 13.4.1 PLAIN — in simple words

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

■ 13.4.2 PLAIN — a picture in your head

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

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

■ 13.4.3 PLAIN — a worked example

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

■ 13.4.4 PLAIN — what is really happening inside

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

■ 13.4.5 TECHNICAL — the engineer's version

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

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

■ 13.4.6 WORDS — remember these

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

13.5 Normal forms and trade-offs

■ 13.5.1 PLAIN — in simple words

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

■ 13.5.2 PLAIN — a picture in your head

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

■ 13.5.3 PLAIN — a worked example

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

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

■ 13.5.4 PLAIN — what is really happening inside

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

■ 13.5.5 TECHNICAL — the engineer's version

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

■ 13.5.6 WORDS — remember these

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

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

13.6 Avoiding accidental multiplication

■ 13.6.1 PLAIN — in simple words

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

■ 13.6.2 PLAIN — a picture in your head

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

■ 13.6.3 PLAIN — a worked example

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

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

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

■ 13.6.4 PLAIN — what is really happening inside

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

■ 13.6.5 TECHNICAL — the engineer's version

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

■ 13.6.6 WORDS — remember these

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

13.97 Practice and worked answers

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

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

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

13.98 Common wrong ideas

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

13.99 Chapter summary in 20 lines

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

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



PART C

Finding Answers

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

Scanning, Sorting and the Cost of a Question

14.0 What this chapter gives you

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

14.1 What a scan does

■ 14.1.1 PLAIN — in simple words

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

■ 14.1.2 PLAIN — a picture in your head

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

■ 14.1.3 PLAIN — a worked example

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

■ 14.1.4 PLAIN — what is really happening inside

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

■ 14.1.5 TECHNICAL — the engineer's version

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

■ 14.1.6 WORDS — remember these

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

14.2 Row width and data volume

■ 14.2.1 PLAIN — in simple words

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

■ 14.2.2 PLAIN — a picture in your head

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

■ 14.2.3 PLAIN — a worked example

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

■ 14.2.4 PLAIN — what is really happening inside

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

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

■ 14.2.5 TECHNICAL — the engineer’s version

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

■ 14.2.6 WORDS — remember these

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

14.3 Sorting and memory

■ 14.3.1 PLAIN — in simple words

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

■ 14.3.2 PLAIN — a picture in your head

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

■ 14.3.3 PLAIN — a worked example

1. Take four invented report rows: (A, 500), (B, 300), (C, 500) and (D, 200). Ordering by amount gives D, B, then the two 500 rows in an unspecified relative order unless another rule is supplied.
2. `ORDER BY amount_minor, order_id` supplies a deterministic order for these distinct identifiers: D, B, A, C. It states the business-readable tie-break rather than relying on a current storage accident.
3. A simplified comparison sort on one million entries has work on the scale of $N \log_2 N$, roughly 20 million comparison steps in an order-of-growth illustration. This is not an exact count for a database sorter.
4. If each temporary entry occupies an assumed 48 bytes, the entries alone need about 48 MB. Pointers, allocator overhead, copies and other plan nodes are outside this simple estimate.

■ 14.3.4 PLAIN — what is really happening inside

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

■ 14.3.5 TECHNICAL — the engineer's version

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

■ 14.3.6 WORDS — remember these

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

14.4 External work and spill

■ 14.4.1 PLAIN — in simple words

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

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

■ 14.4.2 PLAIN — a picture in your head

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

■ 14.4.3 PLAIN — a worked example

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

■ 14.4.4 PLAIN — what is really happening inside

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

■ 14.4.5 TECHNICAL — the engineer’s version

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

■ 14.4.6 WORDS — remember these

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

14.5 Selectivity and filtering

■ 14.5.1 PLAIN — in simple words

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

■ 14.5.2 PLAIN — a picture in your head

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

■ 14.5.3 PLAIN — a worked example

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

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

■ 14.5.4 PLAIN — what is really happening inside

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

■ 14.5.5 TECHNICAL — the engineer's version

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

■ 14.5.6 WORDS — remember these

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

14.6 Cost versus elapsed time

■ 14.6.1 PLAIN — in simple words

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

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

■ 14.6.2 PLAIN — a picture in your head

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

■ 14.6.3 PLAIN — a worked example

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

■ 14.6.4 PLAIN — what is really happening inside

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

■ 14.6.5 TECHNICAL — the engineer's version

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

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

■ 14.6.6 WORDS — remember these

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

14.97 Practice and worked answers

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

14.98 Common wrong ideas

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

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

14.99 Chapter summary in 20 lines

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

Indexes: Building a Faster Way to Find Things

15.0 What this chapter gives you

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

15.1 An extra access path

■ 15.1.1 PLAIN — in simple words

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

■ 15.1.2 PLAIN — a picture in your head

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

■ 15.1.3 PLAIN — a worked example

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

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

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

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

■ 15.1.4 PLAIN — what is really happening inside

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

■ 15.1.5 TECHNICAL — the engineer’s version

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

■ 15.1.6 WORDS — remember these

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

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

15.2 Read benefits and write costs

■ 15.2.1 PLAIN — in simple words

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

■ 15.2.2 PLAIN — a picture in your head

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

■ 15.2.3 PLAIN — a worked example

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

■ 15.2.4 PLAIN — what is really happening inside

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

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

■ 15.2.5 TECHNICAL — the engineer’s version

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

■ 15.2.6 WORDS — remember these

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

15.3 Composite key order

■ 15.3.1 PLAIN — in simple words

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

■ 15.3.2 PLAIN — a picture in your head

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

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

■ 15.3.3 PLAIN — a worked example

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

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

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

■ 15.3.4 PLAIN — what is really happening inside

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

■ 15.3.5 TECHNICAL — the engineer's version

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

■ 15.3.6 WORDS — remember these

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

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

15.4 Covering and partial indexes

■ 15.4.1 PLAIN — in simple words

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

■ 15.4.2 PLAIN — a picture in your head

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

■ 15.4.3 PLAIN — a worked example

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

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

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

■ 15.4.4 PLAIN — what is really happening inside

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

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

■ 15.4.5 TECHNICAL — the engineer’s version

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

■ 15.4.6 WORDS — remember these

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

15.5 Selectivity and distribution

■ 15.5.1 PLAIN — in simple words

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

■ 15.5.2 PLAIN — a picture in your head

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

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

■ 15.5.3 PLAIN — a worked example

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

■ 15.5.4 PLAIN — what is really happening inside

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

■ 15.5.5 TECHNICAL — the engineer's version

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

■ 15.5.6 WORDS — remember these

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

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

15.6 When the engine ignores an index

■ 15.6.1 PLAIN — in simple words

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

■ 15.6.2 PLAIN — a picture in your head

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

■ 15.6.3 PLAIN — a worked example

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

■ 15.6.4 PLAIN — what is really happening inside

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

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

■ 15.6.5 TECHNICAL — the engineer’s version

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

■ 15.6.6 WORDS — remember these

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

15.97 Practice and worked answers

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

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

15.98 Common wrong ideas

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

15.99 Chapter summary in 20 lines

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

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

B-Trees: Ordered Pages, Fast Lookups

16.0 What this chapter gives you

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

16.1 Ordered separators

■ 16.1.1 PLAIN — in simple words

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

■ 16.1.2 PLAIN — a picture in your head

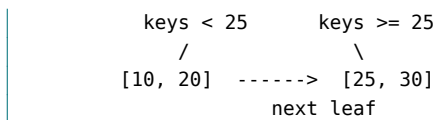
1. A large archive has a sign: numbers below 25 are in the left room; numbers 25 and above are in the right room. Inside a room, another sign may divide the range further.
2. You do not open the wrong room because the signs and filing rules agree. The value 25 belongs to the right side because that is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

■ 16.1.3 PLAIN — a worked example

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

```

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



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

■ 16.1.4 PLAIN — what is really happening inside

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

■ 16.1.5 TECHNICAL — the engineer's version

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

■ 16.1.6 WORDS — remember these

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

16.2 Pages and fan-out

■ 16.2.1 PLAIN — in simple words

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

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

■ 16.2.2 PLAIN — a picture in your head

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

■ 16.2.3 PLAIN — a worked example

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

■ 16.2.4 PLAIN — what is really happening inside

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

■ 16.2.5 TECHNICAL — the engineer’s version

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

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

■ 16.2.6 WORDS — remember these

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

16.3 Search from root to leaf

■ 16.3.1 PLAIN — in simple words

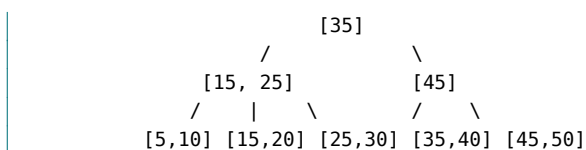
1. Searching is repeated narrowing. At each internal page, compare the target with sorted boundaries and choose exactly the child range that can contain it.
2. At the leaf, compare actual keys and return either the matching entry or an explicit not-found result. A missing key is an ordinary outcome,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

■ 16.3.2 PLAIN — a picture in your head

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

■ 16.3.3 PLAIN — a worked example

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



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

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

■ 16.3.4 PLAIN — what is really happening inside

1. With separators $s_0, s_1, \dots$, child zero contains keys below s_0 ; child one begins at s_0 ; and so on under our convention. Selecting the position after equal separators implements those intervals.
2. A binary search inside a page reduces comparison work compared with a linear search, but a very small page representation may use another strategy. The high-level tree invariant does not mandate one CPU algorithm.
3. For non-unique product keys, a conceptual composite entry (`product_id, row_identity`) supplies a total order among equal products. Another implementation can use a posting list of row locators. Either way, returning the first matching entry alone would not answer “all lines for this product.”
4. A real transaction may need to verify that a located row version is visible. Finding an index entry is a structural result; deciding whether its referenced version belongs to this query's snapshot is a separate step.

■ 16.3.5 TECHNICAL — the engineer's version

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

■ 16.3.6 WORDS — remember these

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

16.4 Inserts and splits

■ 16.4.1 PLAIN — in simple words

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

■ 16.4.2 PLAIN — a picture in your head

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

■ 16.4.3 PLAIN — a worked example

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

■ 16.4.4 PLAIN — what is really happening inside

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

■ 16.4.5 TECHNICAL — the engineer's version

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

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

■ 16.4.6 WORDS — remember these

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

16.5 Range scans

■ 16.5.1 PLAIN — in simple words

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

■ 16.5.2 PLAIN — a picture in your head

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

■ 16.5.3 PLAIN — a worked example

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

■ 16.5.4 PLAIN — what is really happening inside

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

■ 16.5.5 TECHNICAL — the engineer's version

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

■ 16.5.6 WORDS — remember these

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

16.6 Implementation boundaries

■ 16.6.1 PLAIN — in simple words

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

■ 16.6.2 PLAIN — a picture in your head

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

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

■ 16.6.3 PLAIN — a worked example

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

■ 16.6.4 PLAIN — what is really happening inside

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

■ 16.6.5 TECHNICAL — the engineer's version

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

■ 16.6.6 WORDS — remember these

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

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

16.97 Practice and worked answers

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

16.98 Common wrong ideas

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

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

16.99 Chapter summary in 20 lines

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

Hashing, Search Keys and Collisions

17.0 What this chapter gives you

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

17.1 A deterministic mapping

■ 17.1.1 PLAIN — in simple words

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

■ 17.1.2 PLAIN — a picture in your head

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

■ 17.1.3 PLAIN — a worked example

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

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

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

■ 17.1.4 PLAIN — what is really happening inside

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

■ 17.1.5 TECHNICAL — the engineer's version

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

■ 17.1.6 WORDS — remember these

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

17.2 Buckets and collisions

■ 17.2.1 PLAIN — in simple words

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

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

■ 17.2.2 PLAIN — a picture in your head

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

■ 17.2.3 PLAIN — a worked example

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

■ 17.2.4 PLAIN — what is really happening inside

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

■ 17.2.5 TECHNICAL — the engineer's version

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

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

■ 17.2.6 WORDS — remember these

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

17.3 Hash tables and equality

■ 17.3.1 PLAIN — in simple words

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

■ 17.3.2 PLAIN — a picture in your head

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

■ 17.3.3 PLAIN — a worked example

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

■ 17.3.4 PLAIN — what is really happening inside

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

■ 17.3.5 TECHNICAL — the engineer’s version

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

■ 17.3.6 WORDS — remember these

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

17.4 Cryptographic and non-cryptographic purposes

■ 17.4.1 PLAIN — in simple words

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

■ 17.4.2 PLAIN — a picture in your head

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

■ 17.4.3 PLAIN — a worked example

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

■ 17.4.4 PLAIN — what is really happening inside

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

■ 17.4.5 TECHNICAL — the engineer's version

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

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

■ 17.4.6 WORDS — remember these

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

17.5 Distribution and adversarial inputs

■ 17.5.1 PLAIN — in simple words

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

■ 17.5.2 PLAIN — a picture in your head

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

■ 17.5.3 PLAIN — a worked example

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

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

■ 17.5.4 PLAIN — what is really happening inside

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

■ 17.5.5 TECHNICAL — the engineer’s version

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

■ 17.5.6 WORDS — remember these

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

17.6 Hashing in storage designs

■ 17.6.1 PLAIN — in simple words

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

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

■ 17.6.2 PLAIN — a picture in your head

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

■ 17.6.3 PLAIN — a worked example

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

■ 17.6.4 PLAIN — what is really happening inside

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

■ 17.6.5 TECHNICAL — the engineer’s version

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

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

■ 17.6.6 WORDS — remember these

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

17.97 Practice and worked answers

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

17.98 Common wrong ideas

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

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

17.99 Chapter summary in 20 lines

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

Query Plans and the Optimiser

18.0 What this chapter gives you

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

18.1 From SQL to a plan

■ 18.1.1 PLAIN — in simple words

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

■ 18.1.2 PLAIN — a picture in your head

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

■ 18.1.3 PLAIN — a worked example

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

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

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

■ 18.1.4 PLAIN — what is really happening inside

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

■ 18.1.5 TECHNICAL — the engineer's version

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

■ 18.1.6 WORDS — remember these

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

18.2 Statistics and estimates

■ 18.2.1 PLAIN — in simple words

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

■ 18.2.2 PLAIN — a picture in your head

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

■ 18.2.3 PLAIN — a worked example

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

■ 18.2.4 PLAIN — what is really happening inside

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

■ 18.2.5 TECHNICAL — the engineer's version

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

■ 18.2.6 WORDS — remember these

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

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

18.3 Join strategies

■ 18.3.1 PLAIN — in simple words

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

■ 18.3.2 PLAIN — a picture in your head

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

■ 18.3.3 PLAIN — a worked example

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

■ 18.3.4 PLAIN — what is really happening inside

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

■ 18.3.5 TECHNICAL — the engineer's version

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

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

■ 18.3.6 WORDS — remember these

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

18.4 Access-path choices

■ 18.4.1 PLAIN — in simple words

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

■ 18.4.2 PLAIN — a picture in your head

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

■ 18.4.3 PLAIN — a worked example

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

■ 18.4.4 PLAIN — what is really happening inside

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

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

■ 18.4.5 TECHNICAL — the engineer's version

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

■ 18.4.6 WORDS — remember these

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

18.5 Reading EXPLAIN

■ 18.5.1 PLAIN — in simple words

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

■ 18.5.2 PLAIN — a picture in your head

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

■ 18.5.3 PLAIN — a worked example

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

```

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

```

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

■ 18.5.4 PLAIN — what is really happening inside

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

■ 18.5.5 TECHNICAL — the engineer's version

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

■ 18.5.6 WORDS — remember these

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

18.6 Bad estimates and safe intervention

■ 18.6.1 PLAIN — in simple words

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

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

■ 18.6.2 PLAIN — a picture in your head

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

■ 18.6.3 PLAIN — a worked example

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

■ 18.6.4 PLAIN — what is really happening inside

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

■ 18.6.5 TECHNICAL — the engineer's version

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

■ 18.6.6 WORDS — remember these

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

18.97 Practice and worked answers

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

18.98 Common wrong ideas

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

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

18.99 Chapter summary in 20 lines

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

Aggregation, Windows and Useful Reports

19.0 What this chapter gives you

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

19.1 Groups and their grain

■ 19.1.1 PLAIN — in simple words

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

■ 19.1.2 PLAIN — a picture in your head

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

■ 19.1.3 PLAIN — a worked example

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

| ORDER BY order_id;

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

■ 19.1.4 PLAIN — what is really happening inside

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

■ 19.1.5 TECHNICAL — the engineer's version

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

■ 19.1.6 WORDS — remember these

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

19.2 Counts, sums and averages

■ 19.2.1 PLAIN — in simple words

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

■ 19.2.2 PLAIN — a picture in your head

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

■ 19.2.3 PLAIN — a worked example

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

■ 19.2.4 PLAIN — what is really happening inside

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

■ 19.2.5 TECHNICAL — the engineer's version

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

■ 19.2.6 WORDS — remember these

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

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

19.3 Windows versus groups

■ 19.3.1 PLAIN — in simple words

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

■ 19.3.2 PLAIN — a picture in your head

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

■ 19.3.3 PLAIN — a worked example

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

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

■ 19.3.4 PLAIN — what is really happening inside

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

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

■ 19.3.5 TECHNICAL — the engineer's version

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

■ 19.3.6 WORDS — remember these

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

19.4 Time buckets

■ 19.4.1 PLAIN — in simple words

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

■ 19.4.2 PLAIN — a picture in your head

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

■ 19.4.3 PLAIN — a worked example

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

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

■ 19.4.4 PLAIN — what is really happening inside

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

■ 19.4.5 TECHNICAL — the engineer's version

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

■ 19.4.6 WORDS — remember these

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

19.5 Distinctness and duplicates

■ 19.5.1 PLAIN — in simple words

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

■ 19.5.2 PLAIN — a picture in your head

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

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

■ 19.5.3 PLAIN — a worked example

1. The separate O-REPEAT fixture has two different line numbers with the same notebook amount, 7,550 each. Counting distinct amounts gives one value; counting lines gives two; summing amounts gives 15,100.
2. A redelivered copy of (0-REPEAT, 1) with identical payload is different from line 2. Under a declared input policy, the repeated line key can be recognised as a replay while line 2 remains a separate contribution.
3. If the replay changes quantity, it is a conflicting same-key observation, not an ordinary duplicate. The earlier import and event-consumer policies must be respected rather than replaced by a new “last one wins” rule.

■ 19.5.4 PLAIN — what is really happening inside

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

■ 19.5.5 TECHNICAL — the engineer’s version

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

■ 19.5.6 WORDS — remember these

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

19.6 Reconciliation totals

■ 19.6.1 PLAIN — in simple words

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

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

■ 19.6.2 PLAIN — a picture in your head

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

■ 19.6.3 PLAIN — a worked example

1. For the canonical report, expected line identities are (0-1042, 1), (0-1042, 2), (0-1043, 1) and (0-1043, 2). Expected counts are four lines, two orders and six units.
2. Expected order totals are 17,100 and 11,550; expected product totals are 22,650 and 6,000. Both routes sum to 28,650.
3. Add an explicitly separate empty-order fixture. A report sourced only from order lines still has no row for that order. A report sourced from all orders with a left join can include it, with a chosen zero-line interpretation. The denominator must say which report is intended.
4. A useful report header therefore includes population, period, currency, missing-value treatment, source snapshot and transformation version before presenting a large number.

■ 19.6.4 PLAIN — what is really happening inside

1. Reconciliation compares a defined expected set with the observed contributions, then classifies differences. Missing, extra, duplicated and changed contributions require different explanations.
2. Checks at several grains help locate a defect: overall amount, order totals, product quantities and original line identities. Agreement at one grain cannot substitute for all the others.
3. Save the evidence needed to reproduce the result without exposing unnecessary personal data. A diagnostic report should not become an uncontrolled copy of every sensitive field.

■ 19.6.5 TECHNICAL — the engineer's version

1. A reconciliation contract specifies source snapshot, contribution identity, inclusion predicate, units, arithmetic, grouping and tolerances where appropriate. Exact integer-paise examples use exact equality; uncertain measurements may require a separately justified tolerance.
2. Treat report generation as a versioned transformation. Reproducing a number requires both the relevant source state and the transformation's semantics, not merely a saved SQL filename.
3. These educational checks validate the stated fixture and exercises. They do not certify live financial accounts, legal reporting obligations or an organisation's complete data lineage.

■ 19.6.6 WORDS — remember these

1. **Contribution identity:** which source fact a result uses — a stable scoped identifier for a counted or summed input contribution. **Exclusion register:** explain what did not contribute — recorded

counts and reasons for records outside the report's accepted population. **Transformation version:** identify the calculation rules used — a versioned definition of parsing, filtering, joining and aggregation semantics.

19.97 Practice and worked answers

1. **Question:** Calculate mean order amount, mean line amount and weighted unit price for the canonical fixture. **Answer:** 14,325; 7,162.5; and 4,775 paise respectively. The denominators are two orders, four lines and six units.
2. **Question:** Why does averaging two branch averages usually fail? **Answer:** Branches may have different numbers of contributing observations. Combine sums and counts, or apply the correct weights.
3. **Question:** What running totals appear within O-1042 under line-number ordering? **Answer:** 15,100 followed by 17,100. The detail rows remain present.
4. **Question:** Why can a RANGE running sum for 100,100,250 start at 200? **Answer:** The first two rows are peers under value ordering, and the frame includes that peer group. Use an explicit ROWS frame and unique tie-breaker for a specific row-by-row interpretation.
5. **Question:** An event occurs exactly at a period's exclusive end. Where does it go? **Answer:** Into the next adjoining half-open period, assuming the same time basis and no gap.
6. **Question:** Does one distinct amount mean one sale? **Answer:** No. Separate identified sales can have equal amounts.
7. **Question:** A SUM ignores one unknown draft price. Can it be labelled the complete quote? **Answer:** No. Report the known subtotal and missing contribution, or withhold a complete total under the chosen policy.
8. **Question:** Which canonical controls supplement the 28,650-paise grand total? **Answer:** Four line identities, two orders, six units, the two order totals and the two product totals.

19.98 Common wrong ideas

1. **Wrong:** every count measures the same thing. **Right:** rows, known values, distinct identities and units differ.
2. **Wrong:** all averages can be averaged again. **Right:** weights and sufficient statistics are required.
3. **Wrong:** NULL is a free item. **Right:** missing and known zero have different meanings.
4. **Wrong:** a window removes detail rows like GROUP BY. **Right:** window calculations can retain each detail row.
5. **Wrong:** ordering inside OVER guarantees display order. **Right:** the final result needs its own ORDER BY.
6. **Wrong:** a calendar day always means 24 elapsed hours. **Right:** calendar and timezone rules define the bucket.
7. **Wrong:** DISTINCT discovers business identity. **Right:** it applies equality to the expressions supplied.
8. **Wrong:** a matching grand total completes reconciliation. **Right:** identities, populations and explained exclusions matter too.

19.99 Chapter summary in 20 lines

1. A useful report begins with a defined population and grain.

2. Grouping changes what one output row represents.
3. WHERE qualifies input rows and HAVING qualifies groups.
4. Count rows, known values, identities and units deliberately.
5. An average's denominator is part of its meaning.
6. Weighted means preserve the intended contribution sizes.
7. Subgroup means need weights or sufficient statistics before recombination.
8. Missing prices do not become known zero prices.
9. Empty input and observed zero can require different presentation.
10. Windows calculate across related rows while retaining detail.
11. Partition, ordering and frame are separate choices.
12. Peer-inclusive frames differ from row-by-row frames.
13. Ranking functions treat ties differently.
14. Time buckets need a clock basis, timezone and boundary policy.
15. Half-open adjacent intervals avoid double-counting their shared boundary.
16. Late arrivals require an explicit report-revision policy.
17. Distinct expression values are not automatically distinct business events.
18. Deduplication requires scoped identity and conflict handling.
19. Reconcile contribution keys and intermediate totals, not only the grand total.
20. Preserve source and transformation versions so the report can be explained later.

Measuring Performance Without Fooling Yourself

20.0 What this chapter gives you

1. A performance result is an observation about a defined workload, environment and measurement boundary. Without those definitions, a fast number can be true and still misleading.
2. You will specify a benchmark, calculate latency percentiles under an explicit convention, separate throughput from latency, and report errors and limitations instead of selecting only attractive runs.
3. The companion experiment is a local educational microbenchmark, not a capacity test of KedByte, a comparison of database brands or a claim about production hardware.

20.1 Define the workload

■ 20.1.1 PLAIN — in simple words

1. Before timing, say what the system must do correctly. Two queries that return different populations are not competing implementations of the same task.
2. Specify data size and distribution, query parameters, selected fields, concurrency and the mix of reads and writes. “One million rows” alone leaves most of the workload undefined.
3. Decide where the timer starts and stops. Measuring only server execution differs from measuring connection setup, result transfer, decoding and application work as well.

■ 20.1.2 PLAIN — a picture in your head

1. Timing a runner requires a marked course and finish line. One runner cannot stop after 80 metres while another completes 100 and still be compared as though they ran the same race.
2. A race with hurdles is also different from an empty track, even when the distance is equal.
3. **Where the comparison breaks:** database workloads can include many simultaneous operations and changing state. There may be no single finish line for the entire service, so define per-operation and experiment-level boundaries separately.

■ 20.1.3 PLAIN — a worked example

1. Our query experiment compares the same fixed product-like lookup on separately generated teaching rows before and after a candidate index. It checks the complete ordered result against an independent expected list.
2. Dataset creation and index construction are recorded separately from repeated query execution. The query timer includes fetching all result rows into the local client process.

3. Rare, common and absent keys form separate cases. A combined summary, when used, must specify their weighting rather than silently letting the fastest case dominate.
4. The four canonical order lines remain a correctness fixture. They are not expanded by pretending the same four sales happened thousands of times in the story.

■ 20.1.4 PLAIN — what is really happening inside

1. A workload generator determines arrivals and parameters. The database executes work, while the client observes successes, failures and durations within its timer boundary.
2. A closed-loop generator waits for one response before issuing more work at a fixed number of clients. An open-loop generator schedules arrivals independently of previous completion. Under overload, they expose different queueing behaviour.
3. If the generator slows down whenever the system slows down, it can omit the waiting that independently arriving users would experience. State the arrival model rather than claiming that every benchmark measures the same service demand.

■ 20.1.5 TECHNICAL — the engineer's version

1. A benchmark contract includes schema, indexes, data-generation code and seed, parameter distribution, operation mix, concurrency, arrival process, warm-up, timer boundary, sample count and correctness oracle.
2. Use a monotonic timer for elapsed intervals. Python's `perf_counter_ns()` avoids floating-point timestamp subtraction and returns integer nanoseconds; its unit is not an accuracy guarantee. [S102]
3. Retain exact runtime and library versions. A reproducible dataset does not make two machines' execution conditions identical, so reproducibility of inputs and portability of timing results are separate claims.

■ 20.1.6 WORDS — remember these

1. **Benchmark contract:** the rules of the measurement — a specification of inputs, operations, environment, timing boundaries and success criteria. **Closed-loop load:** clients wait before issuing their next request — a workload in which completion controls later arrivals. **Open-loop load:** arrivals follow an independent schedule — a workload whose offered demand does not automatically slow with each response.

20.2 Latency distributions

■ 20.2.1 PLAIN — in simple words

1. Latency is not one fixed property. The same operation can complete quickly most of the time and occasionally take much longer.
2. An average compresses those differences. Percentiles describe positions in an ordered sample, but their calculation convention and sample size matter.
3. Keep failed and timed-out attempts visible. Reporting only successful durations can make an overloaded system look faster by discarding its worst experiences.

■ 20.2.2 PLAIN — a picture in your head

1. Nineteen customers wait ten seconds and one waits a thousand. Saying “the average wait was 59.5 seconds” is mathematically correct but does not describe either group's experience well.

2. Sorting the waits lets us ask what a chosen fraction of customers experienced.
3. **Where the comparison breaks:** a small observed sample does not establish the long-run distribution. A percentile estimated from twenty requests is not reliable evidence about rare one-in-ten-thousand delays.

■ 20.2.3 PLAIN — a worked example

1. Take an invented sample of twenty latencies: nineteen at 10 ms and one at 1,000 ms. The mean is $(19 \times 10 + 1,000) / 20 = 59.5$ ms.
2. This lab uses the nearest-rank convention: for fraction p with $0 < p \leq 1$, select sorted position $\text{ceil}(pN)$, counting from one. The median under this convention is 10 ms; p95 selects position 19 and is 10 ms; p99 selects position 20 and is 1,000 ms.
3. Another percentile interpolation convention can give a different numerical p95. Label the convention instead of treating one library's output as the only mathematically possible percentile.
4. These are calculated teaching values, not measured database latencies. The benchmark's raw observed samples are stored separately in its evidence output.

■ 20.2.4 PLAIN — what is really happening inside

1. Latency combines active work and waits within the chosen boundary. Cache misses, locks, scheduling, storage and result size can create different parts of the distribution.
2. Histograms and raw samples preserve more shape than a single mean. Histogram bucket boundaries limit the precision of derived quantiles; raw samples still have sampling uncertainty.
3. A timeout is a censored observation: the client knows it waited at least until its deadline without obtaining the desired completion. Recording the deadline as though it were the actual completion time understates what is unknown.

■ 20.2.5 TECHNICAL — the engineer's version

1. State whether a percentile describes all attempts, successful attempts or another population. Retain attempt counts, success counts, timeout counts and error classes alongside duration summaries.
2. Quantile convention, sample size and workload stationarity affect interpretation. Do not average p99 values from separate groups and call the result the combined p99; combine compatible underlying observations or histogram counts instead.
3. User-oriented monitoring benefits from latency distributions together with traffic, errors and saturation, rather than one average alone. [S103]

■ 20.2.6 WORDS — remember these

1. **Percentile:** a position in an ordered population or sample — a quantile defined under a stated convention, such as nearest rank. **Tail latency:** the unusually slow end of the distribution — high-percentile delays whose meaning depends on the measured population and sample size. **Censored observation:** completion time is only partly known — an observation limited by a timeout or stopping rule rather than a measured final duration.

20.3 Throughput and saturation

■ 20.3.1 PLAIN — in simple words

1. Throughput counts completed work per unit time. Latency measures how long an operation takes. A system can complete many operations per second while each operation waits in a long queue.
2. As demand approaches a bottleneck's capacity, waiting can grow. Adding clients does not guarantee proportionally more completed work.
3. Count useful, correct completions separately from errors or retries. A server returning failures very quickly has high response throughput but may provide little successful service.

■ 20.3.2 PLAIN — a picture in your head

1. One checkout counter can serve customers only so quickly. More people joining the queue increase waiting, not necessarily the number served each minute.
2. Opening a second counter helps only if the shared payment terminal, stock lookup or packing station is not the real bottleneck.
3. **Where the comparison breaks:** computer workloads can overlap CPU, storage and network work, and requests have different costs. One checkout's fixed service time is an intentionally simplified model.

■ 20.3.3 PLAIN — a worked example

1. In a stable hypothetical system, 100 completed requests per second spend an average 0.2 seconds within a defined service boundary. Little's-law reasoning gives an average $100 \times 0.2 = 20$ requests inside that boundary, under its steady-state accounting assumptions.
2. That count includes any waiting represented by the latency boundary. It is not automatically twenty actively executing CPU tasks or twenty database connections.
3. Suppose offered load increases to 200 requests per second while sustainable completion remains 100 and nothing rejects or slows arrivals. The queue grows by roughly 100 requests per second in this simplified interval.
4. A bounded system must instead apply some policy: admission limits, backpressure, deadlines, rejection or additional capacity. Infinite queues are not a service-quality strategy.

■ 20.3.4 PLAIN — what is really happening inside

1. Bottlenecks can move as load changes. CPU may dominate one workload, storage another, and locks or a connection pool another.
2. Retry traffic can amplify overload. Counting retries as independent fresh business operations can also overstate useful throughput and create correctness problems.
3. Plot or tabulate offered demand, successful completions, errors, queueing and latency together. A throughput plateau with growing latency suggests saturation, but further evidence is needed to identify the constrained resource.

■ 20.3.5 TECHNICAL — the engineer's version

1. Under suitable steady-state conditions and consistent boundaries, Little's law is $L = \lambda W$: average number in the system equals average effective arrival/completion rate times average time in the system. It is an accounting relation, not a universal capacity forecast for transient overload.
2. Distinguish offered rate, admitted rate, completed-attempt rate and successfully completed business-operation rate. Retries and rejected requests separate these quantities.

3. Report saturation signals alongside user outcomes. Resource utilisation alone can miss queueing at a logical bottleneck, and high utilisation is not automatically harmful when latency and reliability requirements remain satisfied. [S103]

■ 20.3.6 WORDS — remember these

1. **Throughput:** completed work per unit time — a rate whose counted operation and success condition must be defined. **Saturation:** a limiting resource is at or near effective capacity — a condition in which additional demand tends to create waiting, rejection or deterioration rather than proportional useful work. **Backpressure:** make upstream work respect downstream limits — a mechanism that slows or bounds production when consumers cannot keep up.

20.4 Warm versus cold state

■ 20.4.1 PLAIN — in simple words

1. A repeated operation can benefit from work and data retained by earlier runs. This warm state may be realistic for frequent requests, but it must be named.
2. Cold state is not one universal condition. A new connection can still use database, operating-system or device caches warmed by another process.
3. Do not clear caches on a live machine merely to manufacture a benchmark. Use an isolated test environment and record what was actually reset.

■ 20.4.2 PLAIN — a picture in your head

1. Dev's first trip finds the right cupboard and brings folders to the desk. A second question about those folders avoids the trip.
2. Starting a new notebook for timings does not move the folders back to a remote archive.
3. **Where the comparison breaks:** there are several cache layers, each with its own lifetime and replacement policy. A single "desk empty" analogy cannot establish the state of all of them.

■ 20.4.3 PLAIN — a worked example

1. In the educational microbenchmark, both variants receive a stated warm-up phase before measured repetitions. The comparison is labelled warm, in-process query execution on generated data.
2. Reopening an in-memory SQLite database with newly inserted rows includes a different preparation history from reopening a file already read by the operating system. Neither should be called a device-cold test without further evidence.
3. Alternate the order of variants or use a recorded random order to reduce systematic first-run advantage. Save all samples, including the first measured one.
4. Index construction remains a separately reported preparation cost. Excluding it from steady-state query timing is acceptable only when that boundary is explicit.

■ 20.4.4 PLAIN — what is really happening inside

1. Warm state can include pages, prepared statements, code paths, allocator state, connection establishment and storage caches. It can also include undesirable accumulated state such as a growing queue.
2. Background work can occur between runs: checkpoints, cleanup or other tenants' activity. A warm-up phase does not make the rest of the environment stationary.

3. Compare the use case you need. Startup latency, infrequent cold retrieval and sustained warm service are different workloads, each potentially important.

■ 20.4.5 TECHNICAL — the engineer's version

1. Define reset scope precisely: process, connection, database buffer pool, filesystem cache, device state or dataset reconstruction. Do not claim a wider reset than the procedure demonstrates.
2. In-memory SQLite timing omits network transport and does not exercise persistent-device durability. That can be appropriate for a teaching comparison but cannot establish server or storage capacity.
3. Counterbalancing run order reduces one source of bias; it does not eliminate all thermal, scheduling or shared-host effects. Record those limitations rather than reporting false environmental control.

■ 20.4.6 WORDS — remember these

1. **Warm state:** earlier work remains useful — cached data or prepared resources available to later operations under a specified scope. **Reset scope:** what was actually returned to a starting condition — the exact layers or resources reinitialised before a measurement. **Counterbalancing:** vary which option runs first — an experimental ordering technique intended to reduce systematic sequence effects.

20.5 Fair comparisons

■ 20.5.1 PLAIN — in simple words

1. Compare equal work with equal correctness requirements. Disabling durability, dropping validation or returning fewer fields changes the service being measured.
2. Use enough repetitions to expose variation, but do not confuse many nearly identical local samples with evidence about every production condition.
3. Preserve unsuccessful runs and reasons for exclusion. A result selected after seeing which measurements look attractive is not a neutral comparison.

■ 20.5.2 PLAIN — a picture in your head

1. Two delivery services cannot be compared fairly if one includes packaging and insurance while the other only moves an unwrapped box across the room.
2. The cheaper service may still be useful, but the difference must be stated rather than hidden inside the word “faster.”
3. **Where the comparison breaks:** software guarantees can be subtle. Two functions returning the same value once may differ under retries, concurrent access or failure, so equal happy-path output is necessary but not sufficient.

■ 20.5.3 PLAIN — a worked example

1. Before timing a query variant, compare its ordered rows with an independently computed expected result. A variant that loses one valid duplicate fails correctness even if its duration is lower.
2. Record the same generated dataset, parameter cases and selected fields for both variants. Keep schema and settings differences limited to the proposed intervention unless additional differences are explicitly part of the comparison.

3. Use a fixed number of warm-ups and measured repetitions decided before inspecting the results. Summarise the samples with the same convention.
4. A result such as “variant B was faster in these warm local samples” is narrower and more defensible than “B scales to ten thousand users.” The latter requires a different workload and capacity experiment.

■ 20.5.4 PLAIN — what is really happening inside

1. A benchmark harness itself consumes resources. Data generation, logging, hashing results and printing can dominate a tiny operation if placed inside its timer boundary.
2. Conversely, timing only a cursor creation can omit the work performed when rows are fetched. Ensure the measurement includes the actual completion condition.
3. A trusted correctness check can be outside the repeated timing loop when the workload is deterministic, but it must still run for every variant and relevant parameter case. Record where checks occur.

■ 20.5.5 TECHNICAL — the engineer’s version

1. Predefine the estimand: the quantity the comparison aims to estimate. Examples include median local fetch latency, successful throughput under a fixed arrival process or p99 end-to-end latency at a stated admitted load.
2. Avoid a durability or isolation mismatch between variants. Different guarantees can be compared as different services, but not silently labelled equivalent implementations.
3. Do not impose a universal speed assertion in a functional test. The educational suite checks result equality and valid measurement records; host scheduling can change timing without making the algorithm incorrect.

■ 20.5.6 WORDS — remember these

1. **Estimand:** the exact quantity a study seeks — the defined performance or outcome measure to which observations are intended to speak. **Harness overhead:** work caused by the measurement setup — time and resources consumed by generation, instrumentation and reporting rather than the target operation alone. **Correctness oracle:** a separate expected-answer check — a mechanism that rejects a fast but semantically wrong variant.

20.6 Reporting limitations

■ 20.6.1 PLAIN — in simple words

1. A useful performance report makes its boundaries easy to see. Readers should know what was measured, what was excluded and what cannot be concluded.
2. Include actual versions, dataset construction, sample counts, result checks and failures. Label simulated or calculated figures as such.
3. Do not turn a local educational run into a claim about production readiness, security, recovery or the number of schools a server can safely host.

■ 20.6.2 PLAIN — a picture in your head

1. A laboratory label says which material was tested, under what load and for how long. It does not simply say “strong.”
2. Performance evidence needs the same discipline: the label travels with the result.

3. **Where the comparison breaks:** software and workloads change quickly. A benchmark's input files and version identifiers are often necessary to understand a result later, even when the machine's model name remains unchanged.

■ 20.6.3 PLAIN — a worked example

1. A minimal record includes: Python and SQLite versions; generated row count and distribution; query text and parameter case; schema/index variant; warm-up and repetition counts; timer boundary; raw nanosecond samples; ordered result digest; and any errors.
2. The result digest is a convenient reproducibility check, not an authentication or semantic proof by itself. The independently expected rows remain the correctness basis.
3. Add an interpretation such as: "Warm single-process SQLite query comparison; no network, multi-user load, persistent-device failure or PostgreSQL server tested."
4. This explicit limitation does not make the result worthless. It tells the reader exactly which small question was answered and which larger experiments remain.

■ 20.6.4 PLAIN — what is really happening inside

1. Reproducibility records separate stable inputs from variable observations. The same code and seed can reproduce the data while observed timings differ on another run.
2. Keep raw measurements so summaries can be recalculated under another convention. Preserve original failures rather than overwriting them with a later passing run.
3. A benchmark informs a decision together with correctness, maintainability, operational costs and service requirements. It is not the entire decision.

■ 20.6.5 TECHNICAL — the engineer's version

1. The companion benchmark emits a machine-readable record and makes no host-independent speed guarantee. Functional tests validate its schema, sample count, non-negative durations and result equivalence, not a predetermined winner.
2. Report measurement uncertainty and environmental limits at the level the experiment supports. A narrow microbenchmark does not warrant an invented confidence interval for production throughput.
3. Future capacity work should connect latency, successful throughput, errors and saturation under realistic demand, including failure and retry behaviour. [S103]

■ 20.6.6 WORDS — remember these

1. **Raw sample:** an individual retained observation — an unaggregated measurement from which summaries can be recomputed. **Measurement boundary:** what the timer includes — the explicitly chosen start and completion conditions of an observed operation. **External validity:** whether a result transfers beyond its test — the justified scope of applying observations to other workloads, environments or populations.

20.97 Practice and worked answers

1. **Question:** Nineteen requests take 10 ms and one takes 1,000 ms. Find the mean and nearest-rank p99. **Answer:** The mean is 59.5 ms; p99 selects the twentieth observation and is 1,000 ms.
2. **Question:** Why can two libraries disagree on p95 for the same twenty samples? **Answer:** Quantile interpolation conventions differ. State the convention and retain raw samples.

3. **Question:** At 100 completions per second and 0.2 seconds average time in a stable boundary, what average population does Little's law imply? **Answer:** Twenty operations within that boundary, not necessarily twenty active CPU workers.
4. **Question:** Can reopening a connection prove a cold-device benchmark? **Answer:** No. Other cache layers can remain warm.
5. **Question:** A variant is faster because it stops fetching after the first row. Is it comparable to a full-result variant? **Answer:** Only if the required task is explicitly first-row retrieval. Otherwise the completion boundaries differ.
6. **Question:** Should failed attempts disappear from a successful-latency chart? **Answer:** They may be excluded from that specifically labelled distribution, but their counts and outcomes must remain visible alongside it.
7. **Question:** Why should a functional test avoid asserting that an index is always faster? **Answer:** Timing depends on host and workload conditions. The test can verify correct results and valid measurement records without inventing a universal performance bound.
8. **Question:** What does the local in-memory benchmark not establish? **Answer:** Networked service capacity, multi-host behaviour, persistent-device durability, production isolation or a supported user count.

20.98 Common wrong ideas

1. **Wrong:** one average describes every user's experience. **Right:** distributions and failures reveal important differences.
2. **Wrong:** percentiles have one universal calculation convention. **Right:** label the method and sample population.
3. **Wrong:** more clients always increase useful throughput. **Right:** saturation can increase waiting and errors instead.
4. **Wrong:** a new connection means everything is cold. **Right:** reset scope must be demonstrated.
5. **Wrong:** a faster result is valid even when guarantees changed. **Right:** compare equal services or disclose the difference.
6. **Wrong:** the benchmark harness has no cost. **Right:** setup, fetching and instrumentation boundaries matter.
7. **Wrong:** enough local repetitions prove production scalability. **Right:** the workload and environment determine transferability.
8. **Wrong:** limitations weaken an otherwise strong result. **Right:** they prevent the result being used to support a claim it never tested.

20.99 Chapter summary in 20 lines

1. Define the required correct operation before timing it.
2. Record data distribution, parameters, concurrency and operation mix.
3. A timer boundary determines what the duration means.
4. Closed-loop and open-loop arrivals expose different overload behaviour.
5. Use a monotonic timer for elapsed intervals.
6. Nanosecond units do not imply nanosecond accuracy.
7. Latency is a distribution rather than one constant.
8. Percentile convention and sample size affect interpretation.

9. Keep timeouts and errors visible.
10. Throughput needs a defined successful unit of work.
11. Queueing can grow while completion rate stops increasing.
12. Little's law requires consistent boundaries and suitable steady-state assumptions.
13. Warm state exists at several layers.
14. Reset only what the procedure can legitimately establish.
15. Compare identical result and guarantee contracts.
16. Account for harness overhead and complete result retrieval.
17. Retain raw samples and predefined exclusion rules.
18. Do not build a universal speed assertion into a functional test.
19. Report actual versions, inputs, observations and limitations together.
20. A bounded experiment answers a bounded question, not every deployment decision.



PART D

Correct Changes

Keep related changes consistent when requests overlap, fail or arrive more than once.

Transactions: One Change, All or Nothing

21.0 What this chapter gives you

1. A useful business action can require several database changes. This chapter explains how to give those changes one commit boundary without pretending that a database transaction controls the whole outside world.
2. You will distinguish atomicity, consistency, isolation and durability; recover from a failed step; use savepoints; and identify effects that remain outside rollback.
3. The stock and request examples introduced here are separate synthetic exercises. They do not explain the earlier unresolved expected-stock-10/observed-stock-9 discrepancy or alter the canonical agreed orders.

21.1 The boundary of a change

■ 21.1.1 PLAIN — in simple words

1. Suppose accepting a reservation must both reduce available stock and record which request received it. Saving only one of those changes leaves an incomplete account of the operation.
2. A transaction gives a selected collection of database work one all-or-nothing boundary. The application decides which related operations belong together and whether their results satisfy its rules.
3. Putting statements inside BEGIN and COMMIT does not make a wrong rule right. A transaction can reliably commit the wrong amount if the application supplied the wrong calculation.

■ 21.1.2 PLAIN — a picture in your head

1. Mira prepares a reservation slip and the corresponding stock adjustment inside one envelope. The envelope is either accepted as a complete unit or rejected before becoming the official record.
2. The envelope boundary makes it harder to accept the slip while forgetting its stock change.
3. **Where the comparison breaks:** a transaction is not literally a hidden envelope containing every effect. Network calls, printed receipts and handed-over goods are outside its ordinary database boundary. Visibility also depends on the engine's isolation rules.

■ 21.1.3 PLAIN — a worked example

1. In a new teaching inventory, product TX-PEN has five available units. Request TX-R1 asks for three. A successful operation should leave two units and one accepted request record for three units.
2. The incomplete outcomes are easy to name: stock becomes two with no request record, or the request is accepted while stock remains five. Either could mislead later reconciliation.

3. Group the guarded stock change and request insertion into one owned transaction. If either step fails, roll back both transactional changes. If both succeed and all required checks pass, commit.
4. This models reservation records, not physical delivery, payment, authentication or a production checkout.

■ 21.1.4 PLAIN — what is really happening inside

1. The database tracks the transaction's changes and coordinates their visibility and persistence under its documented mechanisms.
2. The application must inspect outcomes inside the boundary. An UPDATE affecting zero rows is often a successful SQL statement but an unsuccessful business precondition.
3. Keep the transaction no larger than necessary for the invariant. Waiting for a human to answer a form while holding a transaction open can retain locks, snapshots and resources unnecessarily.

■ 21.1.5 TECHNICAL — the engineer's version

1. Define a transaction's read set, write set, invariants and completion condition. The transaction boundary should include the reads and decisions on which its protected writes depend, subject to a concurrency-control strategy. [S117]
2. State who owns BEGIN, COMMIT and ROLLBACK. A helper that unexpectedly commits its caller's transaction can split a larger intended atomic operation.
3. In SQLite, explicit transaction forms and driver transaction settings interact. The educational code uses a deliberate connection configuration and owns one transaction rather than relying on an unstated autocommit assumption. [S25] [S86]

■ 21.1.6 WORDS — remember these

1. **Transaction boundary:** the work accepted or rejected together — the scope of a database transaction's reads, writes and completion decision. **Business precondition:** what must be true before an operation is accepted — an application rule that may require checking affected rows or queried state. **Transaction owner:** the code responsible for completion — the layer that controls commit and rollback for a defined unit of work.

21.2 Commit and rollback

■ 21.2.1 PLAIN — in simple words

1. Commit asks the database to accept the transaction's changes. Rollback abandons changes still inside an uncommitted transaction.
2. A statement succeeding does not mean the whole transaction committed. A later constraint check, communication failure or other error can change the outcome.
3. After a commit acknowledgement is lost, the client may not know whether the database committed. "I did not receive success" is not the same as "nothing happened."

■ 21.2.2 PLAIN — a picture in your head

1. A clerk sends an envelope for registration. Before registration, it can be withdrawn. After registration, tearing up the clerk's local copy does not erase the official record.
2. If the return receipt is lost, the clerk needs to check the registration identity rather than register the same operation again under a new name.

3. **Where the comparison breaks:** database completion involves protocol and persistence guarantees, not a human registrar. The important analogy is the difference between an uncommitted change and an unknown acknowledgement outcome.

■ 21.2.3 PLAIN — a worked example

```
db.execute("BEGIN IMMEDIATE")
try:
    changed = db.execute(
        "UPDATE tx_stock SET available = available - ? "
        "WHERE product_id = ? AND available >= ?",
        (3, "TX-PEN", 3),
    ).rowcount
    if changed != 1:
        raise ValueError("reservation precondition not satisfied")
    db.execute(
        "INSERT INTO tx_requests(request_id, product_id, quantity) "
        "VALUES (?, ?, ?)",
        ("TX-R1", "TX-PEN", 3),
    )
    db.execute("COMMIT")
except BaseException:
    if db.in_transaction:
        db.execute("ROLLBACK")
    raise
```

1. This is a scoped SQLite teaching pattern for an owned connection and separately created exercise tables. It does not implement idempotent replay; a duplicate request key can reject the insertion and roll back the stock change.
2. Inject an ordinary Python exception after the UPDATE but before the INSERT or COMMIT. After rollback, verify both stock and requests, not just the raised error.
3. This injection tests application-level rollback. It does not simulate device power loss or a lost network acknowledgement.

■ 21.2.4 PLAIN — what is really happening inside

1. Transaction state can differ after different failures. Some errors abort one statement; others invalidate more work or leave a transaction needing explicit recovery. Read the engine and driver contract.
2. Deferred constraints can reject COMMIT after earlier statements appeared successful. The application must handle that failure rather than return success before commit completes.
3. Once a change is committed, a correction is normally another transaction. Rollback is not a time machine for previously committed business actions.

■ 21.2.5 TECHNICAL — the engineer's version

1. SQLite documents that COMMIT can fail, including due to busy conditions, and that transaction state after errors needs deliberate handling. PostgreSQL has its own failed-transaction and save-point behaviour. Do not generalise one engine's exception handling to all engines. [S25] [S117]
2. A communication exception during commit can be an **indeterminate outcome** from the client's perspective. Use stable request identity and a reconciliation path; Chapter 26 develops stored idempotency outcomes.
3. Cleanup should not disguise the original failure. Production libraries need careful exception handling if rollback itself fails, and must not return a possibly damaged or still-active connection to a pool without a defined recovery policy.

■ 21.2.6 WORDS — remember these

1. **Commit:** accept a transaction's database changes — the completion operation whose acknowledgement has engine- and configuration-specific guarantees. **Rollback:** abandon uncommitted transactional work — restoration of the transaction's database effects under its supported semantics. **Indeterminate outcome:** the caller cannot yet tell whether it completed — uncertainty commonly caused by losing communication around a commit or external effect.

21.3 Atomicity and consistency

■ 21.3.1 PLAIN — in simple words

1. Atomicity concerns whether the selected changes take effect together. Consistency concerns the rules that valid states must satisfy.
2. The database enforces the constraints you actually declared and the mechanisms its transaction model provides. It does not infer every intended business rule from a table's name.
3. A transaction that subtracts four units for a three-unit request can be perfectly atomic and still wrong. Both changes agree only if the application checks the correct relationship.

■ 21.3.2 PLAIN — a picture in your head

1. An envelope can contain a complete but incorrect invoice. Keeping every page together prevents missing pages; it does not correct the arithmetic.
2. A reviewer still needs rules about quantities, prices and the meaning of the document.
3. **Where the comparison breaks:** some consistency rules are mechanically enforceable as constraints, while others need application logic or coordination. A human reviewer is not an automatic feature of every transaction.

■ 21.3.3 PLAIN — a worked example

1. Return to the separate capacity example from Chapter 11: capacity is ten, and two allocations are seven and six. Each allocation is positive, so a positive-value CHECK can accept both while the sum reaches thirteen.
2. Wrapping both INSERTs in one transaction makes their acceptance all-or-nothing. It does not add the missing sum-within-capacity rule.
3. A correct design can represent a guarded available quantity, coordinate through a shared record, or use another explicitly justified invariant strategy. The chosen strategy must also work when requests overlap.
4. The earlier counterexample remains evidence of absent protection in that lab; it is not retroactively repaired by this explanation.

■ 21.3.4 PLAIN — what is really happening inside

1. A valid-state rule can concern one value, one row, several rows or an external fact. Different enforcement mechanisms cover different scopes.
2. Transactional atomicity helps keep a multi-row rule intact once the application has a correct algorithm and suitable isolation or locking. It does not supply that algorithm on its own.
3. Verification should include both valid and invalid transitions. Tests that only confirm successful insertion never establish that forbidden states are rejected.

■ 21.3.5 TECHNICAL — the engineer's version

1. In ACID terminology, atomicity, consistency, isolation and durability describe distinct aspects of transactional behaviour. Consistency depends on declared constraints and correctly implemented invariants; it is not a general truth detector.
2. A transaction can preserve structural constraints while violating a business policy omitted from them. Chapter 11's cross-row and historical-price counterexamples demonstrate this distinction with concrete accepted states.
3. Prove the isolated transition preserves the invariant, then analyse overlapping transitions under the chosen concurrency model. A proof that assumes serial execution is incomplete when the deployment permits non-serialisable interactions.

■ 21.3.6 WORDS — remember these

1. **Atomicity:** the selected changes take effect together — all-or-nothing transactional acceptance of the covered database work. **Invariant:** a rule every accepted state must preserve — a condition maintained by constraints and operation protocols across permitted transitions. **Consistency:** accepted states obey the specified rules — validity relative to actual constraints and invariants, not an assurance that every stored claim is true.

21.4 Isolation and durability

■ 21.4.1 PLAIN — in simple words

1. Isolation controls what overlapping transactions can observe and how their actions interact. Durability concerns what an acknowledged change survives.
2. These properties answer different questions. A change may be protected from partial visibility yet configured with weaker persistence acknowledgements. A durable wrong answer remains wrong.
3. State the failure being discussed: a process crash, an operating-system crash, power loss, a destroyed device or a lost region are not equivalent events.

■ 21.4.2 PLAIN — a picture in your head

1. Isolation is about several clerks working without seeing an unacceptable mixture of one another's unfinished edits. Durability is about which official records remain after a failure.
2. A fireproof cabinet does not decide which edits clerks may combine, and a careful editing protocol does not make paper fireproof.
3. **Where the comparison breaks:** actual guarantees depend on software, configuration, storage and replication protocols. Neither "transactional" nor "saved" specifies all failure domains by itself.

■ 21.4.3 PLAIN — a worked example

1. A reader totals two fields while another transaction changes them together. The isolation question is whether the reader can observe an unacceptable mixture of before and after values under its snapshot rules.
2. A server acknowledges that the change committed, then its process exits. The durability question is whether a restarted engine recovers the acknowledged state under its configured persistence path.

3. A memory-only teaching database can demonstrate transactional visibility and rollback while losing all data when the process ends. It is not a durable storage deployment merely because COMMIT works.
4. Chapters 23 and 30 examine the two questions separately rather than hiding them behind one ACID label.

■ 21.4.4 PLAIN — what is really happening inside

1. Isolation can use locks, versions, conflict detection and other coordination. A snapshot is a visibility rule, not necessarily a physical copy of the entire database.
2. Durability usually involves recovery records, ordered persistence requests and assumptions about the operating system and device. Replication can extend the failure boundary only under its actual acknowledgement rules.
3. Testing a Python exception proves less than testing an abrupt process crash, which proves less than a controlled power-loss experiment. Name the experiment actually performed.

■ 21.4.5 TECHNICAL — the engineer's version

1. PostgreSQL's isolation levels and SQLite's writer/reader behaviour are product-specific contracts. Do not infer a universal anomaly table from a generic level name. [S66] [S54]
2. SQLite's atomic-commit documentation explicitly discusses filesystem and device assumptions. The educational in-memory examples do not exercise those assumptions. [S53]
3. A durability claim should identify acknowledgement point, configured persistence mode, recovery procedure and covered failure domain. Later replication acknowledgements must be analysed with the same precision.

■ 21.4.6 WORDS — remember these

1. **Isolation:** rules for overlapping work — guarantees governing visibility and interaction among concurrent transactions. **Durability:** acknowledged work survives specified failures — persistence under a named configuration, recovery model and failure boundary. **Failure domain:** what one failure can affect together — a process, host, device, availability zone or other explicitly bounded component group.

21.5 Savepoints

■ 21.5.1 PLAIN — in simple words

1. A savepoint marks a position inside a transaction. Rolling back to it can discard later transactional changes while retaining earlier uncommitted work.
2. It is not necessarily an independent committed transaction. Releasing an inner savepoint does not make its changes immune to a later rollback of the outer transaction.
3. Savepoints help recover from optional or tentative steps, but they should not be used to silently accept an incomplete business operation whose steps were all required.

■ 21.5.2 PLAIN — a picture in your head

1. While drafting an order, Mira bookmarks a version before trying an optional annotation. She can return to that bookmark without discarding the whole draft.
2. The draft still is not an accepted order merely because the annotation's bookmark was removed.

3. **Where the comparison breaks:** savepoints have precise engine-specific stack and error-recovery semantics. They do not restore external files, user-interface state or every nontransactional counter.

■ 21.5.3 PLAIN — a worked example

1. In a disposable notes table, begin a transaction and insert a required note A. Create savepoint optional_note, then insert tentative note B.

```
BEGIN;
INSERT INTO tx_notes(note_id, body) VALUES ('A', 'required');
SAVEPOINT optional_note;
INSERT INTO tx_notes(note_id, body) VALUES ('B', 'tentative');
ROLLBACK TO optional_note;
RELEASE optional_note;
COMMIT;
```

2. The committed result contains A but not B. The exercise explicitly defines B as optional; this is not permission to drop a failed stock adjustment from a required reservation.
3. Repeat with a full ROLLBACK instead of COMMIT at the end. Then neither A nor B remains from the transaction, even after RELEASE.

■ 21.5.4 PLAIN — what is really happening inside

1. Savepoints form nested recovery boundaries within the transaction's supported semantics. Rolling back to one abandons subsequent work and affects later savepoints according to the engine's rules.
2. A released inner boundary no longer provides that local recovery point, but its changes remain part of the outer transaction until outer completion.
3. SQLite permits an outermost SAVEPOINT to start a transaction, so releasing that outermost boundary can commit. Distinguish this case from releasing an inner savepoint inside an existing transaction. [S127]

■ 21.5.5 TECHNICAL — the engineer's version

1. PostgreSQL SAVEPOINT and ROLLBACK TO support partial recovery inside a transaction, including recovery from eligible failed suboperations. The outer transaction remains the final acceptance boundary. [S121]
2. SQLite RELEASE has stack semantics; an inner release is not a durable commit, while release of the outermost transaction savepoint has a different effect. Code must know which boundary it owns. [S127]
3. A savepoint strategy should document which failures are optional, which invalidate the whole business action and which leave the connection unusable. Catching every exception and continuing is not a valid general recovery policy.

■ 21.5.6 WORDS — remember these

1. **Savepoint:** a local return point inside transaction work — a named boundary to which later transactional changes can be rolled back. **Partial rollback:** discard only a selected later portion — rollback to a savepoint rather than abandonment of the entire outer transaction. **Outer transaction:** the enclosing acceptance boundary — the transaction whose final commit or rollback governs changes retained from inner savepoint work.

21.6 External effects outside the transaction

■ 21.6.1 PLAIN — in simple words

1. Sending an email, charging an external payment service, printing a document or handing over goods is not normally undone by rolling back a local database transaction.
2. Performing the external effect before commit can leave an effect with no accepted database record. Performing it after commit can leave accepted work with an effect that was never completed.
3. The solution is an explicit cross-boundary protocol, not a claim that one local transaction magically covers everything. Chapter 26 introduces durable delivery intent; Chapter 38 covers compensation.

■ 21.6.2 PLAIN — a picture in your head

1. Dev sends a customer a reservation message, then discovers that the database transaction failed. The message is already in the customer's inbox.
2. Erasing the draft on Dev's desk does not recall that message.
3. **Where the comparison breaks:** some external providers support idempotency keys or transactional integration, but those are additional contracts. Their existence must be verified for the exact operation, not assumed from the word API.

■ 21.6.3 PLAIN — a worked example

1. Schedule A: update stock; send confirmation; request insertion fails; roll back. The customer saw a confirmation for a reservation that did not commit.
2. Schedule B: update stock and request; commit; process crashes before sending. The reservation exists but no confirmation was sent.
3. An outbox stores the intention to send alongside the business changes in one database transaction. A later worker retries delivery from that committed intention.
4. This closes the local "committed work with no recorded intent" gap. It does not by itself prevent duplicate external delivery when a worker sends successfully and crashes before recording acknowledgement. [S125]

■ 21.6.4 PLAIN — what is really happening inside

1. Each independent system has its own acceptance point. A local database cannot roll back a remote side effect merely because the application uses a try/except block.
2. Stable operation identity helps reconcile uncertain outcomes. A correction or compensation is a new action with its own evidence, not a claim that the first action never happened.
3. Even within PostgreSQL, sequence allocation has special non-rollback behaviour. Gaps in generated values therefore do not prove that a business record was deleted or lost. [S122]

■ 21.6.5 TECHNICAL — the engineer's version

1. Distinguish local transactional atomicity from distributed atomic commit and from a saga of compensating actions. They provide different failure semantics.
2. An outbox atomically binds business state to durable delivery intent when both writes share the same effective database transaction. Delivery, ordering and consumer deduplication remain separate obligations. [S125]

3. The educational tests inject failures at named points and inspect database state. They do not send live messages, perform payments or prove end-to-end exactly-once effects.

■ 21.6.6 WORDS — remember these

1. **External effect:** something changes beyond the local transaction — an action in another system or the physical world not automatically reversed by database rollback. **Outbox:** committed work carries a delivery instruction — a transactional record of an event or message to be delivered by a separate process. **Compensation:** a new action addresses an earlier effect — an explicit corrective operation, not erasure of the original history.

21.97 Practice and worked answers

1. **Question:** Stock decreases but request insertion fails before commit. What should the owned transaction do? **Answer:** Roll back both covered changes and verify that neither partial outcome remains.
2. **Question:** An UPDATE affects zero rows without a SQL error. Is the reservation accepted? **Answer:** Not under the specified one-row precondition. The application must inspect the affected-row result.
3. **Question:** Can a transaction atomically commit an allocation total above capacity? **Answer:** Yes, if the capacity invariant is not enforced by the schema or operation protocol. Atomicity does not invent the rule.
4. **Question:** Is a lost commit acknowledgement proof of rollback? **Answer:** No. The outcome may be indeterminate to the caller; reconcile using stable operation identity.
5. **Question:** Does releasing an inner savepoint protect its changes from outer rollback? **Answer:** No. Those changes remain inside the outer transaction.
6. **Question:** Does an in-memory COMMIT establish power-loss durability? **Answer:** No. The storage and failure boundary were not exercised.
7. **Question:** Will database rollback recall an email already sent? **Answer:** Not ordinarily. The external effect requires its own protocol or corrective action.
8. **Question:** Why do sequence gaps not prove missing orders? **Answer:** Sequence allocation can survive transaction rollback and has other allocation behaviours. Business-event completeness must be checked against business records, not inferred from gapless numbering.

21.98 Common wrong ideas

1. **Wrong:** successful statements mean the transaction committed. **Right:** final completion and deferred checks still matter.
2. **Wrong:** zero affected rows is always a database error. **Right:** it can be a successful statement whose business precondition failed.
3. **Wrong:** ACID means every business rule is automatically known. **Right:** omitted rules remain omitted.
4. **Wrong:** rollback reverses earlier committed work. **Right:** a later correction is another transaction.
5. **Wrong:** releasing any savepoint commits independently. **Right:** inner and outer boundaries differ.
6. **Wrong:** saved in memory means durable on a device. **Right:** persistence guarantees need a named failure model.

7. **Wrong:** an external API call belongs to the local transaction because it is inside the same function. **Right:** code scope is not a shared commit protocol.
8. **Wrong:** an outbox alone guarantees one external effect. **Right:** delivery ambiguity and consumer behaviour still need handling.

21.99 Chapter summary in 20 lines

1. A transaction groups selected database work under one completion boundary.
2. The application chooses the business action and its invariants.
3. Required reads and decisions need a concurrency strategy as well as grouped writes.
4. One layer should clearly own commit and rollback.
5. A zero-row update can signal a failed business precondition.
6. Commit can fail after earlier statements succeeded.
7. A lost acknowledgement can leave the caller uncertain about completion.
8. Rollback abandons uncommitted work, not all past actions.
9. Atomicity does not correct wrong arithmetic or omitted rules.
10. Consistency is relative to actual enforced constraints and protocols.
11. Isolation governs overlapping observations and changes.
12. Durability concerns survival under specified failures.
13. In-memory tests do not establish persistent-device durability.
14. Savepoints provide local recovery inside an enclosing transaction.
15. Releasing an inner savepoint is not independent final acceptance.
16. Optional-step recovery must not silently drop required business work.
17. External effects usually remain outside local rollback.
18. An outbox binds committed business changes to recorded delivery intent.
19. Sequence gaps are not proof of missing business events.
20. Every guarantee needs a clear boundary and evidence matching that boundary.

Two People Change the Same Record

22.0 What this chapter gives you

1. A program can be correct when run alone and wrong when another program changes the same facts between its steps. This chapter teaches you to write down that overlap instead of explaining it away as a mysterious race.
2. You will trace lost updates, use conditional writes, distinguish a uniqueness rule from an absence check, and design a bounded stock-reservation transaction. The examples concern database records, not an assurance that the physical shelf matches them.
3. Every inventory schedule in this chapter is new synthetic teaching data. None establishes the cause of the earlier unexplained stock discrepancy at Mira's Corner.

22.1 Interleaving actions

■ 22.1.1 PLAIN — in simple words

1. Two people do not need to press a button at exactly the same instant to interfere with one another. It is enough for one person's work to happen between another person's reading and writing.
2. A request that appears to be one action on a screen may contain several separate steps: read a value, decide what to do, calculate a result, write it, and confirm completion.
3. The important question is which of those steps may overlap and what the database promises about the overlap. Counting buttons or adding a loading spinner does not answer it.

■ 22.1.2 PLAIN — a picture in your head

1. Mira reads the number on a stock card, walks away to calculate, and comes back to write her answer. While she is away, Dev reads the same card and begins his own calculation.
2. Each person remembers a different moment, even though both later write on the same card. The card cannot reconstruct the missing conversation by itself.
3. **Where the comparison breaks:** a database may block, reject or reorder operations according to its concurrency controls. A handwritten schedule describes a proposed sequence; it is not evidence that every database permits that sequence.

■ 22.1.3 PLAIN — a worked example

1. Name the requests A and B and split each into read, calculate and write. One possible schedule is A-read, B-read, A-calculate, A-write, B-calculate, B-write.
2. Another is A-read, A-calculate, A-write, B-read, B-calculate, B-write. The second completes A before B reads; it may therefore give B different input.

3. Before claiming an anomaly, state the initial records, each statement, the transaction boundaries, the isolation level, the connection arrangement and the observed results. A drawing with none of these details is a hypothesis to investigate.
4. In a pure model we can deliberately enumerate schedules. In an engine experiment we must record which actions actually succeeded, waited or failed.

■ 22.1.4 PLAIN — what is really happening inside

1. The server and operating system schedule work independently of the order in which people remember clicking. Network delays can also change arrival and response order.
2. A connection may hold one transaction while another connection executes its own. A pool is not a guarantee that related statements use the same connection or transaction unless the application arranges that explicitly.
3. Correctness reasoning follows dependencies: this write used that earlier observation; this decision required that predicate to remain protected. Wall-clock timestamps alone often cannot establish all such relationships.

■ 22.1.5 TECHNICAL — the engineer's version

1. An interleaving is an ordering of elementary operations consistent with the order required inside each participating process or transaction. A schedule model should state which operations are atomic in that model.
2. Engine isolation constrains the legal schedules and visible versions. PostgreSQL 17 and SQLite do not offer identical execution models; a PostgreSQL snapshot example should not be advertised as a reproduced SQLite anomaly. [S66] [S54]
3. Use deterministic barriers or explicit two-connection coordination for a controlled experiment where feasible. A sleep can encourage overlap but does not prove the intended ordering. Retain a step log and transaction outcome for every participant.

■ 22.1.6 WORDS — remember these

1. **Interleaving:** the steps of different requests mixed together — an operation ordering that preserves each participant's required local order. **Race condition:** an answer that depends on an unsafe overlap — behaviour whose correctness changes with the relative execution order of competing operations. **Schedule evidence:** the recorded sequence that actually ran — observations of operations, waits, errors and outcomes under a stated concurrency configuration.

22.2 Lost updates

■ 22.2.1 PLAIN — in simple words

1. A lost update happens when a later write accidentally replaces the effect of another change. Both people may receive a success message even though the final record represents only part of their work.
2. A common cause is calculating a replacement value from an old read. The replacement says “set stock to six,” not “subtract the four units this request reserved.”
3. The database cannot always infer the intention behind a supplied number. Six might be a deliberate correction, an obsolete calculation or an entirely wrong input. The application must express the intended operation and its conditions.

■ 22.2.2 PLAIN — a picture in your head

1. Two editors download the same paragraph. One corrects a spelling error; the other adds a sentence. Uploading the second whole paragraph can restore the spelling error by replacing the first editor's version.
2. The second upload is not necessarily larger or later in meaning. It is merely the last replacement accepted under that editing protocol.
3. **Where the comparison breaks:** text-merging tools sometimes combine edits automatically. A stock counter needs a precisely defined numeric rule; an automatic text-style merge is not an inventory policy.

■ 22.2.3 PLAIN — a worked example

1. A new model begins with RACE-STOCK equal to 10. Request A wants 3 units and request B wants 4. Both read 10 before either writes.
2. A calculates $10 - 3 = 7$. B independently calculates $10 - 4 = 6$. A writes 7, then B writes 6.
3. The final value is 6, but accepting both reservations should have left $10 - 3 - 4 = 3$. A's three-unit subtraction was overwritten. The error is not that subtraction was calculated incorrectly; it was calculated from an observation that was no longer safe to replace.
4. This deliberately unsafe read-then-replace model does not establish that a particular transaction configuration permits both writes. Some engines or isolation levels would reject or serialize relevant operations instead.

■ 22.2.4 PLAIN — what is really happening inside

1. An UPDATE can faithfully store the literal value supplied to it. Atomic execution of that one UPDATE does not retroactively protect a separate earlier SELECT.
2. Beginning a transaction around the SELECT and UPDATE is important but does not, by itself, specify sufficient isolation for every read-dependent rule. The chosen engine and isolation level determine the remaining risks.
3. Different intentions require different controls. An increment can often be expressed as a relative update. Replacing a person's edited document may require a version check and a conflict response rather than silently adding or merging values.

■ 22.2.5 TECHNICAL — the engineer's version

1. Distinguish `SET available = :old_value - :quantity` from `SET available = available - :quantity`. The first binds a client-derived replacement; the second asks the engine to derive the new value from the row it updates.
2. In PostgreSQL 17 Read Committed, a competing updater may wait and then re-evaluate its WHERE condition against an updated row. This supports useful single-row guarded-update patterns, not an unrestricted proof for arbitrarily complex cross-row rules. [S66]
3. A version-based compare-and-set is another strategy: require the expected revision and advance it with the write. Chapter 25 develops that approach and its limits, including deletion and recreation of an identity.

■ 22.2.6 WORDS — remember these

1. **Lost update:** one accepted change is overwritten by another — a write anomaly in which a later write fails to preserve an effect that should remain represented. **Relative update:** describe the

change rather than an old replacement — an expression such as `value = value - amount` evaluated by the database. **Stale read:** an observation that no longer describes the relevant current state — a previously read version that may be unsafe as an unconditional write basis.

22.3 Conditional writes

■ 22.3.1 PLAIN — in simple words

1. A conditional write asks the database to make a change only while a stated condition is satisfied. For stock, the condition may be “this product has at least the requested quantity.”
2. The condition and the change must belong to the protected operation. Checking stock in one request and changing it later without protection leaves the gap open.
3. A statement affecting no rows needs interpretation. It may mean insufficient stock, a missing product, a stale version or a forbidden scope, depending on the statement and application contract. It is not automatically a database malfunction.

■ 22.3.2 PLAIN — a picture in your head

1. A ticket dispenser checks whether tickets remain and releases one through the same controlled mechanism. It does not hand you a note saying that a ticket existed several seconds ago.
2. Combining the check with the action prevents another buyer from treating the same old observation as a separate promise.
3. **Where the comparison breaks:** a database condition concerns stored records. The record may still disagree with damaged, missing or uncounted physical goods. Concurrency control does not replace stocktaking.

■ 22.3.3 PLAIN — a worked example

1. In a separate RES-PEN exercise there are 5 available units. A asks for 3 and B asks for 3. Both cannot be accepted while available stock stays non-negative.
2. The guarded operation subtracts 3 only when at least 3 remain. One successful update leaves 2. A later evaluated condition sees that 2 is not enough and affects zero rows.

```
UPDATE reservation_stock
SET available = available - ?
WHERE product_id = ? AND available >= ?;
```

3. Bind (3, 'RES-PEN', 3) after validating that quantity is a positive bounded integer. Accept the stock step only if exactly one row was affected under the helper's stated schema and driver behaviour.
4. This statement alone does not create the request record, protect an external payment or distinguish a replay. Those responsibilities require the surrounding transaction and request protocol.

■ 22.3.4 PLAIN — what is really happening inside

1. The engine coordinates the qualifying row and its update. Other relevant operations may wait or receive a conflict instead of both acting on the same old value.
2. Keep the product's complete identity in the predicate. Once branches or tenants exist, a product code alone may match the wrong scope or multiple rows.
3. Separate input validation from database enforcement. Reject zero, negative, boolean-like or excessively large quantities according to the API's explicit type contract; retain database constraints as another boundary.

■ 22.3.5 TECHNICAL — the engineer's version

1. A guarded update protects only the predicate and resources covered by the engine's concurrency semantics. It does not automatically enforce a total across other rows or across databases. [S66]
2. With SQLite, serialize the owned write transaction deliberately, for example using `BEGIN IMMEDIATE` in these bounded exercises. Handle contention and inspect transaction state rather than assuming simultaneous independent writers. [S25] [S54]
3. Record the row-count contract for the driver and statement. For the supplied SQLite UPDATE examples, the code checks `cursor.rowcount`; more complex statements, triggers and different drivers require their own documented interpretation. [S86]

■ 22.3.6 WORDS — remember these

1. **Conditional write:** change a record only when its requirement still holds — a write guarded by a predicate evaluated under the database's concurrency rules. **Affected-row check:** verify how many target records changed — inspection of the statement result against an expected cardinality. **Complete scope:** all identity dimensions needed to select the intended record — the tenant, branch and local key components required by the data model.

22.4 Uniqueness under competition

■ 22.4.1 PLAIN — in simple words

1. Asking whether a key exists and receiving “no” is an observation, not a reservation of that key. Another request may receive the same answer before either inserts.
2. A uniqueness constraint gives the database an enforceable rule about which keys may coexist. The application must still decide how to handle the losing request.
3. The meaning of the key matters. One key per event, one key per customer and one key per retryable request describe different things. A constraint cannot repair a confused definition of identity.

■ 22.4.2 PLAIN — a picture in your head

1. Two visitors see an empty seat and each announce that it is theirs. Looking at the seat did not allocate it.
2. A controlled reservation desk can accept one seat assignment and reject the conflicting assignment. The desk must also know whether two messages came from one visitor retrying or from two different visitors.
3. **Where the comparison breaks:** some database uniqueness rules treat missing values differently from ordinary values. A nullable field is not automatically the same as a required reservation identifier.

■ 22.4.3 PLAIN — a worked example

1. A and B both execute a lookup for `(branch='BR-A', receipt_no=17)` and see no row. Both then try to insert that pair.
2. With the required composite unique key, both cannot commit conflicting rows with that key. One may succeed while the other waits, fails or takes an explicitly programmed conflict path.
3. `(BR-B, 17)` is different and may be valid. A global uniqueness constraint on receipt number alone would reject legitimate records from another branch.

4. Receiving a uniqueness error does not prove that the competing record contains the same request payload. Before returning an existing outcome as a replay, check the intended request identity and its recorded meaning.

■ 22.4.4 PLAIN — what is really happening inside

1. The engine's constraint mechanism participates in concurrency control; the application does not implement uniqueness merely by querying first.
2. A pre-check can improve a user message, but it cannot replace the final enforcement point. Handle a conflict even when the pre-check passed.
3. Blindly retrying a permanently conflicting insert wastes resources. A new attempt with the same impossible key and unchanged policy is likely to encounter the same problem again.

■ 22.4.5 TECHNICAL — the engineer's version

1. Define the unique constraint on the complete non-null identity required by the model. Consult the engine's treatment of NULL, deferred constraints and conflict clauses rather than assuming universal behaviour. [S74] [S61] [S73]
2. PostgreSQL's serialization-retry guidance distinguishes transient concurrency situations from persistent uniqueness failures. A uniqueness error is not a universal instruction to retry automatically. [S120]
3. An upsert is a chosen write policy, not proof of semantic equivalence. Replacing an existing payload on conflict may destroy the very evidence needed to identify incompatible uses of a request key.

■ 22.4.6 WORDS — remember these

1. **Check-then-insert race:** two requests act on the same observed absence — an unsafe gap between an existence query and an unprotected insertion decision. **Composite uniqueness:** a combination must be distinct — a constraint enforcing uniqueness over a tuple of columns. **Conflict policy:** what to do when a rule prevents a write — an explicit decision to reject, reconcile, return an equivalent result or apply a defined update.

22.5 Read-modify-write traps

■ 22.5.1 PLAIN — in simple words

1. Some rules depend on more than the row being changed. A branch may allow at most ten active reservations across many request rows. Each individual row can be valid while their total is too large.
2. Protecting one writer's own row does not necessarily protect the shared total. Two writers can change different rows after reading the same acceptable total.
3. The remedy must match the invariant. It may use a shared guarded capacity record, an appropriate locking protocol or serializable transactions with correct retry handling. Choosing a familiar keyword without tracing the rule is not a design.

■ 22.5.2 PLAIN — a picture in your head

1. Two staff members each have a locked drawer for booking forms, but both sell space in the same ten-seat room. Locking the drawers does not coordinate the room's remaining capacity.
2. The scarce thing is shared, even though the pieces of paper are separate.

3. **Where the comparison breaks:** database locks operate on specific engine resources, not abstract business concepts. A design must identify the actual row, predicate or coordination protocol representing the shared capacity.

■ 22.5.3 PLAIN — a worked example

1. Revisit CAP-DEMO as a separate rule demonstration: capacity is 10 and allocations of 7 and 6 are individually positive. Their sum is 13, exceeding capacity by 3.
2. A CHECK requiring each allocation to be between 1 and 10 accepts both. It checks each row, not the combined allocation rule.
3. A possible redesign uses a capacity row with available=10. Every allocating path must atomically reduce that same available value under a sufficient-stock condition and record its allocation in the same transaction.
4. The redesign is incomplete if an administrator, import job or cancellation path can bypass the coordination protocol. Enumerate all mutation paths before claiming the invariant is enforced.

■ 22.5.4 PLAIN — what is really happening inside

1. The read set and write set can differ. A request may read a sum of many rows but write only its new allocation. Another request can have a disjoint write set while depending on the same shared read condition.
2. A lock on a row that does not exist cannot casually be treated as a universal lock on future matching rows. Engines have different predicate and range-locking behaviours.
3. Computing a decision outside the protected transaction and merely repeating the final INSERT inside it can preserve the original unsafe assumption. Recompute or validate the decision at the correct boundary.

■ 22.5.5 TECHNICAL — the engineer's version

1. Express the invariant mathematically, such as `sum(active quantities for capacity_key) <= capacity`. Then specify a protocol that every competing writer follows.
2. PostgreSQL Serializable can detect dependency patterns that threaten serializability, but applications must retry the whole transaction when required. Row-locking alone and snapshot isolation alone do not imply the same guarantee. [S66] [S120]
3. Avoid holding a transaction open across an unbounded external call to preserve a read. Separate durable intent and external execution where possible; Chapter 26 explains the outbox boundary.

■ 22.5.6 WORDS — remember these

1. **Cross-row invariant:** a rule about several records together — a correctness condition involving a set of rows rather than one row's fields. **Shared guard record:** one coordination point for a shared limit — a record whose guarded mutation serializes changes to the represented resource. **Mutation path:** a way records can change — an application route, job, import, administrator action or other writer that must respect the invariant.

22.6 A stock-reservation case

■ 22.6.1 PLAIN — in simple words

1. A reservation should have a clear outcome: accepted with a specific quantity, rejected for a stated business reason, or unresolved because the caller has not learned the result.

2. “The connection disappeared” belongs to the third category. It does not prove either acceptance or rejection. The database may have committed before the response was lost.
3. Build the operation in layers: validate the request, coordinate the stock change, record the outcome, commit, and expose a way to resolve retries. Do not let a success message outrun the evidence supporting it.

■ 22.6.2 PLAIN — a picture in your head

1. The reservation desk records a numbered decision before sending a confirmation. A customer who loses the confirmation can ask about that same decision rather than buying another ticket by accident.
2. A decision number is useful only if the desk keeps its meaning stable and can tell a retry from a new purchase.
3. **Where the comparison breaks:** database and notification systems may fail separately. Recording a decision does not guarantee that an email was delivered or that the physical item was handed over.

■ 22.6.3 PLAIN — a worked example

1. Start a new exercise with RES-PEN=5. R-A requests 3; R-B requests 3. An accepted R-A leaves 2 and a matching accepted reservation record. R-B cannot acquire another 3 from that state.
2. Inject a failure after reducing stock but before recording R-A. Rolling back the owned transaction should restore stock to 5 and leave no accepted R-A record.
3. Next allow R-A to commit but pretend its response was lost. Simply submitting another subtraction is unsafe. Chapter 26 records a scoped request identity and stable outcome so the same intent can be resolved without allocating twice.
4. Reconcile the accepted reservation quantities with the stock change in the exercise. This proves agreement within the model, not agreement with an unobserved real shelf.

■ 22.6.4 PLAIN — what is really happening inside

1. The database transaction supplies a local atomic boundary. The request protocol adds a durable connection between the caller’s intent and the recorded outcome.
2. Failure handling must know whether a transaction is still active, rolled back or committed. Where the client cannot determine that state remotely, it needs reconciliation rather than a guessed status.
3. Cancellation is another business operation, not deletion of inconvenient history. Define whether, when and how it releases stock and whether a repeated cancellation has any additional effect.

■ 22.6.5 TECHNICAL — the engineer’s version

1. Minimum tests include success, insufficient stock, missing scoped product, invalid quantity, duplicate request identity, failure before commit and reconciliation after a lost response. Add actual coordinated competing-connection tests before claiming an engine-specific concurrency result.
2. The companion code separates deterministic schedule models from real SQLite transactions. A sequential model demonstrates why a protocol is needed; a passing in-memory transaction test is not multi-host or power-loss evidence.
3. Request deduplication requires a stable identifier and an explicit equivalence contract. Atomic recording of that identity, its outcome and the business write is developed in Chapter 26. [S126]

■ 22.6.6 WORDS — remember these

1. **Reservation:** a recorded allocation under defined rules — a business state transition that reduces available capacity while linking the allocation to a request. **Unknown outcome:** the caller lacks reliable completion evidence — a state in which a request may have committed even though its response was not received. **Reconciliation:** compare related records to resolve disagreement — a procedure that checks quantities, identities and outcomes against stated invariants and evidence.

22.97 Practice and worked answers

1. **Trace the lost update.** Start at 10. A reads 10 and reserves 3; B reads 10 and reserves 4; A writes 7 and B writes 6. **Answer:** the model ends at 6, whereas both accepted reservations require 3. Three units of A's effect were overwritten.
2. **Change the schedule.** Let A finish before B reads. **Answer:** B reads 7 and calculates 3. This safe-looking schedule does not prove that the unprotected program is safe under every permitted overlap.
3. **Interpret zero affected rows.** The guarded update selects by product and sufficient stock. **Answer:** zero alone cannot distinguish a missing product from insufficient stock. The API must define a safe response or perform an appropriately scoped diagnostic read.
4. **Find the incomplete key.** A multi-branch UPDATE filters only by product code. **Answer:** include the branch and, where applicable, tenant identity. A unique local product code is not a globally complete key.
5. **Evaluate a pre-check.** Two requests each see receipt 17 absent in BR-A. **Answer:** the lookup does not reserve the key. The composite uniqueness constraint must decide conflicting insertions, and the losing path must handle its result.
6. **Repair the capacity argument.** Each of two allocation rows passes `quantity <= 10`. **Answer:** that does not enforce their sum. Coordinate a shared resource or use another proven protocol covering all participating writers.
7. **Classify a disconnected caller.** The caller did not receive COMMIT's result. **Answer:** the outcome may be unknown. Resolve the original request identity rather than assuming rollback and issuing an unrelated new intent.
8. **State the evidence boundary.** A deterministic Python schedule produces a lost update. **Answer:** it proves the model's arithmetic and ordering, not that PostgreSQL or SQLite under an unspecified configuration admitted that schedule.

22.98 Common wrong ideas

1. **Wrong:** a race requires exactly simultaneous clicks. **Right:** an unsafe overlap between dependent steps is enough.
2. **Wrong:** an atomic UPDATE protects an earlier unprotected SELECT. **Right:** the earlier observation needs its own appropriate concurrency protocol.
3. **Wrong:** BEGIN and COMMIT imply serializable behaviour. **Right:** the engine and isolation level determine the actual guarantee.
4. **Wrong:** a pre-check implements uniqueness. **Right:** a database constraint must enforce competing claims at the write boundary.
5. **Wrong:** a locked individual row protects every total involving it. **Right:** the shared invariant may involve other rows or future insertions.

6. **Wrong:** a timeout proves that nothing was saved. **Right:** the caller may have an unknown outcome after a committed operation.
7. **Wrong:** retries should always create a new request key. **Right:** retrying one intent should preserve its defined identity; a new intent is a separate decision.
8. **Wrong:** correct reservation records prove correct physical inventory. **Right:** the shelf remains an external source of evidence.

22.99 Chapter summary in 20 lines

1. Competing requests interleave at their internal steps, not only at button presses.
2. State the initial records, statements and transaction boundaries before analysing an overlap.
3. A schedule model is different from a recorded engine experiment.
4. A replacement calculated from a stale read can overwrite another accepted change.
5. Starting at 10, stale replacements of 7 then 6 lose the intended combined result of 3.
6. A relative update expresses a change instead of an obsolete replacement.
7. A guarded write combines a condition with the protected update.
8. Inspect affected rows against the operation's expected cardinality.
9. Validate positive bounded quantities before attempting the database mutation.
10. Select records using every required tenant, branch and local identity component.
11. Seeing that a key is absent does not reserve it.
12. Enforce uniqueness in the database and handle the losing request deliberately.
13. A conflict does not prove that two payloads represent the same intent.
14. Cross-row limits need protocols that protect the shared invariant.
15. Every mutation path must follow that protocol, including jobs and administrative writers.
16. Record the reservation and stock change within the intended atomic boundary.
17. A failure before commit should not leave one half of the local operation accepted.
18. A lost response after commit can leave the caller uncertain rather than rejected.
19. Resolve retries through stable request identity and recorded outcomes.
20. Database correctness and physical-stock accuracy require different evidence.

Isolation Levels and What You Can Observe

23.0 What this chapter gives you

1. A transaction needs an answer to a deceptively simple question: which changes made by other transactions may it see? Isolation levels describe parts of that answer and the anomalies that remain possible.
2. You will distinguish statement snapshots from transaction snapshots, trace non-repeatable reads and write skew, and understand what serializable execution promises without mistaking it for a single physically sequential worker.
3. Product-specific statements in this chapter refer to PostgreSQL 17 unless explicitly marked otherwise. SQLite examples are kept separate. The written schedules are teaching models, not fabricated server traces.

23.1 Visibility rules

■ 23.1.1 PLAIN — in simple words

1. While Mira is changing an order, Dev may be reading it. The database needs rules about whether Dev sees the old version, the new version or a wait before receiving an answer.
2. “A transaction hides changes” is too vague. We must ask whose changes, until which boundary, and whether two statements in the same transaction can see different committed worlds.
3. An isolation level does not decide whether a recorded sale actually occurred. It governs interactions among database operations. Good visibility rules cannot transform invented input into a verified real-world event.

■ 23.1.2 PLAIN — a picture in your head

1. Imagine viewing a noticeboard through a window while someone updates its cards. One rule gives you a fresh photograph whenever you ask a question. Another gives you one photograph to use throughout a consultation.
2. Both avoid watching half-erased handwriting, but they do not give the same answers after the board changes.
3. **Where the comparison breaks:** transactions can normally see their own earlier writes, so their view is not simply a frozen photograph of everyone else’s board. Locking and conflict checks may also affect what happens when they write.

■ 23.1.3 PLAIN — a worked example

1. Suppose product ISO-PEN has price 100 in a separate example. Transaction A changes it to 120 but has not committed. Transaction B asks for the price.

2. In the PostgreSQL levels discussed here, B does not read A's uncommitted replacement as an ordinary visible committed value. What B later sees after A commits depends on B's snapshot and statement timing.
3. If A rolls back, its transactional price replacement must not become an accepted committed price. A read that relied on an uncommitted replacement would have relied on a change that never became committed history.
4. Keep this separate from sequences: PostgreSQL sequence advancement has special non-rollback behaviour. A generated number is not itself proof of a committed business record. [S122]

■ 23.1.4 PLAIN — what is really happening inside

1. Engines combine snapshots, locks, row versions and conflict detection in different ways. An isolation-level name is not a substitute for the relevant product documentation.
2. The SQL standard's traditional anomaly categories are useful vocabulary, but implementations can offer stronger guarantees than the minimum associated with a name.
3. A practical contract names the engine, version, level, transaction boundaries and participating writers. It also states what the application does when the engine refuses a conflicting transaction.

■ 23.1.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 implements Read Uncommitted as Read Committed. Its Repeatable Read prevents phantom reads as well as non-repeatable reads, while serialization anomalies remain possible. Serializable adds protection against those anomalies through its concurrency-control implementation. [S66]
2. A dirty read observes data written by an uncommitted transaction. A non-repeatable read obtains a different value when rereading a row after another transaction commits a change. A phantom concerns changed membership of a repeated predicate query. These categories describe observations, not every possible application failure.
3. SQLite normally serializes writers; WAL mode permits readers to retain snapshots while a writer works. Its documented exceptions and upgrade failures must be treated as SQLite-specific behaviour, not inferred from PostgreSQL names. [S54]

■ 23.1.6 WORDS — remember these

1. **Isolation level:** the chosen rules for overlapping transactions — a named contract restricting visibility and permitted concurrency anomalies. **Dirty read:** seeing a change before it commits — observation of another transaction's uncommitted data. **Phantom:** a repeated condition matches a changed set — a predicate-query membership change caused by another transaction's committed work.

23.2 Statement and transaction snapshots

■ 23.2.1 PLAIN — in simple words

1. A statement snapshot gives one query a consistent view of relevant committed data at a particular point. A later query may receive a newer view.
2. A transaction snapshot lets a transaction continue using an earlier view of other transactions' committed work. This helps related reads agree, but it does not mean the transaction can safely make every possible decision from that view.

3. Snapshot age also matters operationally. A long-running reader can keep old information relevant to the engine long after ordinary short requests have moved on.

■ 23.2.2 PLAIN — a picture in your head

1. For each question, Mira either asks for today's newly printed report or keeps consulting the report she collected when the meeting began.
2. A fresh report can include later corrections; a retained report supports consistent comparisons within the meeting. Neither is automatically preferable for every purpose.
3. **Where the comparison breaks:** a database snapshot is not usually a complete copied report. Visibility is determined using version and transaction metadata, and the transaction can see changes it made itself.

■ 23.2.3 PLAIN — a worked example

1. Start with ISO-PEN=100. Reader R runs its first SELECT and sees 100. Writer W then commits a change to 120. R runs the same SELECT again.
2. Under PostgreSQL 17 Read Committed, ordinary separate SELECT statements can return 100 and then 120 because the second statement obtains a later snapshot.
3. Under PostgreSQL 17 Repeatable Read, R's snapshot begins with its first non-transaction-control statement. If that snapshot precedes W's commit, the repeated ordinary read continues to see 100, excluding R's own changes for simplicity.
4. Neither result is inherently a lie. The question is which visibility contract the application needed and actually selected. [S66]

■ 23.2.4 PLAIN — what is really happening inside

1. The engine checks whether a row version is visible to the snapshot. A later committed replacement can exist while an older transaction still reads an earlier visible version.
2. Read Committed does not mean every operation inside one statement behaves like a simplistic immutable photograph. PostgreSQL's modifying statements have rules for waiting on and rechecking concurrently updated rows.
3. A report requiring several related totals should explicitly choose how their common observation boundary is established. Repeated independent queries cannot be assumed to describe one shared instant.

■ 23.2.5 TECHNICAL — the engineer's version

1. In PostgreSQL 17, Read Committed ordinary SELECT uses a snapshot as of the query's start and sees the transaction's own preceding writes. Repeatable Read uses the snapshot established at the transaction's first non-transaction-control statement. [S66]
2. A stable snapshot is a visibility mechanism, not a full serializability proof. Transactions can read the same condition and write disjoint rows in ways that violate a combined invariant.
3. For reproducible demonstrations, record when the snapshot-establishing statement ran. Writing only BEGIN in a timeline can assign the snapshot to the wrong moment in this implementation.

■ 23.2.6 WORDS — remember these

1. **Statement snapshot:** a view chosen for one statement — the visibility boundary used by a query under statement-level snapshot rules. **Transaction snapshot:** a retained view for related state-

ments — a visibility boundary reused through a transaction under the chosen implementation. **Snapshot establishment:** the event that chooses the view — the implementation-defined point at which a transaction's snapshot is acquired.

23.3 Non-repeatable reads

■ 23.3.1 PLAIN — in simple words

1. A non-repeatable read means that reading the same row twice can produce different values because another transaction committed between the reads.
2. This is sometimes useful: a screen refresh should often reveal a newly changed value. It is dangerous when a calculation assumes that an earlier value remained unchanged across several steps.
3. The solution is not always “use the strongest level everywhere.” First decide whether the operation needs a common snapshot, a guarded write, a lock or a different workflow.

■ 23.3.2 PLAIN — a picture in your head

1. Mira checks the price card, walks to the till, and checks it again after Dev updates the card. The second answer differs because the world she is consulting changed.
2. A quotation process must decide whether the first observed price was merely information or an agreement that should be preserved.
3. **Where the comparison breaks:** a database snapshot can preserve an earlier stored view, but it cannot create a customer agreement that the application never recorded. Historical meaning and transaction visibility are related but different responsibilities.

■ 23.3.3 PLAIN — a worked example

1. Consider an analytical exercise with A's balance=60 and B's balance=40. A transfer moves 10 from A to B in one transaction, leaving 50 and 50. The total is 100 before and after.
2. A reader first queries A before the transfer commits and receives 60. It then separately queries B after the transfer commits and receives 50. Adding the two observations gives 110.
3. Neither query necessarily returned an individually incorrect committed value. The reader combined different observation boundaries and called the result one total.
4. A single appropriately defined query or a suitable retained snapshot can align the reads. This example is fictional arithmetic, not a banking implementation or financial advice.

■ 23.3.4 PLAIN — what is really happening inside

1. The transfer's atomicity protects its own related writes. It does not automatically force a separate reader's two statements to share a snapshot.
2. A transaction can therefore be locally atomic while a poorly designed multi-statement report mixes versions. Different guarantees protect different boundaries.
3. Document whether a report is “as of one snapshot,” “latest available per query,” or an eventually assembled view. The label affects how readers should interpret discrepancies.

■ 23.3.5 TECHNICAL — the engineer's version

1. Under PostgreSQL Read Committed, successive commands can see different committed data. This makes it unsuitable for a multi-command invariant calculation that silently assumes a stable view unless additional controls establish the required condition. [S66]

2. Predicate membership also matters. Repeating `WHERE status='active'` may change the rows returned even when no previously returned row changes value. Analyse both row values and the set being counted.
3. Do not “fix” a mixed-snapshot report by clamping an unexpected total to a preferred number. Preserve evidence, identify the observation boundary and rerun the defined query under a suitable contract.

■ 23.3.6 WORDS — remember these

1. **Non-repeatable read:** rereading a row yields a later committed value — a permitted observation change under some isolation contracts. **Mixed-snapshot calculation:** combining facts observed at different boundaries — a derived result that may correspond to no single database state. **Observation contract:** the stated point or interval a result describes — the temporal and visibility meaning assigned to a query or report.

23.4 Write skew

■ 23.4.1 PLAIN — in simple words

1. Write skew is a different trap from two people overwriting the same row. Two transactions can change different rows and still break a rule about those rows together.
2. Each transaction may see a stable, internally consistent snapshot. The trouble is that both make decisions without seeing the other’s eventual change.
3. Protecting only the changed rows may miss the shared condition. The rule, not just the UPDATE target, must guide the concurrency design.

■ 23.4.2 PLAIN — a picture in your head

1. Two assistants are on duty, and the rule requires at least one to remain. Each sees that the other is present and decides to leave.
2. They lock their own lockers carefully, but neither coordinates the shared requirement that somebody stay.
3. **Where the comparison breaks:** real staffing has human communication and policy enforcement outside a database. This is a small logical model of a cross-row invariant, not a recommended scheduling system.

■ 23.4.3 PLAIN — a worked example

1. A new table contains `(Mira,on_duty=1)` and `(Dev,on_duty=1)`. The invariant is `count(on_duty=1) >= 1`.
2. T1’s snapshot sees two assistants and sets Mira’s row to 0. T2’s snapshot also sees two and sets Dev’s row to 0. Their writes concern different rows.
3. If both transactions are allowed to commit under this snapshot model, the final count is 0. Either serial order would have made the second transaction see only one assistant and refuse its departure.
4. The example illustrates a serialization anomaly possible under snapshot isolation, including PostgreSQL’s Repeatable Read model. It is not claimed as a reproduced SQLite two-writer trace. [S66]

■ 23.4.4 PLAIN — what is really happening inside

1. Each transaction's decision depends on a fact changed by the other. Those cross-dependencies create a cycle that cannot be explained by a safe serial order of the stated decision logic.
2. Locking only Mira's row in T1 and only Dev's row in T2 still leaves distinct protected targets. A protocol must cover the shared duty condition, not merely each private update.
3. Possible designs include a common guard record, locking an appropriately defined shared set with consistent ordering, or serializable transactions with full retries. Each proposal must include every path that can change the duty state.

■ 23.4.5 TECHNICAL — the engineer's version

1. Snapshot isolation can prevent same-row concurrent update patterns while still allowing write skew through disjoint write sets and intersecting read dependencies. Stable reads therefore do not imply serializability. [S66]
2. A correct serializable execution of the stated decision logic cannot commit both departures from the initial two-person state. At least one transaction must observe the changed condition, wait appropriately or abort and retry under the implementation's protocol.
3. A serializable transaction containing faulty decision logic can still commit a bad business outcome. Isolation cannot enforce an invariant that the program neither checks nor represents correctly.

■ 23.4.6 WORDS — remember these

1. **Write skew:** separate writes jointly break a shared rule — an anomaly in which transactions act on overlapping read conditions while changing different records. **Read dependency:** a decision relies on an observed fact — a relationship between a transaction's read and another transaction's relevant write. **Serializable outcome:** a result explainable by a legal one-at-a-time execution — an outcome equivalent to some serial order of the participating transactions.

23.5 Serialisable execution

■ 23.5.1 PLAIN — in simple words

1. Serializable means that committed transactions behave as though they occurred in some one-at-a-time order. The engine may still run their internal work concurrently.
2. The word "some" matters. Serializability alone is not a universal promise about real-time order observed by external callers across every system component.
3. An engine may preserve the guarantee by refusing a transaction and asking the application to retry. A refusal is part of the concurrency contract, not proof that the database has become unreliable.

■ 23.5.2 PLAIN — a picture in your head

1. Several clerks prepare paperwork in parallel. The final accepted collection must be explainable as though the desk handled each complete case in a valid sequence.
2. If two prepared cases cannot both fit such a sequence, one may need to be prepared again using newer information.
3. **Where the comparison breaks:** the engine does not necessarily find and print a literal serial schedule for the application. Implementations use locks or dependency detection to prevent forbidden outcomes.

■ 23.5.3 PLAIN — a worked example

1. Return to the duty model. If Mira's departure is treated first, one assistant remains. Dev's later decision must then refuse to leave. Reverse the order and Mira must stay instead.
2. These are two different acceptable serial outcomes. The system need not choose the same assistant every time unless a separate fairness policy requires that.
3. Committing both departures is unacceptable under the stated serial decision logic. A retry must repeat the read and decision, not merely resubmit the old "set off duty" statement.
4. The operation's correctness includes the retry path because that path may be the normal way the engine resolves competition.

■ 23.5.4 PLAIN — what is really happening inside

1. PostgreSQL's serializable implementation tracks potentially dangerous read/write dependencies and may abort a transaction to prevent a serialization anomaly.
2. Its predicate-lock information is not the same as an ordinary blocking row lock. Treating every lock-related term as "another request must wait here" gives the wrong mental model.
3. Results read inside a transaction that later aborts cannot be presented as though that transaction's business decision committed. Keep externally visible success aligned with the final outcome.

■ 23.5.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 Serializable uses Serializable Snapshot Isolation. Its SIReadLock predicate-lock records support dependency checking; they do not block ordinary operations or cause deadlocks in the manner of blocking locks. [S66]
2. Correctness arguments require appropriate participation by the transactions that interact with the invariant. Mixing arbitrary bypass writers into a supposedly protected application can invalidate the application's proof.
3. Distinguish serializability from strict serializability and linearizability. Chapter 36 introduces real-time ordering explicitly; do not advertise stronger guarantees merely because a transaction used the Serializable label.

■ 23.5.6 WORDS — remember these

1. **Serializability:** concurrent work has a valid serial explanation — equivalence of committed transaction effects to an execution in some serial order. **Serialization failure:** a transaction is refused to preserve the isolation guarantee — an error requiring an appropriate full-operation retry or failure response. **Predicate tracking:** remember the read conditions that matter — implementation metadata used to detect dependencies involving rows or ranges a transaction read.

23.6 Retries and operational cost

■ 23.6.1 PLAIN — in simple words

1. Stronger isolation can make the application simpler in one respect: it may avoid some fragile manual coordination. It does not remove the need to handle failures and retries.
2. A retry consumes work and can repeat unsafe external effects. Re-running a database decision is different from sending another payment instruction or email.
3. The operational question is whether the complete protocol meets its correctness and performance requirements under the actual workload. A label alone cannot answer that question.

■ 23.6.2 PLAIN — a picture in your head

1. A clerk whose draft conflicts with another case returns to the beginning, reads the current facts and prepares a new decision. Re-signing the old draft would preserve the original conflict.
2. The clerk also checks whether a confirmation was already dispatched. Starting again must not mean blindly sending everything twice.
3. **Where the comparison breaks:** software needs explicit error classes, deadlines and request identities. A human's memory of "I already handled that" is not a durable concurrency protocol.

■ 23.6.3 PLAIN — a worked example

1. Suppose 100 submitted operations produce 100 initial transaction attempts, and 8 encounter retryable serialization failures. If each of those succeeds on one additional attempt, the database executed 108 attempts for 100 completed operations.
2. The retry count is 8, not an 8% business failure rate. But the extra attempts consumed resources and increased some callers' latency.
3. If 2 operations exhaust their retry deadline, report 98 completions and 2 failures separately. Do not compute latency only for the fastest successes and call it the service's complete performance.
4. These figures are invented arithmetic for reporting definitions, not a measured property of Serializable isolation.

■ 23.6.4 PLAIN — what is really happening inside

1. A retry must re-enter a clean transaction and repeat every read and decision that depended on the previous attempt's view.
2. Classify errors. A serialization failure and an invalid required field have different remedies. Repeating an invalid field cannot make it valid.
3. Bound retry attempts by a deadline and cancellation policy, and use suitable backoff when contention warrants it. Keep a stable business request identity so uncertainty does not become accidental duplication.

■ 23.6.5 TECHNICAL — the engineer's version

1. PostgreSQL recommends retrying the complete transaction after SQLSTATE 40001, including application logic that selected the statements and values. Deadlock error 40P01 can also justify a retry, while other errors require context-specific classification. [S120]
2. Measure attempt counts, conflict classes, wait time, successful-operation latency and deadline exhaustion. Do not equate higher raw attempt throughput with more useful business completions.
3. The companion exercises model anomaly schedules and test bounded SQLite transaction behaviour. PostgreSQL isolation examples here are documentation-grounded, not executed server experiments. Independent engine-specific testing remains necessary before deploying a matching protocol.

■ 23.6.6 WORDS — remember these

1. **Full-transaction retry:** repeat the decision from fresh protected reads — a new attempt that does not reuse an invalidated transaction's conclusions blindly. **Attempt amplification:** extra work caused by retries — the ratio or count of transaction attempts relative to completed business operations. **Retry deadline:** the point beyond which another attempt is not allowed — an operation-level bound on time spent resolving transient failures.

23.97 Practice and worked answers

1. **Choose the snapshot boundary.** PostgreSQL Repeatable Read begins at 09:00, but its first ordinary statement runs at 09:01. **Answer:** the documented snapshot is established by the first non-transaction-control statement, not merely by BEGIN.
2. **Trace two reads.** R reads 100, W commits 120, R reads again under Read Committed. **Answer:** 100 then 120 is permitted for ordinary separate SELECT statements.
3. **Explain a total of 110.** A transfer preserves $60+40=50+50=100$, but a reader combines old $A=60$ with new $B=50$. **Answer:** the reader mixed observation boundaries; the transfer's own atomicity did not force the reader's statements to share a snapshot.
4. **Identify write skew.** Two transactions see two on-duty assistants and each disables a different assistant. **Answer:** disjoint writes can jointly violate the shared minimum-one invariant under the stated snapshot model.
5. **Assess a row-lock proposal.** Each assistant locks only their own row. **Answer:** that does not by itself protect the shared count. Specify a protocol covering the invariant and all writers.
6. **Interpret Serializable.** Must the engine run every transaction physically one after another? **Answer:** no. Committed results must have a valid serial explanation; internal execution may overlap.
7. **Retry correctly.** A serialization failure occurs after a stock decision. **Answer:** start a clean transaction and repeat the relevant reads and decision. Do not resubmit only the final write with stale assumptions.
8. **Report attempts.** One hundred operations need 108 attempts and all complete. **Answer:** report 100 completions and 8 extra attempts, with latency and retry distribution. Do not call the extra attempts eight failed business operations.

23.98 Common wrong ideas

1. **Wrong:** isolation names have identical behaviour in every engine. **Right:** consult the chosen implementation and version.
2. **Wrong:** Repeatable Read universally permits phantoms. **Right:** PostgreSQL 17 provides stronger snapshot behaviour than that minimum description.
3. **Wrong:** a stable snapshot prevents every business anomaly. **Right:** write skew can involve disjoint writes based on a shared snapshot condition.
4. **Wrong:** an atomic transfer guarantees every separately assembled report sees one total. **Right:** the reader needs its own observation contract.
5. **Wrong:** Serializable means one physical worker and no parallelism. **Right:** it constrains the explanation of committed outcomes.
6. **Wrong:** a serialization failure can be fixed by replaying only the last UPDATE. **Right:** repeat the entire dependent decision in a new transaction.
7. **Wrong:** every database error deserves a retry. **Right:** permanent validation and identity conflicts need different handling.
8. **Wrong:** a documented PostgreSQL schedule is a measured SQLite result. **Right:** implementation claims and experiment evidence must remain separate.

23.99 Chapter summary in 20 lines

1. Isolation describes how overlapping transactions interact and what they may observe.
2. Dirty reads concern another transaction's uncommitted work.

3. Non-repeatable reads concern changed committed values across repeated reads.
4. Phantoms concern changed membership of a repeated predicate query.
5. PostgreSQL 17 treats Read Uncommitted as Read Committed.
6. PostgreSQL Read Committed ordinary SELECT obtains a statement-start snapshot.
7. Successive statements can therefore see different committed states.
8. PostgreSQL Repeatable Read retains its first non-control-statement snapshot.
9. A transaction can see its own preceding writes under the documented rules.
10. Snapshot consistency is not a universal serializability guarantee.
11. Mixing snapshots can produce a total corresponding to no single state.
12. Write skew can break a shared rule through changes to different rows.
13. Protect the invariant, not merely each writer's individual target.
14. Serializable outcomes must be explainable by a valid serial order.
15. Serializable execution need not be physically sequential.
16. PostgreSQL's dependency tracking can abort a transaction to preserve that guarantee.
17. Repeat the complete dependent decision after a retryable serialization failure.
18. Keep external effects outside blind transaction retries.
19. Measure retry work and deadline failures alongside successful-operation latency.
20. Match every claim to the engine, version, configuration and evidence actually used.

Locks, Deadlocks and Safe Retries

24.0 What this chapter gives you

1. A lock is a coordination mechanism, not a synonym for failure. Waiting can be the correct way to protect a record; waiting forever or repeating the wrong operation is not.
2. This chapter separates lock scope, compatibility, blocking, deadlock, timeouts and retries. You will draw a wait graph, choose a consistent acquisition order and identify effects that must not be repeated blindly.
3. Lock names and error codes are implementation-specific. PostgreSQL 17 examples are documented illustrations. The supplied SQLite exercises use fresh teaching databases and do not establish PostgreSQL lock behaviour.

24.1 What is locked

■ 24.1.1 PLAIN — in simple words

1. A lock reserves a kind of access to a resource while an operation is in progress. The resource might be a row, a table or an application-defined coordination key.
2. Saying “the database is locked” hides the important details. Which operation is waiting, which resource does it need, who currently holds conflicting access, and when should that holder finish?
3. A lock is effective only within the protocol that honours it. An application-defined lock does not magically stop an unrelated writer that ignores that agreement.

■ 24.1.2 PLAIN — a picture in your head

1. A staff member places a sign on a particular filing drawer while correcting its contents. Other staff follow the drawer-access rule until the sign is removed.
2. Locking the drawer is different from closing the entire room. A room closure affects far more work.
3. **Where the comparison breaks:** database locks are not all exclusive signs, and normal readers may use older visible row versions without waiting for a writer. Some lock names also describe tables even when they contain the word “row.”

■ 24.1.3 PLAIN — a worked example

1. Request A is updating product LOCK-PEN. Request B wants to update the same product. Request C is reading an unrelated product. A useful diagnosis identifies whether B conflicts with A and whether C needs a conflicting resource at all.
2. A table-wide schema change may have a broader lock requirement than the ordinary row update. It can therefore wait behind activity that would not block another ordinary row change.

3. Record the actual lock mode and object identifier in an engine investigation. Do not infer a table lock from a slow screen or infer a row lock solely from a statement's WHERE clause.
4. The example names are synthetic; it does not assert a particular observed server lock trace.

■ 24.1.4 PLAIN — what is really happening inside

1. The engine checks requested access against existing incompatible locks. It may grant the request, queue it, time it out or later select a transaction for abort.
2. Locks can be acquired implicitly by SQL statements or explicitly through locking clauses. A statement may acquire more than one kind of lock.
3. The business resource and engine resource need not match one-to-one. Protecting “remaining branch capacity” requires identifying the concrete record or protocol representing that capacity.

■ 24.1.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 has table-level and row-level lock modes. Names such as ROW EXCLUSIVE in the table-lock family still denote table-level locks; the name alone does not establish row granularity. [S118]
2. PostgreSQL advisory locks coordinate application-defined resources. Their usefulness depends on every relevant participant following the convention; they are not substitutes for ordinary data constraints or authorization. [S118]
3. SQLite's database-level writer coordination differs substantially from PostgreSQL row-locking patterns. Describe its transaction and journal mode explicitly rather than translating a PostgreSQL lock diagram unchanged. [S54]

■ 24.1.6 WORDS — remember these

1. **Lock scope:** the resource access being coordinated — the row, table, key or other engine/application object covered by a lock. **Lock holder:** the participant currently retaining protected access — the session or transaction owning a granted lock. **Advisory lock:** a coordination agreement writers must choose to follow — an application-defined locking facility not automatically tied to every relevant data mutation.

24.2 Lock modes and duration

■ 24.2.1 PLAIN — in simple words

1. Not every lock excludes every other lock. A compatibility rule determines which kinds of access may coexist.
2. Duration matters as much as scope. A brief conflict can be harmless; a transaction left open while someone answers a phone can hold resources long after useful database work stops.
3. Know what releases a lock. Transaction completion, statement completion, session termination and explicit release are different boundaries.

■ 24.2.2 PLAIN — a picture in your head

1. A reading room may allow several readers at once but stop new visitors during a structural repair. The rules depend on what each person is doing, not just whether someone is present.
2. A repair permit that lasts until the repair finishes is different from a building key retained by a staff member all day.

3. **Where the comparison breaks:** an engine's compatibility matrix is exact and product-specific. Everyday "reader" and "writer" labels are not precise enough to reproduce all lock modes.

■ 24.2.3 PLAIN — a worked example

1. In PostgreSQL 17, an ordinary SELECT obtains ACCESS SHARE on its referenced table. ACCESS EXCLUSIVE conflicts with it; most other table-lock modes do not conflict with ACCESS SHARE.
2. Thus a schema operation requiring ACCESS EXCLUSIVE can be delayed by a long-running reader, even when that reader is not changing any row.
3. Conversely, a row locked for an update does not necessarily block an ordinary MVCC SELECT from reading an appropriate visible version. "A writer exists" is not enough to predict every reader's wait. [S118]
4. The practical response is to inspect the actual requested and held modes, not to memorize that all readers either always block or never block writers.

■ 24.2.4 PLAIN — what is really happening inside

1. Transaction-scoped locks generally remain until the transaction ends, subject to documented savepoint and implementation behaviour.
2. Application code can accidentally lengthen that interval through network calls, slow loops, user interaction or an abandoned transaction in a pooled connection.
3. Session-level advisory locks have different lifetime rules from transaction-level advisory locks. Returning a connection to a pool without clearing session state can pass unwanted coordination state to another logical request.

■ 24.2.5 TECHNICAL — the engineer's version

1. PostgreSQL distinguishes session-level advisory locks from transaction-level advisory locks. Session-level advisory locks do not follow ordinary transaction rollback semantics; transaction-level advisory locks are released at transaction end. [S118]
2. Define a maximum intended transaction duration and observe time spent idle inside transactions separately from active query work. A short individual query does not imply a short lock-holding transaction.
3. Savepoint rollback can release locks acquired after that savepoint under PostgreSQL's documented rules. Do not extrapolate this into a claim that any arbitrary external resource acquired in the same code block is also released. [S118] [S121]

■ 24.2.6 WORDS — remember these

1. **Lock compatibility:** which access modes can coexist — the engine's conflict relation between requested and held lock modes. **Lock lifetime:** how long protected access remains held — the release boundary determined by transaction, statement, session or explicit protocol. **Idle in transaction:** a transaction remains open without active useful work — a state that can retain locks and snapshots while the application waits elsewhere.

24.3 Blocking versus deadlock

■ 24.3.1 PLAIN — in simple words

1. Blocking means one operation is waiting for access held incompatibly by another. The holder may finish normally and release the resource.

2. A deadlock is a cycle of waits: each participant needs something held by another participant in the cycle, so none can finish without intervention.
3. A long wait is not automatically a deadlock. It may be an overloaded server, a slow holder, an abandoned transaction or an external dependency.

■ 24.3.2 PLAIN — a picture in your head

1. Mira waits for Dev to finish using the label printer. Dev is printing and will finish. That is blocking.
2. Now Mira holds the label printer and needs Dev's scanner, while Dev holds the scanner and needs Mira's printer. Neither releases the held device until acquiring the other. That is a deadlock in this model.
3. **Where the comparison breaks:** databases can detect some lock cycles and abort a participant. A cycle spanning outside services may not be visible to one database's detector.

■ 24.3.3 PLAIN — a worked example

1. Let A hold resource X and request Y. Let B hold Y and request X. Draw an arrow from the waiter to the holder: $A \rightarrow B$ and $B \rightarrow A$.
2. The arrows form a cycle. Neither can obtain its requested resource while both retain their current locks under the stated protocol.
3. Compare $A \rightarrow B \rightarrow C$ with no arrow returning from C. That chain is blocking, not a cycle. C may complete and release the chain.
4. The companion graph exercise checks directed cycles in a small synthetic wait graph. It is not a full model of an engine's lock manager, lock upgrades or distributed deadlock detection.

■ 24.3.4 PLAIN — what is really happening inside

1. A database may inspect waits and choose a victim transaction to abort. Releasing that transaction's locks lets another participant continue.
2. The application must handle the victim's failure even when its business request was otherwise valid. It should not assume that the first or newest request is always chosen.
3. A statement timeout or lock timeout can terminate waiting without proving that a cycle existed. Preserve the actual error classification in logs and retry decisions.

■ 24.3.5 TECHNICAL — the engineer's version

1. A wait-for graph represents transactions as vertices and incompatible-resource waits as directed edges. A cycle is the key condition in the simplified single-resource ownership model used here.
2. PostgreSQL automatically detects deadlocks and aborts one involved transaction. Its documentation advises against relying on a predictable choice of victim. [S118]
3. A client-side deadline, server statement timeout, lock timeout and detected deadlock are different events. Keep their error codes and transaction-state consequences separate instead of collapsing them into "database busy."

■ 24.3.6 WORDS — remember these

1. **Blocking:** waiting for conflicting access to be released — a resource wait that need not contain a dependency cycle. **Deadlock:** a cycle of participants waiting on one another — a condition requiring some participant to release, abort or otherwise break the cycle. **Wait-for graph:** a map of who waits for whom — a directed graph used to reason about resource dependencies.

24.4 Detection and ordering

■ 24.4.1 PLAIN — in simple words

1. A consistent rule for acquiring multiple resources can prevent many deadlocks. Everyone asks for them in the same agreed order instead of choosing an order independently.
2. This works only when the protocol is complete. Hidden trigger work, alternate code paths or lock upgrades can introduce additional dependencies.
3. Prevention and recovery belong together. Even a carefully ordered application should handle deadlock errors because other participants and maintenance operations may not follow the same small model.

■ 24.4.2 PLAIN — a picture in your head

1. Whenever two drawers are needed, staff open the lower-numbered drawer first. Someone may wait, but two staff following that rule cannot each hold the higher drawer while waiting for the lower one in the simple two-drawer model.
2. The order gives everyone a shared coordination language.
3. **Where the comparison breaks:** real statements can touch rows through plans, constraints or triggers. Sorting application IDs is not by itself a complete proof of all engine lock acquisition orders.

■ 24.4.3 PLAIN — a worked example

1. A transfer-like teaching operation needs accounts AC-02 and AC-09. A second operation needs the same pair in the opposite business direction.
2. Both acquire their required coordination locks in key order: AC-02, then AC-09. One may wait for AC-02 before holding AC-09, avoiding the earlier reversed-order pattern.
3. Business direction remains independent of lock order. The destination account can be locked first if its key sorts first; locking order does not change the meaning of the transfer.
4. Include complete scoped keys in the ordering rule. Ordering only by a local number while ignoring tenant or branch can create ambiguous protocol descriptions.

■ 24.4.4 PLAIN — what is really happening inside

1. A common acquisition order removes certain cycles from the allowed protocol. It is a structural argument, not merely an attempt to make collisions less likely.
2. Keep transactions short after acquiring locks. Holding the first lock while performing unrelated expensive work increases waiting even if no cycle forms.
3. During diagnosis, capture the waiting statement, holder, transaction age, lock objects and recent changes. Avoid killing arbitrary sessions without understanding the rollback and business-outcome consequences.

■ 24.4.5 TECHNICAL — the engineer's version

1. PostgreSQL recommends acquiring locks on multiple objects in a consistent order and acquiring the most restrictive needed mode first where appropriate. This reduces common deadlock patterns but does not eliminate the need for error handling. [S118]
2. Document explicit locking SQL separately from the business updates. Include trigger, foreign-key and index-related interactions in a real review rather than treating them as invisible implementation magic.

3. A diagnostic procedure should favour bounded observation before intervention. Record which transaction was cancelled, which request it represented and how its outcome will be reconciled after rollback or connection loss.

■ 24.4.6 WORDS — remember these

1. **Acquisition order:** the sequence in which resources are requested — a shared ordering protocol intended to prevent cyclic waits. **Lock upgrade:** requesting stronger access while already holding weaker access — a mode transition that can introduce additional conflicts. **Deadlock victim:** the transaction selected to break a cycle — a participant aborted by the engine so other work can proceed.

24.5 Bounded retries

■ 24.5.1 PLAIN — in simple words

1. A retry is another attempt at the same intended operation, not permission to repeat every action blindly until something works.
2. Some failures are temporary, such as a chosen serialization conflict. Others are permanent until the input or policy changes, such as an invalid quantity or an incompatible request-key reuse.
3. A retry needs a stopping rule. Without one, a failing system can spend more and more resources repeating work while the caller receives no useful outcome.

■ 24.5.2 PLAIN — a picture in your head

1. A clerk who finds the desk briefly occupied waits a little and returns. A clerk whose form lacks a required signature does not solve the problem by submitting the same unsigned form a hundred times.
2. The reason for failure determines the next action.
3. **Where the comparison breaks:** software cannot rely on intuition about whether a failure is temporary. It needs documented error classification, transaction cleanup, a deadline and recorded attempt outcomes.

■ 24.5.3 PLAIN — a worked example

1. Define a teaching retry policy allowing at most four total attempts within a two-second operation deadline. The first attempt is not called a retry; at most three further attempts remain.
2. Suggested wait ceilings might grow from 25 to 50 to 100 milliseconds, with a bounded random delay beneath each ceiling. These are invented parameters for illustrating the policy, not recommended universal production values.
3. Before each new attempt, check cancellation and remaining deadline. Start a clean transaction and reread the facts used in the decision.
4. If the deadline expires, return a defined failure or unresolved status as appropriate. Do not silently extend the caller's deadline just because the retry counter has room left.

■ 24.5.4 PLAIN — what is really happening inside

1. Backoff reduces immediate repeated competition. Jitter spreads retry timing so a crowd does not wake and collide again in perfect synchrony.
2. Retrying only the failed final statement can preserve stale decisions from the previous transaction. Retry the entire dependent operation where the engine's error contract requires it.

3. Record attempts separately from business requests. The same request may have several failed attempts and one committed outcome; those counts answer different operational questions.

■ 24.5.5 TECHNICAL — the engineer's version

1. PostgreSQL SQLSTATE 40001 requires retrying the complete transaction, including application decision logic. 40P01 deadlock detection can also justify a full retry. Uniqueness failures need context: some arise from competing decisions, while others are persistent conflicts. [S120]
2. A robust retry wrapper takes a monotonic deadline, maximum attempts, an allowlist of retryable failure classes and an operation that owns its transaction. Cleanup must complete before a new attempt starts.
3. A lost connection around commit creates a different problem from a confirmed abort. Use request identity and reconciliation to resolve uncertain completion rather than assuming the next attempt starts from an uncommitted world. [S126]

■ 24.5.6 WORDS — remember these

1. **Backoff:** wait longer before another attempt — a retry-spacing strategy that reduces repeated contention or overload. **Jitter:** deliberately vary the waiting time — bounded randomness used to avoid synchronized retries. **Retry budget:** the allowed additional effort — limits on attempts, elapsed time or resources spent resolving transient failures.

24.6 Avoiding repeated external effects

■ 24.6.1 PLAIN — in simple words

1. Database rollback does not recall an email, unprint a receipt or reverse an already accepted external instruction.
2. A retryable transaction that performs such an effect midway can cause it again when the transaction restarts. Even moving the effect after commit leaves a crash gap before sending it.
3. The design needs a durable connection between the accepted business action and the external work still to be attempted. An outbox records that intent, but delivery and consumer behaviour still need their own rules.

■ 24.6.2 PLAIN — a picture in your head

1. Mira writes “send this confirmation” into the accepted reservation record before a messenger takes responsibility for delivery. Another messenger can later find unfinished deliveries.
2. A messenger who delivered the letter but failed to mark the list may cause another delivery attempt. The list solves forgetting, not every duplicate.
3. **Where the comparison breaks:** a recipient's system can sometimes deduplicate by a stable identifier, but a real person's inbox or an arbitrary external service may not offer the exact atomicity required by your application.

■ 24.6.3 PLAIN — a worked example

1. Attempt A sends a confirmation, then encounters a serialization failure before committing its reservation. A full retry sends the confirmation again. Two messages now describe a reservation that committed only on the second attempt, or perhaps never committed at all.
2. Instead, record the reservation and an outbox row in one transaction. If that transaction aborts, neither accepted reservation nor accepted delivery intent remains.

3. A dispatcher later sends the outbox message. If it crashes after sending but before marking completion, it may send again. Give the message a stable identity and define the receiver's duplicate-handling contract.
4. Chapter 26 demonstrates a local inbox record and database effect committed together. That bounded design does not prove exactly-once email delivery.

■ 24.6.4 PLAIN — what is really happening inside

1. A transaction boundary ends at the participating storage system. An HTTP call to another service is not automatically enlisted in that transaction.
2. The outbox transforms an immediate external side effect into durable local intent. Delivery becomes a separate state machine with attempts, acknowledgements, retries and unresolved outcomes.
3. A retry framework should therefore receive an operation designed for repetition, not wrap arbitrary business code and hope rollback covers everything it did.

■ 24.6.5 TECHNICAL — the engineer's version

1. The transactional outbox pattern records business changes and message intent in the same local transaction. Dispatch can still produce duplicate messages, so consumers must handle the delivery semantics explicitly. [S125]
2. Provider-supported idempotency keys are useful only under the provider's actual scope, payload-equivalence and retention contract. A key header with no such contract is just an uninterpreted string. [S126]
3. Tests should inject failures before commit, after commit, before send and after send but before acknowledgement recording. State which components are models and which are actual processes; a simulated crash point is not a power-loss experiment.

■ 24.6.6 WORDS — remember these

1. **External effect:** a change outside the local transaction — an email, print, remote API action or other operation not undone by ordinary database rollback. **Outbox:** a durable list of external work implied by accepted changes — message-intent records committed atomically with local business state. **Delivery gap:** uncertainty between an external action and its recorded acknowledgement — a failure window that can require replay or reconciliation.

24.97 Practice and worked answers

1. **Read a lock name.** PostgreSQL reports ROW EXCLUSIVE in its table-lock family. **Answer:** this is a table-level mode despite its name. Determine scope from the documented lock family and object.
2. **Explain a waiting schema change.** A long SELECT holds ACCESS SHARE while a change needs ACCESS EXCLUSIVE. **Answer:** those modes conflict; a reader can delay the broader operation without changing rows.
3. **Find a cycle.** $A \rightarrow B, B \rightarrow C, C \rightarrow A$. **Answer:** the graph contains a cycle. $A \rightarrow B \rightarrow C$ with no return edge is only a chain in this simplified model.
4. **Separate a timeout.** A client gives up after two seconds. **Answer:** elapsed time alone does not prove deadlock. Inspect the actual failure class and holder state.

5. **Choose an order.** Opposite-direction operations need AC-02 and AC-09. **Answer:** both should follow the same resource-acquisition order, independent of business direction, while reviewing all additional implicit locks.
6. **Count attempts.** A maximum of four attempts succeeds on the fourth. **Answer:** one original attempt and three retries occurred. Record four attempts and one completed business operation.
7. **Classify an invalid request.** Quantity is negative. **Answer:** reject or correct the input under the contract; contention backoff cannot make it valid.
8. **Find the external-effect gap.** A dispatcher sends then crashes before marking the outbox row delivered. **Answer:** replay may duplicate delivery. Stable message identity and receiver-side handling are still needed.

24.98 Common wrong ideas

1. **Wrong:** any lock is a database failure. **Right:** coordination often requires temporary incompatible access.
2. **Wrong:** all readers block all writers. **Right:** compatibility and MVCC behaviour depend on the operation and engine.
3. **Wrong:** every lock ends at COMMIT. **Right:** session-scoped advisory locks have different lifetime rules.
4. **Wrong:** a long wait proves a deadlock. **Right:** a deadlock requires the relevant dependency cycle, not merely elapsed time.
5. **Wrong:** sorting two IDs proves the whole application deadlock-free. **Right:** implicit locks and alternate paths still matter.
6. **Wrong:** retry every error. **Right:** distinguish transient conflicts, permanent rejection and unknown completion.
7. **Wrong:** retrying only the last statement always preserves the business operation. **Right:** dependent reads and decisions may need to run again.
8. **Wrong:** an outbox guarantees exactly-once external delivery. **Right:** it records local intent atomically; delivery duplicates remain a separate problem.

24.99 Chapter summary in 20 lines

1. A lock coordinates access to a specified resource.
2. Diagnose the waiter, holder, resource, mode and expected release boundary.
3. Lock names do not always reveal their granularity.
4. Compatibility matrices decide which modes can coexist.
5. A broad schema lock can wait behind ordinary reading activity.
6. Row locks need not block ordinary MVCC reads of visible versions.
7. Transaction duration controls how long many locks remain relevant.
8. Idle transactions can retain resources without doing useful work.
9. Session-level advisory locks differ from transaction-scoped locks.
10. Blocking is a wait; deadlock is a cycle of waits under the relevant model.
11. A timeout alone does not establish a deadlock.
12. A detector may abort a victim so other transactions can proceed.
13. Consistent resource-acquisition order prevents common cyclic patterns.

14. Include implicit locks and all mutation paths in a real protocol review.
15. Retry only failures classified as retryable under the actual contract.
16. Repeat dependent reads and decisions inside a clean transaction.
17. Bound retries by attempts, a monotonic deadline and cancellation.
18. Backoff and jitter can reduce synchronized repeated contention.
19. Database rollback does not undo ordinary external effects.
20. Atomic outbox intent still requires a delivery and duplicate-handling protocol.

Versions, Snapshots and Optimistic Concurrency

25.0 What this chapter gives you

1. A record can have more than one relevant version: the engine's internal versions, a saved business revision and the copy someone is editing on a screen. Confusing these versions creates subtle bugs.
2. You will trace snapshot visibility, understand why old readers affect cleanup, implement an explicit revision check and choose a conflict policy that does not silently discard someone's work.
3. The examples introduce a new draft quotation and do not change the agreed historical prices in O-1042 or O-1043. Engine version metadata is not treated as an everlasting business audit log.

25.1 Old and new row versions

■ 25.1.1 PLAIN — in simple words

1. When a database changes a record, an earlier version may remain available for readers that still need it. The engine can decide which version each reader is allowed to see.
2. This helps readers and writers coexist without requiring every reader to wait for every change. It does not mean that all old versions are kept forever.
3. A business history has a different purpose. If the shop must explain who approved an agreed price, it needs deliberately recorded history and retention rules rather than assuming the database's temporary internal versions will answer later.

■ 25.1.2 PLAIN — a picture in your head

1. Imagine a shared document where a reader can finish the edition they opened while an editor prepares a newer edition for later readers.
2. The printing office keeps old sheets only as long as its current readers require them. An archive keeper has a separate job if editions must be preserved for years.
3. **Where the comparison breaks:** a database does not necessarily copy an entire document for each edit. Physical version storage and visibility checks vary by engine, and cleanup can reclaim old storage.

■ 25.1.3 PLAIN — a worked example

1. In a new draft example, D-EDIT has quantity 3 at business revision 7. A transaction changes it to quantity 4 at revision 8.

2. An older permitted snapshot may still read the earlier quantity while a later snapshot reads the committed replacement. The application revision identifies the edit state chosen by the application; the engine may use other internal metadata to decide visibility.
3. After old snapshots are no longer relevant, the engine may reclaim obsolete versions. A future request for “the reason revision 7 existed” cannot rely on that reclaimed physical version.
4. If reasons and approvals matter, record them in an explicit history model with a defined scope and retention policy.

■ 25.1.4 PLAIN — what is really happening inside

1. Multi-version concurrency control, or MVCC, associates versions with transactional visibility information. Readers evaluate visibility according to their snapshot rules.
2. An UPDATE can therefore create storage and cleanup work beyond changing one displayed number. Index maintenance and space reuse also depend on the engine’s implementation.
3. MVCC is not one universal algorithm. Use the general concept to ask useful questions, then consult the actual engine for version layout, cleanup and conflict behaviour.

■ 25.1.5 TECHNICAL — the engineer’s version

1. PostgreSQL uses MVCC so statements can see appropriate snapshots while concurrent transactions modify data. Its internal tuple metadata participates in visibility, but it is not a substitute for application-level history. [S119] [S124]
2. PostgreSQL’s ctid identifies a physical row-version location and can change. Transaction identifiers also have lifecycle and wraparound considerations. Neither should casually replace an immutable business key or an explicitly designed revision protocol. [S124] [S123]
3. An application revision should have a documented domain, increment rule and generation boundary. Its correctness depends on every relevant writer advancing or otherwise respecting it.

■ 25.1.6 WORDS — remember these

1. **MVCC**: let readers use suitable versions while changes occur — multi-version concurrency control using versioned data and transactional visibility rules. **Business revision**: the edit state named by the application — an explicit version value used for conflict detection or documented history. **Physical row location**: where one stored version currently resides — engine metadata such as PostgreSQL’s ctid, not an enduring logical identity.

25.2 Snapshot visibility

■ 25.2.1 PLAIN — in simple words

1. A snapshot decides which committed work from other transactions belongs in a reader’s view. “Newest record on disk” is not necessarily the version that reader should receive.
2. A reader can therefore obtain an older valid version without receiving a corrupt or random answer. The answer must be interpreted under the reader’s observation contract.
3. A screen’s age is another matter. After the database returns a result, the browser can keep displaying it indefinitely. Database snapshot guarantees do not continuously refresh every previously returned screen.

■ 25.2.2 PLAIN — a picture in your head

1. A visitor enters an exhibition arranged as it was at opening time. New exhibits can be prepared for later visitors without rearranging the earlier visitor's tour halfway through.
2. Once the visitor leaves with a photograph, the exhibition staff no longer control how long they keep using that photograph.
3. **Where the comparison breaks:** snapshots also interact with a transaction's own writes and conflict checks. They are not merely wall-clock photographs, and not every engine starts them at BEGIN.

■ 25.2.3 PLAIN — a worked example

1. Reader R establishes a retained snapshot while D-EDIT is at revision 7. Writer W commits revision 8. R may still read revision 7 under the retained-snapshot contract, while a new reader N sees revision 8.
2. R's user later submits an edit based on revision 7. The server should not assume the submitted revision is current merely because it once came from a valid database read.
3. A write protocol can require "apply this change only if the current business revision is still 7." If revision 8 now exists, return a conflict and preserve the user's proposed change for review.
4. This separates reading an old valid view from overwriting newer work without permission.

■ 25.2.4 PLAIN — what is really happening inside

1. The snapshot belongs to the database operation or transaction, not to every future action of the client that received its result.
2. A revision returned with a resource is therefore useful as a precondition for a later update. It connects the update to the state on which the editor actually based the change.
3. A revision supplied by a client is still untrusted input. Validate its type and scope, and authorize the action independently before treating it as a precondition.

■ 25.2.5 TECHNICAL — the engineer's version

1. PostgreSQL Read Committed and Repeatable Read establish and reuse snapshots differently, as detailed in Chapter 23. An ordinary earlier SELECT does not reserve its observed version for a later independent HTTP request. [S66]
2. SQLite WAL readers can retain a snapshot while another connection commits changes. A reader attempting to upgrade a stale snapshot to a writer can encounter `SQLITE_BUSY_SNAPSHOT`; restarting the transaction is different from overwriting the newer state. [S54] [S77]
3. Treat a client edit as (resource identity, expected revision, proposed change), plus separately established authority. Do not use the revision field as a password or authorization token.

■ 25.2.6 WORDS — remember these

1. **Visible version:** the row state allowed by a reader's snapshot — a version satisfying the engine's transaction-visibility rules. **Edit precondition:** the state an update expects to replace — a condition checked at the write boundary to detect intervening changes. **Stale client copy:** an earlier result still held outside the database — data whose original read validity does not guarantee current edit validity.

25.3 Cleanup and long readers

■ 25.3.1 PLAIN — in simple words

1. Keeping every internal version forever would consume growing amounts of storage. Engines need to reclaim versions that no relevant reader or recovery rule still requires.
2. A long-running transaction can delay that cleanup because its older view may still need versions newer readers no longer use.
3. The operational cost is easy to miss: the reader may appear quiet while keeping old storage relevant. A dashboard showing only active query duration can overlook an idle open transaction.

■ 25.3.2 PLAIN — a picture in your head

1. A library keeps an old edition on a return trolley because one registered reader still has a claim to consult it. Hundreds of newer editions cannot be fully cleared while that claim remains open.
2. Closing the reader's session can allow the cleanup process to proceed under its rules.
3. **Where the comparison breaks:** version cleanup is not simply deleting the oldest timestamp. Visibility horizons, replication and engine-specific transaction metadata can affect what is safe to reclaim.

■ 25.3.3 PLAIN — a worked example

1. Suppose a synthetic workload replaces 50,000 rows per hour, each producing 200 bytes of obsolete row-version payload in a simplified accounting model. That is 10,000,000 bytes per hour before indexes, page overhead and reuse.
2. If a long reader prevents relevant cleanup for six hours, the simplified payload accumulation is 60,000,000 bytes, about 60 MB decimal. This is a planning calculation, not a measured PostgreSQL storage prediction.
3. Real storage growth must be measured because row layout, updates, indexes, fill space and vacuum behaviour change the result.
4. The lesson is not that every long reader causes exactly 60 MB of bloat. It is that reader lifetime can become a storage and maintenance input.

■ 25.3.4 PLAIN — what is really happening inside

1. PostgreSQL vacuuming reclaims space from obsolete row versions when it is safe to do so and performs other essential maintenance, including transaction-ID-related work.
2. Ordinary VACUUM commonly makes space reusable inside the table rather than promising that the operating-system file immediately shrinks by the same amount.
3. Cancelling a long reader is an operational decision with consequences. Identify its owner, purpose and restart/recovery behaviour before treating it as disposable.

■ 25.3.5 TECHNICAL — the engineer's version

1. PostgreSQL's routine vacuuming documentation covers dead-row-version cleanup, planner statistics and transaction-ID wraparound protection. Autovacuum is a maintenance system, not a guarantee that every workload remains healthy without monitoring. [S123]
2. Distinguish reusable free space, dead tuples, table-file size and live logical data size. They are related but not interchangeable measurements.

3. Monitor transaction age and relevant maintenance progress under the chosen engine. Avoid prescribing a global timeout or aggressive cleanup command without workload evidence and a reviewed rollback or restart plan.

■ 25.3.6 WORDS — remember these

1. **Obsolete version:** an earlier stored state no longer needed by relevant readers — a version eligible for reclamation under the engine's visibility and maintenance rules. **Vacuuming:** maintenance that reclaims and manages old storage — PostgreSQL's process for handling obsolete tuples and other required maintenance tasks. **Cleanup horizon:** the boundary beyond which old state can be reclaimed — the visibility or retention limit constraining safe version removal.

25.4 Version columns

■ 25.4.1 PLAIN — in simple words

1. A version column gives an editable record a counter or other explicit revision marker. The server advances it whenever a change relevant to the editing contract is accepted.
2. A client sends the revision it originally read. The server compares that expectation with the current revision before applying the proposed replacement.
3. The counter works only when its rules are followed consistently. A maintenance script that changes the record without advancing the revision can make an old client appear current.

■ 25.4.2 PLAIN — a picture in your head

1. A quotation has an edition number printed in its corner. Dev's correction says, "This change is based on edition seven."
2. If Mira has already approved edition eight, the desk does not silently paste Dev's old replacement over it. The difference must be resolved.
3. **Where the comparison breaks:** an edition counter can wrap, reset or be reused after deletion. A complete protocol must define the resource's lifetime as well as its current number.

■ 25.4.3 PLAIN — a worked example

1. D-EDIT starts with quantity=3 and revision=7. Mira submits quantity=4 with expected revision=7. The accepted update sets quantity=4 and revision=8.
2. Dev then submits quantity=5 with expected revision=7. The revision condition no longer matches, so his update affects zero rows and must not be reported as a successful replacement.
3. The response can return a conflict and the current revision, subject to access policy. Preserve Dev's proposed quantity so he can compare and intentionally resubmit against the newer state.
4. This policy rejects stale replacement; it does not decide whether quantity 4 or 5 is the better business choice.

■ 25.4.4 PLAIN — what is really happening inside

1. The revision check and increment belong in the same protected write. Reading the revision first and then performing an unconditional replacement reintroduces the gap.
2. Define which changes advance the revision. A counter for editable quotation fields might intentionally ignore a background "last viewed" timestamp, but that decision must be explicit.

3. Bound the revision domain and handle exhaustion rather than wrapping silently into a previously used value. Large counters make exhaustion unlikely; they do not remove the need for a defined limit.

■ 25.4.5 TECHNICAL — the engineer's version

1. A typical pattern is `UPDATE ... SET quantity=?, revision=revision+1 WHERE id=? AND revision=?`. The uniqueness of `id`, bounded revision domain and affected-row check are part of the protocol.
2. Add a generation identifier when a logical identifier can be deleted and recreated. Compare (`resource_id`, `generation_id`, `revision`) so a new object's revision 1 is not confused with the old object's revision 1.
3. Do not substitute PostgreSQL `xmin` or `ctid` casually for this application contract. Their implementation purpose and lifecycle differ from a deliberately stable business revision. [S124] [S123]

■ 25.4.6 WORDS — remember these

1. **Version column:** a field naming the accepted edit state — an application-managed revision value checked and advanced by relevant writes. **Generation identifier:** distinguish separate lifetimes of a reused key — an immutable identity component that changes when a resource is recreated. **Revision exhaustion:** no unused value remains in the chosen domain — a boundary requiring an explicit policy rather than silent counter reuse.

25.5 Compare-and-set

■ 25.5.1 PLAIN — in simple words

1. Compare-and-set means “make this change only if the current state still matches my expectation.” It turns an old observation into an explicit precondition instead of an unconditional command.
2. Comparing only the displayed value may miss intervening changes that return to the same value. A revision can reveal that history of accepted edits even when the final quantity looks unchanged.
3. A successful comparison still does not authenticate the caller, validate every business rule or identify a repeated request. Those are separate controls.

■ 25.5.2 PLAIN — a picture in your head

1. Dev leaves a note saying, “Change the sign only if it is still the edition I saw.” That is stronger than saying, “Change it if it still happens to display the same number.”
2. The sign could have changed twice and returned to that number while the edition advanced.
3. **Where the comparison breaks:** a counter records only changes that the protocol includes. It is not an omniscient record of every physical event or bypass mutation.

■ 25.5.3 PLAIN — a worked example

1. Start with `quantity=3`, `revision=7`. An accepted edit changes it to `quantity=4`, `revision=8`. Another accepted edit changes it back to `quantity=3`, `revision=9`.
2. A stale client expecting `quantity=3` passes a value-only comparison even though two edits intervened. This is the ABA pattern: A became B and later became A again.
3. A stale client expecting `revision=7` fails against `revision=9`. The explicit revision detected intervening protocol-respecting edits.

4. If the record was deleted and recreated with revision=7 again, the revision alone could mislead the client. A fresh generation identifier separates the lifetimes.

■ 25.5.4 PLAIN — what is really happening inside

1. The write predicate encodes the expectation; the database decides whether the current target satisfies it at the protected write boundary.
2. On failure, avoid guessing the reason from zero affected rows alone. Missing identity, different generation, stale revision or a scope condition can all make the predicate fail.
3. After a successful write whose response is lost, replaying the old expected revision will usually conflict. That conflict does not prove the original attempt failed. Stable request identity is needed to resolve the original outcome.

■ 25.5.5 TECHNICAL — the engineer's version

1. The educational compare-and-set helper uses bound values, a complete identity, a positive bounded proposed quantity and an expected revision. It owns a transaction and accepts exactly one changed row under its schema.
2. The protocol protects stale replacement of that resource. It does not enforce unrelated cross-row limits or prevent authorized users from intentionally making undesirable but permitted changes.
3. Separate optimistic concurrency from idempotency: the former detects conflicting edit state; the latter identifies repeated attempts at the same intent and resolves their outcomes. Chapter 26 combines these concerns where appropriate. [S126]

■ 25.5.6 WORDS — remember these

1. **Compare-and-set:** write only when the expectation still matches — an atomic conditional mutation based on an expected value, revision or generation. **ABA problem:** a value changes and returns, hiding intervening work — a limitation of equality-only checks that do not identify the relevant state history. **Optimistic concurrency:** detect a conflict when saving rather than holding a long edit lock — a protocol that validates an earlier observation at the write boundary.

25.6 Choosing a conflict policy

■ 25.6.1 PLAIN — in simple words

1. Detecting a conflict is not the end of the user experience. The application must decide whether to reject, ask for review, merge safely or retry a newly computed operation.
2. Different data needs different treatment. Two independent additions to a counter can sometimes combine naturally; two replacements of a delivery address may require a person to choose.
3. “Last write wins” is a policy that discards something when changes compete. It can be appropriate for selected low-risk data, but it should never be mistaken for the absence of data loss.

■ 25.6.2 PLAIN — a picture in your head

1. Two editors correct different spelling errors in a document; a careful merge may keep both. Two editors choose different delivery destinations; combining the street from one and the city from the other can create a destination neither intended.
2. The merge rule must understand the meaning of the fields, not just avoid a software error.
3. **Where the comparison breaks:** automatic text merging can still produce a syntactically valid but semantically wrong result. Business invariants need explicit validation after any merge.

■ 25.6.3 PLAIN — a worked example

1. Mira changes a draft quantity from 3 to 4. Dev changes the same draft's delivery note, both from revision 7.
2. A whole-record revision policy rejects Dev's stale update even though the fields differ. This is conservative and simple, but may create avoidable review work.
3. A field-aware merge can preserve both only if the application checks that the changes are independent under its rules. If the note says "pack exactly three," it depends on quantity and is not independent after all.
4. Record the chosen resolution and resulting revision. Do not hide an automatic overwrite behind a generic "synchronised" message.

■ 25.6.4 PLAIN — what is really happening inside

1. Conflict granularity is a design trade-off. One revision per document is easier to reason about; finer-grained revisions may reduce false conflicts while increasing metadata and invariant complexity.
2. A retry of a commutative operation, such as adding a separately identified increment, differs from retrying an absolute replacement. Duplicate delivery still needs its own identity handling.
3. Preserve the user's unsaved proposal and show the relevant current state only within their permissions. Conflict messages must not become a way to reveal another tenant's records.

■ 25.6.5 TECHNICAL — the engineer's version

1. Specify the conflict response schema, retry eligibility, merge rules, audit requirements and authorization checks. A version mismatch should have a stable API meaning rather than being flattened into an unexplained server error.
2. Tests should cover two stale editors, ABA, identity recreation, bypass-writer assumptions, response loss, malformed revision and unauthorized scope. The provided lab exercises a trusted local data protocol, not a complete authenticated API.
3. Evaluate conflict rate and user resolution cost alongside throughput. Reducing visible conflicts by silently overwriting edits is not a correctness improvement.

■ 25.6.6 WORDS — remember these

1. **Conflict resolution:** decide what to retain after incompatible changes — a policy for rejection, reviewed replacement, safe merge or recomputation. **Merge granularity:** the unit at which changes are compared — a document, row, field or operation boundary used by the conflict protocol. **Last-write-wins policy:** accept the selected later replacement — a conflict rule that can discard competing state and requires a defined ordering basis.

25.97 Practice and worked answers

1. **Separate version purposes.** Can internal MVCC versions replace an approval history? **Answer:** no. Cleanup and implementation lifecycles differ from a deliberate business-history and retention contract.
2. **Name a stable key.** Should PostgreSQL ctid identify a customer's permanent account? **Answer:** no. It identifies a physical row-version location, not an immutable business identity.
3. **Trace an edit conflict.** Quantity 3/revision 7 becomes 4/revision 8. A client still expects revision 7. **Answer:** the conditional replacement should fail without overwriting revision 8.

4. **Find ABA.** Quantity changes $3 \rightarrow 4 \rightarrow 3$. **Answer:** a quantity-only comparison can miss both edits. A consistently advanced revision distinguishes the states.
5. **Find identity reuse.** A deleted key is recreated with the old revision number. **Answer:** include a new generation identifier or otherwise prohibit ambiguous lifetime reuse.
6. **Interpret a lost response.** A saved edit advances revision 7 to 8, but the caller receives no reply. **Answer:** retrying expected revision 7 can conflict even though the original succeeded. Resolve request identity rather than treating the conflict as proof of original failure.
7. **Calculate simplified old-version payload.** Fifty thousand changes/hour at 200 bytes for six hours. **Answer:** 60,000,000 bytes before overhead and reuse; this is not a measured engine-size forecast.
8. **Review a merge.** One user changes quantity and another changes a note saying “pack exactly three.” **Answer:** the fields have a semantic dependency. A blind field merge can violate the intended meaning.

25.98 Common wrong ideas

1. **Wrong:** MVCC means every historical row is kept forever. **Right:** internal versions are subject to visibility and cleanup rules.
2. **Wrong:** a physical row location is a permanent record identity. **Right:** storage location and business identity serve different purposes.
3. **Wrong:** an earlier valid read authorizes a later overwrite. **Right:** later writes need current preconditions and independent authority checks.
4. **Wrong:** VACUUM always shrinks the table file immediately. **Right:** ordinary maintenance often makes internal space reusable instead.
5. **Wrong:** comparing the current value detects every intervening edit. **Right:** ABA can return to the same value.
6. **Wrong:** a revision counter works even when some writers ignore it. **Right:** every relevant mutation must follow the revision contract.
7. **Wrong:** optimistic concurrency is idempotency. **Right:** stale-state detection and repeated-intent resolution solve different problems.
8. **Wrong:** automatic merging is always more correct than rejecting. **Right:** the merge must preserve the meaning and invariants of the data.

25.99 Chapter summary in 20 lines

1. Internal row versions, business revisions and client copies are different concepts.
2. MVCC uses versions and visibility rules to coordinate readers and writers.
3. An older visible version can be valid under a retained snapshot.
4. Internal version retention is not an everlasting business audit log.
5. Physical tuple locations are not permanent logical identities.
6. A client’s earlier read does not reserve the record for a later edit.
7. Return explicit revisions when later writes need stale-state detection.
8. Cleanup reclaims obsolete versions when relevant visibility rules permit it.
9. Long readers can delay reclamation and affect maintenance cost.
10. Reusable space and operating-system file size are different measurements.
11. A version column requires a defined increment and exhaustion policy.

12. Every relevant writer must follow the same revision contract.
13. Compare the expected revision and mutate the record in one protected operation.
14. Affected-row failure must not be reported as successful replacement.
15. ABA can hide intervening changes from value-only comparisons.
16. Generation identifiers distinguish separate lifetimes of a reused key.
17. A lost response can make a successful original edit appear as a later conflict.
18. Request identity resolves retry outcomes separately from edit concurrency.
19. Choose rejection, review or merge according to the data's meaning.
20. Preserve user proposals and validate invariants after conflict resolution.

Invariants, Idempotency and the Outbox

26.0 What this chapter gives you

1. A reliable operation must survive more than a clean first attempt. Requests can arrive twice, replies can disappear, and external delivery can fail after the database has committed.
2. This chapter combines three ideas: an invariant defining what must stay true, an identity defining which attempts represent one intent, and an outbox recording external work implied by an accepted local change.
3. The resulting pattern has precise boundaries. It can make a bounded local reservation protocol repeatable; it does not establish universal exactly-once execution, physical-stock accuracy, authentication or fault-free external delivery.

26.1 Rules across operations

■ 26.1.1 PLAIN — in simple words

1. An invariant is a rule that must hold across the relevant accepted operations. “Available stock never becomes negative” is one possible rule. “Every accepted reservation has a matching stock allocation” is another.
2. A workflow can satisfy one while breaking the other. Stock can remain non-negative even when the application forgets to record who reserved it.
3. Write the rules before selecting the mechanism. A transaction, a unique key and a queue each protect different boundaries; none is a general replacement for understanding the intended state changes.

■ 26.1.2 PLAIN — a picture in your head

1. Mira keeps a stock ledger, a reservation register and a dispatch list. A useful rule connects them: every accepted reservation explains a stock reduction and creates the required confirmation task.
2. Checking each notebook separately is not enough. Their relationships are part of the work.
3. **Where the comparison breaks:** the notebooks can represent promises about records, but actual goods and delivered messages remain outside the local database. A consistent ledger is evidence about the ledger first.

■ 26.1.3 PLAIN — a worked example

1. In a new local teaching system, available stock begins at 5. Request IDEM-A asks for 3. Acceptance should leave available=2, one accepted request outcome for 3, one reservation and one outbox message identity.

2. A repeated delivery of IDEM-A should return its recorded outcome without reducing stock again or creating a second confirmation intent.
3. A different request IDEM-B asking for 3 should be rejected while only 2 remain. A duplicate of IDEM-B should follow the explicitly chosen rejection-replay policy rather than unexpectedly becoming accepted later.
4. These example quantities do not rewrite the canonical agreed orders or explain the earlier unknown stock discrepancy.

■ 26.1.4 PLAIN — what is really happening inside

1. The protocol connects request identity to the state transition it authorizes. Unique constraints help prevent two concurrent records claiming the same identity, while a transaction groups the related local writes.
2. The code must handle accepted, rejected and exceptional outcomes distinctly. A deliberate business rejection can be recorded; an unexpected exception should not accidentally create a successful-looking outcome.
3. Cancellation, replenishment and correction need their own rules. Adding idempotency to one endpoint does not prove that every other mutation preserves the same invariant.

■ 26.1.5 TECHNICAL — the engineer's version

1. Define the operation as a state transition over (stock, requests, reservations, outbox). State the preconditions and which fields may change for accepted, rejected and replayed operations.
2. In the educational implementation, a single owned SQLite write transaction coordinates the guarded stock update and the relevant request/outbox records. SQLite writer serialization is part of this local implementation, not a claim about a distributed service. [S25] [S54]
3. Add reconciliation identities, such as initial stock plus replenishments minus accepted reservations plus valid releases equals expected available stock, under explicitly bounded record classes. Unrecorded physical loss remains outside that equation's evidence.

■ 26.1.6 WORDS — remember these

1. **Invariant:** a rule accepted operations must preserve — a predicate over the relevant system state that the protocol is designed to maintain. **State transition:** one defined move from an old state to a new state — an operation with explicit preconditions, writes and outcomes. **Reconciliation identity:** an equation connecting related records — a check that quantities and recorded state changes agree under stated inclusions and exclusions.

26.2 Request identity

■ 26.2.1 PLAIN — in simple words

1. A retry needs a stable name for the original intent. Without that name, the server may be unable to distinguish “try that same reservation again” from “make another identical reservation.”
2. Two equal payloads are not automatically the same intent. A customer can legitimately place two separate orders for the same product and quantity.
3. Scope the name. The same local request string used by two different tenants should not make their operations collide or reveal one tenant's outcome to another.

■ 26.2.2 PLAIN — a picture in your head

1. A customer receives a claim ticket for one request. Presenting the same ticket asks about the same decision; receiving a new ticket creates a separately identified request.
2. The ticket belongs to a particular desk and customer scope. Ticket 17 at another shop is not necessarily your ticket 17.
3. **Where the comparison breaks:** a request key is not automatically a secret or proof of ownership. Authorization must be established independently before the server reveals or changes anything associated with it.

■ 26.2.3 PLAIN — a worked example

1. Tenant T-A submits key REQ-17 for branch BR-A, product P-DEMO and quantity 3. A network retry repeats the same scoped key and the same validated meaning.
2. A new purchase with the same branch, product and quantity uses a new intent identity, such as REQ-18. Deduplicating solely by a hash of the payload would wrongly collapse these legitimate separate purchases.
3. Tenant T-B can independently use REQ-17 under a key scoped by tenant. A request outcome lookup must include the tenant established by the trusted service boundary, not merely a client-supplied string.
4. The companion's local function accepts a trusted tenant parameter for teaching; it does not implement that authentication boundary.

■ 26.2.4 PLAIN — what is really happening inside

1. Validate the input into a well-defined semantic representation before comparing attempts. Decide whether whitespace, field order or equivalent numeric spelling changes meaning under the API contract.
2. Store enough validated request meaning to detect incompatible reuse. A digest can help indexing or comparison, but a collision-prone shortcut must not replace the equivalence rule.
3. The server must define who generates keys, their allowed length, their retention and whether a key can ever be reused. An unlimited arbitrary string is also an input-resource concern.

■ 26.2.5 TECHNICAL — the engineer's version

1. Model identity as a tuple such as (tenant_id, operation_kind, request_key). Include additional scope only when required by the contract, and enforce the tuple's uniqueness in storage.
2. Idempotent APIs commonly distinguish caller-provided request intent from merely equal parameters. The equivalence and key-retention contract is central to safe retries. [S126]
3. Canonicalization is not generic "sort JSON and hash." It must preserve domain meaning, reject ambiguous duplicate members where relevant, represent units explicitly and avoid silently equating values that the application treats differently. Chapters 3, 5 and 7 supply the prerequisites.

■ 26.2.6 WORDS — remember these

1. **Request identity:** the stable name of one intended operation — a scoped identifier linking repeated attempts to one business decision. **Payload equivalence:** when two attempts mean the same thing — an explicit comparison contract over validated request semantics. **Identity scope:** the domain in which a key is unique — the tenant, operation and other components defining a request key's namespace.

26.3 Replay detection

■ 26.3.1 PLAIN — in simple words

1. When a known request returns, the server checks whether it represents the same intent and then returns the recorded outcome. It does not perform the business change again merely because the caller asked again.
2. Reusing the same key with different meaning is a conflict, not a request to rewrite history. The server should reject the incompatible reuse under a clear contract.
3. Decide what happens to recorded rejections. In this book's local protocol, a valid business rejection is stable for that request identity. A later change in stock does not silently change the old decision.

■ 26.3.2 PLAIN — a picture in your head

1. A reservation desk keeps the decision attached to the claim ticket. Returning with the ticket retrieves that decision.
2. Changing the requested quantity while presenting the same ticket is not merely asking again. It is attempting to attach a different request to an existing identity.
3. **Where the comparison breaks:** some real APIs deliberately choose other policies, including retention limits or selected retryable rejections. Those choices must be documented; this chapter's policy is not a universal standard.

■ 26.3.3 PLAIN — a worked example

1. IDEM-A with quantity 3 is accepted from stock 5, leaving 2. Replaying IDEM-A with quantity 3 returns the same accepted outcome and leaves stock 2.
2. IDEM-A with quantity 4 conflicts with the stored request meaning. It must neither allocate a fourth unit nor overwrite the original record.
3. IDEM-B with quantity 3 is rejected for insufficient stock. Suppose a separate replenishment later raises available stock to 7. Replaying IDEM-B still returns its original rejection under the chosen contract.
4. A caller who now intends a new allocation submits a new request identity. That is an explicit new decision, not an automatic retry that changes the meaning of the old outcome.

■ 26.3.4 PLAIN — what is really happening inside

1. Replay detection must occur within a protocol that coordinates competing first attempts. A lookup followed by an unprotected insertion has the same check-then-insert race discussed in Chapter 22.
2. Storing an outcome separately after the stock commit leaves a gap: a replay could see no outcome even though stock was already reduced. Group the outcome and business mutation atomically.
3. Do not return another scope's result to resolve a conflict. Identity comparison, authorization and response filtering all remain relevant on the replay path.

■ 26.3.5 TECHNICAL — the engineer's version

1. The local example persists both accepted and valid business-rejected outcomes, with normalized payload fields. A unique request key and an owned transaction prevent inconsistent duplicate outcome records within the tested database protocol.

2. Unexpected exceptions roll back that attempt's local changes. They are not converted into stable success or an invented business rejection. An externally unknown commit outcome is resolved by querying the same request identity.
3. Retention bounds the deduplication horizon. Deleting request records while late retries remain possible can allow a formerly recognized intent to be treated as new. State that limit explicitly. [S126]

■ 26.3.6 WORDS — remember these

1. **Replay:** another attempt carrying the same defined intent — a repeated request that should resolve through its existing identity and outcome. **Incompatible key reuse:** one identity is presented with different meaning — a conflict requiring rejection or explicit reconciliation, not silent replacement. **Deduplication horizon:** how long repeated intent remains recognizable — the retention interval and protocol conditions under which replay records are available.

26.4 Atomic intent recording

■ 26.4.1 PLAIN — in simple words

1. The reservation, its request outcome and its required message intent should be accepted together when the protocol requires all three.
2. This does not mean the message is already delivered. It means that a committed reservation has a durable local record of the delivery work that follows from it.
3. The order of application steps matters. A failure at any pre-commit point must not leave an accepted half-operation in the database.

■ 26.4.2 PLAIN — a picture in your head

1. Mira accepts a reservation only when its stock entry, decision slip and dispatch instruction are filed as one case.
2. The messenger may arrive later, but the instruction no longer depends on Mira remembering it after a crash or interruption.
3. **Where the comparison breaks:** an in-memory educational database does not demonstrate physical durability. The protocol's atomic grouping can be tested locally while durable storage assumptions remain a separate chapter's responsibility.

■ 26.4.3 PLAIN — a worked example

1. Begin an owned transaction. Look up the scoped request key. If it exists, compare meaning and return its recorded outcome without another allocation.
2. For a new valid request, attempt the guarded stock reduction. If accepted, insert the reservation, accepted outcome and one outbox row. If rejected under the business contract, insert the stable rejected outcome without a reservation or accepted-confirmation intent.
3. Commit only after every required local write succeeds. Inject an exception after stock reduction and verify that rollback removes all changes from that attempt.
4. The transaction is not allowed to call an email provider before commit. It records what should be sent, not an external send that rollback cannot undo.

■ 26.4.4 PLAIN — what is really happening inside

1. An outbox record needs a stable message identity, type, relevant payload or durable reference, and enough state to manage delivery attempts under the chosen design.
2. Payload design has retention consequences. Copying an entire customer profile into every message makes later correction or deletion more complicated than sending a bounded required subset.
3. A worker that reads the outbox also needs a coordination protocol if several workers can claim the same work. This book's simplest local dispatcher is deliberately single-worker; it does not imply a production lease implementation.

■ 26.4.5 TECHNICAL — the engineer's version

1. The transactional outbox places business state and delivery intent in the same local atomic boundary, avoiding the basic dual-write gap between a database commit and an unrelated message send. [S125]
2. Use constraints to connect outbox identity and request identity as required. Do not regenerate a new message identifier on every delivery attempt; that defeats consumer deduplication by message identity.
3. For multiple dispatchers, specify claiming, lease expiry, crash recovery, ordering and fencing where needed. Merely adding a `sending=true` flag can strand work permanently after a worker dies.

■ 26.4.6 WORDS — remember these

1. **Atomic intent recording:** accept the business change and its follow-up instruction together — committing state and outbox intent in one local transaction. **Dual-write gap:** one system changes while another does not — a failure window between separately committed operations in different resources. **Dispatcher:** the worker attempting recorded external tasks — a process that reads delivery intent and manages sends and acknowledgements.

26.5 Outbox delivery and consumers

■ 26.5.1 PLAIN — in simple words

1. A dispatcher can send the same message more than once if it loses certainty about a previous attempt. The receiver must know what repeated delivery means.
2. A database-backed consumer can record “I processed message M” together with its local database effect. Replaying M then finds the record and avoids applying that effect again.
3. This protects the consumer's participating local database work. It does not make a separate external effect, such as sending a physical parcel, part of that database transaction.

■ 26.5.2 PLAIN — a picture in your head

1. The receiving desk stamps a message identity into a register at the same time it files the corresponding local instruction. A second copy with the same identity finds the stamp.
2. If the desk crashes before accepting either, it can safely try again. If it accepted both, it can recognize the replay.
3. **Where the comparison breaks:** a stamp made before an external action can falsely suggest completion, while a stamp made after it can leave a duplication gap. The atomic boundary must contain the actual effect you claim to deduplicate.

■ 26.5.3 PLAIN — a worked example

1. Outbox message MSG-A represents IDEM-A's accepted reservation. The dispatcher sends it, and the consumer commits an inbox record plus a local notification-task record.
2. Before the dispatcher marks delivery complete, it crashes. On restart it sends MSG-A again.
3. The consumer finds MSG-A in its inbox with equivalent meaning and returns the recorded result without creating another notification task. There are two deliveries but one committed local task.
4. If that task later sends an email, email delivery remains another boundary. The local inbox result does not prove that exactly one email reached a person.

■ 26.5.4 PLAIN — what is really happening inside

1. Inbox identity and effect must share the consumer's transaction. Marking a message processed first and performing the effect later can lose work; performing the effect first and marking later can duplicate it.
2. A duplicate identity carrying different payload is suspicious or incompatible under the protocol. Do not silently treat it as an equivalent replay simply because the identifier matches.
3. Order is separate from duplication. A consumer may need per-order sequence checks so "cancel reservation" does not apply before the reservation it references. A global message ID alone does not establish causal order.

■ 26.5.5 TECHNICAL — the engineer's version

1. Consumer-side deduplication typically uses a unique inbox key and a transaction containing both the inbox decision and the local effect. Its horizon is bounded by retained identity records and participating storage guarantees. [S125]
2. An acknowledgement should correspond to the consumer's defined accepted boundary. A queue acknowledgement sent before the local effect commits can lose work if the consumer then fails.
3. Model delivery as at-least-once attempts with possible unknown outcomes unless the actual infrastructure contract establishes something else. "Exactly once" must name the effect, identity scope, storage boundary and retention interval it describes.

■ 26.5.6 WORDS — remember these

1. **Inbox record:** remember which message identity was accepted — a consumer-side deduplication record committed with the participating local effect. **At-least-once delivery:** retries may deliver a message repeatedly — a delivery contract that prioritizes eventual attempts while permitting duplicates under its stated assumptions. **Acknowledgement boundary:** what a positive receipt actually confirms — the defined acceptance point represented by a queue, consumer or provider response.

26.6 End-to-end limits

■ 26.6.1 PLAIN — in simple words

1. The complete journey contains several boundaries: caller intent, database acceptance, outbox dispatch, consumer acceptance and any later real-world action.
2. A local guarantee should be described at its own boundary. Extending it verbally to the whole journey does not make the missing coordination appear.

3. Strong engineering names the remaining uncertainty and provides recovery procedures. It does not hide uncertainty behind a confident success label.

■ 26.6.2 PLAIN — a picture in your head

1. A parcel has separate records for booking, warehouse acceptance, courier collection and recipient delivery. A booking receipt is useful, but it is not proof that the parcel reached its destination.
2. Each handover needs a defined meaning and a way to investigate missing evidence.
3. **Where the comparison breaks:** this book's reservation lab does not implement a logistics system. The parcel story illustrates boundaries, not a tested physical delivery protocol.

■ 26.6.3 PLAIN — a worked example

1. IDEM-A commits locally, but the response is lost. The caller retries the same identity and learns the accepted outcome. No second allocation occurs under the tested local protocol.
2. MSG-A is delivered twice, but the consumer's inbox and local task transaction apply one local task. The dispatcher eventually records its acknowledgement.
3. A later email provider returns an ambiguous timeout. Without a suitable provider identity contract or a reconciliation interface, the system cannot infer from that timeout whether the email was accepted.
4. The honest status separates "reservation accepted," "notification task recorded," "provider outcome unresolved" and "delivery verified," rather than collapsing all four into one invented certainty.

■ 26.6.4 PLAIN — what is really happening inside

1. Retention, permissions and failure domains constrain the protocol. Erasing an inbox record can remove replay protection; losing both the business database and its only backup can remove outcome evidence.
2. Replays must remain authorized. Knowing an old request key does not grant permanent access after a user's rights change.
3. Operational recovery needs bounded procedures: identify stuck intents, verify payload identity, inspect consumer state, decide safe replay scope and preserve the evidence of the decision.

■ 26.6.5 TECHNICAL — the engineer's version

1. The supplied implementation is a local educational state machine. It uses synthetic records and trusted function parameters; it is not a multi-process lease service, authenticated API, immutable audit store or proof of durable power-loss behaviour.
2. Test invariants after every injected failure point, replay and incompatible key reuse. Include stable rejection replay, inbox payload mismatch, tenant scope and stock reconciliation. A passing test means its stated behaviour occurred, not that unseen external systems were verified.
3. Before production, separately review transaction isolation, key retention, worker coordination, external provider contracts, privacy, monitoring and restore evidence. Idempotency is a protocol property with assumptions, not a decorative endpoint label. [S126]

■ 26.6.6 WORDS — remember these

1. **End-to-end boundary:** the complete chain whose result is claimed — all participating systems and effects required for a stated business outcome. **Outcome reconciliation:** resolve uncertainty

using recorded identity and state — a procedure for determining what a prior attempt actually accepted. **Scoped guarantee:** a promise with explicit limits — a correctness claim tied to defined effects, participants, failures and retention assumptions.

26.97 Practice and worked answers

1. **Name the invariant.** Stock falls from 5 to 2, but no reservation exists. **Answer:** non-negative stock holds, but the required allocation-to-reservation relationship fails. One invariant does not imply the other.
2. **Separate equal requests.** A customer intentionally buys the same three pens twice. **Answer:** equal payloads can represent two legitimate intents. Give the intents separate request identities.
3. **Replay acceptance.** IDEM-A for 3 is accepted once and delivered again. **Answer:** return its stored outcome without a second subtraction or a new outbox identity.
4. **Reject incompatible reuse.** IDEM-A returns with quantity 4. **Answer:** reject the mismatched meaning under the protocol; do not replace the original request record.
5. **Replay rejection.** IDEM-B was rejected, then stock was replenished. **Answer:** under this book's explicit policy, the old identity retains its rejection. A genuinely new allocation request uses a new intent identity.
6. **Inject a local failure.** An exception occurs after stock reduction but before outbox insertion. **Answer:** the owned transaction rolls back all of that attempt's local changes.
7. **Trace duplicate delivery.** MSG-A is sent twice after a dispatcher crash. **Answer:** an equivalent inbox identity and local effect in one consumer transaction can preserve one local task despite two deliveries.
8. **Limit the claim.** Does one local task prove exactly one delivered email? **Answer:** no. The provider and recipient boundaries require separate contracts and evidence.

26.98 Common wrong ideas

1. **Wrong:** a transaction automatically knows the business invariant. **Right:** the application must define and protect the relevant relationships.
2. **Wrong:** equal payload hashes always mean one intent. **Right:** identical purchases can be legitimate separate requests.
3. **Wrong:** a request key is an authorization token. **Right:** scope and permission checks remain independent.
4. **Wrong:** a duplicate key with different data should overwrite the old request. **Right:** incompatible identity reuse requires an explicit conflict policy.
5. **Wrong:** every rejected request should become successful when retried later. **Right:** replay semantics must be stable under the chosen contract.
6. **Wrong:** an outbox means the message has been delivered. **Right:** it records durable local intent within the participating storage boundary.
7. **Wrong:** a consumer inbox deduplicates arbitrary outside effects. **Right:** it protects only effects inside the same proven atomic boundary.
8. **Wrong:** exactly once is meaningful without naming the effect and scope. **Right:** define identity, participants, failure assumptions and retention horizon.

26.99 Chapter summary in 20 lines

1. Invariants define what accepted state transitions must preserve.
2. Related stock, reservation, outcome and message records need explicit relationships.
3. A retry must carry the stable identity of the original intent.
4. Equal payloads can represent different legitimate intents.
5. Scope request identity by the required tenant and operation namespace.
6. Validate and compare request meaning under a documented equivalence contract.
7. Matching identity with different meaning is a conflict, not an ordinary replay.
8. Store the business outcome with the local state change atomically.
9. A replay returns the recorded outcome without performing the allocation again.
10. This book's valid business rejections remain stable for their original request identity.
11. Unexpected pre-commit exceptions roll back the attempt's local changes.
12. A lost response after commit can be resolved through the same request identity.
13. An outbox records external work implied by an accepted local change.
14. Outbox intent is not evidence of completed external delivery.
15. Dispatch can duplicate messages after an ambiguous send or lost acknowledgement.
16. An inbox can deduplicate participating local consumer effects in one transaction.
17. Message identity does not by itself guarantee causal ordering.
18. Retention bounds how long replay protection and outcome evidence remain available.
19. Authorization and privacy checks apply to replays as well as first attempts.
20. State every guarantee at its actual boundary and preserve unresolved outcomes honestly.



PART E

Storage and Recovery

Trace a write into storage and learn how to recover after failures
without inventing certainty.

Pages, Buffers and the Storage Engine

27.0 What this chapter gives you

1. SQL describes logical records, but a storage engine moves and manages physical units. This chapter connects rows to pages, memory buffers, free space and the work triggered by a small update.
2. You will estimate page occupancy under stated assumptions, distinguish a database cache hit from a device read, and inspect storage metadata without pretending a toy page is a real PostgreSQL or SQLite file.
3. The arithmetic layouts below are teaching designs. Product-specific sizes and behaviours are labelled separately, and the labs operate only on synthetic data.

27.1 Logical rows and physical pages

■ 27.1.1 PLAIN — in simple words

1. A row is a logical record. A page is a physical storage unit used by many database engines to organize records and indexes.
2. Several small rows can share a page. A large value may require an overflow or separate-storage mechanism. The relationship is not simply one row equals one file or one disk sector.
3. Reading one requested field can involve bringing a larger unit into memory. That is one reason data layout and access patterns affect performance even when the SQL result is tiny.

■ 27.1.2 PLAIN — a picture in your head

1. Think of records as individual forms and pages as folders that carry several forms together. Fetching one form may require retrieving its folder.
2. A folder also has labels and indexing information, leaving less than its full capacity for form contents.
3. **Where the comparison breaks:** database pages have exact binary layouts, and the device can have different physical transfer units. A folder analogy does not define page atomicity or hardware write guarantees.

■ 27.1.3 PLAIN — a worked example

1. Design a toy page of 4,096 bytes with a 64-byte header. Each stored row uses 96 payload bytes plus a 4-byte slot entry.
2. Available space is $4,096 - 64 = 4,032$ bytes. At 100 bytes per row-plus-slot, the page holds at most 40 such rows, leaving 32 bytes unused under this simplified layout.
3. One thousand rows therefore need at least 25 data pages if all pages achieve that occupancy. Real engines may use more because of metadata, free space, variable-length values and update history.

4. These are not SQLite's or PostgreSQL's actual tuple-overhead figures. They are a transparent model whose assumptions can be changed and recalculated.

■ 27.1.4 PLAIN — what is really happening inside

1. A page can contain a header, an array of item references, free space and record bytes. The engine interprets those bytes using its file-format and schema information.
2. An index page may organize keys and child pointers rather than business rows. A query can visit both index pages and table pages to produce one result.
3. Physical storage can change while logical identity stays the same. Compaction, updates and maintenance should not force the application to identify customers by their current byte offsets.

■ 27.1.5 TECHNICAL — the engineer's version

1. PostgreSQL's standard heap layout commonly uses 8 kB pages, with a compile-time page-size choice. The documented page header is 24 bytes and item identifiers use 4 bytes each. These are implementation facts, not universal database constants. [S128]
2. SQLite's database page size is a power of two between 512 and 65,536 bytes under its file format. Header encoding treats the value 1 specially to represent 65,536. [S133]
3. Separate logical key, page identifier, slot reference and device block address in a design. Each names a different layer and may have different stability guarantees.

■ 27.1.6 WORDS — remember these

1. **Database page:** a unit in the engine's physical layout — a fixed-size storage block containing records, index entries or other engine structures. **Slot entry:** a small reference locating an item within a page — metadata mapping an item identifier to its stored position and length. **Page occupancy:** how much of a page is usefully filled — the proportion consumed by payload and required structures under a stated layout.

27.2 Buffer pools

■ 27.2.1 PLAIN — in simple words

1. A buffer pool keeps database pages in memory so repeated work can avoid fetching them again from lower storage layers.
2. Memory is limited. The engine must decide which pages to retain, which to replace and how to coordinate pages currently being used or changed.
3. A database-cache miss does not necessarily mean a physical device read. The operating system may already have a copy in its own cache.

■ 27.2.2 PLAIN — a picture in your head

1. Mira keeps frequently used folders on her desk and retrieves others from a cabinet. A desk hit is faster than a cabinet trip.
2. The cabinet assistant may also have a trolley of recently fetched folders. Missing from Mira's desk does not mean the folder must travel all the way from the distant archive.
3. **Where the comparison breaks:** the layers contain copies and coordination metadata, not one movable physical folder. Cache coherence, pinning and dirty-state handling are more exact than ordinary desk tidying.

■ 27.2.3 PLAIN — a worked example

1. A toy buffer pool has three slots and uses least-recently-used replacement. Read page IDs 1, 2, 3, 1, 4, 2.
2. The first three reads miss and fill the pool. Reading 1 hits and makes it most recent. Reading 4 evicts 2; the later read of 2 misses again.
3. The model records one hit and five misses. It demonstrates a replacement policy, not the policy or device-read count of a named engine.
4. A different access sequence or replacement algorithm can change the result. A single global hit ratio cannot tell you which important request suffered the expensive miss.

■ 27.2.4 PLAIN — what is really happening inside

1. The engine locates the requested page in its buffer mapping or chooses a buffer to load it into. It coordinates concurrent access and prevents a page from being evicted while an operation still requires it.
2. A dirty page cannot be discarded as though it were merely an unchanged cached copy. Its modifications need the engine's recovery and flushing protocol.
3. Large scans can compete with frequently reused pages for cache capacity. Engines may use specialized strategies to avoid letting one scan disrupt every other workload.

■ 27.2.5 TECHNICAL — the engineer's version

1. PostgreSQL I/O statistics distinguish operations at the database/kernel boundary but do not by themselves distinguish actual device reads from data already in the kernel's page cache. Combine them with appropriate operating-system evidence. [S144]
2. Cache metrics require a scope and interval. A counter since process start can hide a recent regression; ratios aggregated across workloads can hide a slow critical query.
3. The companion LRU exercise is a deterministic model with no real I/O, locks or dirty-page write-back. Do not use its hit count as a measured PostgreSQL buffer-pool result.

■ 27.2.6 WORDS — remember these

1. **Buffer pool:** the database's in-memory page workspace — a managed cache of pages and metadata used by the storage engine. **Cache hit:** the needed item is already in the named cache — an access resolved at a specified cache layer. **Eviction:** remove a cached item to make room — replacement subject to usage, dirty-state and recovery constraints.

27.3 Row layout

■ 27.3.1 PLAIN — in simple words

1. A stored row contains more than the visible values returned by SELECT. The engine needs information about field boundaries, missing values, versions and other implementation details.
2. Variable-length text cannot always be located by multiplying a column number by a fixed width. Earlier fields, alignment and length information can affect where it begins.
3. Large values may be compressed or stored elsewhere with a reference in the main row. Reading only small fields can therefore avoid some large-value work, depending on the engine and query.

■ 27.3.2 PLAIN — a picture in your head

1. An application form has boxes of different sizes and a note saying that the attached document is in another folder. The visible answers are not the whole filing structure.
2. A short reference can stand in for a large attachment until someone actually needs it.
3. **Where the comparison breaks:** binary length headers, alignment and compression are not hand-written annotations. Reading raw bytes safely requires the exact format and bounds, not guess-work based on how a screen displays the row.

■ 27.3.3 PLAIN — a worked example

1. Define a toy record with an 8-byte header, a 4-byte ID, a 2-byte text length and five UTF-8 bytes of text. Without alignment or other fields, its size is $8 + 4 + 2 + 5 = 19$ bytes.
2. Replace the text with ten UTF-8 bytes and the record becomes 24 bytes. The number of displayed characters need not equal the encoded byte count.
3. If a format rounds total size up to a multiple of eight bytes, both 19 and 24 occupy 24 bytes, while 25 occupies 32. That rule belongs to the toy format only.
4. The exercise links Chapter 3's representations to physical space: units and encoding remain important after values enter a database.

■ 27.3.4 PLAIN — what is really happening inside

1. Row metadata tells the engine how to interpret fields under the relevant schema and storage representation. Missing values may use bitmap information rather than a full ordinary payload value.
2. Schema evolution adds another layer: old and new row representations may coexist while the engine supplies the logical view promised to SQL.
3. Do not infer storage savings solely from a column's declared maximum length. Actual representation, compression, nulls and engine rules determine the physical result.

■ 27.3.5 TECHNICAL — the engineer's version

1. PostgreSQL heap rows have a fixed header, optional null bitmap and aligned user data under the documented layout. Physical interpretation depends on type lengths and alignment metadata. [S128]
2. PostgreSQL TOAST handles eligible large values through compression and/or out-of-line storage. Ordinary tuples do not simply span arbitrary heap pages as one contiguous record. [S143]
3. SQLite records use their own header and serial-type representation. Treating a PostgreSQL tuple decoder as an SQLite parser would be a format error, even when both systems return similar SQL values. [S133]

■ 27.3.6 WORDS — remember these

1. **Row header:** metadata around the visible field values — implementation information needed to locate, interpret or manage a stored row version. **Alignment:** place values at required byte boundaries — padding and layout rules supporting an implementation's representation or access requirements. **Out-of-line value:** a large value stored separately from its main row — payload reached through a reference rather than fully embedded in the ordinary tuple.

27.4 Dirty pages and flushing

■ 27.4.1 PLAIN — in simple words

1. A dirty page is a memory copy containing changes that have not yet been written to the corresponding main data-file location.
2. Dirty does not mean corrupt. It means that the in-memory page and its lower-level stored copy differ in a way the engine is managing.
3. A transaction can commit before every changed data page reaches its final data-file location if the recovery protocol has durably recorded enough information to reconstruct the accepted changes.

■ 27.4.2 PLAIN — a picture in your head

1. Mira corrects a working folder on her desk and records the correction in a protected journal. The cabinet copy can be updated later because the journal tells the recovery clerk what changed.
2. The order is essential: losing the desk copy must not also lose the only description of an accepted correction.
3. **Where the comparison breaks:** a recovery log uses engine-specific records, ordering and commit rules. A casual handwritten note is not enough to recover every possible partial write.

■ 27.4.3 PLAIN — a worked example

1. A toy database page on persistent storage contains stock 5. A transaction changes its buffered version to 2 and creates sufficient recovery information for that accepted update.
2. If the log and required commit evidence are durable while the data page still contains 5, recovery can redo the committed change under the toy protocol.
3. If the engine overwrites the data page before making the required recovery information durable, a crash can leave an incomplete state with no reliable reconstruction path.
4. Chapter 28 makes the toy log rules explicit. This chapter does not claim that merely writing the text “stock=2” implements a real WAL engine.

■ 27.4.4 PLAIN — what is really happening inside

1. The engine schedules writeback and coordinates it with recovery records. A background writer or checkpoint can move dirty pages toward their data-file locations.
2. A page may be modified again while flushing is in progress. Correct engines track the relevant synchronization and dirtiness state rather than assuming one write makes every future version clean.
3. Flushing to the operating system and forcing data through lower caches are distinct boundaries. A successful function call must be interpreted under its actual contract.

■ 27.4.5 TECHNICAL — the engineer’s version

1. Write-ahead logging requires relevant log records to be persisted before the corresponding data-file modifications are allowed to become durable under the engine’s protocol. This permits redo-based recovery without forcing every data page at each commit. [S129]
2. PostgreSQL checkpoints bound the WAL region needed for crash recovery while creating their own I/O work. WAL archiving and other retention requirements can keep older log material relevant beyond local checkpoint needs. [S135] [S131]
3. A database buffer flush is not a universal proof against device loss or dishonest hardware acknowledgements. Chapter 30 follows the persistence chain explicitly. [S130]

■ 27.4.6 WORDS — remember these

1. **Dirty page:** a changed cached page awaiting corresponding writeback — a buffer whose managed contents differ from its lower stored representation. **Flush:** push pending changes toward a specified lower layer — an operation whose persistence guarantee depends on the API and storage contract. **Redo:** reconstruct an accepted change from recovery information — replay that restores data not fully reflected in the main data files before failure.

27.5 Free space and fragmentation

■ 27.5.1 PLAIN — in simple words

1. Deleting a row does not necessarily make the database file smaller immediately. Its space may become reusable within the file while the file retains its allocated size.
2. Free space can be scattered across pages. A large new record may need a suitable contiguous region or an overflow mechanism even when total free bytes look sufficient.
3. Maintenance can reorganize storage, but it consumes resources and may require locks or extra temporary space. Rebuilding a file is not a harmless cosmetic operation.

■ 27.5.2 PLAIN — a picture in your head

1. Removing forms from several folders creates spare room without removing any folder from the cabinet. A new thick form may still not fit in any one folder.
2. Repacking the folders can help, but someone must read, move and rewrite the contents while preserving their identities.
3. **Where the comparison breaks:** engines have exact slot and free-space rules, and some can reuse fragmented regions efficiently. “Fragmented” is a diagnosis requiring measurements, not a general explanation for every slow query.

■ 27.5.3 PLAIN — a worked example

1. Four toy pages each have 100 free bytes. The total is 400 bytes, but a 250-byte item cannot fit on any one page if the format requires the item to fit entirely within one page.
2. A compaction or overflow strategy might change the result. The sum of free bytes alone cannot tell you which strategy exists.
3. Now delete half the rows from a 20-page toy file. The logical row count halves, but the file can remain 20 pages while its internal free space becomes available for later inserts.
4. Measure logical rows, allocated pages and reusable space separately rather than expecting all three to shrink together.

■ 27.5.4 PLAIN — what is really happening inside

1. Engines keep metadata about pages with reusable space and may compact items within a page or recycle entirely free pages.
2. Old row versions and long snapshots can delay when space becomes reusable. A DELETE’s commit is not always the end of its storage consequences.
3. Rebuilding or vacuuming operations differ by product. A command with the same name can have different locking, file-size and operational effects in two engines.

■ 27.5.5 TECHNICAL — the engineer's version

1. PostgreSQL ordinary VACUUM and SQLite VACUUM are not interchangeable operations. PostgreSQL routine vacuuming commonly reclaims reusable space, while SQLite VACUUM rebuilds the database file and has its own space and transaction constraints. [S123] [S145]
2. Include indexes and large-value storage when measuring total footprint. A compact heap with oversized indexes is not a small database merely because one table-size metric looks low.
3. The lab's page arithmetic and SQLite metadata observations do not justify production maintenance commands. Capture workload, free-space evidence, backup/restore readiness and acceptable lock duration before a real intervention.

■ 27.5.6 WORDS — remember these

1. **Reusable space:** allocated storage available for later records — free capacity inside an existing database structure rather than necessarily returned to the filesystem. **Fragmentation:** useful space is distributed inconveniently — a layout condition whose performance effect depends on the engine and access pattern. **Rebuild:** write a reorganized physical representation — maintenance that can improve layout while requiring I/O, temporary capacity and coordination.

27.6 Reading engine-specific evidence

■ 27.6.1 PLAIN — in simple words

1. A useful storage investigation begins with a question: are we reading too much, missing a cache, retaining old versions, waiting for writes or running out of space?
2. Choose observations that distinguish those possibilities. A single file size, hit ratio or elapsed time rarely identifies the cause by itself.
3. Observe through supported interfaces first. Editing a live database file with a hex editor is not a reasonable first diagnostic step.

■ 27.6.2 PLAIN — a picture in your head

1. A mechanic first checks the dashboard, service record and accessible measurements. They do not drill a hole into a running engine because a noise sounds unfamiliar.
2. Storage metadata is the instrument panel, not a complete explanation of every failure.
3. **Where the comparison breaks:** database files can contain private records and credentials. Diagnostic copies, logs and screenshots need access controls as well as technical care.

■ 27.6.3 PLAIN — a worked example

1. In a fresh SQLite teaching database, record `PRAGMA page_size`, `PRAGMA page_count` and `PRAGMA freelist_count` before and after a bounded synthetic workload.
2. Multiply page size by page count to describe the logical main-file page allocation under that observation. Do not add a claim that this equals every physical byte consumed by WAL, journals, temporary files and backups.
3. Compare row counts and expected totals too. A smaller file containing missing records is not an optimization success.
4. The companion records actual runtime and query outputs; no page count is invented in the prose because it can vary with the engine and exact fixture.

■ 27.6.4 PLAIN — what is really happening inside

1. Statistics can be cumulative, delayed, reset or sampled. Their observation interval and reset point are part of their meaning.
2. Product documentation can also change. The official SQLite WAL page consulted for this edition discloses a 2026 WAL-reset bug affecting older releases under specific concurrent WAL write/checkpoint conditions.
3. The historical lab runtime is SQLite 3.46.1. Its successful bounded tests are not a recommendation to deploy that release or a test of the disclosed concurrent WAL bug. The new exercises do not attempt that workload. [S132]

■ 27.6.5 TECHNICAL — the engineer's version

1. Record engine version, journal/recovery mode, relevant settings, connection count, exact statements and measurement interval. Use supported metadata and a disposable copy for low-level experiments. [S75] [S144]
2. As consulted on 25 September 2026, SQLite documents the WAL-reset fix in 3.51.3 and later, with named backports. Verify the current supported release and applicable advisories before real deployment; do not infer safety from this book's historical runtime. [S132]
3. Corruption investigations require evidence preservation. Keep the original data and associated recovery files intact under a reviewed acquisition procedure; Chapter 32 explains why an attempted repair can destroy the information needed to diagnose the problem. [S139]

■ 27.6.6 WORDS — remember these

1. **Storage observation:** a measured fact about one physical configuration — metadata or instrumentation interpreted with version, scope and timing. **Freelist:** storage recorded as available for reuse — an engine-managed collection of free pages or blocks. **Runtime provenance:** which software actually produced the result — the recorded engine and interpreter versions and relevant configuration of an experiment.

27.97 Practice and worked answers

1. **Count toy rows per page.** A 4,096-byte page has 64 header bytes and 100 bytes per row-plus-slot. **Answer:** $\text{floor}(4,032/100)=40$ rows, with 32 bytes left.
2. **Trace the three-slot LRU cache.** Read 1,2,3,1,4,2. **Answer:** one hit and five misses under the stated toy policy, not five proven device reads.
3. **Apply alignment.** Round record lengths 19,24,25 up to multiples of eight. **Answer:** 24,24,32 bytes under that chosen layout.
4. **Interpret dirty.** A buffered page differs from its main-file copy. **Answer:** this is managed pending writeback, not necessarily corruption.
5. **Find the space trap.** Four pages each have 100 free bytes; a 250-byte item must fit on one page. **Answer:** total free space is insufficiently localized; no page can hold the item without another mechanism.
6. **Interpret a DELETE.** Row count halves but file size does not. **Answer:** storage may be reusable internally, retained for old versions or still allocated for engine structures.
7. **Name the cache boundary.** PostgreSQL reports an I/O read. **Answer:** that does not by itself distinguish a device access from an operating-system cache hit.

8. **Limit the runtime claim.** The old SQLite runtime passes the educational tests. **Answer:** this does not establish current production suitability or exercise the disclosed concurrent WAL-reset bug.

27.98 Common wrong ideas

1. **Wrong:** one row occupies one page. **Right:** pages can contain many rows, while large values can use separate storage.
2. **Wrong:** page size is a universal database constant. **Right:** formats and builds differ.
3. **Wrong:** a database cache miss proves a disk read. **Right:** lower caches may satisfy it.
4. **Wrong:** dirty means damaged. **Right:** it normally means changed and awaiting managed write-back.
5. **Wrong:** COMMIT must flush every changed data page. **Right:** a correct recovery protocol can persist log evidence and redo later.
6. **Wrong:** deleted rows immediately shrink the file. **Right:** allocated and reusable space are different measurements.
7. **Wrong:** all VACUUM commands have the same semantics. **Right:** consult the specific engine and operation.
8. **Wrong:** passing old-runtime tests certifies current deployment safety. **Right:** version advisories and real workload validation remain necessary.

27.99 Chapter summary in 20 lines

1. Logical records and physical storage units belong to different layers.
2. Pages organize rows, index entries and engine metadata.
3. Headers and slot entries reduce space available for payload.
4. Toy occupancy calculations must state their layout assumptions.
5. Large values can use compression or out-of-line storage.
6. Physical row locations should not replace stable business keys.
7. Buffer pools retain pages in memory for reuse.
8. Cache replacement depends on workload and engine policy.
9. A database-cache miss is not necessarily a physical-device read.
10. Row representations include metadata, lengths, null information and alignment.
11. Dirty pages contain managed changes awaiting corresponding writeback.
12. Recovery ordering connects log persistence to later data-page flushing.
13. A committed transaction need not have every data page at its final location immediately.
14. Checkpoints and background writeback have measurable resource costs.
15. Free space inside a file differs from space returned to the filesystem.
16. Scattered free bytes do not guarantee room for one large item.
17. Maintenance commands differ in semantics across products.
18. Supported metadata is the starting point for storage diagnosis.
19. Record runtime, settings and observation intervals with every experiment.
20. Preserve recovery evidence and verify current advisories before real deployment.

Write-Ahead Logs and Recovery

28.0 What this chapter gives you

1. A crash can erase memory while leaving only part of the recent work in storage. Recovery needs a protocol for deciding what those surviving bytes mean.
2. This chapter explains write-ahead ordering, log positions, checkpoints and redo. A small logical recovery model makes the steps visible without pretending to implement PostgreSQL's or SQLite's physical log format.
3. The model has explicit simplifying assumptions: a known starting state, ordered valid records, serial teaching transactions and an explicitly selected surviving log prefix. Its tests are not process-crash or power-loss experiments.

28.1 Why recovery information exists

■ 28.1.1 PLAIN — in simple words

1. An application can change several pages before the storage system finishes writing all of them. A crash in the middle can leave a mixture of old and new data.
2. Recovery information records enough about changes and their acceptance boundaries to reconstruct a valid state under the engine's rules.
3. A log is useful only when its own validity, ordering and persistence are protected. An incomplete or corrupted note cannot be treated as trustworthy merely because it is called a log.

■ 28.1.2 PLAIN — a picture in your head

1. Before updating several filing cabinets, a clerk records the required changes in a protected journal. After an interruption, another clerk uses the journal to finish or resolve the interrupted work.
2. The journal must distinguish an accepted instruction from a draft that was never approved.
3. **Where the comparison breaks:** real logs often describe physical or physiological engine changes, not ordinary business sentences. Their records, checksums and transaction metadata must be interpreted by the matching engine.

■ 28.1.3 PLAIN — a worked example

1. A toy operation changes stock from 5 to 2 and records a reservation. If the stock page reaches storage but the reservation page does not, a crash leaves an incomplete local operation.
2. A recovery protocol can avoid forcing both pages at the exact same instant by preserving sufficient log evidence and a defined commit boundary.
3. After restart, the engine follows its recovery rules rather than guessing from whichever page happens to look newest.

4. A business audit log containing “reservation accepted” may lack the physical information needed to rebuild pages. Conversely, a WAL record may lack the human explanation needed for an audit. The two logs serve different purposes.

■ 28.1.4 PLAIN — what is really happening inside

1. The engine coordinates data-page modifications, log generation and transaction completion. Log persistence and page writeback obey a required order.
2. Recovery can encounter work from committed, aborted or incomplete transactions. How it handles each depends on the engine’s storage and visibility design.
3. A correct recovery process must also recognize incomplete or invalid log tails. It cannot blindly execute every byte remaining in a file as a valid operation.

■ 28.1.5 TECHNICAL — the engineer’s version

1. PostgreSQL’s WAL supports redo-based recovery of data-file changes. SQLite WAL uses page frames and commit information under its own format; SQLite rollback journaling is a different mechanism. [S129] [S132] [S133]
2. The logical model in this chapter replays only complete committed transactions. That is not a description of PostgreSQL’s exact redo algorithm: physical recovery can replay changes whose transaction visibility is subsequently governed by transaction-status and MVCC rules.
3. Keep recovery logs, application event logs and audit records distinct in a system inventory. Similar append-only shapes do not imply identical contents, retention or authority.

■ 28.1.6 WORDS — remember these

1. **Recovery log:** information needed to reconstruct a valid stored state — engine records supporting recovery after incomplete persistence or interruption. **Log tail:** the final portion of the recorded sequence — a region that can be incomplete or require validity checks after failure. **Recovery protocol:** the rules for interpreting surviving evidence — a specified procedure connecting log validity, transaction status and stored data.

28.2 Log before data

■ 28.2.1 PLAIN — in simple words

1. Write-ahead means that the required recovery information reaches the promised persistent boundary before the corresponding data-file change can make recovery depend on it.
2. This ordering is more important than the fact that a log file exists. A log written too late cannot explain a data change after the only in-memory explanation disappears.
3. Commit acknowledgement adds another condition: the engine must meet the configured durability promise before telling the caller that the transaction is accepted under that promise.

■ 28.2.2 PLAIN — a picture in your head

1. A clerk must secure the correction instruction before altering the official cabinet copy. Otherwise an interruption can leave a changed cabinet and no reliable account of the intended operation.
2. Securing the instruction can allow the cabinet changes to be completed later in a more efficient batch.
3. **Where the comparison breaks:** “secure” means a precise storage contract, not merely placing a note in another memory buffer. Operating-system and device caches may still be volatile.

■ 28.2.3 PLAIN — a worked example

1. Consider four milestones: create the log record in memory, persist the required log record, flush the changed data page, and acknowledge commit under the selected contract.
2. A valid WAL design can persist the log and required commit information, acknowledge a durable commit, and flush some data pages later. Recovery uses the surviving log if the later flush never happened.
3. An invalid ordering flushes a changed page while its required recovery record exists only in volatile memory. Losing that memory can make the page change impossible to interpret or complete safely.
4. This is an ordering argument, not a claim that every engine uses exactly four system calls or one log record per business action.

■ 28.2.4 PLAIN — what is really happening inside

1. Log records are often appended in a sequence, making it possible to synchronize a contiguous prefix efficiently.
2. Several commits can share one lower-level synchronization when the engine ensures that the required records for each are included. This is group commit, not a relaxation of each acknowledged transaction's selected promise.
3. Durability settings can intentionally change when acknowledgement occurs. Such a change belongs in the application contract and benchmark report, not hidden in a performance tweak.

■ 28.2.5 TECHNICAL — the engineer's version

1. PostgreSQL's WAL ordering allows data pages to be written after the necessary log records have been flushed. Its documentation explains that a WAL synchronization can cover multiple concurrent commits. [S129]
2. PostgreSQL asynchronous commit can acknowledge before the transaction's WAL reaches durable storage, allowing recent acknowledged transactions to be lost after a crash while preserving database consistency under the documented design. This differs from disabling `fsync`, which can threaten consistency. [S136]
3. An experiment comparing commit latency must record these settings. A faster result obtained by weakening the durability contract is not a like-for-like optimization.

■ 28.2.6 WORDS — remember these

1. **Write-ahead ordering:** recovery evidence must precede dependent data persistence — the ordering constraint linking log durability and data-page writes. **Group commit:** one synchronization supports several commits — batching log persistence while meeting each transaction's selected acknowledgement condition. **Asynchronous commit:** acknowledge before full local log persistence — a configured contract that can trade recent-transaction durability for lower latency.

28.3 Log sequence positions

■ 28.3.1 PLAIN — in simple words

1. A log position identifies a place in the ordered recovery stream. It lets the engine say how far it has generated, written, flushed or replayed information.

2. Those progress points are different. A record can be generated but not yet persisted, or persisted on a replica but not yet applied there.
3. A position is not automatically a timestamp, a transaction number or a count of rows. Interpret it within the log stream and timeline to which it belongs.

■ 28.3.2 PLAIN — a picture in your head

1. A long instruction roll has marked positions. One clerk has written through position 100, another has secured through 90, and a recovery clerk has applied through 80.
2. Saying “we are at 100” hides which clerk and which boundary you mean.
3. **Where the comparison breaks:** real log sequence numbers may refer to byte positions rather than numbered instructions. Comparing positions across unrelated streams or timelines needs additional identity information.

■ 28.3.3 PLAIN — a worked example

1. Our toy log numbers records 1 through 11 for easy hand tracing. Suppose records through 9 survive a simulated failure; records 10 and 11 exist only in the discarded model memory.
2. Recovery must use the surviving prefix ending at 9. It cannot infer the contents of later records from the fact that the application intended to append them.
3. A checkpoint at position 6 supplies a known base state. Replaying valid records 7 through 9 can advance that state without rereading the model’s earlier completed prefix.
4. These small integers are record ordinals in the teaching model, not actual PostgreSQL LSN encodings.

■ 28.3.4 PLAIN — what is really happening inside

1. Engine metadata can associate pages with log positions so recovery knows which changes a page already reflects.
2. The engine must make redo safe under its actual record semantics. Simply replaying an increment twice would be wrong unless the protocol recognizes that the corresponding change is already applied.
3. Position continuity matters. A missing required segment is not repaired by skipping to the next available filename and hoping the data agrees.

■ 28.3.5 TECHNICAL — the engineer’s version

1. PostgreSQL page headers include a WAL-related LSN field. Recovery and replication use log positions with engine-specific meanings; the toy ordinal notation deliberately avoids imitating their byte-level format. [S128] [S131]
2. Distinguish generated, written, flushed and replayed progress. A lag measurement must name both endpoints and whether the difference is expressed in bytes, time or another unit. [S144]
3. Retain stream identity and timeline context with recovery positions. A numerically similar offset in an unrelated history is not interchangeable evidence.

■ 28.3.6 WORDS — remember these

1. **Log sequence number:** a position in a named recovery stream — an engine-defined offset used to identify and compare WAL progress. **Durable prefix:** the surviving accepted beginning of a log — the valid ordered region known to have reached the required persistence boundary. **Re-**

play position: how far recovery has applied the stream — progress measured under the engine's interpretation of log records.

28.4 Checkpoints

■ 28.4.1 PLAIN — in simple words

1. A checkpoint records a recovery starting boundary so the engine need not reconstruct everything from the beginning of time after every crash.
2. It coordinates data-page persistence and recovery metadata. It is not simply the same thing as a user transaction committing.
3. Older logs may still be needed for backups, replicas or other consumers even after local crash recovery can start later. "Checkpoint complete" is not permission to delete arbitrary recovery files.

■ 28.4.2 PLAIN — a picture in your head

1. A clerk periodically produces a verified cabinet state and marks where the journal continues from that state. A later recovery starts from the cabinet and reads the remaining instructions.
2. An archive keeper may still need older instructions to reconstruct an earlier date.
3. **Where the comparison breaks:** real checkpoints can coordinate work over an interval rather than freeze the entire system at one instant. Their correctness depends on the engine's detailed ordering rules.

■ 28.4.3 PLAIN — a worked example

1. In the toy model, start with $P=5$. T_1 commits $P=2$ and T_2 aborts its proposed $P=9$. After position 6, no teaching transaction is active and the accepted state is $P=2$.
2. Save that state as a model checkpoint with position 6. Later T_3 commits $P=4$ at position 9; T_4 remains incomplete at position 11.
3. Recovery from the checkpoint and valid suffix returns $P=4$. The checkpoint does not make T_4 's incomplete proposed $P=1$ committed.
4. This model deliberately checkpoints only at a simple quiescent boundary. It is not an implementation of a real fuzzy checkpoint algorithm.

■ 28.4.4 PLAIN — what is really happening inside

1. More frequent checkpoints can reduce later redo work but increase current writeback and related log overhead.
2. Less frequent checkpoints can improve some write workloads while increasing recovery work and retained log volume. The best interval depends on recovery objectives and actual measurements.
3. Long readers can also affect checkpoint progress in engines such as SQLite WAL. A checkpoint that cannot complete under its mode should not be reported as a fully reclaimed log merely because the call returned.

■ 28.4.5 TECHNICAL — the engineer's version

1. PostgreSQL checkpoints establish redo boundaries and spread writeback work under configurable limits. Full-page-image behaviour interacts with checkpoint frequency, so tuning one parameter can change WAL volume as well as recovery time. [S135]

2. SQLite WAL checkpointing transfers eligible page content back to the main database and has modes with different blocking/completion behaviour. Active readers can prevent complete reset of the WAL. [S132]
3. Checkpoint retention and archive retention are different. Keep every segment required by the actual backup, recovery and replication contracts. [S131]

■ 28.4.6 WORDS — remember these

1. **Checkpoint:** a recorded starting boundary for recovery — coordinated data and metadata progress that limits required redo under an engine's protocol. **Redo horizon:** the earliest recovery position still needed — the log boundary from which reconstruction must begin for a particular purpose. **Checkpoint pressure:** current work caused by reaching that boundary — I/O and related overhead generated by checkpoint activity.

28.5 Redo and rollback concepts

■ 28.5.1 PLAIN — in simple words

1. Redo means reconstructing changes that the recovery protocol says must be represented. Rollback means preventing an unsuccessful transaction's changes from becoming its accepted result.
2. Engines can achieve those goals differently. Some recovery designs use explicit undo information; others combine physical redo with transaction visibility so incomplete transactions do not become visible committed work.
3. Do not assume that every database literally runs an opposite SQL statement for every failed statement. Recovery operates at the engine's own representation and transaction rules.

■ 28.5.2 PLAIN — a picture in your head

1. A recovery clerk can finish approved corrections and exclude unapproved drafts. The exact paperwork determines how each is recognized.
2. One office keeps before-and-after copies; another preserves new pages with approval metadata. Both need a coherent protocol, but their recovery steps differ.
3. **Where the comparison breaks:** "undo" is not necessarily a business compensation. Refunding a delivered purchase is a new real-world action, not erasing an uncommitted database write.

■ 28.5.3 PLAIN — a worked example

1. The toy model stages SET records by transaction identity. A COMMIT applies that transaction's staged after-values; an ABORT discards them. Incomplete staged work at the end is ignored.
2. For T1: BEGIN, SET P=2, COMMIT changes accepted state from 5 to 2. For T2: BEGIN, SET P=9, ABORT leaves accepted state at 2.
3. Replaying the same complete log from the same initial state produces the same result. That is determinism of the model, not permission to apply arbitrary increments repeatedly to an already advanced state.
4. The lab validates record ordering and transaction state. A COMMIT for an unknown transaction is malformed input, not a harmless record to skip silently.

■ 28.5.4 PLAIN — what is really happening inside

1. Recovery must know both the base state's provenance and the log region being applied. Replaying a suffix against the wrong checkpoint can produce plausible but incorrect records.

2. Physical redo may rebuild pages containing versions that are not visible as committed rows. Visibility and physical reconstruction are related but separate engine responsibilities.
3. A recovery tool should fail clearly when required evidence is absent or incompatible, rather than inventing a clean state by dropping unexplained records.

■ 28.5.5 TECHNICAL — the engineer's version

1. The toy recovery function uses logical after-values and serial transaction fixtures. It does not model page LSN checks, torn pages, interleaved conflicting writers, transaction-ID wraparound or an engine's log parser.
2. PostgreSQL crash recovery is redo-oriented; MVCC and transaction-status information govern visibility of tuple versions. Do not equate the toy's "apply only committed transactions" loop with the exact PostgreSQL implementation. [S129] [S119]
3. SQLite rollback journals preserve original page content, whereas WAL frames record revised page content. These mechanisms illustrate why recovery terminology must be attached to a concrete format. [S133]

■ 28.5.6 WORDS — remember these

1. **After-value:** the state an accepted update should leave — a logical replacement used by this teaching recovery model. **Undo information:** evidence for reversing unaccepted changes — recovery data used by designs that require explicit restoration of earlier state. **Recovery base:** the known state from which replay begins — a checkpoint or backup identified together with its matching log boundary.

28.6 Crash-recovery traces

■ 28.6.1 PLAIN — in simple words

1. A useful recovery trace lists what survived, what did not, and what the protocol can conclude. It does not fill missing evidence with the application's intention.
2. Testing different surviving prefixes exposes the importance of commit boundaries. An update record alone is not necessarily an accepted transaction.
3. A simulated prefix is a teaching experiment. A real crash test additionally needs control over processes, storage, flush settings and the failure being injected.

■ 28.6.2 PLAIN — a picture in your head

1. Imagine the journal is cut at a chosen line. The recovery clerk may use only the lines still present and the known starting cabinet state.
2. A line that would have been written next is not evidence merely because the clerk remembers planning it.
3. **Where the comparison breaks:** real failures do not always leave a neat cut at a record boundary. Checksums, framing, partial writes and device behaviour determine which bytes are valid.

■ 28.6.3 PLAIN — a worked example

1. Use the following complete synthetic log. SET means a staged after-value, and the initial state is P=5.

1 BEGIN T1	2 SET T1 P=2	3 COMMIT T1
4 BEGIN T2	5 SET T2 P=9	6 ABORT T2

7 BEGIN T3	8 SET T3 P=4	9 COMMIT T3
10 BEGIN T4	11 SET T4 P=1	[no commit]

2. A surviving prefix ending at 2 recovers P=5. Ending at 3 recovers P=2. Ending at 5 or 6 still recovers P=2.
3. Ending at 9 recovers P=4. Ending at 11 also recovers P=4 because T4 lacks a commit. Recovering from the position-6 checkpoint P=2 plus the remaining suffix also yields P=4.
4. Every expected result follows from the model's explicit rules. The example makes no claim that a real device preserved exactly those prefixes during a power cut.

■ 28.6.4 PLAIN — what is really happening inside

1. A real recovery investigation preserves the original files, relevant log segments, configuration and diagnostic output before attempting changes.
2. A successful restart proves that the engine reached a state it accepted under its checks. Business reconciliation and expected-record checks are still needed to establish the recovery target's meaning.
3. Failure to find a record can mean it was never committed, not included in the surviving history, restored to an earlier target or queried incorrectly. The trace must distinguish these possibilities rather than assigning a convenient cause.

■ 28.6.5 TECHNICAL — the engineer's version

1. The companion tests every listed prefix and malformed transaction ordering in memory. No process is killed, no power is cut, and no hardware durability claim follows from those tests.
2. Before a real failure experiment, state the fault model: process termination, operating-system crash, abrupt power loss, device loss or a network partition. Each exercises different layers of the persistence chain. [S130]
3. For PostgreSQL recovery from backups, use matching physical base-backup and continuous WAL evidence under the documented procedure. A logical SQL dump is not a physical WAL-replay base. [S131]

■ 28.6.6 WORDS — remember these

1. **Crash trace:** the ordered evidence surrounding an interruption — a record of persisted state, surviving log material and recovery decisions. **Fault model:** the failures an experiment assumes or injects — the specific process, operating-system, device or communication conditions being tested. **Recovery target:** the intended state or boundary to restore — a chosen time, log position or other engine-supported objective with matching evidence.

28.97 Practice and worked answers

1. **Find the unsafe ordering.** A changed data page reaches storage while its required log record remains only in memory. **Answer:** a crash can remove the only recovery explanation; the write-ahead requirement is violated under the stated model.
2. **Interpret group commit.** One sync covers five transactions' required records. **Answer:** batching can preserve each selected durability promise; five separate sync calls are not inherently required.
3. **Trace prefix 2.** T1 has begun and staged P=2 but not committed. **Answer:** recover P=5 from the known initial state.
4. **Trace prefix 5.** T1 committed P=2; T2 staged P=9 without commit. **Answer:** recover P=2.

5. **Trace prefix 11.** T3 committed P=4; T4 remains incomplete. **Answer:** recover P=4.
6. **Use a checkpoint.** Base P=2 at position 6, then replay 7–11. **Answer:** P=4 under the same model and matching boundary.
7. **Classify a malformed log.** COMMIT appears for a transaction never begun. **Answer:** reject the invalid model input; do not silently invent a transaction.
8. **Limit the experiment.** All prefix tests pass. **Answer:** the logical recovery model behaved as expected. Physical log parsing, process crashes and device persistence remain untested.

28.98 Common wrong ideas

1. **Wrong:** any append-only application log is a database recovery log. **Right:** contents and recovery guarantees differ.
2. **Wrong:** write-ahead means only that the log file was created first. **Right:** the required persistence ordering must hold for dependent changes.
3. **Wrong:** every COMMIT forces every data page. **Right:** durable recovery evidence can permit later page writeback.
4. **Wrong:** a log position is always a timestamp or row count. **Right:** its meaning belongs to the named stream and implementation.
5. **Wrong:** a checkpoint makes all older logs disposable. **Right:** backups, replicas and other recovery objectives can still require them.
6. **Wrong:** all engines recover by the toy committed-transaction loop. **Right:** physical redo, undo and visibility designs differ.
7. **Wrong:** a restart proves every expected business record survived. **Right:** reconcile the restored state against the intended recovery target.
8. **Wrong:** a simulated log cut is a power-loss test. **Right:** it exercises only the stated logical model.

28.99 Chapter summary in 20 lines

1. Recovery needs evidence about changes that may be only partly reflected in data files.
2. A recovery log has a different purpose from an ordinary business audit log.
3. Log validity, ordering and persistence are part of the protocol.
4. Required recovery records must precede dependent data-page persistence.
5. Commit acknowledgement must match the configured durability promise.
6. Group commit can synchronize several transactions together.
7. Asynchronous commit changes the recent-transaction durability contract.
8. Log positions identify progress within a named stream and history.
9. Generated, written, flushed and replayed positions are different boundaries.
10. Recovery cannot use records that did not survive.
11. Checkpoints provide a matching base and redo starting boundary.
12. Checkpoint frequency trades current I/O against later recovery work.
13. Older logs can remain necessary for other recovery and replication purposes.
14. Redo reconstructs required state; rollback prevents unaccepted transactional effects.
15. Engine recovery mechanisms are not universally identical.
16. The toy model stages after-values until a valid COMMIT appears.
17. Its incomplete T4 does not change the recovered P=4 result.

18. A checkpoint must be paired with the correct suffix and provenance.
19. Preserve original evidence before attempting recovery interventions.
20. State which fault model was actually tested and which remains unverified.

LSM Trees and Compaction

29.0 What this chapter gives you

1. Some storage engines accept new writes into memory and later produce sorted immutable files instead of continually updating one in-place search tree. This chapter explains that family of designs.
2. You will follow a key through a memory table, sorted runs, older versions and deletion markers, then examine how compaction trades write work, read work and temporary space.
3. The companion model is an in-memory sorted-run exercise, not a RocksDB implementation. It has no real WAL, concurrent snapshots, file manifest or crash-safe installation protocol.

29.1 Sequential writes

■ 29.1.1 PLAIN — in simple words

1. A system receiving many scattered key updates can collect them before writing larger organized groups to storage. That changes the physical work performed for each incoming request.
2. Writing sequentially can avoid some scattered access costs, but the work does not disappear. The engine must later find the latest values and reorganize accumulated files.
3. An LSM design is therefore not “writes are free.” It moves and batches work, creating trade-offs that depend on the workload and storage system.

■ 29.1.2 PLAIN — a picture in your head

1. Rather than walking to the filing cabinet for every arriving note, Mira collects notes in an organized tray and periodically files a sorted batch.
2. Later searches may need to consult several batches until the office merges them into a cleaner arrangement.
3. **Where the comparison breaks:** real engines need durable logs, ordered installation of files and concurrency controls. An ordinary tray that vanishes in a fire does not provide durable acknowledged writes.

■ 29.1.3 PLAIN — a worked example

1. Ten thousand synthetic updates each contain 100 bytes of logical key/value payload. The incoming payload totals 1,000,000 bytes, or 1 MB decimal.
2. If storage writes those values first into new sorted files and later rewrites them during compaction, total bytes written can exceed 1 MB. WAL, indexes, filters, compression and metadata add further differences.

3. Define write amplification as physical bytes written divided by logical bytes written over a stated interval and boundary. If the measured boundary writes 5 MB for 1 MB of logical input, the ratio is 5.
4. This arithmetic does not predict RocksDB's ratio. The real ratio depends on configuration, key distribution, updates and which storage layers the measurement includes.

■ 29.1.4 PLAIN — what is really happening inside

1. New entries are organized in a memory structure and may also be logged for recovery. A flush creates a sorted persistent run under the engine's durability protocol.
2. Older runs remain available while newer runs accumulate. A lookup must choose the correct visible version across these structures.
3. Background compaction merges selected runs, discards eligible obsolete versions and installs replacement files safely. This is ongoing maintenance work, not a one-time setup step.

■ 29.1.5 TECHNICAL — the engineer's version

1. RocksDB's documented architecture includes a memtable, log files and sorted-string-table files. It is an embedded storage-engine library, not by itself a complete networked database service. [S140]
2. Log-structured merge designs batch updates into sorted runs and reorganize them over time. Their compaction policy determines the shape of the stored levels and which runs reads may need to consult. [S134]
3. Distinguish the engine's write amplification from a flash device's internal write amplification. Multiplying or comparing ratios requires compatible boundaries and denominators.

■ 29.1.6 WORDS — remember these

1. **LSM tree:** organize writes through memory and merged sorted runs — a log-structured merge family of storage designs with deferred reorganization. **Write amplification:** storage writes exceed logical input — a measured ratio of bytes written at a named layer to logical bytes accepted over a stated interval. **Sorted run:** a sequence already ordered by its search key — a file or logical collection that can be searched and merged using key order.

29.2 Memory tables and immutable files

■ 29.2.1 PLAIN — in simple words

1. A memory table holds recent entries in a searchable structure. When it reaches a chosen limit, the engine can freeze it and flush its contents into an immutable sorted file.
2. Immutable means the installed file is not edited in place by ordinary key updates. Later updates appear in newer structures, and compaction eventually replaces groups of files.
3. An immutable file is not automatically a complete immutable history. Old files can be retired, and the engine can discard obsolete versions under its retention rules.

■ 29.2.2 PLAIN — a picture in your head

1. Mira writes into today's tray. When it fills, she seals it for filing and starts a new tray so incoming work can continue.
2. The sealed batch does not receive handwritten corrections. New correction notes go into the active tray and take precedence under the office's version rules.

3. **Where the comparison breaks:** a real flush can fail partway. The engine needs metadata identifying which files are complete and installed, not merely files whose names happen to exist in a directory.

■ 29.2.3 PLAIN — a worked example

1. Let the active table hold entries (A, 1, 'red'), (B, 2, 'blue') and (A, 3, 'green'), where the middle number is a strictly ordered model version.
2. A current lookup for A returns green because version 3 supersedes version 1. A snapshot lookup at version 2 returns red under the model's visibility rule.
3. After flush, the sorted representation groups A's versions and B's entry. The same logical answers should hold before and after the flush.
4. A flush that accidentally drops version 1 while a permitted version-2 snapshot still exists would violate that snapshot contract.

■ 29.2.4 PLAIN — what is really happening inside

1. Engines can keep active and immutable memory tables concurrently while background workers flush completed batches.
2. Recovery logging protects recent acknowledged work according to the chosen settings. Disabling or weakening logging changes the failure contract; sorted files alone do not protect entries still only in memory.
3. A manifest or equivalent metadata identifies the active file set and related state. Installing new files and retiring old ones must survive interruption without exposing an arbitrary mixture.

■ 29.2.5 TECHNICAL — the engineer's version

1. RocksDB describes memtable flushes producing sorted table files and associated log lifecycle management. Its configurable recovery and synchronization behaviour must be read separately from its high-level architecture. [S140]
2. Internal keys commonly combine user-key identity with sequence or version information in LSM implementations. The exact encoding and comparison order are product details; this book's tuples are a teaching representation.
3. A correct flush preserves every version required by active snapshots and the current visibility contract. A latest-value-only export is not a valid substitute when historical snapshot reads remain supported.

■ 29.2.6 WORDS — remember these

1. **Memtable:** the searchable workspace for recent writes — an in-memory structure holding entries before or alongside persistent sorted files. **Immutable table file:** a sorted file replaced rather than updated in place — an installed run whose ordinary contents do not change after creation. **Manifest:** the record of which storage files belong to the current state — metadata coordinating installed runs and their lifecycle.

29.3 Sorted runs

■ 29.3.1 PLAIN — in simple words

1. Sorting makes each run easier to search and combine with another run. But a key may appear in several runs, so finding one occurrence is not necessarily enough.

2. The engine must select the version visible to the reader. A newer deletion marker can override an older value that still exists in a lower run.
3. Range queries must merge ordered candidates and resolve duplicates or versions while preserving the requested key order.

■ 29.3.2 PLAIN — a picture in your head

1. Three alphabetized card boxes contain successive batches of corrections. Looking up a name requires considering the newest relevant card, not accepting the first old card you find.
2. Printing all names in order requires merging the boxes and removing superseded copies under a clear rule.
3. **Where the comparison breaks:** engine lookups use indexes, filters and metadata to skip irrelevant runs. They do not necessarily scan every card in every box.

■ 29.3.3 PLAIN — a worked example

1. Run R1 contains A at version 1 with value red and C at version 2 with value black. Run R2 contains A at version 3 with value green and B at version 4 with value blue.
2. A current ordered merge returns A=green, B=blue, C=black. It does not return A twice merely because two physical versions exist.
3. A permitted snapshot at version 2 returns A=red and C=black; B did not yet exist in that snapshot. The version limit changes the answer even though the same files may be consulted.
4. The model explicitly orders versions numerically. Wall-clock timestamps from unsynchronized writers are not silently substituted for this ordering rule.

■ 29.3.4 PLAIN — what is really happening inside

1. Key-range metadata can rule out runs that cannot contain the requested key. Within a run, an index can locate the relevant block.
2. A Bloom filter can help reject absent keys without reading their data blocks, subject to its construction and lookup contract. A possible match still needs verification against real entries.
3. Snapshot visibility, key comparison and deletion handling must agree across memory tables, file indexes and compaction. A mismatch can create lost or resurrected records.

■ 29.3.5 TECHNICAL — the engineer's version

1. A merge iterator can combine ordered runs using a priority structure while grouping equal user keys and selecting the highest visible version under the defined comparator.
2. In the simple model with k runs and n emitted candidates, a heap-based merge can organize candidate selection in approximately $O(n \log k)$ comparisons, excluding decoding, I/O and version-resolution costs. This is an algorithmic model, not a device-latency estimate.
3. Real compaction strategies constrain run overlap and hence lookup work. RocksDB's documentation distinguishes leveled and tiered/universal approaches rather than one universal LSM layout. [S134]

■ 29.3.6 WORDS — remember these

1. **Merge iterator:** read several ordered runs as one ordered sequence — an iterator that combines candidates while resolving key and version rules. **Visible sequence limit:** the newest version a snapshot may use — a model or engine boundary excluding later updates from that reader. **Run**

overlap: several files cover some of the same key range — a layout property affecting how many candidates a lookup may need to inspect.

29.4 Read amplification

■ 29.4.1 PLAIN — in simple words

1. A request for one small value can require several checks and reads across memory, indexes, filters and runs. That extra work is part of read amplification.
2. The term needs a definition. It can refer to bytes read relative to bytes returned, or to the number of storage probes required for one logical lookup. These are different metrics.
3. Optimizing only write speed can leave readers doing excessive work later. A storage design should be judged against the complete workload, not one convenient operation.

■ 29.4.2 PLAIN — a picture in your head

1. Filing each arriving batch immediately is easy, but finding one customer's latest note becomes harder if the office keeps hundreds of overlapping boxes.
2. A directory and "not in this box" filters reduce unnecessary opening, while merging boxes reduces the number of places to check.
3. **Where the comparison breaks:** one physical read can bring many useful entries into cache. Counting boxes without naming the cache and block boundary can misrepresent actual device work.

■ 29.4.3 PLAIN — a worked example

1. A toy absent-key lookup checks four overlapping runs. Filters reject three runs, leaving one candidate block read that ultimately finds no matching key.
2. Without those filters, the same model might read four candidate blocks. The logical result is still "not found"; the physical work differs.
3. If each block is 4,096 bytes and the model reads four, it transfers 16,384 bytes at that modeled boundary. A real cache can reduce device reads below this count.
4. A filter false positive costs extra checking. A false negative caused by a broken filter protocol can hide a real record and is a correctness failure, not merely a performance cost.

■ 29.4.4 PLAIN — what is really happening inside

1. Read cost depends on run count, overlap, filters, block indexes, cache state and how many obsolete versions must be examined.
2. Negative lookups can be especially revealing because they may need to establish absence across all relevant structures.
3. Range scans have different behaviour from point lookups. A filter designed for exact-key membership may help less when the request spans a broad interval.

■ 29.4.5 TECHNICAL — the engineer's version

1. Define separate metrics for run probes, data-block reads, bytes read and user-visible latency. Report warm and cold cache assumptions rather than treating these metrics as interchangeable.
2. Leveled and tiered compaction make different read/write/space trade-offs; workload skew and implementation optimizations can change the practical result. [S134]

3. The companion records model probe counts only. It does not measure RocksDB file I/O or claim a particular production read-amplification factor.

■ 29.4.6 WORDS — remember these

1. **Read amplification:** extra storage work for a logical read — a defined ratio or probe count measured at a specified layer. **Negative lookup:** a search that establishes absence — a lookup returning no visible matching key after checking relevant structures. **Filter false positive:** a filter suggests a candidate that is absent — extra verification work without a false final membership claim.

29.5 Compaction and tombstones

■ 29.5.1 PLAIN — in simple words

1. Compaction merges selected runs and removes versions that are no longer needed under the visibility and retention rules.
2. Deletion often begins as a tombstone: a newer marker saying that an older value must not be returned. The older bytes may still exist until safe cleanup occurs.
3. Dropping the marker too early can resurrect an older value from an unmerged file. Deletion correctness depends on which older versions remain reachable.

■ 29.5.2 PLAIN — a picture in your head

1. A new card says “this account entry is deleted,” while an older card remains in a lower archive box. Throwing away only the deletion card makes the old card look current again.
2. The cleanup clerk must know whether all older relevant copies have been covered before removing the marker.
3. **Where the comparison breaks:** real deletion also involves snapshots, backups and replicas. Safe removal from one LSM structure does not prove erasure from every system copy.

■ 29.5.3 PLAIN — a worked example

1. R1 contains (A, 1, 'red'). R2 contains (A, 3, TOMBSTONE). A current lookup returns no A; a permitted snapshot at version 2 returns red.
2. Compacting only R2 and dropping its tombstone while R1 remains would wrongly expose red to a current lookup.
3. In a latest-only toy mode with no older snapshots and all relevant runs included, compaction can remove both the obsolete value and its deletion marker. The assumptions are part of the result.
4. If a later (A, 4, 'green') is added, current A is green while a version-3 snapshot remains deleted. Recreation is another versioned state, not evidence that the earlier deletion never occurred.

■ 29.5.4 PLAIN — what is really happening inside

1. Compaction reads selected runs, resolves versions and writes replacement runs. The old files remain necessary until the new file set is safely installed and no reader still requires them.
2. This can require temporary space for both input and output files. A nearly full disk can prevent the cleanup work intended to reclaim space.
3. Tombstone removal requires knowledge of older versions and snapshot requirements. A simple age threshold is not universally sufficient without the protocol’s supporting assumptions.

■ 29.5.5 TECHNICAL — the engineer's version

1. Leveled compaction typically trades more rewriting for tighter run organization, while tiered/universal approaches can reduce some write amplification at the cost of read and space amplification. These are design tendencies, not universal numeric rankings. [S134]
2. The model exposes two operations: a visibility-preserving merge retaining versions, and latest-only compaction allowed only when the caller declares complete run coverage and no retained snapshots. Tests demonstrate the unsafe tombstone-drop counterexample.
3. Neither operation implements crash-safe file installation, concurrent reader references or a real deletion guarantee across backups. Those are separate protocol layers.

■ 29.5.6 WORDS — remember these

1. **Compaction:** merge and reorganize stored runs — background work that installs a new representation and discards only eligible obsolete state. **Tombstone:** a newer marker hiding an older value — a deletion record retained until older visible copies and snapshot requirements permit removal. **Resurrection:** an old deleted value becomes visible again — a correctness failure caused by losing the deletion information while older state remains reachable.

29.6 Workload trade-offs

■ 29.6.1 PLAIN — in simple words

1. A useful storage choice asks what the application actually does: frequent small updates, large range scans, mostly reads, repeated changes to a few hot keys or mostly new keys.
2. Background maintenance must keep up with the long-term write rate. A short benchmark can look fast while leaving a growing backlog of compaction work.
3. The right comparison includes steady-state latency, throughput, space, recovery and operational effort. One peak write number is not a complete description of the system.

■ 29.6.2 PLAIN — a picture in your head

1. An office can accept incoming boxes quickly by stacking them in the corridor. That speed is not sustainable if nobody can sort the boxes and the corridor eventually fills.
2. A steady-state test asks whether incoming work and cleanup can continue together without an ever-growing backlog.
3. **Where the comparison breaks:** engines can throttle or stall writes deliberately to avoid exhausting memory or storage. A slowdown can be a protective response to accumulated work, not arbitrary malfunction.

■ 29.6.3 PLAIN — a worked example

1. A synthetic service accepts 10 MB/s of logical writes. At a chosen measured storage boundary, steady-state write amplification is 6, implying about 60 MB/s of writes before other uncounted work.
2. If the device and competing workload leave only 40 MB/s for that work, the long-run demand exceeds the available budget by 20 MB/s under these assumptions.
3. A brief test may temporarily absorb the difference in memory and pending files. It cannot continue indefinitely without reducing demand, increasing capacity or changing the layout and workload.

4. These values are illustrative. Measure actual amplification and sustained capacity rather than selecting a product from this arithmetic alone.

■ 29.6.4 PLAIN — what is really happening inside

1. Compaction competes with foreground reads and writes for CPU, memory, bandwidth and temporary space.
2. Skew matters: repeatedly updating a small key range can behave differently from uniformly rewriting the entire dataset. Compression and value sizes also change the balance.
3. A benchmark should include sufficient duration for maintenance to occur and should report pending work, not merely completed foreground requests.

■ 29.6.5 TECHNICAL — the engineer's version

1. Report workload distribution, key/value sizes, update ratio, point/range reads, cache state, compaction policy, durability settings and storage headroom. Keep acknowledged-write guarantees equivalent across comparisons. [S134] [S140]
2. Test restore and recovery alongside steady-state performance. An engine that meets latency targets but cannot meet the required recovery objective does not meet the complete application requirement.
3. The book's model develops the mechanisms; it is not a benchmark ranking of LSM and B-tree products. Both families have sophisticated implementations and workload-dependent trade-offs.

■ 29.6.6 WORDS — remember these

1. **Compaction backlog:** reorganization work waiting to be completed — pending maintenance that can increase read cost, space use or write stalls. **Steady state:** workload and maintenance remain sustainable together — an operating regime without unbounded growth of deferred work under the stated conditions. **Space amplification:** physical footprint exceeds live logical data — a ratio whose scope includes the selected files, versions and temporary maintenance state.

29.97 Practice and worked answers

1. **Compute write amplification.** One MB of logical input causes five MB of writes at the measured boundary. **Answer:** the ratio is 5; name whether WAL and device-internal writes are included.
2. **Read versions.** A has red at sequence 1 and green at sequence 3. **Answer:** latest reads green; a sequence-2 snapshot reads red under the model.
3. **Merge runs.** R1 has A1=red,C2=black; R2 has A3=green,B4=blue. **Answer:** latest ordered output is A=green,B=blue,C=black.
4. **Interpret filter work.** Three filters reject and one candidate block is read. **Answer:** one modeled block read, not necessarily one physical-device read.
5. **Find resurrection.** Drop A's sequence-3 tombstone while its sequence-1 value remains in another run. **Answer:** a current read can wrongly expose the old value.
6. **Allow latest-only cleanup.** All relevant runs are covered and no older snapshots are supported. **Answer:** the model can remove obsolete versions and a fully covered tombstone; the assumptions must remain explicit.
7. **Budget sustained work.** Ten MB/s logical input with amplification six needs sixty MB/s at the named boundary, but forty is available. **Answer:** demand exceeds that budget by twenty MB/s; temporary buffering does not solve the long-run mismatch.

8. **Limit the model.** Does the sorted-run exercise validate RocksDB crash recovery? **Answer:** no. It lacks real files, logging, manifest installation and concurrency.

29.98 Common wrong ideas

1. **Wrong:** LSM writes create no later work. **Right:** flushing and compaction move and rewrite data.
2. **Wrong:** immutable files mean eternal immutable history. **Right:** files and obsolete versions can be retired.
3. **Wrong:** finding any key occurrence gives its current value. **Right:** visibility and newer versions or tombstones matter.
4. **Wrong:** a filter's possible match proves a record exists. **Right:** verify against the actual entries.
5. **Wrong:** a tombstone can be discarded immediately. **Right:** older reachable values and snapshots can still require it.
6. **Wrong:** compaction always reduces instantaneous disk usage. **Right:** input and replacement files can coexist temporarily.
7. **Wrong:** a short burst benchmark demonstrates sustainable throughput. **Right:** deferred maintenance must keep up in steady state.
8. **Wrong:** LSM or B-tree is universally faster. **Right:** compare complete implementations under the actual workload and guarantees.

29.99 Chapter summary in 20 lines

1. LSM designs organize recent writes in memory and persistent sorted runs.
2. Batching changes the timing and shape of work rather than eliminating it.
3. Recovery logging protects recent work according to the selected durability contract.
4. Immutable files are replaced through controlled installation rather than ordinary in-place edits.
5. A manifest identifies which runs belong to the installed state.
6. Multiple versions of one key can coexist across memory and files.
7. A reader chooses the highest version visible to its snapshot.
8. Sorted runs support ordered search and merging.
9. Filters and range metadata can avoid some unnecessary probes.
10. Read amplification needs a defined metric and cache boundary.
11. Write amplification compares physical writes with logical input at a named layer.
12. Compaction reorganizes runs and removes only eligible obsolete versions.
13. A tombstone prevents an older deleted value from being returned.
14. Dropping a tombstone too early can resurrect old data.
15. Older snapshots can require versions that current readers no longer need.
16. Compaction can require temporary space for old and new files together.
17. Leveled and tiered strategies trade read, write and space costs differently.
18. Background maintenance must keep up with the sustained workload.
19. Measure backlog, latency, throughput and recovery, not just peak ingestion.
20. The educational run model is not a production storage engine or crash-safety proof.

Durability From Application to Device

30.0 What this chapter gives you

1. “Saved” can mean copied into an application buffer, accepted by the operating system, acknowledged by a storage device or preserved across a specified failure. Those meanings are not interchangeable.
2. This chapter follows the acknowledgement chain and teaches you to state exactly which failures a durability claim covers. It separates process failure, operating-system failure, power loss, device loss and correlated loss of copies.
3. The supplied exercises do not cut power, reset hardware or modify a user’s storage configuration. They test models and bounded local file/database operations, with their limits stated explicitly.

30.1 The acknowledgement chain

■ 30.1.1 PLAIN — in simple words

1. A user sees a success message only after several layers have exchanged requests and responses. Each response confirms something at its own boundary.
2. A server saying “accepted into memory” is different from a database saying “committed under this durability setting.” Both can be useful contracts, but the interface must not describe one as the other.
3. A lost response creates uncertainty. The operation may have completed before the reply disappeared, so the caller’s silence cannot be treated as proof of rollback.

■ 30.1.2 PLAIN — a picture in your head

1. A shop hands a document to a clerk, the clerk hands it to a filing department, and the department places it in a protected archive. Each handover can produce a receipt.
2. The first receipt does not prove that the archive step finished unless the organization explicitly waits for that boundary before issuing it.
3. **Where the comparison breaks:** software layers can buffer, reorder and batch work. The chain is a protocol, not necessarily one person waiting motionless at each step.

■ 30.1.3 PLAIN — a worked example

1. Define three API statuses for a hypothetical service: queued, locally committed and externally confirmed. Queued means the request is retained under the queue’s stated contract; locally committed means the database accepted the transaction; externally confirmed means a named external participant acknowledged its defined action.

2. A request can reach locally committed while the client never receives that response. It can also be locally committed while the outbox delivery remains pending.
3. The API should expose the original request identity so the caller can reconcile these states rather than creating an accidental second operation.
4. The status names are a proposed teaching contract, not claims about KedByte's live website or a deployed service.

■ 30.1.4 PLAIN — what is really happening inside

1. The application maps lower-level results into business responses. Incorrect mapping can weaken a strong database guarantee by showing success too early or can hide an already committed result behind a generic failure.
2. Database settings decide which lower-level persistence conditions COMMIT waits for. Replication settings can add remote acknowledgement requirements, each with its own meaning.
3. Document the exact success boundary and make tests observe it. A diagram saying “disk” without naming the promised failure survival is incomplete.

■ 30.1.5 TECHNICAL — the engineer's version

1. A durability contract should name the acknowledged operation, persistence boundary, fault model and assumptions about storage correctness. PostgreSQL's reliability documentation explicitly discusses the cache layers between memory and non-volatile storage. [S130]
2. Asynchronous commit intentionally changes when acknowledgement occurs relative to WAL persistence. It is not equivalent to ordinary durable commit merely because both statements eventually return success. [S136]
3. Request idempotency and outcome lookup address uncertain completion, not the physical persistence of the underlying storage. Both protocols are needed when the service requires both properties.

■ 30.1.6 WORDS — remember these

1. **Acknowledgement chain:** the sequence of accepted handovers — responses from application, database, operating system and storage layers with distinct meanings. **Success boundary:** the event a positive response promises has occurred — the precise acceptance or persistence condition represented to the caller. **Durability contract:** what accepted data is promised to survive — a guarantee defined against specified failures and storage assumptions.

30.2 Process memory and operating-system caches

■ 30.2.1 PLAIN — in simple words

1. A program can hold data in its own memory before passing it to the operating system. The operating system can then hold data in a cache before writing it to lower storage.
2. Ending one program does not necessarily erase the operating system's cache. Therefore a process-kill test is not the same as a sudden power-loss test.
3. Conversely, a successful buffered write call may only mean that bytes were copied into an intermediate buffer. The program must understand the interface it used.

■ 30.2.2 PLAIN — a picture in your head

1. Mira's notebook, the office's outgoing tray and the archive cabinet are three different places. Losing Mira's notebook does not destroy papers already in the office tray; losing the whole office's power-protected storage is another event.
2. Testing only the notebook's loss does not establish the archive's behaviour in every disaster.
3. **Where the comparison breaks:** memory and cache failures do not follow simple room boundaries. Hardware, kernel, filesystem and controller behaviour all matter.

■ 30.2.3 PLAIN — a worked example

1. A Python file object receives text in a user-space buffer. Calling its `flush()` pushes buffered output through the underlying stream interface; it is not by itself a universal guarantee that the device has durably stored the bytes.
2. On a supported ordinary file, an application can request a lower persistence boundary using the platform's synchronization API, such as `os.fsync()` after flushing the language buffer.
3. Even then, inspect errors and the platform contract. The API cannot promise survival of the complete loss of the only storage device holding the data.
4. This is an explanation of layers, not a replacement for a database engine's carefully designed logging and file-management protocol. [S43] [S64] [S141]

■ 30.2.4 PLAIN — what is really happening inside

1. Buffered interfaces combine small writes for efficiency. A lower write can be partial or fail later, so robust software cannot assume that every requested byte was accepted merely because it attempted one call.
2. Operating-system writeback may occur after the application has continued. Delayed errors need to be surfaced at the relevant synchronization or closing boundary according to the platform.
3. Database engines already implement these details under their supported configurations. Application authors should not bypass the engine by editing its files directly.

■ 30.2.5 TECHNICAL — the engineer's version

1. Linux `write()` can return fewer bytes than requested. Successful return does not, by itself, guarantee that the data has been committed to persistent storage. [S142]
2. Language-buffer flushing and file synchronization are separate operations. A correct low-level file protocol accounts for partial writes, synchronization errors, file metadata and directory entry persistence where required. [S141]
3. Tests must identify whether they terminate only a process, crash the operating system or remove power. The surviving cache layers differ between those fault models.

■ 30.2.6 WORDS — remember these

1. **User-space buffer:** data waiting inside the program or library — memory used to batch output before passing it to the operating system. **Page cache:** operating-system memory holding file data — a cache between application I/O and lower storage devices. **Partial write:** fewer bytes were accepted than requested — an I/O result requiring explicit handling rather than an assumption of complete transfer.

30.3 Flush requests

■ 30.3.1 PLAIN — in simple words

1. A flush request asks a layer to advance pending work toward a lower boundary. Different APIs use the word for different guarantees.
2. Synchronizing file contents does not always synchronize the directory entry that makes a newly created or renamed file discoverable after a crash.
3. Atomic visibility and durability are also different. A rename can appear all at once to concurrent readers while still requiring a persistence protocol for the promised crash behaviour.

■ 30.3.2 PLAIN — a picture in your head

1. The archive stores a document securely, but its catalogue card naming the document has not yet been secured. After an interruption, the document and its discoverable name can have different states.
2. A reliable filing procedure considers both the contents and the catalogue.
3. **Where the comparison breaks:** filesystem rules differ, and not every platform accepts the same directory-synchronization operations. Copying a Linux recipe into another operating system without verification can be wrong.

■ 30.3.3 PLAIN — a worked example

1. A conceptual safe-publication workflow writes a new file, verifies all intended bytes were written, synchronizes the file under the platform contract, installs its name using the appropriate atomic operation, and synchronizes required directory metadata.
2. If the process crashes before the name installation, readers should still use the previous installed version under the chosen design. If the installation completes durably, readers can use the new complete version.
3. The exact sequence and guarantees depend on the filesystem, operating system and overwrite semantics. This paragraph is a protocol outline, not executable cross-platform recovery code.
4. Database backup and storage tools should use their documented procedures rather than a home-made file replacement routine applied to a live database.

■ 30.3.4 PLAIN — what is really happening inside

1. The filesystem coordinates data and metadata, while the device may have its own caches and flush commands. The application observes a returned result, not the physical movement of each electron.
2. Error handling is part of correctness. A failed sync is not a warning to ignore while still reporting the promised durable success.
3. Repeated sync calls do not fix a fundamentally incompatible or dishonest lower storage contract. Verify the supported stack and its documented assumptions.

■ 30.3.5 TECHNICAL — the engineer's version

1. Linux `fsync()` synchronizes file data and associated metadata and waits for the device to report completion. Its manual explicitly notes that synchronizing a file does not necessarily synchronize its containing directory entry. [S141]

2. `fdasync()` has a related but narrower metadata requirement: it synchronizes metadata needed for subsequent data retrieval, rather than all unrelated metadata. This distinction is platform-specific API detail, not a generic database setting. [S141]
3. Do not confuse an I/O synchronization guarantee with application-level transaction atomicity. Persisting two independent files successfully does not make their updates one atomic business operation.

■ 30.3.6 WORDS — remember these

1. **File synchronization:** request persistence of pending file state — an operating-system operation with documented data, metadata and completion semantics. **Directory durability:** preserve the file's naming relationship — persistence of directory metadata needed to find the intended file after failure. **Atomic visibility:** observers do not see an intermediate replacement state — a concurrency property distinct from surviving a crash or power loss.

30.4 Device promises

■ 30.4.1 PLAIN — in simple words

1. A storage device can acknowledge a write while retaining data in a volatile cache unless the protocol and configuration require a stronger boundary.
2. Power-loss protection can help preserve certain cached state, but its scope must be understood. It does not protect against every device fault or the loss of the entire storage location.
3. A database depends on the lower stack honouring its promises. No database setting can make a failed or dishonest device become reliable merely by naming it durable.

■ 30.4.2 PLAIN — a picture in your head

1. A courier says a parcel is secured, but it is actually still in an unlocked van. Whether the receipt is trustworthy depends on what “secured” promised and whether the courier fulfilled it.
2. A backup power supply may keep the van's lock working, but it does not prevent the van itself from being destroyed.
3. **Where the comparison breaks:** device persistence has precise command semantics and firmware behaviour. Consumer metaphors cannot substitute for tested hardware and supported filesystem configurations.

■ 30.4.3 PLAIN — a worked example

1. A database writes an 8,192-byte page. Suppose the storage path can fail after only part of that page reaches its final location. The surviving page may contain a mixture of old and new portions.
2. This is a torn-page risk, not merely the loss of the entire latest page. Recovery needs sufficient information to reconstruct or detect the damaged state under its design.
3. PostgreSQL's full-page-image mechanism addresses partial-page-write recovery within its documented assumptions. A page checksum can detect some damage but does not itself contain the original page needed to repair it. [S130] [S137]
4. The numerical page size here matches the common PostgreSQL page size; it does not claim that all devices provide 8,192-byte atomic writes.

■ 30.4.4 PLAIN — what is really happening inside

1. Controllers, devices and filesystems can reorder or cache work. Flush and barrier semantics constrain that behaviour when correctly implemented and configured.
2. A battery-backed cache has maintenance requirements. An exhausted battery or an unverified firmware behaviour can invalidate assumptions that previously held.
3. Avoid casually disabling synchronization or barriers to obtain a better benchmark. First identify exactly which guarantee would change and whether the application can tolerate that loss.

■ 30.4.5 TECHNICAL — the engineer's version

1. PostgreSQL's reliability guidance discusses operating-system, controller and device caches, full-page writes and the need for a trustworthy storage subsystem. Those assumptions are part of its durability story. [S130]
2. Checksums support detection, not general reconstruction or authentication. A checksum-valid page can still encode a logically wrong business change. [S137]
3. The educational tests do not validate a device's flush implementation, controller battery, firmware, atomic-write unit or power-loss protection. Such claims require separate hardware-specific evidence.

■ 30.4.6 WORDS — remember these

1. **Volatile write cache:** acknowledged data still depends on power — intermediate storage that can lose pending writes when its power disappears. **Torn page:** only part of a page update survives — a mixed old/new physical page caused by an interrupted multi-part write. **Power-loss protection:** preserve specified device state during power interruption — a hardware capability whose scope and health must be verified.

30.5 Failure domains

■ 30.5.1 PLAIN — in simple words

1. Two copies are useful only to the extent that the failures you care about do not destroy both together.
2. Two files on one disk are separate filenames but share the disk's failure. Two disks in one machine share other risks, such as a mistaken deletion command or a stolen machine.
3. Replication can spread copies across machines while faithfully spreading an accidental deletion. Recovery requires both independent failure planning and a way to return to an earlier valid state.

■ 30.5.2 PLAIN — a picture in your head

1. Keeping two photocopies in the same folder helps if one sheet is smudged. It does not help if the folder is lost.
2. Keeping another copy elsewhere changes the risk, but someone must still verify that the remote copy is readable and current enough for its purpose.
3. **Where the comparison breaks:** digital copies can fail through shared credentials, software defects and automation, not only physical disasters. Geography alone does not make failures independent.

■ 30.5.3 PLAIN — a worked example

1. A hypothetical system has a primary database and a replica in the same building. It tolerates some individual-machine failures but not necessarily a building-wide outage.
2. Add a backup in a separate administrative and storage boundary. This can improve recovery options, but its access credentials, encryption keys, retention and restore procedure must also survive the incident.
3. If the same compromised administrator can erase the primary, replica and backup, the copies share a destructive authority domain despite their different locations.
4. The example is a design exercise, not a claim that any named provider guarantees independence between its products.

■ 30.5.4 PLAIN — what is really happening inside

1. Failure domains include processes, hosts, devices, racks, sites, regions, identities, software versions and operational procedures.
2. Independence is an assumption to test and document, not something proved by drawing two boxes on a diagram.
3. A complete recovery plan also includes dependencies such as configuration, secrets, keys and application versions. Restored database bytes can remain unusable if the only decryption key was lost.

■ 30.5.5 TECHNICAL — the engineer's version

1. Build a fault matrix listing which copies and dependencies survive each proposed failure. Distinguish availability during the fault from recoverability after it.
2. Include logical corruption and operator error as well as hardware loss. A replica of the latest state is not necessarily a restore point preceding a bad transaction. [S131]
3. Avoid multiplying independent-failure probabilities when the independence assumption is unsupported. Shared infrastructure and authority can dominate the practical risk.

■ 30.5.6 WORDS — remember these

1. **Failure domain:** components that can fail together — a shared physical, software, administrative or operational dependency boundary. **Correlated failure:** one cause affects several copies — a failure pattern that invalidates an assumption of independent redundancy. **Recovery dependency:** something required to make restored data usable — keys, configuration, software, access and other prerequisites beyond database bytes.

30.6 Testing a durability claim

■ 30.6.1 PLAIN — in simple words

1. A test should state the failure it injects, the data acknowledged before that failure and the evidence checked afterward.
2. A clean shutdown is not a crash test. A process kill is not a power cut. A restored in-memory copy is not an off-site disaster rehearsal.
3. Small tests are still useful when their claims stay small. The problem is not limited evidence; it is describing limited evidence as proof of a larger untested guarantee.

■ 30.6.2 PLAIN — a picture in your head

1. Testing whether a door closes is useful. It does not prove that the building survives a flood.
2. A test report becomes more useful when it says exactly which door, under which condition and with what observed result.
3. **Where the comparison breaks:** data systems can hide failure until later reads or reconciliation. “The program restarted” is only one observation in a recovery test.

■ 30.6.3 PLAIN — a worked example

1. Prepare a disposable database with known records and a separate record of acknowledged request identities. Run a bounded operation, induce the approved fault, then recover into an isolated inspection environment.
2. Check structural integrity, required relationships, acknowledged outcomes, request replay behaviour and business totals. Record missing or uncertain evidence explicitly.
3. The book’s actual companion performs in-memory rollback, backup/restore and logical surviving-prefix tests. It does not execute the hardware-fault procedure above.
4. A real power-loss test must use authorized disposable equipment and a reviewed safety procedure; this book does not ask readers to interrupt power to their ordinary computer or production database.

■ 30.6.4 PLAIN — what is really happening inside

1. A test harness must avoid becoming the source of misleading evidence. Recording acknowledgements only in the same volatile memory that the test destroys can leave no trustworthy oracle afterward.
2. Failure injection timing and reproducibility matter. Randomly killing a process once may miss the vulnerable window entirely.
3. Preserve logs and configuration together with the outcome. A test that passed under stronger synchronization settings does not validate a later weakened configuration.

■ 30.6.5 TECHNICAL — the engineer’s version

1. Separate safety properties, such as “no acknowledged durable transaction is lost under the stated fault,” from liveness and recovery-time properties. A test can establish one observation without proving all schedules or failures.
2. Record the exact engine/runtime, filesystem, device, settings, fault point, acknowledged set, recovered set and reconciliation procedure. Compare identities and values, not just row counts.
3. PostgreSQL and SQLite document storage and recovery assumptions that must remain part of the claim. Successful educational checks are not independent hardware certification. [S130] [S53] [S139]

■ 30.6.6 WORDS — remember these

1. **Failure injection:** deliberately create a specified fault — a controlled experiment against an authorized disposable system. **Recovery oracle:** independent evidence of the expected result — identities and values used to judge what should remain after the tested failure. **Durability evidence:** observations supporting a bounded persistence claim — a test record tied to configuration, fault model and verification procedure.

30.97 Practice and worked answers

1. **Read the response contract.** An API acknowledges “queued in memory.” **Answer:** this is not a promise of survival after process or power loss unless another documented mechanism establishes it.
2. **Separate buffers.** Python `flush()` succeeds. **Answer:** language-buffer progress alone is not a universal device-persistence guarantee.
3. **Handle a short write.** A low-level write returns 600 for a 1,000-byte request. **Answer:** only 600 bytes were reported accepted; the remaining bytes and error conditions require explicit handling.
4. **Find missing metadata.** A new file’s data is synchronized but its directory entry is not covered by the platform’s persistence procedure. **Answer:** content durability and discoverable-name durability may differ.
5. **Classify a torn page.** Half an updated page survives. **Answer:** recovery needs appropriate reconstruction information; a checksum alone can detect some damage but cannot recreate the lost original.
6. **Identify correlated authority.** One compromised credential can erase all three copies. **Answer:** the copies share a destructive administrative failure domain.
7. **Limit a process-kill test.** The process dies but the operating system continues. **Answer:** lower caches can survive; this is not a power-loss test.
8. **Choose a recovery oracle.** Compare only row counts after restore. **Answer:** counts can match while identities or values differ. Check the acknowledged set, relationships and defined business totals.

30.98 Common wrong ideas

1. **Wrong:** every success response means the same kind of saved. **Right:** name the acknowledgement and failure-survival boundary.
2. **Wrong:** a process kill and a power cut test the same thing. **Right:** different cache and storage layers survive.
3. **Wrong:** successful write means all bytes are permanently stored. **Right:** partial writes and persistence semantics must be handled explicitly.
4. **Wrong:** atomic rename automatically proves crash durability everywhere. **Right:** platform data and metadata synchronization rules still matter.
5. **Wrong:** power-loss protection prevents every storage failure. **Right:** its scope is narrower than total device or site survival.
6. **Wrong:** three copies are three independent failure domains. **Right:** infrastructure and authority can correlate their loss.
7. **Wrong:** a checksum repairs damaged data. **Right:** detection and reconstruction require different evidence.
8. **Wrong:** passing bounded local tests certifies production durability. **Right:** the tested configuration and fault model define the claim.

30.99 Chapter summary in 20 lines

1. Saved data passes through several acknowledgement boundaries.
2. The application must map those boundaries into honest business statuses.
3. A lost response can leave a committed operation unresolved to its caller.

4. User-space buffers, operating-system caches and device caches are separate layers.
5. Language-buffer flushing is not the same as device synchronization.
6. Low-level writes can be partial and require error handling.
7. File data and directory metadata have distinct persistence requirements.
8. Atomic visibility does not by itself establish durability.
9. Synchronization depends on the lower stack honouring its documented contract.
10. Volatile controller and device caches affect power-loss assumptions.
11. Partial-page writes require appropriate recovery protection.
12. Checksums detect some damage but do not reconstruct missing content.
13. Power-loss protection has a specific scope and maintenance requirements.
14. Copies can share physical, software and administrative failure domains.
15. Replication does not replace a recoverable earlier version.
16. Keys, configuration and application dependencies belong in recovery planning.
17. Tests must state the exact fault they inject.
18. Preserve an independent record of acknowledged identities and expected values.
19. Reconcile recovered meaning as well as structural validity.
20. Report bounded evidence without extending it to untested hardware or failures.

Backups, Restore and Point-in-Time Recovery

31.0 What this chapter gives you

1. A backup is a copy kept for a recovery purpose. Its value depends on whether the required state can actually be restored, validated and made usable within the intended limits.
2. This chapter distinguishes logical exports, physical backups, snapshots, log archives and restore rehearsals. You will calculate recovery objectives and identify dependencies that a database file alone does not contain.
3. The actual lab restores synthetic SQLite data into a fresh isolated destination. PostgreSQL point-in-time recovery is explained from its documentation, not claimed as an executed server restore.

31.1 Copies with a recovery purpose

■ 31.1.1 PLAIN — in simple words

1. A backup is not simply any second copy. It has a defined source, time or recovery boundary, retention policy and procedure for turning it back into usable data.
2. A replica usually follows current changes, including mistakes. A backup can preserve an earlier state needed after accidental deletion or corruption.
3. An export can be useful without containing everything required for a complete service recovery. Know what it includes and what it omits.

■ 31.1.2 PLAIN — a picture in your head

1. A photocopy of yesterday's ledger can help recover from today's mistaken erasure. A second clerk copying today's erasure immediately may leave two equally damaged ledgers.
2. The recovery copy needs a date, an identity and enough context to interpret its contents.
3. **Where the comparison breaks:** digital backups may contain physical engine structures, logical SQL or incremental changes. They are not interchangeable sheets that can be opened by any database version.

■ 31.1.3 PLAIN — a worked example

1. Mira exports the four canonical order lines. Their total is 28,650 paise across six units. That export can help reconstruct or verify those lines.
2. It does not automatically include customers, permissions, product metadata, outbox state, schema definitions, secrets or the application version needed to run the service.

3. A complete recovery inventory lists every required component and how it is acquired. Some dependencies should be backed up separately under stronger access restrictions rather than bundled into a public teaching archive.
4. The canonical export contains invented data only. No live customer records or credentials are included in this book's files.

■ 31.1.4 PLAIN — what is really happening inside

1. Logical backups describe objects and values in a form the target can reconstruct. Physical backups preserve the engine's storage representation under its supported consistency procedure.
2. The two approaches have different portability, granularity and recovery characteristics. A logical export can support selected-object restoration, while a physical base backup can participate in a matching WAL-replay recovery chain.
3. Backups themselves need monitoring. A scheduled job that stopped producing usable copies weeks ago is not a current recovery capability.

■ 31.1.5 TECHNICAL — the engineer's version

1. PostgreSQL distinguishes SQL dumps from filesystem-level base backups. `pg_dump` output is logical and is not a physical base for continuous WAL replay. [S67] [S131]
2. SQLite's online backup API supports making a consistent database copy through the engine. Copying only a live main file without the appropriate journal/WAL coordination can produce an invalid or incomplete backup. [S55] [S139]
3. A backup manifest should name source identity, engine version, acquisition method, completion state, checksums and required accompanying material. A filename containing a date is not sufficient provenance.

■ 31.1.6 WORDS — remember these

1. **Backup:** a copy retained for a defined recovery purpose — data and metadata acquired under a procedure that supports a stated restoration objective. **Logical backup:** reconstruct objects from their logical definitions and values — an export such as SQL or another supported logical representation. **Physical backup:** preserve an engine's storage representation — a copy requiring the matching consistency, version and recovery procedure.

31.2 Consistent snapshots

■ 31.2.1 PLAIN — in simple words

1. A recovery copy must represent a state the database can interpret consistently. Copying files one after another while they change can mix incompatible moments.
2. A supported snapshot or backup protocol establishes the required boundary, sometimes using recovery logs to reconcile files copied over an interval.
3. A storage snapshot is not automatically an application-consistent service snapshot. Other databases, object stores and external actions may have separate boundaries.

■ 31.2.2 PLAIN — a picture in your head

1. Copying the first half of a ledger before a transfer and the second half afterward can create an impossible combined total.

2. A controlled copying procedure either fixes the view or preserves the instructions needed to reconstruct the matching state.
3. **Where the comparison breaks:** database backups can be consistent even when file copying spans time, provided the engine's backup and log protocol supplies the required recovery evidence. They need not all be instantaneous photographs.

■ 31.2.3 PLAIN — a worked example

1. A toy service stores an order in one database and its attachment in an object store. The database backup records attachment key ATT-7, but the object-store snapshot predates ATT-7's upload.
2. Each component can be internally valid while the restored service contains a missing attachment reference.
3. The recovery plan must define the relationship: coordinate acquisition, retain referenced object versions or verify and reconcile missing objects under an explicit policy.
4. Replacing the missing attachment with a blank file would conceal the failure rather than restore the original evidence.

■ 31.2.4 PLAIN — what is really happening inside

1. Recovery consistency has layers: file-format validity, transaction consistency, cross-store references and business meaning.
2. Engine-supported backup procedures specify which logs, metadata and completion markers are required. A partially completed backup must not be labelled usable merely because many files exist.
3. The destination should be isolated while it is validated. Starting ordinary outbound workers immediately after restore can repeat historical notifications or other external effects.

■ 31.2.5 TECHNICAL — the engineer's version

1. PostgreSQL continuous archiving combines a supported physical base backup with the required WAL sequence. The base copy can span time because replay resolves the corresponding physical inconsistencies under the documented procedure. [S131]
2. SQLite's backup API operates through database connections rather than treating a changing file as an uncoordinated byte source. The companion uses this API for its actual synthetic restore test. [S55] [S86]
3. Define application-consistency checks beyond engine startup: foreign keys, expected objects, request/outbox relationships, attachment references and reconciliation totals.

■ 31.2.6 WORDS — remember these

1. **Consistent backup boundary:** the state the recovery procedure can validly reconstruct — a snapshot or log-supported acquisition point with matching metadata. **Application consistency:** related service components agree — correctness of references and workflows beyond one engine's internal file validity. **Restore isolation:** keep the recovered copy from affecting live systems — a controlled environment with outbound effects and production access disabled during validation.

31.3 Log retention

■ 31.3.1 PLAIN — in simple words

1. Point-in-time recovery starts from an earlier supported base and replays the required changes only as far as the chosen target.
2. The necessary log chain must be continuous. A missing required segment can prevent reaching the target even if newer segments remain available.
3. Retaining a base backup without its required logs, or retaining logs without a usable matching base, can leave a collection of files that cannot meet the promised recovery objective.

■ 31.3.2 PLAIN — a picture in your head

1. A verified ledger at Monday morning plus every accepted correction through Thursday can reconstruct Thursday's state. Missing Tuesday's corrections cannot be repaired by reading Friday's entries alone.
2. The base and the journal belong to one history, including any later branches created by recovery.
3. **Where the comparison breaks:** real log segments have engine-specific identities, timelines and validation rules. Sorting filenames lexically is not a recovery algorithm.

■ 31.3.3 PLAIN — a worked example

1. A synthetic recovery chain has a valid base at 10:00 and complete archived changes through 10:30. An incident occurs at 10:40.
2. The chain can support an appropriate target no later than its verified coverage, subject to the engine's transaction boundaries. It cannot claim recovery through 10:40 merely because the primary generated changes after 10:30.
3. If the intended target is 10:25, the plan must preserve every required part from the base through that target. A missing segment at 10:15 cannot be ignored.
4. These times illustrate coverage; actual WAL recovery targets and inclusive/exclusive settings require the documented engine procedure.

■ 31.3.4 PLAIN — what is really happening inside

1. Archiving is a transfer and retention workflow with its own failures. Generated WAL is not the same as successfully archived, verified and retrievable WAL.
2. Recovery can create a new timeline or history branch. Later backups and log references must identify which history they continue.
3. Retention deletion should evaluate dependencies before removing a base or log range. Deleting the oldest-looking file can break several otherwise retained restore points.

■ 31.3.5 TECHNICAL — the engineer's version

1. PostgreSQL PITR requires a continuous sequence of archived WAL extending back at least to the required start of the base backup. Its documentation separately explains recovery targets and timelines. [S131]
2. Record archive success, failures, coverage gaps and restore-tested ranges. A counter saying that some archive command succeeded does not prove every desired target remains recoverable. [S144]
3. Do not replay a physical log against a different logical export or unrelated cluster. Match base identity, version compatibility and timeline evidence before attempting recovery.

■ 31.3.6 WORDS — remember these

1. **Point-in-time recovery:** restore a supported earlier state — replay from a matching base to a chosen engine-supported recovery target. **Archive coverage:** the recovery history actually retained and retrievable — verified continuity of required log material over a stated range. **Recovery timeline:** a named branch of physical recovery history — metadata distinguishing histories after recovery or promotion under the engine's rules.

31.4 Recovery objectives

■ 31.4.1 PLAIN — in simple words

1. Recovery point objective, or RPO, describes the tolerated loss of recent data measured against a defined incident boundary. Recovery time objective, or RTO, describes how quickly the required service should be restored.
2. An objective is a requirement, not evidence that the current backup system achieves it. A restore rehearsal measures actual performance under stated conditions.
3. The service is not necessarily recovered when file copying finishes. Validation, application startup, access checks and controlled return to users can consume substantial time.

■ 31.4.2 PLAIN — a picture in your head

1. Mira says, “We can tolerate losing at most fifteen minutes of accepted work, and we need usable service within one hour.” These are two different targets.
2. Finding a recent ledger quickly does not help if nobody can unlock the cabinet or run the software that reads it.
3. **Where the comparison breaks:** recovery objectives need precise definitions for accepted work, incident start, degraded service and completion. Everyday phrases such as “back online” can hide incompatible meanings.

■ 31.4.3 PLAIN — a worked example

1. An incident occurs at 10:40. The verified recoverable state reaches 10:30, so the recent-data gap is ten minutes under this simplified timeline. That fits a fifteen-minute RPO only if the coverage and acceptance definitions truly match.
2. A rehearsal takes 32 minutes to restore, 18 minutes to validate and 12 minutes to return the required service safely. Total recovery time is 62 minutes.
3. A one-hour RTO is missed by two minutes. Reporting only the 32-minute copy time would conceal the operational result.
4. These invented values show how to calculate the measures; they are not a promise about the book's lab or any hosting plan.

■ 31.4.4 PLAIN — what is really happening inside

1. Restore time depends on data volume, retrieval bandwidth, decompression, replay, index work, validation and operational coordination.
2. A backup in low-cost archival storage can have a retrieval delay that dominates the recovery timeline. Keys or approvals can also become bottlenecks.
3. Objectives can differ by service function. A read-only emergency view may return before full write service, but the completion definition must state that distinction honestly.

■ 31.4.5 TECHNICAL — the engineer's version

1. A lower-bound transfer calculation for 1 TB decimal at a sustained 100 MB/s is 10,000 seconds, about 2.78 hours, before replay, validation or contention. It is not an end-to-end restore estimate.
2. Define RPO relative to acknowledged operations and verified recoverable history, not merely backup-job start times. Define RTO's start and finish events before measuring it.
3. Keep objectives, measured rehearsal results and residual assumptions in separate fields. A green backup-job status is not a measured RTO.

■ 31.4.6 WORDS — remember these

1. **RPO**: how much recent accepted work may be lost — a recovery-point requirement defined against a specified incident and acceptance boundary. **RTO**: how long required service restoration may take — a recovery-time requirement with explicit start and completion events. **Restore rehearsal**: actually exercise the recovery procedure — a controlled test measuring whether the required state and service can be recovered.

31.5 Restore rehearsals

■ 31.5.1 PLAIN — in simple words

1. A backup that has never been restored is an untested recovery input. Reading its filename and checking that it is large enough is not a substitute for trying the procedure.
2. Rehearse in isolation, verify the restored state and record the results. Keep outbound side effects disabled until their replay risks are understood.
3. Tests should include failure cases: missing files, wrong keys, incompatible versions and a target beyond the retained log coverage.

■ 31.5.2 PLAIN — a picture in your head

1. An emergency key is useful only if it opens the right door. A rehearsal tries it before the emergency rather than admiring the key's label.
2. The rehearsal also checks what is behind the door and whether the staff can perform the required work.
3. **Where the comparison breaks**: restoring historical state can reactivate pending jobs, old permissions or stale request records. A restored service can create new harm if connected to live systems without review.

■ 31.5.3 PLAIN — a worked example

1. The companion creates the canonical synthetic shop, makes an engine-supported SQLite backup into a fresh destination and checks that order totals remain O-1042=17,100 and O-1043=11,550 paise.
2. It also checks the combined 28,650-paise total, foreign-key violations and structural integrity under the available SQLite checks.
3. A later separate mutation of the source should not silently mutate the already completed backup destination. The test verifies separation under this local setup.
4. This is an actual bounded backup/restore exercise, not an off-site recovery rehearsal, encrypted-archive test or PostgreSQL PITR run.

■ 31.5.4 PLAIN — what is really happening inside

1. Structural verification and business verification answer different questions. A readable database can still contain the wrong target time or missing expected records.
2. Compare identities and values, not just totals: two offsetting errors can leave a sum unchanged. Include representative application workflows and boundary tests.
3. Record the exact backup input, software versions, procedure, timings and checks. A later successful restore from another file does not retroactively validate an earlier failed backup.

■ 31.5.5 TECHNICAL — the engineer's version

1. PostgreSQL `pg_verifybackup` checks a base backup against its manifest and performs defined WAL-related validation. Its documentation explicitly states that this does not replace test restores and application-data verification. [S138]
2. A restore acceptance report should include acquisition identity, hashes, target boundary, isolated environment, executed commands, checks, timing and unresolved findings.
3. Restored outbox and idempotency records require special review before external dispatch resumes. A backup can precede actions already performed outside the restored database; reconciliation must prevent unsafe replay.

■ 31.5.6 WORDS — remember these

1. **Backup verification:** check the retained input against defined expectations — structural, manifest or checksum validation that may precede a full restore. **Restore acceptance:** decide whether the recovered state meets its objective — a documented result based on actual recovery and relevant application checks. **Replay review:** inspect historical pending work before reactivation — reconciliation of restored tasks with effects that may already exist outside the backup boundary.

31.6 Access and retention of backups

■ 31.6.1 PLAIN — in simple words

1. Backups can contain sensitive information long after the live application has changed or deleted it. They need access controls, retention rules and a way to prevent old data from quietly returning to normal use.
2. Encryption helps protect stored bytes, but recovery also depends on keeping the required keys available to authorized responders.
3. Retention should preserve the required restore points without claiming that every copy lasts forever. Storage cost, purpose and deletion obligations need an explicit policy.

■ 31.6.2 PLAIN — a picture in your head

1. An old ledger in a locked archive is still a ledger containing people's information. Locking it does not make its contents disappear.
2. Destroying the only key may prevent recovery, while leaving a spare key on the door defeats the access control.
3. **Where the comparison breaks:** digital deletion and encryption-key lifecycle are complex across replicas, caches and services. A local file deletion is not proof that every retained copy was erased.

■ 31.6.3 PLAIN — a worked example

1. A teaching policy retains daily backups for 30 days and monthly backups for a separately defined period. The policy is an example, not legal advice or a universal retention recommendation.
2. A deletion request affects the live system today, while an older restricted backup still contains the record until its permitted retention expires.
3. A restore procedure must reapply the relevant deletion or suppression decisions before that old record becomes available to ordinary users again, under the organization's approved policy.
4. Keep evidence of which stores were checked and which retained exceptions remain. Do not report universal erasure based only on the live-table DELETE.

■ 31.6.4 PLAIN — what is really happening inside

1. Backup credentials should be scoped to their tasks. Separate the ability to create backups from the ability to erase every historical copy where the storage system and operational model support that separation.
2. Retention automation needs dependency checks so it does not delete logs required by retained base backups. It also needs monitoring for failures and unexpectedly retained data.
3. Recovery access should be rehearsed under realistic restrictions. A plan that works only with an unavailable administrator's personal credentials is operationally fragile.

■ 31.6.5 TECHNICAL — the engineer's version

1. Record backup classes, purpose, retention, encryption-key dependencies, allowed operators and restore-time suppression procedures. Chapter 48 develops lineage and deletion boundaries in more detail.
2. Protect manifests and integrity metadata against unauthorized replacement. A checksum stored beside a file can detect accidental damage but does not authenticate the file if an attacker can rewrite both.
3. The book does not prescribe a jurisdiction-specific retention period. Legal and contractual requirements must be checked for the actual organization and data classes; the technical protocol should expose those decisions rather than invent them.

■ 31.6.6 WORDS — remember these

1. **Backup retention:** how long recovery copies remain available — a policy governing restore-point preservation and eventual retirement. **Key dependency:** recovery requires particular cryptographic material — an access and availability requirement for decrypting retained data. **Restore-time suppression:** prevent retired data from re-entering ordinary use — reapplication of approved deletion or access decisions after historical recovery.

31.97 Practice and worked answers

1. **Classify an export.** A CSV contains only four order lines. **Answer:** it is a useful logical data extract, not automatically a complete service backup.
2. **Choose a PITR base.** Can a PostgreSQL SQL dump be used directly as the physical base for WAL replay? **Answer:** no; use the supported matching physical-backup procedure.
3. **Find cross-store inconsistency.** The database references ATT-7, absent from the object-store restore. **Answer:** component validity does not establish application consistency; reconcile the missing dependency.

4. **Calculate the data gap.** Incident at 10:40, verified coverage through 10:30. **Answer:** ten minutes under the stated acceptance and coverage assumptions.
5. **Calculate recovery time.** Restore 32 minutes, validate 18, return service 12. **Answer:** 62 minutes, missing a one-hour objective by two minutes.
6. **Compute a transfer floor.** One TB decimal at 100 MB/s. **Answer:** 10,000 seconds, about 2.78 hours before other recovery work.
7. **Interpret manifest success.** `pg_verifybackup` reports success. **Answer:** perform an actual test restore and application checks; the tool does not establish every recovery property.
8. **Review restored workers.** Historical outbox rows appear pending. **Answer:** keep dispatch isolated until external effects and replay identities are reconciled.

31.98 Common wrong ideas

1. **Wrong:** any second copy is a usable backup. **Right:** provenance, consistency and a tested restore procedure matter.
2. **Wrong:** replication protects against every mistaken deletion. **Right:** replicas can copy the mistake.
3. **Wrong:** individually valid component snapshots form a valid service snapshot. **Right:** cross-store references can disagree.
4. **Wrong:** newer WAL can compensate for missing required older segments. **Right:** the recovery chain must be continuous.
5. **Wrong:** RPO and RTO are achieved because they are written in a policy. **Right:** objectives need measured evidence.
6. **Wrong:** copying finished means recovery finished. **Right:** validation and safe service return also take time.
7. **Wrong:** matching row counts prove correct restoration. **Right:** identities, values and business relationships must be checked.
8. **Wrong:** deleting live data proves every backup copy is gone. **Right:** retained copies and restore-time controls need separate accounting.

31.99 Chapter summary in 20 lines

1. A backup has a defined recovery purpose and provenance.
2. Logical exports and physical backups support different procedures.
3. A PostgreSQL logical dump is not a physical WAL-replay base.
4. Supported backup interfaces coordinate changing data safely.
5. Cross-store application consistency extends beyond one database's validity.
6. Keep restored environments isolated while validating them.
7. Point-in-time recovery requires a matching base and continuous required logs.
8. Generated log data is not automatically archived and retrievable coverage.
9. Recovery timelines distinguish branches of physical history.
10. Retention deletion must preserve dependencies of retained restore points.
11. RPO defines tolerated recent-data loss under a stated boundary.
12. RTO defines the allowed time to restore required service.
13. Objectives are different from measured rehearsal results.
14. Transfer, replay, validation and controlled service return all consume time.

15. Manifest verification does not replace a test restore.
16. Check identities, values, constraints and business reconciliation after recovery.
17. Historical outbox tasks need replay review before external dispatch resumes.
18. Backups require access, encryption-key and retention controls.
19. Restoring old data must not silently undo approved deletion or suppression decisions.
20. Report exactly which recovery procedure was executed and which remains untested.

Corruption, Reconciliation and Repair

32.0 What this chapter gives you

1. A wrong answer can come from damaged bytes, an invalid relationship, a faulty query, a mistaken business entry or a misunderstanding of the report. These are different problems with different evidence.
2. This chapter teaches a disciplined path from suspected disagreement to verified diagnosis and bounded repair. It explains checksums, structural checks, reconciliation and evidence preservation without treating a repair command as a substitute for understanding.
3. The exercises use synthetic records and disposable byte strings. They do not modify a live database, bypass engine safeguards or claim to recover a user's damaged files.

32.1 Detecting disagreement

■ 32.1.1 PLAIN — in simple words

1. Disagreement is a reason to investigate, not a diagnosis by itself. A report total can be wrong even when every stored byte is intact.
2. Begin by stating the expected relationship and the observed difference. Then identify which layer could explain it: query, input, schema, transaction, storage or outside-world measurement.
3. Preserve uncertainty. The earlier expected-stock-10/observed-stock-9 example remains unexplained unless new evidence actually establishes a cause.

■ 32.1.2 PLAIN — a picture in your head

1. Two calculators show different totals. One may contain a different list, one may use a different tax rule, or one may be broken. The disagreement alone does not identify which.
2. Comparing the inputs and rules is often more informative than replacing the calculator immediately.
3. **Where the comparison breaks:** a database may have several simultaneous faults. Finding one query error does not prove that every underlying record is healthy.

■ 32.1.3 PLAIN — a worked example

1. The canonical agreed lines total 28,650 paise. A report joining lines to product tags produces 51,300 paise because some lines appear once per tag.
2. The difference is 22,650 paise, exactly the repeated notebook contribution in that fixture. The stored agreed prices need not be corrupted for the report to be wrong.
3. A repair aimed at reducing stored order totals to compensate for the report would damage correct source data. The query's grain and join must be fixed instead.

4. This demonstrates a verified logical cause in the synthetic fixture; it does not resolve any unrelated real inventory discrepancy.

■ 32.1.4 PLAIN — what is really happening inside

1. Reproduce the smallest relevant observation using a stable data boundary. A changing live dataset can make two correctly run queries disagree because they observed different states.
2. Keep the exact query, parameters, schema version and result. A screenshot of one number without its population or exclusions is weak evidence.
3. Classify findings separately: suspected, reproduced, explained, repaired and revalidated. These are stages, not interchangeable labels for “done.”

■ 32.1.5 TECHNICAL — the engineer’s version

1. Distinguish physical corruption, structural inconsistency, violated business invariants and incorrect derived queries. A diagnostic plan should select tests that discriminate among these classes.
2. A stable snapshot or isolated copy can support reproducible comparisons, subject to the acquisition procedure. Query plans and joins must be reviewed at their actual grain. [S58] [S66]
3. Do not modify authoritative rows merely to make a dashboard match a preferred number. Identify the source of authority and preserve the original evidence before proposing a correction.

■ 32.1.6 WORDS — remember these

1. **Disagreement:** two observations or expectations do not match — a finding that requires diagnosis rather than automatically proving storage damage. **Physical corruption:** stored representation is damaged or invalid — a failure involving bytes or engine structures rather than merely an undesirable business value. **Logical inconsistency:** records or calculations violate their intended meaning — a mismatch that can exist even when all files are structurally readable.

32.2 Checksums and their limits

■ 32.2.1 PLAIN — in simple words

1. A checksum or digest summarizes bytes so a later comparison can detect many changes. It answers a question about byte identity under the chosen algorithm.
2. It does not explain whether the original bytes were correct, whether a transaction was authorized or how to reconstruct missing information.
3. If someone can replace both a file and its stored checksum, a matching pair does not authenticate the file’s origin. Integrity comparison and trusted provenance are separate requirements.

■ 32.2.2 PLAIN — a picture in your head

1. A parcel’s inventory code helps detect that its contents changed after packing. It does not prove that the packer selected the right goods.
2. A dishonest person who repacks the parcel and rewrites the code can defeat a comparison against that same untrusted label.
3. **Where the comparison breaks:** cryptographic digests have formal security properties and collision limits. An everyday inventory code is only an analogy, not an implementation of those properties.

■ 32.2.3 PLAIN — a worked example

1. The lab hashes a fixed synthetic byte string, changes one byte in a separate copy and hashes it again. The observed digests differ for that fixture.
2. It also verifies that hashing the unchanged bytes reproduces the original digest. These checks establish deterministic behaviour and the detected alteration in the example.
3. They do not prove that no two different byte strings can ever share the digest or that the original string represented a true sale.
4. A trusted manifest must bind the expected digest to the intended source and version through an appropriate authority or custody process.

■ 32.2.4 PLAIN — what is really happening inside

1. Checksums are computed over a defined byte range. Omitted files, metadata or fields are outside that range's protection.
2. Database page checksums can detect some storage damage when pages are checked, but a logically valid wrong UPDATE can produce a perfectly checksum-valid page.
3. Verification timing matters. A successful check yesterday does not prove that a file is unchanged today unless the relevant bytes were checked again or protected by another justified mechanism.

■ 32.2.5 TECHNICAL — the engineer's version

1. PostgreSQL data checksums provide page-level corruption detection within their documented scope. They are not a general repair facility or a proof of business correctness. [S137]
2. Cryptographic digest comparison, such as SHA-256 through Python's `hashlib`, can identify exact artifact bytes when paired with a trusted expected digest. It does not supply authentication by itself. [S99] [S100]
3. Define the manifest's coverage and protect its provenance. Hashing a ZIP after delivery identifies that ZIP's bytes; it does not imply every internal claim or test result is true.

■ 32.2.6 WORDS — remember these

1. **Checksum coverage:** the bytes included in an integrity calculation — the exact range or file set a checksum can help verify. **Trusted expected digest:** an integrity reference with justified provenance — a digest bound to an intended artifact through a trusted source or process. **Authentication:** establish a source or actor under a security protocol — a property not supplied merely by an unkeyed matching checksum.

32.3 Structural checks

■ 32.3.1 PLAIN — in simple words

1. A structural check asks whether the database's internal representation satisfies the checker's defined rules. It can detect problems invisible to an ordinary SELECT.
2. Different checks cover different things. A general integrity check may not test every foreign-key relationship or every business invariant.
3. A successful result means the selected checks found no covered problem. It does not mean that every possible fault has been ruled out.

■ 32.3.2 PLAIN — a picture in your head

1. A building inspection can verify that shelves are stable without verifying that every form is filed under the right customer. A separate filing audit checks those relationships.
2. Both checks matter, and neither should be described as the other.
3. **Where the comparison breaks:** database checkers have precise implementation scopes, may require permissions and can consume substantial resources. Running every checker on a busy production system is not automatically safe.

■ 32.3.3 PLAIN — a worked example

1. In an isolated SQLite demonstration, a child row can exist without its intended parent if foreign-key enforcement was bypassed during insertion.
2. A general `PRAGMA integrity_check` and a separate `PRAGMA foreign_key_check` answer different questions. The earlier labs demonstrate that a structurally accepted database can still contain the orphan relationship.
3. Enabling foreign keys afterward does not retroactively create the missing parent or prove all old rows valid. Inspect existing relationships explicitly.
4. Repairing the orphan requires a business decision based on evidence; inventing a parent record just to silence the checker is not necessarily correct.

■ 32.3.4 PLAIN — what is really happening inside

1. Checkers traverse selected engine structures and compare them against invariants encoded in the checker.
2. They can miss application-level rules because those rules may not exist in the schema. A capacity total exceeding ten is invisible to a checker that only knows each row's positive quantity constraint.
3. Checker errors should be retained exactly, together with version and options. Summarizing them as "database broken" loses information needed for a bounded diagnosis.

■ 32.3.5 TECHNICAL — the engineer's version

1. SQLite documents `integrity_check` and `foreign_key_check` separately. Use both when their respective coverage is needed, while respecting the workload and acquisition boundary. [S75] [S59]
2. PostgreSQL `amcheck` provides defined table/index consistency checks. Its options, lock implications and limitations must be reviewed before use; it is not a universal repair command. [S146]
3. A checker's successful exit should be recorded with the exact object scope. Testing one index or one database does not certify every other store in the service.

■ 32.3.6 WORDS — remember these

1. **Structural integrity check:** inspect engine representation rules — a diagnostic procedure for covered page, tree or table/index invariants. **Referential check:** verify relationships between records — a test that child references resolve according to the schema's key rules. **Checker scope:** the objects and properties actually examined — the boundary limiting what a successful diagnostic result establishes.

32.4 Business reconciliation

■ 32.4.1 PLAIN — in simple words

1. Business reconciliation compares records that should agree under a defined rule. It can reveal missing, duplicated or misclassified work even when storage is structurally healthy.
2. Start with grain and inclusion rules. Comparing all-time accepted orders with today's delivered items creates disagreement even when both lists are correct for their own purpose.
3. A matching total is useful but not sufficient. Two opposite errors can cancel numerically while the wrong customers or products remain affected.

■ 32.4.2 PLAIN — a picture in your head

1. Mira compares each reservation slip with its stock allocation, not just the total number of slips in two piles.
2. Ten slips in each pile can still refer to different reservations.
3. **Where the comparison breaks:** real reconciliation may involve delayed arrivals and approved exceptions. An unmatched record is not automatically fraud, corruption or a completed failure until timing and scope are examined.

■ 32.4.3 PLAIN — a worked example

1. The canonical dataset has four agreed lines, six units and a total of 28,650 paise. Verify each complete (order_id, line_no) key and its product, quantity and agreed price.
2. Suppose one copied line is increased by 100 paise and another decreased by 100 paise. The combined total can still match while both line values are wrong.
3. A record-level comparison catches the mismatch. A total-only comparison does not.
4. For reservations, compare each accepted request to its reservation and stock effect. Keep rejected requests and pending delivery intents in their own categories rather than forcing every count to be equal.

■ 32.4.4 PLAIN — what is really happening inside

1. Reconciliation often joins independent representations at explicit identity boundaries. Missing matches, duplicate keys and incompatible values should be categorized separately.
2. Define the allowed timing lag and late-arrival policy. A recently committed order may legitimately await a downstream report refresh under the system's contract.
3. Keep the source of truth explicit for each field. A derived report cannot automatically overrule the original agreement merely because the report was generated later.

■ 32.4.5 TECHNICAL — the engineer's version

1. A reconciliation specification includes population, grain, keys, units, time boundary, expected equations, allowed exceptions and authoritative sources.
2. Use anti-joins or complete-key comparisons to identify missing and extra records before comparing aggregate totals. Chapter 13's join-grain rules apply directly. [S58]
3. Reconciliation findings should be reproducible from captured inputs or a defined snapshot. A live query whose population changes during investigation needs an explicit observation contract. [S66]

■ 32.4.6 WORDS — remember these

1. **Business reconciliation:** compare representations that should agree — an identity- and meaning-aware check of records, totals and state transitions. **Authoritative field source:** the record allowed to decide a disputed value — a defined authority for a particular fact, not necessarily one universal source for every field. **Offsetting errors:** wrong values cancel in an aggregate — discrepancies that preserve a total while corrupting individual records.

32.5 Evidence-preserving repair

■ 32.5.1 PLAIN — in simple words

1. Repair changes evidence. Before changing a damaged or disputed system, preserve enough of the original state to explain what was found and why the proposed correction is justified.
2. A repair should be bounded: identify the records, preconditions, exact changes, expected post-conditions and a safe stopping rule.
3. When the evidence does not establish the correct value, the repair must not invent one. Escalate the unresolved decision or restore from an appropriate verified source.

■ 32.5.2 PLAIN — a picture in your head

1. An archivist photographs a damaged page before repairing it and records which words were restored from a verified duplicate. They do not silently fill illegible words with a plausible story.
2. The repair record distinguishes observed text from reconstructed text and unresolved gaps.
3. **Where the comparison breaks:** copying a live database incorrectly can itself create an inconsistent artifact. Evidence acquisition needs a supported procedure that includes relevant recovery files and metadata.

■ 32.5.3 PLAIN — a worked example

1. In a disposable copy, identify one imported row whose source record and import trace prove that quantity 2 was mistakenly stored as 3. Preserve the source evidence, original row and acquisition identity.
2. Propose a conditional correction targeting the complete key and expected old value. Check that exactly the intended row changes and that related totals reconcile afterward.
3. If the old value no longer matches, stop and review rather than broadening the UPDATE until it affects something. The precondition protected the repair from silently overwriting intervening work.
4. This is a teaching repair pattern, not authorization to change the canonical agreed orders or any live customer record.

■ 32.5.4 PLAIN — what is really happening inside

1. Evidence preservation includes original files, logs, schema, version, configuration and observations relevant to the finding. A checksum records artifact identity but does not independently validate the diagnosis.
2. Some storage-level repairs can discard data to regain structural readability. Such a result must be reported as recovery with known loss, not a complete restoration.
3. Perform proposed changes on an isolated copy first where possible. Validate them before a separately authorized production intervention with a tested recovery path.

■ 32.5.5 TECHNICAL — the engineer's version

1. A repair manifest should contain input hashes, object scope, expected preimages, patch or statements, transaction boundary, postcondition checks, rollback/recovery plan and decision authority. Do not invent reviewer identities or approvals.
2. SQLite's corruption guidance warns about unsafe file operations, journal handling and interference with its locking assumptions. Preserve associated recovery material rather than deleting unfamiliar sidecar files. [S139]
3. Prefer restoring verified data or rebuilding derived structures from authoritative sources when that is the justified remedy. Never present a generic low-level edit as universally safe for an unknown damaged database.

■ 32.5.6 WORDS — remember these

1. **Preimage:** the exact state expected before a repair — captured values or bytes used to verify that a proposed change targets the reviewed input. **Repair manifest:** the record of a bounded correction — evidence, scope, changes, checks and authority supporting an intervention. **Evidence-preserving repair:** fix a problem without concealing its original state — a procedure retaining provenance and disclosing reconstructed or lost information.

32.6 Incident follow-through

■ 32.6.1 PLAIN — in simple words

1. A repair is not complete merely because the first error message disappeared. Check whether the cause remains, whether downstream copies are consistent and whether the same fault can recur.
2. Separate immediate containment, restored service, verified data correction and long-term prevention. They can finish at different times.
3. A useful incident record states facts, uncertainty and actions without assigning unsupported motives or blame.

■ 32.6.2 PLAIN — a picture in your head

1. Replacing a wet ledger page helps today, but a leaking roof can ruin the replacement tomorrow. The repair and the cause need separate attention.
2. Staff also need to know which reports used the damaged page before it was replaced.
3. **Where the comparison breaks:** software causes can be interacting conditions rather than one broken part. An incident may have several contributing factors and no single simple root cause.

■ 32.6.3 PLAIN — a worked example

1. A synthetic report is corrected from the tag-multiplied 51,300 paise to the properly scoped 28,650 paise. The source order lines remain unchanged.
2. Follow-through identifies other reports using the same faulty join, adds a regression fixture with multiple tags, and records which exported results may need correction.
3. The incident closes only after the agreed scope is checked and remaining unreviewed consumers are disclosed. Fixing one dashboard does not prove every downstream spreadsheet was replaced.
4. The original unresolved physical stock discrepancy remains a separate finding, not absorbed into this explained reporting error.

■ 32.6.4 PLAIN — what is really happening inside

1. Data flows create downstream effects: caches, reports, exports, messages and decisions. Lineage helps identify which outputs depended on the disputed input or query.
2. A regression test should reproduce the actual failure condition, not merely repeat the happy path that already passed before the incident.
3. Monitor the corrected invariant over an appropriate period and preserve the old and new definitions. Silent metric-definition changes can make a problem appear solved without changing the underlying behaviour.

■ 32.6.5 TECHNICAL — the engineer's version

1. An incident report should include detection, impact scope, evidence, timeline, containment, correction, validation, remaining uncertainty and prevention actions with owners. Owners are actual designated people or roles, not invented identities.
2. Track which claims were verified: byte integrity, structural validity, record-level reconciliation, downstream regeneration and operational recovery. Do not collapse them into one unqualified PASS.
3. The final book package's own validation follows the same principle: checksums identify files, tests exercise examples and layout checks inspect exports. None alone proves that every sentence is perfect or independently reviewed.

■ 32.6.6 WORDS — remember these

1. **Containment:** limit further harm while investigating — an action that can precede full diagnosis or repair. **Regression fixture:** data and steps reproducing a past failure — a test case intended to prevent recurrence of the specific defect. **Downstream impact:** outputs and decisions depending on disputed data — the propagation scope that must be reviewed after a correction.

32.97 Practice and worked answers

1. **Classify the inflated total.** Joining order lines to multiple tags yields 51,300 instead of 28,650 paise. **Answer:** the fixture demonstrates query-grain multiplication, not necessarily damaged stored prices.
2. **Limit a digest.** A file and its digest are both replaced by an attacker. **Answer:** their agreement does not authenticate the original source; trusted provenance is missing.
3. **Separate checks.** SQLite integrity checking reports no covered structural issue, but a foreign-key check finds an orphan. **Answer:** the checks have different scopes and can legitimately produce those results.
4. **Find offsetting errors.** Two values differ by +100 and -100. **Answer:** the total can match while record-level comparison fails.
5. **Protect a correction.** The reviewed old quantity is 3, but the live row now contains 4. **Answer:** stop the conditional repair and re-review; do not overwrite the intervening change blindly.
6. **Handle an unknown value.** No evidence establishes whether stock should be 9 or 10. **Answer:** retain the uncertainty and obtain a justified observation or decision rather than inventing a repair.
7. **Follow a query fix.** One dashboard is corrected. **Answer:** inspect other consumers, exports and regression coverage before claiming complete downstream correction.

8. **Interpret test success.** A checksum check, integrity check and query test all pass. **Answer:** each supports its bounded claim; together they still do not prove every business fact or unseen system copy is correct.

32.98 Common wrong ideas

1. **Wrong:** every wrong total means corrupted storage. **Right:** query grain, timing and meaning can produce disagreement with intact bytes.
2. **Wrong:** a checksum proves the original data was true. **Right:** it helps compare bytes against a reference.
3. **Wrong:** a checksum can reconstruct a missing page. **Right:** repair needs suitable redundant information or an authoritative source.
4. **Wrong:** one integrity checker covers all constraints. **Right:** structural, referential and business checks differ.
5. **Wrong:** equal totals prove equal records. **Right:** offsetting errors and mismatched identities can hide underneath.
6. **Wrong:** a plausible replacement is a justified repair. **Right:** evidence or explicit authority must support the correction.
7. **Wrong:** deleting recovery sidecars is harmless cleanup. **Right:** they may contain information required for consistency or diagnosis.
8. **Wrong:** removing the symptom closes the incident. **Right:** cause, downstream impact and recurrence need follow-through.

32.99 Chapter summary in 20 lines

1. Disagreement is an observation, not an automatic diagnosis of corruption.
2. Separate physical damage, structural inconsistency and wrong business meaning.
3. Capture exact queries, parameters and observation boundaries.
4. The tag-join fixture inflates a correct 28,650-paise source total to 51,300.
5. Repair the faulty derivation rather than damaging correct source records.
6. Checksums compare covered bytes against an expected reference.
7. Matching digests do not by themselves authenticate origin or truth.
8. Page checksums detect some damage but do not reconstruct lost content.
9. Structural and foreign-key checks answer different questions.
10. Business reconciliation requires explicit grain, scope, keys and units.
11. Matching totals can conceal offsetting record-level errors.
12. Preserve original evidence before proposing a repair.
13. Bind each correction to reviewed preimages and a complete identity.
14. Stop when repair preconditions no longer match.
15. Do not invent missing values to make a checker pass.
16. Preserve recovery files and use supported acquisition procedures.
17. Validate proposed repairs in isolation before separately authorized intervention.
18. Review downstream reports, caches, exports and decisions after correction.
19. Add regression fixtures reproducing the actual failure condition.
20. Close only the verified scope and keep unresolved findings visible.



PART F

Data Across Machines

Understand copies, partitions, coordination and the limits of promises made over a network.

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.

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.

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 $k \bmod 3$. 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 $k \bmod 4$. 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.

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, $|R \cap W| \geq |R| + |W| - N$. 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.

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 $\text{floor}(N/2)+1$ 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 — $\text{floor}(N/2)+1$ 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.

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.

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.



PART G

From Records to Insight

Build pipelines, analytical systems and search while preserving the meaning of the source data.

Data Pipelines: Capture, Transform and Deliver

40.0 What this chapter gives you

1. A data pipeline carries information from one representation or system into another. It may copy, validate, reshape or summarise the records, but it must not silently change what they mean.
2. You will follow a frozen export of the shop's four agreed order lines into a small reporting store. The questions are practical: Which source was read? What was accepted? What was changed? Where does progress stop after a failure? Can the result be reproduced?
3. The companion exercise is a local, bounded batch import. The discussion of change capture uses published PostgreSQL and Debezium documentation; it does not claim that a connector, Kafka cluster or production pipeline was deployed.

40.1 Sources and contracts

■ 40.1.1 PLAIN — in simple words

1. A pipeline needs an agreement about its input before it needs a schedule. That agreement describes what one record means, which fields identify it, which values are allowed and which changes can happen later.
2. "Sales data" is not a sufficient description. It could mean agreed orders, dispatched goods, payments or refunds. Those are different events, with different amounts and dates.
3. A receiving system should preserve enough context to explain each result. A total without its population, units, source boundary and transformation rule is difficult to verify, even when the arithmetic is correct.

■ 40.1.2 PLAIN — a picture in your head

1. Mira sends Dev a sealed envelope containing copies of order lines. On the envelope she writes which register supplied them, the export date and the rule used to choose them.
2. Dev writes a second note describing how he grouped the copies. Together, the two notes explain both what arrived and what he did with it.
3. **Where the comparison breaks:** a digital checksum can bind a report to exact bytes, but it cannot prove who created those bytes or that the source claims are true. Authentication and business verification remain separate responsibilities.

■ 40.1.3 PLAIN — a worked example

1. Our new teaching export, PIPE-01, contains the same four agreed lines introduced earlier. It is a frozen copy, not a claim about a live capture time. Its grain is one agreed order line, identified by (order_id, line_no).
2. The quantity unit is individual items. Prices are integer paise in INR. Multiplying each quantity by its agreed unit price gives 15,100, 2,000, 7,550 and 4,000 paise.
3. The input contract therefore predicts four distinct line keys, six units, two orders and 28,650 paise. These are useful reconciliation checks, not evidence that all possible real orders were included.
4. The contract deliberately excludes payments, discounts, taxes, shipping and refunds. A report derived from PIPE-01 must not acquire a label such as “net cash collected” merely because that label sounds useful.

■ 40.1.4 PLAIN — what is really happening inside

1. The producer serialises records according to a schema. The consumer decodes them, checks structural rules and then applies domain rules. These are different stages: valid JSON can still contain an invalid price.
2. An identifier for the schema or transformation helps the consumer select the right interpretation. A later field rename or unit change must not be inferred solely from a familiar-looking value.
3. Provenance records connect the input version, processing code, parameters and output version. This connection makes reproduction and investigation possible without pretending that provenance proves correctness.

■ 40.1.5 TECHNICAL — the engineer’s version

1. Define grain, identity, domain types, null semantics, ordering scope, update/delete representation and compatibility policy. A contract should also specify size bounds and what happens to malformed or conflicting input.
2. The pipeline manifest in this book records a byte digest and transformation version. The digest establishes content identity under the selected hash, not origin authenticity or semantic completeness. [S99]
3. Source-table changes and business events must not be conflated. Logical decoding exposes database changes; deciding that a change means “a customer paid” requires additional application semantics. [S166]

■ 40.1.6 WORDS — remember these

1. **Data contract:** the agreement about incoming information — explicit structure, semantics, identity and compatibility rules for a data interface. **Provenance:** the recorded origin and processing history — information linking source entities, transformations and resulting data. **Transformation version:** which processing rule was used — an identifier binding output to a particular implementation and its parameters.

40.2 Batch extraction

■ 40.2.1 PLAIN — in simple words

1. A batch is a bounded collection processed together. Its boundary may be a frozen file, a database snapshot or an explicitly closed interval of source events.

2. Reading a changing database in several pieces can produce a mixture that never existed at one moment. Each piece may be individually valid while the combination is misleading.
3. A successful extraction needs both complete coverage of its intended boundary and a consistent interpretation of that boundary. “The command finished” establishes neither by itself.

■ 40.2.2 PLAIN — a picture in your head

1. Dev photographs the first half of a noticeboard. Mira then replaces several notices before he photographs the second half. Joining the photographs does not create a reliable picture of the board at one time.
2. Freezing the board during both photographs would solve that particular problem. Keeping a versioned copy of the board would solve it differently.
3. **Where the comparison breaks:** database snapshot visibility can avoid stopping all writers. The exact isolation and export mechanisms are engine-specific, and a file copied outside those mechanisms is not automatically consistent.

■ 40.2.3 PLAIN — a worked example

1. Use a separate two-account teaching model, not the shop’s financial records. Initially A contains 60 units and B contains 40, for a total of 100.
2. An extractor reads A as 60. Another operation transfers 10 from A to B. The extractor then reads B as 50 and reports 110.
3. The source still contains 100. The error is the inconsistent observation boundary, not an addition mistake. A snapshot that sees either the complete before-state or complete after-state reports 100.
4. PIPE-01 avoids this race by using a fixed byte sequence as its input. That makes the local exercise repeatable, but it does not establish that an arbitrary real export was originally captured consistently.

■ 40.2.4 PLAIN — what is really happening inside

1. A database extractor issues queries under a particular transaction and isolation configuration. Different statements may see different committed versions unless the chosen mechanism supplies a suitable shared snapshot.
2. Pagination is another boundary. An offset into a changing result set can skip or repeat rows as earlier rows appear or disappear. A stable ordering key helps, but a key alone does not freeze the population.
3. A manifest should identify the selection predicate, ordering, snapshot or source cut, row count and any exclusions. When the source cannot provide a consistent cut, the limitation belongs in the result, not only in an operator’s memory.

■ 40.2.5 TECHNICAL — the engineer’s version

1. PostgreSQL 17 Read Committed gives statement-level snapshots; stronger transaction-level snapshot behaviour requires an appropriate isolation choice and operational design. A multi-statement export must account for that distinction. [S66]
2. Incremental extraction based only on an increasing row ID can miss updates and deletions to older rows. A timestamp cursor also needs tie-breaking, precision, clock and late-write assumptions.
3. This book’s immutable batch uses (batch_id, input_digest, transform_version) as declared identity. Reusing the batch ID with different input bytes is rejected, even when someone believes

the new bytes are equivalent. That is an original teaching policy, not a universal connector requirement.

■ 40.2.6 WORDS — remember these

1. **Extraction boundary:** exactly which source state or records were selected — the snapshot, interval or immutable object defining a batch's coverage. **Consistent cut:** observations that fit one permitted source history — a boundary that avoids incompatible mixtures of concurrent states. **Incremental extraction:** reading changes since recorded progress — a method whose cursor must account for the source's actual change semantics.

40.3 Change capture

■ 40.3.1 PLAIN — in simple words

1. Instead of repeatedly copying everything, a pipeline can follow changes: a row was inserted, updated or deleted. This is change data capture, usually shortened to CDC.
2. A new subscriber still needs a starting picture. Otherwise, it sees tomorrow's changes without knowing what existed today.
3. Joining that starting picture to the continuing changes is the difficult part. A gap loses information; an overlap can repeat information. Both need an explicit protocol rather than an approximate timestamp guess.

■ 40.3.2 PLAIN — a picture in your head

1. Dev copies the entire catalogue while Mira keeps a numbered list of changes. The copy has a clear marker saying which changes are already reflected in it.
2. Dev then follows the list from the agreed boundary. If a page of the list is missing, he cannot safely claim that his copy is current.
3. **Where the comparison breaks:** database logs can describe physical or logical operations rather than human catalogue edits. Transaction boundaries, replica identity and retention rules determine what a real connector can reconstruct.

■ 40.3.3 PLAIN — a worked example

1. In a separate model, a snapshot at sequence 100 says product X has quantity 5. Sequence 101 changes it to 3; sequence 102 deletes it.
2. Applying both changes after the snapshot leaves X absent. Starting at 102 and missing 101 happens to give the same final presence here, but would lose an intermediate change required by an audit-oriented consumer.
3. Starting at 103 misses the deletion and incorrectly retains X with quantity 5. A final-state check for this key now catches the gap, but a few spot checks cannot prove full coverage.
4. Replaying 101 after 102 must not recreate the old state. The consumer needs a suitable version/order policy, or an ordered replay mechanism that prevents this stale update from becoming current.

■ 40.3.4 PLAIN — what is really happening inside

1. A connector obtains a starting snapshot and records an associated log position. It then consumes changes from a boundary coordinated with that snapshot.

2. Source logs are finite operational resources. A stalled consumer can require old log segments to remain available, increasing storage use or eventually invalidating the ability to resume.
3. A delete requires enough identity to identify the destination record. A partial update may require previous state or configuration-specific fields. The consumer must understand the actual event envelope rather than treating every message as a complete new row.

■ 40.3.5 TECHNICAL — the engineer's version

1. Debezium's PostgreSQL connector documents a coordinated initial snapshot followed by log streaming. Snapshot modes, restart behaviour and captured fields are connector/version concerns, not properties of every pipeline called CDC. [S165]
2. PostgreSQL logical slots can resend recent changes after a crash because persisted progress may precede the last delivered position. Clients must avoid harmful repeated handling. A received message is not proof of a unique business effect. [S166]
3. PostgreSQL logical replication does not replicate schema DDL and sequence state in the same manner as table rows. Schema evolution and failover therefore require separate planning. Do not infer a complete database clone from a table-change feed. [S164]

■ 40.3.6 WORDS — remember these

1. **Change data capture:** following source mutations — obtaining inserts, updates and deletions for downstream processing. **Snapshot-to-stream handover:** joining the initial copy to later changes — a protocol preventing gaps and managing overlap at a capture boundary. **Replication slot:** retained source progress for a consumer — a PostgreSQL mechanism whose log retention and restart behaviour require monitoring.

40.4 Transformations

■ 40.4.1 PLAIN — in simple words

1. A transformation changes representation or derives a new value. It might convert units, choose columns, join descriptions or group lines into totals.
2. Some transformations lose detail. Once four lines become two order totals, the original line quantities cannot be recovered from the totals alone.
3. The processing rule must say what happens to missing, invalid and duplicate data. Silently turning all three into zero makes the output simpler by destroying distinctions the reader may need.

■ 40.4.2 PLAIN — a picture in your head

1. Dev sorts four order slips into two envelopes and writes a total on each. The envelope totals are useful, but they are not substitutes for every detail on the slips.
2. If one slip is unreadable, writing zero on it is an invented correction. Keeping it for review preserves the uncertainty instead.
3. **Where the comparison breaks:** software can retain exact source references and repeat the calculation automatically. That helps only when the code, inputs and policy are recorded and the process does not depend on unrecorded outside state.

■ 40.4.3 PLAIN — a worked example

1. Group PIPE-01 by order ID. O-1042 contributes 15,100 plus 2,000, giving 17,100 paise. O-1043 contributes 7,550 plus 4,000, giving 11,550.

2. The two totals sum to 28,650. Dividing by two orders gives 14,325 paise per order. Dividing by six items instead gives 4,775 paise per item. These answer different questions.
3. Joining the current notebook catalogue price of 8,250 into the calculation would produce the previously discussed hypothetical 30,750 total. That is not the historical agreed total and must not replace it.
4. Joining product tags before summing can repeat line values and produce 51,300. The transformation may be syntactically valid while changing the grain and therefore the answer.

■ 40.4.4 PLAIN — what is really happening inside

1. A deterministic transformation gives the same result for the same complete inputs and rules. Looking up today's exchange rate or catalogue value creates another input, even when it is not written in the main file.
2. Filtering changes the population. Joins can remove rows or multiply them. Rounding can make summing rounded parts differ from rounding a final exact sum.
3. A useful transformation record states the output grain, keys, units, retained exclusions and expected invariants. Its test cases include counterexamples, not only a clean happy-path batch.

■ 40.4.5 TECHNICAL — the engineer's version

1. Treat transformation code, configuration and reference data as versioned inputs. Reproducibility requires all dependencies affecting the result, not merely a script filename.
2. Relational joins and aggregation operate on the selected rows and multiplicities. `DISTINCT` is not a general repair for a wrong join: it can remove legitimate equal-valued facts. [S58] [S80]
3. The local reporting transformation uses integer arithmetic and original line identity. It does not infer a sale's financial, legal or physical completion from an order record. Those remain outside the teaching contract.

■ 40.4.6 WORDS — remember these

1. **Deterministic transformation:** the same complete inputs yield the same output — a rule without uncontrolled time, randomness or external-state dependencies. **Lossy transformation:** some source detail is discarded — a mapping that cannot generally be inverted to recover the original records. **Output grain:** one resulting row means one what — the level at which a transformed record states a fact.

40.5 Delivery and checkpoints

■ 40.5.1 PLAIN — in simple words

1. A checkpoint records where processing can safely resume. It must correspond to work actually preserved, not work merely started or displayed on a progress bar.
2. If the checkpoint moves ahead before the output is saved, a restart can skip missing output. If output is saved before progress is recorded, a restart can repeat it.
3. A correct design either commits output and progress together within a suitable boundary, or makes repeated delivery safe. The correct choice depends on which systems participate.

■ 40.5.2 PLAIN — a picture in your head

1. Dev ticks a slip after filing it. Ticking first risks losing the slip after the tick. Filing first risks filing it again after an interruption.

2. A numbered filing system can recognise an already filed slip and check whether the repeated contents agree. That turns some repeats into harmless replays.
3. **Where the comparison breaks:** a database transaction can bind several local records atomically. Two unrelated remote systems do not become one atomic filing cabinet merely because one program calls them in sequence.

■ 40.5.3 PLAIN — a worked example

1. Our local batch publisher begins a transaction, inserts the four reporting lines and records PIPE-01's identity in the same database. It commits only after all checks pass.
2. Inject an exception after the second insert. Rollback removes the new destination lines and the uncommitted batch record. The retry starts from the same frozen input, not from an invented "half complete" marker.
3. After a successful commit, submit PIPE-01 again with the same digest and transformation version. The stored result is returned without inserting four more lines or doubling 28,650 to 57,300.
4. Submit different bytes under PIPE-01. The publisher rejects the identity conflict. This does not overwrite earlier output, and it does not treat a matching batch label as permission to replace history.

■ 40.5.4 PLAIN — what is really happening inside

1. Local atomic publication stores destination data and the receipt for that data within one transaction. Unique keys constrain repeated insertion, while explicit payload checks distinguish replay from conflict.
2. Remote destinations require their own acknowledgement and replay rules. A checkpoint in a source connector does not automatically prove that every downstream report has incorporated the same prefix.
3. Partial progress also needs a defined scope. A checkpoint for partition A says nothing about partition B unless a higher-level checkpoint records both and explains their relationship.

■ 40.5.5 TECHNICAL — the engineer's version

1. Bind progress to the destination effect it claims. In the local exercise, the transaction owns both rows and batch receipt; no cross-system exactly-once guarantee is asserted. [S25] [S86]
2. A production streaming checkpoint may capture offsets, operator state and sink-commit information. End-to-end correctness depends on compatible source and sink protocols, not just a framework's internal recovery mechanism.
3. Retrying an external side effect requires stable request identity and a policy for unknown outcomes. The outbox/inbox construction from Chapter 26 applies when its local atomicity and consumer deduplication assumptions hold. [S125] [S126]

■ 40.5.6 WORDS — remember these

1. **Checkpoint:** recorded safe progress — a recovery marker bound to preserved state and explicitly scoped effects. **Atomic publication:** output becomes accepted as one local change — committing records and their completion evidence within one transaction boundary. **Replay-safe delivery:** repetition does not duplicate the intended effect — handling a repeated identity consistently while detecting conflicting payloads.

40.6 Reconciliation and replay

■ 40.6.1 PLAIN — in simple words

1. Reconciliation compares the source and result using checks chosen for the transformation. It asks whether the expected records and quantities survived, not merely whether a job reported success.
2. A matching total is useful but incomplete. Two wrong amounts can cancel each other. Missing one record and duplicating another can preserve the row count.
3. Replay rebuilds output from recorded inputs. It is a controlled experiment when the inputs and rules are fixed; it is a different computation when either changes.

■ 40.6.2 PLAIN — a picture in your head

1. Mira checks that Dev filed four slips, then checks their identifiers and values. Counting envelopes alone would miss a slip placed twice and another omitted.
2. She can ask him to repeat the grouping from the sealed copies. If the totals change, they can inspect the procedure rather than arguing from memory.
3. **Where the comparison breaks:** digital equality can compare every included key and value, but it still cannot reveal records that the source never captured or an undisclosed exclusion outside the manifest.

■ 40.6.3 PLAIN — a worked example

1. PIPE-01's report should have exactly the four source keys, quantities totalling six and agreed amounts totalling 28,650. Check each line's amount as well as the aggregate.
2. In a deliberately damaged copy, add 100 paise to one line total and subtract 100 from another. The grand total stays 28,650, but keyed comparison reveals both disagreements.
3. Rebuild the clean output with the same input digest and transformation version. Compare ordered key/value records or a canonical representation, not a hash of unspecified row order.
4. A proposed new report definition should receive a new transformation version and its own results. Keep a comparison explaining the difference rather than relabelling the old report as though it always meant the new thing.

■ 40.6.4 PLAIN — what is really happening inside

1. Reconciliation combines coverage checks, uniqueness checks, value comparisons and domain invariants. The right combination follows the contract: a filtered report should reconcile its included and excluded populations separately.
2. A reproducible run stores parameters, versions and output evidence. This allows failures to be investigated without touching the original source records or silently changing the test expectations.
3. Operational monitoring tracks lag, rejected input, unavailable log history, replay conflicts and destination failures. A green schedule is not enough when useful data is steadily falling behind.

■ 40.6.5 TECHNICAL — the engineer's version

1. Record source and destination watermarks with their meanings. A maximum observed source position is not necessarily a contiguous applied prefix, and offsets from different streams are not universally comparable integers.

2. Use keyed reconciliation to supplement aggregate checks. Canonical serialisation must specify field order, type representation, text encoding and row order before a digest can meaningfully compare runs. [S45] [S99]
3. The lab's failure injection and replay checks establish local behaviour for frozen teaching inputs. They do not test connector failover, source-log loss, unbounded streams, external delivery or every possible malformed-file attack.

■ 40.6.6 WORDS — remember these

1. **Reconciliation:** comparing expected and observed information — a set of coverage, identity, value and invariant checks across representations. **Contiguous progress:** every required earlier position is included — a stronger condition than merely having seen a large position number. **Re-processing:** applying rules again to retained inputs — a controlled rebuild whose interpretation depends on input and transformation versions.

40.97 Practice and worked answers

1. **Question:** Why is a pipeline labelled “sales” ambiguous? **Answer:** Orders, dispatches, payments and refunds describe different facts. Specify which event or state is included before assigning the metric a name.
2. **Question:** An extractor reads 60 from A and later 50 from B after a 10-unit transfer. What failed? **Answer:** The observation boundary mixed before and after states. Addition is correct; the claimed population is not one consistent state.
3. **Question:** Can an increasing primary-key cursor capture all changes? **Answer:** Not by itself. Updates and deletions to existing smaller keys can be missed.
4. **Question:** Why does the batch receipt share a transaction with reporting rows? **Answer:** A receipt must not claim completion without the output, and a retry must recognise already committed output without duplicating it.
5. **Question:** How many rows and paise should a replay of PIPE-01 add? **Answer:** Zero new rows and zero new amount when the stored batch identity agrees. The retained four rows still total 28,650.
6. **Question:** Does a matching grand total prove record equality? **Answer:** No. Opposite 100-paise errors cancel. Compare complete keys and values as well as aggregates.
7. **Question:** Does a source connector's checkpoint prove a dashboard is current? **Answer:** No. Every downstream stage has its own applied progress and publication boundary.
8. **Question:** What should change when the transformation definition changes? **Answer:** Its version and derived result identity, with a recorded comparison and a deliberate replacement or coexistence policy.

40.98 Common wrong ideas

1. **Wrong:** A pipeline is just a scheduled copy. **Right:** It also carries contracts, progress, failure policy and evidence about meaning.
2. **Wrong:** A checksum authenticates the producer. **Right:** It identifies bytes under a hash; authenticity needs a separate trust mechanism.
3. **Wrong:** Every valid input record belongs in the report. **Right:** Eligibility follows the declared population and purpose.

4. **Wrong:** A snapshot followed by any recent log position is sufficient. **Right:** The boundary must prevent gaps and handle overlap.
5. **Wrong:** Repeating a job is harmless because it reads the same file. **Right:** Destination effects need replay safety too.
6. **Wrong:** CDC captures the full meaning of a business action. **Right:** Row changes require application interpretation.
7. **Wrong:** Matching counts and totals prove all records are correct. **Right:** Keyed differences can remain hidden.
8. **Wrong:** A successful local import proves production delivery guarantees. **Right:** It proves only the bounded behaviours actually exercised.

40.99 Chapter summary in 20 lines

1. A pipeline moves and transforms information under a contract.
2. Define source grain before writing extraction code.
3. Preserve units, identity and missing-value meaning.
4. A digest binds bytes, not truth or origin.
5. A batch needs an explicit observation boundary.
6. Changing sources can produce inconsistent multi-query exports.
7. An increasing ID alone does not capture every mutation.
8. CDC follows inserts, updates and deletions.
9. An initial snapshot must connect correctly to the change stream.
10. Log retention limits how a consumer can resume.
11. Redelivery is a normal recovery possibility.
12. Transformations can discard detail and change grain.
13. Historical prices are not today's catalogue prices.
14. Record all inputs that affect a derived answer.
15. Progress must correspond to preserved output.
16. One local transaction can bind output and its receipt.
17. Remote effects need additional delivery and replay rules.
18. Reconcile keys and values as well as counts and totals.
19. Reprocessing with new rules creates a new result version.
20. State exactly which pipeline guarantees the evidence demonstrates.

Streams, Event Time and Late Arrivals

41.0 What this chapter gives you

1. A stream keeps bringing records while a question is being answered. There may be no natural final record telling the system that a day's information is complete.
2. You will separate the time an event claims to describe from the time a processor receives it. Then you will build windowed answers that can be revised when information arrives late, without counting a replay twice.
3. The chapter uses an original deterministic time-window model. Its explicit watermarks, lateness limits and output revisions are teaching policies, not an implementation or performance test of Apache Beam or another distributed stream processor.

41.1 Continuous records

■ 41.1.1 PLAIN — in simple words

1. A stream is a continuing sequence of records. A till, delivery scanner or application can keep producing new information while yesterday's records are still being processed.
2. Continuous arrival does not guarantee continuous progress. A source can pause, a connection can fail, and a processor can fall behind while the producer keeps working.
3. Before processing a stream, define what one record represents and how repetitions, corrections and gaps are recognised. Those responsibilities do not disappear when a system is described as real-time.

■ 41.1.2 PLAIN — a picture in your head

1. Mira receives numbered slips through a slot while she is counting the previous slips. There is no sign saying that the last slip has arrived forever.
2. She can still produce a useful interim answer, provided she says which slips it includes and what might change later.
3. **Where the comparison breaks:** a distributed stream can have several independently ordered inputs rather than one slot. A position in one input is not automatically a position in a single global sequence.

■ 41.1.3 PLAIN — a worked example

1. Suppose source A has processed records through position 50 and source B through position 80. These positions belong to different sequences and cannot be added to mean "global position 130."
2. An honest progress record is a pair: A through 50 and B through 80, together with any required gap checks or source-generation identifiers.

3. If A stops sending, B may reach 500 while A remains at 50. A large maximum position does not establish that a report using both sources is complete.
4. A report can either wait, publish a provisional result with the lag disclosed, or follow an explicit incomplete-data policy. The choice changes the product's meaning, not just its speed.

■ 41.1.4 PLAIN — what is really happening inside

1. A processor reads records, updates state and emits results. Durable progress and state recovery determine which work is repeated after a failure.
2. Partitioning permits parallel work, but each partition may advance independently. Coordination is needed when one answer depends on several partitions.
3. A bounded batch can be viewed as a finite collection. An unbounded stream needs continuing policies for time, state size, late input and result revision rather than assuming that all data will eventually fit in memory.

■ 41.1.5 TECHNICAL — the engineer's version

1. Apache Beam distinguishes bounded and unbounded collections and provides windowing and triggers for grouping and emitting results over continuing input. The runtime details depend on the selected runner and connectors. [S167]
2. Progress is typically partition-scoped. Include stream identity, generation and offset semantics when designing recovery; a numeric offset alone can be ambiguous after recreation or migration.
3. The local exercises use finite sequences to expose streaming rules reproducibly. They do not claim that a finite successful run establishes unbounded operational stability or distributed fault tolerance.

■ 41.1.6 WORDS — remember these

1. **Stream:** an ongoing supply of records — input processed as a continuing sequence rather than only as a completed finite batch. **Unbounded input:** no known final record — a collection whose completion is not generally available during normal operation. **Partition progress:** how far one input segment has advanced — a scoped position that must not be confused with universal completion.

41.2 Event and processing time

■ 41.2.1 PLAIN — in simple words

1. Event time is the time attached to the event being described. Processing time is when a particular system handles the record.
2. They can differ because of offline devices, queues, retries, transmission delays or incorrect source clocks. A late arrival is not necessarily a late real-world event.
3. Choose which time answers the question. “What happened before noon?” and “What did the server learn before noon?” are both useful, but they are different reports.

■ 41.2.2 PLAIN — a picture in your head

1. A delivery driver writes 10:05 on a paper receipt and returns to the shop at 12:20. The delivery time and the time Mira receives the receipt are different facts.
2. Filing the receipt under 12:20 may help explain office workload. Using it to claim the delivery happened at 12:20 would change the event's meaning.

3. **Where the comparison breaks:** a source timestamp may come from a poorly set or manipulated clock. Preserving it does not require treating it as unquestionably accurate.

■ 41.2.3 PLAIN — a worked example

1. Consider a separate stream fixture with integer minutes from an agreed origin. E1 has event time 2 and value 100; E2 has event time 7 and value 200.
2. The processor receives both before it announces completion progress for minute 10. Later it receives E3, whose event time is 6 and value is 50.
3. By event time, all three belong to the first ten-minute interval. By arrival order, E3 follows the first published answer. Both descriptions can be stored without rewriting either fact.
4. These values are invented abstract amounts, not additional sales in the canonical shop ledger. The example must not increase the earlier agreed total of 28,650 paise.

■ 41.2.4 PLAIN — what is really happening inside

1. Timestamp assignment may come from source fields, transport metadata or a processor's clock. The assignment rule is part of the pipeline contract.
2. Conversion must preserve the relevant time zone and precision. A local date without a zone cannot always be mapped to one instant, and a clock reading is not a causal ordering proof.
3. Implausible future timestamps can distort progress estimates if the system blindly uses the largest observed event time. Validating timestamps and recording clock uncertainty is therefore an operational concern, not only a presentation choice.

■ 41.2.5 TECHNICAL — the engineer's version

1. Event-time semantics require a defined timestamp extractor and temporal domain. Processing-time semantics instead follow the executing system's clock and can change when the same input is replayed. [S167]
2. Preserve original event time, ingestion time and processing metadata when their distinct meanings matter. Do not overwrite an inconvenient source timestamp with arrival time and keep the old label.
3. Ordering by timestamp is not equivalent to happened-before. Clock error and concurrent events remain possible, so causal or per-entity ordering may require other identifiers or sequence information. [S41]

■ 41.2.6 WORDS — remember these

1. **Event time:** when the described event is said to have happened — a timestamp assigned under the source and pipeline's semantic contract. **Processing time:** when a processor handles a record — time measured by the executing system, potentially different on replay. **Ingestion time:** when a system accepted the input — a separately useful observation that does not replace the original event time.

41.3 Windows

■ 41.3.1 PLAIN — in simple words

1. A window gives a bounded group over which a stream can answer a question. It might mean each ten-minute interval or each period of activity separated by a long gap.

2. Window membership needs exact boundaries. Otherwise an event precisely at 10:00 can be counted twice or in neither interval.
3. Windowing and publication are separate. Deciding which group contains a record does not decide when the result should be shown or whether it may later change.

■ 41.3.2 PLAIN — a picture in your head

1. Mira places time-stamped slips into trays labelled 09:00 up to, but not including, 09:10; then 09:10 up to, but not including, 09:20.
2. A slip at exactly 09:10 belongs to the second tray. The labels make the handover unambiguous.
3. **Where the comparison breaks:** sliding windows can deliberately place one event into several groups. Session windows can merge after a late event connects periods that previously looked separate.

■ 41.3.3 PLAIN — a worked example

1. With fixed windows of width 10, event times 2, 6 and 7 belong to $[0, 10)$. Time 10 belongs to $[10, 20)$. The square bracket includes the start; the round bracket excludes the end.
2. For non-negative integer time t , the window start is $(t // 10) * 10$. At $t = 27$, integer division gives 2, so the window is $[20, 30)$.
3. With width 10 and a new sliding window every 5 minutes, time 7 belongs to both $[0, 10)$ and $[5, 15)$. Summing the totals of overlapping windows would count some events more than once.
4. In a separate session rule, each event creates an interval $[t, t + 5)$, and overlapping intervals merge. Events at 1 and 4 form $[1, 9)$. An event at 12 forms $[12, 17)$. A late event at 8 bridges them into $[1, 17)$.

■ 41.3.4 PLAIN — what is really happening inside

1. A window function assigns each record to one or more windows. Grouping usually also includes an entity or report key, so one branch's total is not accidentally combined with another's.
2. Fixed windows simplify alignment. Sliding windows answer overlapping recent-history questions. Session windows follow bursts of activity and require rules for merging state and revising previously emitted results.
3. Calendar windows need additional definitions. A local day is not always a fixed duration in regions with clock changes. For a business report, the relevant zone and calendar policy belong in the metric definition.

■ 41.3.5 TECHNICAL — the engineer's version

1. Window assignment, trigger policy, accumulation mode and allowed lateness are distinct configuration dimensions. Matching only the window width does not make two streaming reports equivalent. [S167]
2. The local model supports fixed ten-unit windows and explicitly non-negative integer event times. Its session example is a separate interval-union calculation, not a claim to implement every streaming session-merge protocol.
3. Key the published result by report definition, entity scope and window boundary. A revision identifier can then distinguish successive answers to the same question without treating them as independent amounts to add.

■ 41.3.6 WORDS — remember these

1. **Window:** a defined group of time-related records — a membership rule used to bound aggregation over a stream. **Sliding window:** overlapping intervals — windows whose width exceeds their step, allowing one event to contribute to multiple results. **Session window:** a group linked by activity gaps — a window that may merge when an arriving event connects earlier groups.

41.4 Watermarks

■ 41.4.1 PLAIN — in simple words

1. A watermark is a progress statement about event time. It helps a processor decide that the ordinary waiting period for earlier records has passed.
2. It is not a magical observation of every device in the world. Its strength depends on the source and the assumptions used to produce it.
3. A record arriving behind the watermark is late under that policy. It may still describe a valid event, so “late” and “wrong” must not be used as synonyms.

■ 41.4.2 PLAIN — a picture in your head

1. Mira announces that the normal courier has delivered slips through the morning interval. She publishes a provisional count while keeping a procedure for exceptional late envelopes.
2. If a branch is offline, she must decide whether to wait for it or disclose that the count can still change.
3. **Where the comparison breaks:** some sources can offer stronger progress guarantees than an estimated courier schedule. Other watermarks are heuristic estimates. Their meanings must be read from the actual source and framework contract.

■ 41.4.3 PLAIN — a worked example

1. In our model, an external test driver advances a non-decreasing watermark. When it reaches 10, the model publishes revision 1 for $[0, 10)$ using E1 and E2: $100 + 200 = 300$.
2. Advancing the watermark is a declared input to the exercise. The model does not derive it from a network, a real clock or an unbounded source.
3. Suppose two required input partitions have watermarks 20 and 7. A conservative combined frontier is 7, because the second partition has not provided the same progress evidence as the first.
4. Marking the slow partition idle may permit progress under a chosen framework policy. When it resumes with older records, those records still need a late-data policy; idleness does not make them impossible.

■ 41.4.4 PLAIN — what is really happening inside

1. Sources and operators maintain progress estimates or guarantees. Downstream stages combine them with outstanding work, timers and state dependencies.
2. A watermark should not move backward within the same progress history. Recovered or recreated sources also need identity and state rules so that an old position is not mistaken for current progress.
3. Monitoring should distinguish event-time lag, processing backlog and idle inputs. A processor may be fast at handling available records while a disconnected source makes its answer incomplete.

■ 41.4.5 TECHNICAL — the engineer's version

1. Beam describes a watermark as its notion of when data for earlier event times can be expected to have arrived. Its guide separates this from trigger configuration and allowed late input. [S167]
2. A common estimate based on observed event time minus a delay is a policy, not a proof of source completeness. A forged future timestamp can advance such an estimate unsafely unless the system applies appropriate validation and source-specific safeguards.
3. Our integer model rejects a decreasing watermark and records late dispositions. Those checks establish internal consistency of the chosen model, not the truth of an externally supplied progress claim.

■ 41.4.6 WORDS — remember these

1. **Watermark:** a statement of event-time progress — a source- and framework-dependent frontier used for window and timer decisions. **Late data:** a record behind the accepted progress frontier — input whose event time precedes the relevant watermark or closed-window boundary. **Idle input:** a source currently producing no records — a condition requiring a policy, not proof that earlier events can never return.

41.5 Late corrections

■ 41.5.1 PLAIN — in simple words

1. Publishing quickly and waiting for every possible late record are competing goals. A system must choose how long to accept revisions and what to do after that limit.
2. A later result should say whether it replaces the earlier result or adds a difference to it. Mixing those meanings can double a report even when each emitted number is correct.
3. Repeated delivery of the same event is different from a new late event. Recognising the difference requires identity and payload checks, not only comparing timestamps.

■ 41.5.2 PLAIN — a picture in your head

1. Mira writes “morning total: 300, revision 1.” A late slip adds 50, so she writes “morning total: 350, revision 2.”
2. Dev must replace the old total with 350. Adding the two displayed totals would produce 650, even though the actual included amounts sum to only 350.
3. **Where the comparison breaks:** systems can instead emit a +50 change. Both replacement and delta designs can work, but the consumer must know which protocol it receives and how repeats are handled.

■ 41.5.3 PLAIN — a worked example

1. The model permits late updates while the watermark is below window end plus 5. For $[0, 10)$, the retention cutoff is therefore 15.
2. At watermark 11, E3 arrives with event time 6 and value 50. The model accepts it, changes the stored sum from 300 to 350 and publishes replacement revision 2.
3. Repeating E3 with the same identity and content creates no revision and no additional 50. Reusing E3 with value 70 is a conflict, not a correction. A genuine correction needs its own explicit identity and relation to the earlier event.

4. At watermark 15, a new E4 with event time 8 and value 25 is beyond the model's acceptance boundary. It is recorded as too late and does not silently change 350 to 375. This equality-at-cutoff rule is our declared policy.

■ 41.5.4 PLAIN — what is really happening inside

1. A trigger decides when to emit. Accumulating output may contain all accepted data so far; discarding output may contain only a new pane's contribution. Consumers need the matching interpretation.
2. Revisions need stable window identity and ordered version handling. Receiving revision 2 before revision 1 must not allow the older replacement to overwrite the newer answer.
3. Records rejected by the streaming lateness policy may still enter a separately governed batch correction process. That process must preserve the difference between the earlier published view and a later restatement.

■ 41.5.5 TECHNICAL — the engineer's version

1. Beam's trigger and accumulation settings affect when and how output panes are emitted. Allowed lateness provides a bounded opportunity to react to records after the watermark passes a window end. These settings must be coordinated with sink semantics. [S167]
2. Our publisher uses (`window_start`, `revision`, `replacement_total`) and a remembered event ID/payload mapping. Duplicate identity with identical content is a replay; conflicting content raises an error without mutating the accepted state.
3. The exercise retains final summaries and disposition evidence for inspection. It is not an unbounded deduplication store or a complete state-expiration implementation. Real retention must account for replay horizons and privacy requirements.

■ 41.5.6 WORDS — remember these

1. **Trigger:** the rule deciding when to publish — a condition for emitting a window result or pane. **Replacement result:** a new complete answer for the same key — output that supersedes an earlier revision rather than adding to it. **Allowed lateness:** the accepted revision interval — a configured policy governing how long late window data may affect retained computation.

41.6 State, checkpoints and replay

■ 41.6.1 PLAIN — in simple words

1. A streaming answer depends on remembered state: partial totals, seen identities, window boundaries and progress. Saving only the total can lose the information needed to resume safely.
2. A restart may repeat records. If the restored deduplication state is older than the restored total, a repeated event can be added again.
3. Recovery should restore a compatible combination of input progress, state and output publication evidence. Separate components must agree on what the checkpoint actually covers.

■ 41.6.2 PLAIN — a picture in your head

1. Mira copies her total into a notebook but forgets to copy the list of slips already counted. After returning from a break, she cannot reliably distinguish counted slips from new ones.
2. Saving the total and the counted-slip list together avoids that particular mismatch. She also needs to know which displayed revision customers may already have seen.

3. **Where the comparison breaks:** distributed systems may checkpoint many operators and coordinate with transactional sinks. A single notebook illustrates the dependency but does not solve the distributed checkpoint protocol.

■ 41.6.3 PLAIN — a worked example

1. After accepting E3, the local state contains sum 350, revision 2, seen identities E1/E2/E3 and watermark 11. Snapshot that complete model state.
2. Restore it into a fresh model and replay E3 unchanged. The result remains 350 at revision 2. A new valid E5 in the same window can still be accepted while the watermark remains below 15.
3. Now imagine restoring sum 350 but only seen identities E1/E2. E3 would look new and could incorrectly produce 400. This counterexample explains why state components must share a recovery boundary.
4. If an external reader already received revision 2 before a crash, resending revision 2 should be recognised as the same result. Saving internal state alone does not prevent a remote consumer from adding a replacement twice.

■ 41.6.4 PLAIN — what is really happening inside

1. Checkpointing serialises the state required to continue the computation. Source replay then reconstructs work after that checkpoint under the same rules.
2. State schema evolution matters too. A newer program must either understand the older checkpoint format or use a reviewed migration/rebuild process.
3. End-to-end correctness includes the sink. A framework can restore its own state consistently while an ordinary external API repeats a side effect that was acknowledged ambiguously.

■ 41.6.5 TECHNICAL — the engineer's version

1. The model snapshot includes its schema version, width, lateness policy, watermark, per-window state and accepted event identities. Restore validates the representation rather than executing arbitrary serialized objects.
2. Use bounded, explicit data encodings for untrusted or persisted state. Python pickle is not a safe interchange format for untrusted inputs because loading it can execute code. [S62]
3. The companion tests cover deterministic replay and revision handling on finite inputs. They do not exercise a distributed checkpoint coordinator, exactly-once external delivery, clock synchronization or recovery from physical power loss.

■ 41.6.6 WORDS — remember these

1. **Operator state:** remembered information used by a computation — partial aggregates, identities, timers and metadata needed to process later input. **State checkpoint:** a recoverable state boundary — a saved representation coordinated with the progress and effects it claims. **Restatement:** a later version of an earlier report — a deliberate correction whose relationship to previously published results remains visible.

41.97 Practice and worked answers

1. **Question:** Does a record received at 12:20 necessarily describe a 12:20 event? **Answer:** No. Event and receipt times are distinct, and the source clock's reliability must also be considered.

2. **Question:** Where does event time 10 belong with fixed width 10? **Answer:** In $[10, 20)$, not $[0, 10)$, under the declared half-open convention.
3. **Question:** Why should overlapping sliding-window totals not simply be added? **Answer:** The same event may belong to several windows, producing deliberate overlap.
4. **Question:** The two input watermarks are 20 and 7. What is the conservative combined frontier? **Answer:** 7, when both inputs are required and no separate idleness policy changes the interpretation.
5. **Question:** What replaces revision 1's 300 after E3 adds 50? **Answer:** Revision 2's complete total 350. Adding replacement totals would be incorrect.
6. **Question:** What happens to E4 at watermark 15 in this model? **Answer:** It is recorded as too late for the first window and does not alter the published total.
7. **Question:** Why save seen event IDs together with totals? **Answer:** Restoring inconsistent versions can cause already included events to be counted again.
8. **Question:** Does a passing deterministic window test prove a source watermark is truthful? **Answer:** No. The test checks the model's response to supplied progress, not the external source's completeness.

41.98 Common wrong ideas

1. **Wrong:** Real-time means no delay. **Right:** Continuing systems still have queues, unavailable inputs and processing lag.
2. **Wrong:** Arrival time and event time are interchangeable. **Right:** They answer different questions.
3. **Wrong:** A watermark makes late records impossible. **Right:** Its meaning depends on the source's guarantee or estimate.
4. **Wrong:** Window membership determines publication timing. **Right:** Triggers and revision policy are separate choices.
5. **Wrong:** Every result pane is an additional amount. **Right:** Some panes replace earlier results.
6. **Wrong:** The same timestamp proves duplicate identity. **Right:** Different events can share a timestamp, and replays can arrive later.
7. **Wrong:** Keeping a final sum is enough for recovery. **Right:** Progress, identities and publication state may also be required.
8. **Wrong:** A framework's state recovery automatically makes an external effect exactly once. **Right:** The sink and retry protocol must support the intended guarantee.

41.99 Chapter summary in 20 lines

1. Streams can continue without a natural final record.
2. Define record identity and correction meaning before processing.
3. Progress is scoped to its source and partition.
4. Event time describes the event under a timestamp contract.
5. Processing time describes the processor's handling of a record.
6. Source clocks can be wrong or manipulated.
7. Windows assign records to bounded groups.
8. Half-open boundaries prevent accidental double membership at an edge.
9. Sliding windows deliberately overlap.

10. Session windows may merge after a bridging arrival.
11. Watermarks express event-time progress under stated assumptions.
12. Slow or idle inputs need an explicit policy.
13. Late does not necessarily mean invalid.
14. Triggers decide when results are emitted.
15. Consumers must distinguish replacements from deltas.
16. Stable identities separate replays from new events.
17. A lateness cutoff bounds revisions, not real-world truth.
18. Recover totals, identities and progress consistently.
19. External sinks have their own acknowledgement and replay limits.
20. Preserve the history of published answers and later restatements.

Warehouses, Lakes and Analytical Tables

42.0 What this chapter gives you

1. The database that accepts one customer's order and the system that studies five years of orders solve different workloads. They may share data, but they need not share every storage layout, query pattern or operating boundary.
2. You will distinguish an analytical model from the files holding it, then connect facts, dimensions, catalogues and snapshots. The aim is to understand responsibilities rather than memorize product labels.
3. The examples retain the shop's four agreed lines and introduce separately labelled analytical fixtures. No warehouse service, object-storage account or Iceberg engine is deployed by the companion exercises.

42.1 Operational versus analytical work

■ 42.1.1 PLAIN — in simple words

1. Operational work asks small questions while doing the business: Can this order be accepted? Which items are available? Has this request already succeeded?
2. Analytical work asks questions across many records: How did quantities vary by month? Which report definitions disagree? What changed after a process revision?
3. The same underlying facts can support both, but a large report should not accidentally prevent the shop from accepting orders. Sharing data creates a need to manage resource use and freshness expectations.

■ 42.1.2 PLAIN — a picture in your head

1. Mira's counter register is arranged for finding and changing one order quickly. Her annual review uses copies spread across a large table so she can compare many months at once.
2. Doing the annual review directly on the busy counter would obstruct customers, even if every calculation were correct.
3. **Where the comparison breaks:** one database can support mixed workloads with suitable design. Separate operational and analytical systems are an architectural choice, not a law requiring every small shop to buy two products.

■ 42.1.3 PLAIN — a worked example

1. Suppose the counter issues 100 short requests per second, while a monthly report scans 50 million historical lines. These are invented workload assumptions, not measurements of KedByte's systems.

2. The report's acceptable completion time might be minutes, while a counter request has a much shorter service objective. Running both without resource controls can make their interests conflict.
3. A reporting copy could isolate some work, but it introduces a delay between accepted orders and report availability. A label such as "data applied through 10:30" becomes part of the report's interpretation.
4. The canonical four-line exercise is much smaller. It does not justify a distributed warehouse. Its value is teaching how the responsibilities change as the workload and organization grow.

■ 42.1.4 PLAIN — what is really happening inside

1. Operational designs often emphasize constrained updates and targeted access. Analytical designs often emphasize scanning selected columns, joining descriptive data and aggregating large populations.
2. Copying data separates some resource and failure boundaries, but adds capture, reconciliation and access-management duties. A second system is not a free performance improvement.
3. Materialized results can reduce repeated computation. Their usefulness depends on whether the refresh policy is compatible with the question's required freshness.

■ 42.1.5 TECHNICAL — the engineer's version

1. OLTP and OLAP are workload categories, not universal product classes. Characterize transaction size, scan volume, concurrency, update patterns, latency targets and consistency requirements before choosing an architecture.
2. PostgreSQL materialized views store query results for later access and may be stale until refreshed. A materialized result is not automatically synchronized with every source change. [S169]
3. Isolation through replicas or separate stores must be evaluated against capture lag, query visibility and recovery procedures. A report's source cut is part of its correctness claim, not merely an operations statistic.

■ 42.1.6 WORDS — remember these

1. **Operational workload:** queries and changes that run the business — work often focused on small, current, constrained transactions. **Analytical workload:** queries that study populations of records — work often involving scans, grouping and historical comparison. **Materialized view:** a stored query result — derived data whose refresh policy determines how current it is.

42.2 Warehouse modelling

■ 42.2.1 PLAIN — in simple words

1. An analytical model organizes measurable facts and the descriptions used to group them. An order-line quantity is a fact; a product category is a description that might help group facts.
2. Start by saying what one fact row means. A row for an order line cannot be mixed casually with a row for an entire order or a daily stock snapshot.
3. Historical descriptions need a policy. A customer changing category today does not automatically mean that last year's report should use the new category.

■ 42.2.2 PLAIN — a picture in your head

1. Mira places each order slip beside cards describing its product, branch and date. The slip holds the agreed quantity and price; the cards help arrange the slips for different questions.

2. If a descriptive card changes, she must decide whether old slips keep their earlier description or use today's classification.
3. **Where the comparison breaks:** a dimension join can multiply rows when several description versions match one fact. Correct analytical modelling requires enforced or checked uniqueness of the matching interpretation.

■ 42.2.3 PLAIN — a worked example

1. Define the fact table's grain as one agreed order line. The four canonical lines retain quantities 2, 1, 1 and 2 and line amounts 15,100, 2,000, 7,550 and 4,000 paise.
2. Grouping by product gives notebook quantity 3 and amount 22,650, and pen quantity 3 and amount 6,000. These still reconcile to six items and 28,650 paise.
3. Introduce a separate customer C-DEMO whose category is Retail for times [0, 100) and Trade from 100 onward. A synthetic fact at time 90 matches Retail; a fact at 110 matches Trade under an event-time classification policy.
4. If the Retail interval accidentally ends at 120, the time-110 fact matches both versions. A join now duplicates it. The defect lies in overlapping validity intervals, not in the addition function.

■ 42.2.4 PLAIN — what is really happening inside

1. A star-shaped model places a fact table near descriptive dimension tables. Keys connect the fact to one intended dimension interpretation.
2. Some dimensions overwrite attributes and answer questions using current descriptions. Others retain versions with validity boundaries, often called slowly changing dimensions. The selected history policy changes the answer.
3. Measures also have aggregation rules. Line amounts can be added within compatible currency and scope. A daily closing-stock value cannot be summed across days and called current stock, because each day is another observation of a state.

■ 42.2.5 TECHNICAL — the engineer's version

1. Declare fact grain, dimensional keys, measurement units and additive scope. A measure may be additive across products but not across time or incompatible currencies; a ratio usually requires preserving its numerator and denominator.
2. In the versioned-dimension example, enforce or validate one matching interval for each fact's key and reference time. A surrogate dimension key identifies a version but does not prove that its business-time assignment is correct.
3. Relational joins preserve the multiplicities implied by their matching conditions. The model's correctness therefore includes dimension coverage and non-overlap checks, not only a foreign key to an existing dimension row. [S58] [S74]

■ 42.2.6 WORDS — remember these

1. **Fact table:** measurable observations at a declared grain — analytical records whose keys and units define what may be aggregated. **Dimension:** descriptive context for grouping facts — attributes connected to facts under a current or historical interpretation. **Slowly changing dimension:** a policy for evolving descriptions — a modelling approach that may overwrite attributes or preserve dated versions.

42.3 Files and catalogues

■ 42.3.1 PLAIN — in simple words

1. A collection of files does not automatically form a reliable table. A reader needs to know which files belong, what their fields mean and which versions are current.
2. A catalogue supplies organized metadata about tables and their locations or definitions. It is a map, not the data itself and not automatically an access-control system.
3. “Lake” commonly describes a storage approach built around collections of data files. “Warehouse” commonly emphasizes managed analytical tables and queries. Real architectures can combine these ideas, so labels alone do not settle capabilities.

■ 42.3.2 PLAIN — a picture in your head

1. A storeroom contains boxes of order copies. The boxes have value, but a researcher cannot know which ones form the approved September collection without a catalogue.
2. A catalogue entry says which boxes to read and which schema explains their contents. It can also distinguish an old collection from a new one.
3. **Where the comparison breaks:** files can be uploaded concurrently, replaced or left orphaned after failed work. A directory listing is not necessarily the same thing as an atomic table publication protocol.

■ 42.3.3 PLAIN — a worked example

1. Suppose a table version consists of immutable files F1 and F2. A writer prepares replacement files F3 and F4 but has not yet published the new table metadata.
2. A reader that scans every visible file can accidentally combine F1, F2, F3 and F4, duplicating facts. A reader using the committed manifest reads only the files named by its chosen table version.
3. An abandoned upload F5 should not become part of the table simply because it exists beside the other files. It needs a successful publication boundary or a cleanup decision.
4. These file labels are an original conceptual model. The exercise does not rely on claims about a particular cloud provider’s directory-listing consistency or object-storage API.

■ 42.3.4 PLAIN — what is really happening inside

1. Data files store encoded records. Table metadata identifies schema, partitioning and the files or manifests making up a table state. A catalogue helps locate and update that metadata.
2. Readers follow a selected committed metadata version rather than guessing from whatever objects happen to be visible. This separates preparation from publication.
3. Storage permissions, catalogue permissions and query-engine permissions may be different boundaries. A user denied access through a dashboard might still read raw files if the underlying storage is broadly exposed.

■ 42.3.5 TECHNICAL — the engineer’s version

1. Apache Iceberg specifies table metadata, snapshots and manifests above data-file formats. It distinguishes the logical table state from individual stored objects. [S168]
2. Apache Parquet specifies a columnar file representation with metadata and column chunks. A Parquet file alone does not provide a multi-file transaction manager or a universal business schema. [S170]

3. Evaluate catalogues for identity, atomic update support, concurrency handling, permissions and interoperability. An architectural label such as lakehouse is not itself evidence that each requirement has been implemented and tested.

■ 42.3.6 WORDS — remember these

1. **Catalogue:** organized metadata about data assets — a service or representation locating tables and describing their definitions and versions. **Manifest:** an explicit inventory for a data version — metadata naming the files or entries that belong to a committed state. **Orphan file:** a stored object not referenced by an accepted table state — prepared or abandoned data requiring a deliberate retention or cleanup policy.

42.4 Table snapshots

■ 42.4.1 PLAIN — in simple words

1. A table snapshot identifies one coherent version of an analytical table. A reader can continue using that version while a writer prepares another.
2. Publishing a new snapshot should not accidentally discard another writer's accepted work. The update needs a rule for detecting that its starting version has become stale.
3. Keeping older snapshots allows historical reading only while the required files and metadata still exist and access remains permitted. A snapshot name is not an eternal recovery guarantee.

■ 42.4.2 PLAIN — a picture in your head

1. Mira publishes a numbered catalogue page listing the boxes for edition 10. Dev reads that page while she prepares edition 11 elsewhere.
2. Another assistant also starts from edition 10. Before publishing, both must check whether edition 10 is still the current starting point; otherwise the second publication could erase the first assistant's additions.
3. **Where the comparison breaks:** real table systems use storage and catalogue-specific atomic operations and conflict checks. A paper edition number explains the condition but does not implement distributed concurrency control.

■ 42.4.3 PLAIN — a worked example

1. Our metadata model starts at version 10 with files {F1, F2}. Writer A prepares {F1, F2, F3} based on version 10 and publishes version 11.
2. Writer B prepares {F1, F2, F4}, also based on 10. An unconditional replacement would lose F3 from the current table view.
3. A compare-and-set publication checks that the current version still equals the expected base. B's expected 10 no longer matches 11, so the stale publication is rejected without changing the current manifest.
4. B can review the new state and prepare a compatible result. Whether appending F4 can be safely combined depends on the operation and conflict policy; not every failed transaction can be retried by blindly taking a set union.

■ 42.4.4 PLAIN — what is really happening inside

1. Immutable data and metadata versions permit readers to hold a stable view while writers produce new objects. The accepted metadata pointer selects which version becomes current.

2. Field identity matters during schema changes. A field renamed from `unit_price` to `agreed_unit_price` should not be confused with a newly introduced field that happens to reuse an old name or position.
3. Expiration and cleanup remove historical states according to policy. A file still required by a retained snapshot cannot simply be deleted because the latest snapshot no longer uses it.

■ 42.4.5 TECHNICAL — the engineer's version

1. Iceberg uses field IDs to preserve column identity across schema naming and ordering changes. Its metadata-commit operation must atomically replace the version on which a change was based; the catalogue-specific atomic mechanism is outside the core format specification. [S168]
2. The local compare-and-set manifest example tests a stale-writer condition only. It does not implement Iceberg's file-level conflict validation, delete files, catalogues, storage durability or full schema evolution.
3. A time-travel query is bounded by retained metadata and files, supported reader versions and permissions. Recovery, audit retention and deletion obligations require separate design decisions beyond retaining a snapshot list.

■ 42.4.6 WORDS — remember these

1. **Table snapshot:** a coherent table version — metadata selecting the files and interpretation visible to a reader. **Atomic metadata publication:** one accepted version replaces its expected predecessor — a concurrency boundary preventing incompatible partial or stale updates. **Field ID:** stable column identity independent of a display name — metadata used to preserve meaning through supported schema changes.

42.5 Governance boundaries

■ 42.5.1 PLAIN — in simple words

1. Copying data for analysis creates another place where it must be protected, explained and eventually retained or removed according to policy.
2. A clean dashboard does not prove that the raw files, temporary exports and query logs are equally controlled. Governance follows the information through all relevant representations.
3. Ownership should identify who defines a metric, who may change its meaning and who handles a disagreement. Otherwise two carefully built reports can become competing versions of the truth without an agreed resolution process.

■ 42.5.2 PLAIN — a picture in your head

1. Mira locks the main register but leaves its photocopies on an open table. Protecting the original does not protect the copies.
2. She also needs labels explaining whether a report is provisional, corrected or approved for a particular purpose. A polished cover is not a governance state.
3. **Where the comparison breaks:** digital data can be copied into caches, notebooks and exported files automatically. A complete inventory needs technical discovery and process ownership, not only visible office shelves.

■ 42.5.3 PLAIN — a worked example

1. An analyst needs monthly product quantities. The four canonical order lines are sufficient to derive quantities 3 and 3 for notebook and pen; customer names are unnecessary for this particular calculation.
2. Including every customer attribute in the export increases exposure without helping that stated question. A narrower approved dataset can reduce the information handled.
3. A report row should retain its definition and source version, so a later reviewer can trace 22,650 notebook pause to the three agreed notebook units rather than a current-price lookup.
4. If the underlying line data is later removed under policy, the system must state which aggregate evidence remains and whether detailed reconstruction is still possible. It must not claim reproducibility from inputs it no longer holds.

■ 42.5.4 PLAIN — what is really happening inside

1. Governance combines metadata, access rules, retention decisions, ownership and evidence. No single catalogue field enforces all of these automatically.
2. Derived tables may preserve sensitive information even when direct identifiers are removed. Small groups, unique combinations and retained links can make a supposedly anonymous aggregate revealing.
3. Testing should include access through storage paths, query engines, extracts and recovery environments. Restoring an old snapshot may reintroduce information that has since been restricted or deleted unless the restore process accounts for that history.

■ 42.5.5 TECHNICAL — the engineer's version

1. Treat lineage as a dependency graph from source versions through transformations to outputs. Record both the useful audit relationships and the access/retention implications of keeping those relationships.
2. Iceberg snapshot expiration describes removal of historical table references and eventual eligibility of unused files for cleanup. It is not a universal guarantee of erasure from every backup, export or consumer. [S168]
3. The chapter provides engineering boundaries, not legal retention periods or a certification of anonymization. Actual retention, exceptions and access decisions need the organization's applicable requirements and authorized review.

■ 42.5.6 WORDS — remember these

1. **Data lineage:** how outputs depend on inputs — a recorded graph of source versions, transformations and derived assets. **Metric owner:** the accountable interpreter of a reported measure — a role responsible for definition changes, exclusions and disputes. **Data minimization:** limiting information to a justified purpose — a design practice reducing unnecessary collection, copying and exposure.

42.6 Selecting a design by workload

■ 42.6.1 PLAIN — in simple words

1. Select a design by the questions, scale, update rate and operational responsibilities it must handle. A fashionable label is not a requirement.

2. A small consistent database with clear reports may be better suited to a modest task than a complicated collection of distributed services. A much larger workload may justify additional machinery.
3. Compare complete operating systems of work: ingestion, queries, corrections, access, recovery and cost. Fast query demonstrations alone leave many important responsibilities untested.

■ 42.6.2 PLAIN — a picture in your head

1. Mira does not build a warehouse complex to store four slips. She starts with a system she can understand and maintain.
2. When she adds many branches and years of history, she measures which part becomes difficult before expanding the design.
3. **Where the comparison breaks:** software migrations can be costly even before physical scale becomes large. Compatibility, team skills and future requirements matter, but they should be recorded as assumptions rather than treated as certainty.

■ 42.6.3 PLAIN — a worked example

1. Define three hypothetical requirements: refresh within 15 minutes, answer the monthly quantity report within 20 seconds under ten concurrent readers, and reproduce a selected earlier report from retained inputs.
2. Candidate A might use one database with controlled reporting queries. Candidate B might use a reporting replica. Candidate C might use captured files and a separate analytical engine.
3. For each candidate, test the same input, report definition and freshness condition. Include the cost of maintaining captures, updates, deletions and recovery, not only query elapsed time.
4. A candidate that answers in two seconds but omits the latest hour fails the stated freshness requirement. Another that meets both timing goals but cannot explain its input version fails the stated reproduction requirement.

■ 42.6.4 PLAIN — what is really happening inside

1. Architecture selection is a constraint problem. Some designs reduce scan cost but complicate updates; others simplify transactions but make very large historical scans expensive.
2. Separate reversible decisions from difficult commitments. File format, catalogue semantics, key design and source-history retention can constrain later migration.
3. Record failure cases as part of evaluation. A system should be observed while a load is late, a writer conflicts, a source changes schema and a restore is required, not only while every component is healthy.

■ 42.6.5 TECHNICAL — the engineer's version

1. A workload specification should include data volume and distribution, query mix, concurrency, update/delete rates, service objectives, source lag, retention and recovery requirements. Avoid treating row count as a complete capacity model.
2. File layout, table format, catalogue and query engine are separate components with separate compatibility and failure properties. Measure their combined behaviour rather than assuming a capability from a format's name. [S168] [S170]
3. The book's analytical examples validate selected arithmetic, model boundaries and snapshot predicates. They are not a benchmark ranking or a procurement recommendation for a particular managed service.

■ 42.6.6 WORDS — remember these

1. **Workload specification:** a description of the actual work — volumes, distributions, query patterns and operating requirements used to evaluate a design. **Freshness objective:** how current a result must be — an explicit bound connecting publication to its required source state. **Architecture trade-off:** a benefit obtained with another cost or constraint — a choice evaluated against the complete task rather than one isolated metric.

42.97 Practice and worked answers

1. **Question:** Why separate operational and analytical workloads? **Answer:** They can compete for resources and have different response, history and freshness needs. Separation is a choice with additional copy-management costs.
2. **Question:** What is the canonical fact-table grain? **Answer:** One agreed order line, not one order, payment or daily stock snapshot.
3. **Question:** Why can a historical dimension join double a fact? **Answer:** Overlapping validity intervals can make several dimension versions match the same fact.
4. **Question:** Does a Parquet file provide table transactions by itself? **Answer:** No. Multi-file table state and publication require additional metadata and coordination.
5. **Question:** What should writer B do after its expected base version 10 no longer matches current version 11? **Answer:** Reject the stale publication, inspect the new state and retry only under a compatible conflict policy.
6. **Question:** Can an expired snapshot always be queried later? **Answer:** No. Its metadata or required files may no longer be retained or accessible.
7. **Question:** Does the monthly product-quantity report need customer names? **Answer:** Not for the stated calculation. Additional fields need a separate justified purpose.
8. **Question:** A query is fast but one hour stale against a 15-minute requirement. Does it pass? **Answer:** No. Speed does not substitute for freshness.

42.98 Common wrong ideas

1. **Wrong:** OLTP and OLAP must always use different products. **Right:** They are workload categories; architecture follows measured requirements.
2. **Wrong:** A warehouse fact table can mix grains freely. **Right:** Mixed grains produce ambiguous aggregation.
3. **Wrong:** Today's category must describe every historical fact. **Right:** Historical interpretation requires an explicit policy.
4. **Wrong:** Every file in a folder belongs to the current table. **Right:** Committed metadata defines accepted membership.
5. **Wrong:** A catalogue automatically protects raw storage. **Right:** Access boundaries must be checked separately.
6. **Wrong:** A snapshot identifier guarantees permanent recovery. **Right:** Recovery depends on retained files, metadata, compatibility and permission.
7. **Wrong:** An aggregate contains no sensitive information. **Right:** Small or linkable groups can still reveal information.
8. **Wrong:** The most elaborate architecture is the most complete solution. **Right:** Unnecessary components add responsibilities and failure modes.

42.99 Chapter summary in 20 lines

1. Operational and analytical work ask different kinds of questions.
2. Shared resources can make those workloads interfere.
3. Reporting copies introduce freshness and reconciliation duties.
4. Materialized results must have a refresh policy.
5. State one fact row's grain precisely.
6. Dimensions provide descriptive grouping context.
7. Historical dimension changes need an interpretation rule.
8. Overlapping dimension versions can multiply facts.
9. Measures have explicit additive scopes and units.
10. A collection of files is not automatically a coherent table.
11. Catalogues describe and locate data assets.
12. Manifests identify accepted file membership.
13. Snapshots give readers a selected table version.
14. Atomic metadata updates detect stale writers.
15. Stable field IDs separate identity from naming.
16. Snapshot retention bounds time travel and cleanup.
17. Governance follows raw and derived copies alike.
18. Minimize data to the justified analytical purpose.
19. Evaluate query speed together with freshness and recovery.
20. Choose machinery by workload evidence, not architectural fashion.

Columnar Storage and Compression

43.0 What this chapter gives you

1. A report often needs a few fields from many records. Columnar storage arranges values so the engine can avoid reading unrelated fields and can exploit similarities among values of the same kind.
2. You will work through row and column layouts, dictionary and run-length encodings, compression blocks, predicate pushdown and conservative skipping. Every saving will be tied to an explicit workload and representation.
3. The companion byte-layout experiment is an original fixed-width format using Python's standard library. It is not a Parquet writer, a database benchmark or evidence that columnar storage always outperforms row storage.

43.1 Rows versus columns

■ 43.1.1 PLAIN — in simple words

1. Row storage keeps the fields of one record close together. Column storage keeps values of one field close together across many records.
2. A row-oriented layout can suit fetching most fields of one order. A column-oriented layout can suit adding one amount field across millions of orders.
3. The benefit comes from avoiding unnecessary work, not from changing the answer. Both layouts must preserve the same record relationships and values.

■ 43.1.2 PLAIN — a picture in your head

1. Mira can store complete order slips in a stack, or copy all quantities into one list and all amounts into another list with matching positions.
2. To add amounts, the second arrangement lets her read only the amount list. To inspect one complete order, she must gather corresponding values from several lists.
3. **Where the comparison breaks:** real columnar formats include row groups, metadata, null representation and nested structures. They are not simply separate text files with the same line numbers.

■ 43.1.3 PLAIN — a worked example

1. Imagine 10 million records with 50 fixed-width fields of 8 bytes each. Ignoring all headers and compression, the data occupies $10,000,000 \times 50 \times 8 = 4,000,000,000$ bytes, or 4 decimal GB.
2. A query needing only two fields has 160,000,000 bytes of relevant field values. Ideal projection avoids 48 of 50 fields, or 96% of that raw field payload.

3. That is an opportunity, not a promise of 25 times faster execution. Metadata reads, filtering, decompression, computation, result handling and storage access patterns still contribute to elapsed time.
4. The local experiment uses 10,000 rows and four signed 64-bit fields. A complete row representation contains 320,000 raw bytes. The branch and amount columns together contain 160,000 bytes before metadata or compression.

■ 43.1.4 PLAIN — what is really happening inside

1. A reader selects the fields required by the query and locates their chunks. Values from unneeded columns may remain unread.
2. The engine combines corresponding positions to reconstruct selected records or compute aggregates. Preserving row alignment is essential; independently sorting columns would destroy the original relationships.
3. Row groups divide a file into manageable horizontal pieces, within which columns are stored separately. This permits some parallelism and bounded reads without requiring one enormous chunk for each column.

■ 43.1.5 TECHNICAL — the engineer's version

1. Parquet's file organization contains row groups and column chunks, with metadata describing their locations and interpretation. Projection can avoid unrelated column payloads when the reader and query support it. [S170]
2. A column layout is not inherently a transaction or update policy. Point updates, deletes, snapshots and concurrency may be implemented by an engine or table layer above the file format.
3. The laboratory counts bytes that its explicit access algorithm inspects and verifies equal query results across layouts. It does not measure operating-system read amplification, real storage-device traffic or a Parquet engine's performance.

■ 43.1.6 WORDS — remember these

1. **Columnar storage:** values grouped by field — a layout that can reduce irrelevant reads for analytical projections. **Row group:** a horizontal subset stored in column pieces — a unit connecting corresponding column chunks within a columnar file. **Projection pruning:** avoiding unneeded fields — selecting only the columns required to compute the query result.

43.2 Encodings

■ 43.2.1 PLAIN — in simple words

1. An encoding represents values in another form. It may save space by replacing repeated strings with short codes or by recording how values change rather than repeating their full representations.
2. The decoder must recover the same intended values. A smaller representation that silently changes a price or loses missing-value information is not an equivalent storage improvement.
3. Different patterns favour different encodings. Repeated values, small ranges and sorted sequences offer different opportunities; random-looking unique values may offer few.

■ 43.2.2 PLAIN — a picture in your head

1. Instead of writing “north” five hundred times, Mira writes a small dictionary saying code 1 means north and then records the code on each relevant slip.
2. For a long uninterrupted run of code 1, she might write “1 repeated 500 times.” That is another encoding layered on the repeated pattern.
3. **Where the comparison breaks:** digital encodings require precise widths, byte order, length fields and error handling. A verbal shortcut is not a complete interoperable file specification.

■ 43.2.3 PLAIN — a worked example

1. A column contains 500 instances of north, 300 of south and 200 of west. The raw UTF-8 character payload is $500 \times 5 + 300 \times 5 + 200 \times 4 = 4,800$ bytes.
2. A dictionary’s three strings need 14 character bytes, and 1,000 one-byte codes need 1,000 bytes. The combined 1,014 bytes exclude lengths, counts and other framing; they are not a claimed complete file size.
3. If the codes appear in three runs, a toy run-length format using one byte for the code and four for its count needs $3 \times 5 = 15$ run bytes. Alternating codes 1 and 2 for 1,000 values instead needs 5,000 run bytes, worse than the uncompressed codes.
4. A delta example stores 1,000 once, then changes +2, +2 and +3 to represent 1,000, 1,002, 1,004 and 1,007. The benefit depends on how those deltas and their metadata are actually encoded.

■ 43.2.4 PLAIN — what is really happening inside

1. Dictionary encoding separates distinct values from their occurrences. Its usefulness decreases when most values are unique or when dictionary lookup and storage costs dominate.
2. Run-length encoding records consecutive repetitions. Sorting or clustering can improve runs, but changing record order must preserve the relationship among all fields.
3. Delta and bit-packing schemes exploit small differences or bounded ranges. The decoder needs the appropriate base, bit width and signedness rules; these are part of the format contract.

■ 43.2.5 TECHNICAL — the engineer’s version

1. Parquet specifies several encodings, including plain values, dictionary codes, run-length/bit-packing hybrids and delta representations. Their actual bitstream rules must be read from the specification, not inferred from the teaching examples. [S171]
2. The toy run codec validates code range and positive bounded counts before decoding. Resource limits matter: a tiny encoded input claiming billions of repetitions should not allocate unbounded memory.
3. Encoding and general-purpose compression are distinct stages. An encoding may reduce size directly or rearrange bytes so that a later compressor finds more repetition. Neither should be credited with the other’s effect without measuring the pipeline.

■ 43.2.6 WORDS — remember these

1. **Dictionary encoding:** replacing values with reusable codes — a representation containing a distinct-value dictionary and an occurrence-code sequence. **Run-length encoding:** storing repetitions as a value and count — an encoding effective when identical values are consecutive. **Delta encoding:** storing changes from earlier values — a representation whose savings depend on the size and regularity of differences.

43.3 Compression blocks

■ 43.3.1 PLAIN — in simple words

1. Compression reduces a byte sequence using patterns that a decoder can reconstruct. Lossless compression must return the original bytes exactly.
2. Compression usually works within blocks or pages. To read one value, the reader may need to fetch and decompress a larger block containing it.
3. Larger blocks can provide more repeated context and fewer headers, but can also increase the minimum work for a small read. Block size is therefore a trade-off, not a universal “bigger is better” setting.

■ 43.3.2 PLAIN — a picture in your head

1. Mira packs many slips into a tightly sealed bundle. The bundle occupies less shelf space, but extracting one slip may require opening and unpacking the whole bundle.
2. Several smaller bundles can make selective access easier, at the cost of more wrapping and sometimes less efficient packing.
3. **Where the comparison breaks:** software compression uses mathematical representations, not physical squeezing. Its CPU cost, memory use and exact framing depend on the codec and implementation.

■ 43.3.3 PLAIN — a worked example

1. Suppose a raw 1 MB block compresses to 250 kB. Reading and decoding that block to obtain one 8-byte value may still require 250 kB of input and roughly 1 MB of decompressed output handling.
2. If a query needs every value in the block, that overhead may be worthwhile because three quarters of the stored payload was avoided. If it needs one scattered value from each of many blocks, the trade-off is different.
3. In the companion experiment, the same generated records are packed in row-major and column-major byte order, then compressed with the same standard-library zlib settings.
4. The output records actual encoded lengths and verifies exact decompression equality. It does not assume in advance which layout has the better compression ratio, and it does not infer application throughput from file size alone.

■ 43.3.4 PLAIN — what is really happening inside

1. The compressor encodes patterns in a bounded input unit. The decoder reconstructs that unit before or while the reader interprets the values inside it.
2. Compressed length, uncompressed length and integrity metadata serve different purposes. A length field helps frame data but must not be trusted as permission for unlimited memory allocation.
3. A reader may decompress only selected pages when metadata allows it. Efficient access depends on the arrangement of blocks, the query and which metadata the writer actually supplied.

■ 43.3.5 TECHNICAL — the engineer’s version

1. Keep the stages explicit: logical values, value encoding, compression and file/container framing. Distinguish stored bytes, bytes read, bytes decompressed and values examined in a measurement.

2. A codec benchmark should state implementation, level, block size, data distribution and whether encoding/decoding time is included. Comparing different data or unequal settings confounds the result.
3. Parquet pages are the units whose encoded payloads may be compressed under the file's codec settings. The teaching zlib blob has no Parquet page headers, dictionary-page rules or compatible reader contract. [S170] [S171]

■ 43.3.6 WORDS — remember these

1. **Lossless compression:** a smaller representation with exact recovery — byte encoding whose decoder reproduces the original input. **Compression block:** a unit compressed and decoded together — a boundary affecting storage efficiency and selective-read cost. **Read amplification:** reading more bytes than the requested logical data — additional work caused by layout, blocks, metadata or access patterns.

43.4 Predicate pushdown

■ 43.4.1 PLAIN — in simple words

1. A predicate is a condition such as “branch equals 1” or “amount is at least 5,000.” Pushdown means applying useful parts of that condition closer to the stored data.
2. This can avoid returning or decoding data that the rest of the query would immediately discard. It is helpful only when the lower layer understands the condition with compatible semantics.
3. Not every expression can be pushed down safely. A custom function, different text comparison or missing-value rule can change which rows qualify.

■ 43.4.2 PLAIN — a picture in your head

1. Dev asks the storeroom clerk for only the boxes belonging to branch 1 rather than carrying every box to the office and sorting them there.
2. The clerk must use the same branch definition and labels. A convenient approximation can send the wrong boxes or omit needed ones.
3. **Where the comparison breaks:** engines may combine metadata pruning, value filtering and later verification. “Pushed down” does not necessarily mean the storage layer proves the entire query condition without reading values.

■ 43.4.3 PLAIN — a worked example

1. The toy file contains four 8-byte fields per row: sequence, branch, quantity and amount. A query asks for the sum of amount where branch is 1.
2. A full-row scan examines 32 bytes per row, or 320,000 bytes for 10,000 rows. A column-oriented scan can inspect the branch and amount columns, totalling 160,000 raw bytes in this simple implementation.
3. A more selective implementation could first inspect branch values, then fetch amount values only for matching positions. Whether that reduces physical reads depends on block layout and the access API, so the simple lab does not claim it automatically.
4. Both query paths must return exactly the same sum for the same generated records. A smaller byte count with a different result is a correctness failure, not an optimization.

■ 43.4.4 PLAIN — what is really happening inside

1. The query planner determines which conditions the storage reader can evaluate. The reader uses partition information, statistics or decoded values to reduce candidates.
2. Conditions left unresolved continue through the execution plan and are evaluated later. Correctness requires retaining all possible qualifying rows until the relevant condition is decided.
3. Filtering also changes downstream work. Fewer candidate rows can reduce joins, aggregation memory and network transfer, even when the initial file read saves little.

■ 43.4.5 TECHNICAL — the engineer's version

1. Pushdown requires semantic compatibility across layers, including data types, null handling, collation and supported operators. A functionally similar expression is not necessarily equivalent for every input.
2. Distinguish partition pruning, row-group/page pruning and row-level filtering. They remove work at different granularities and rely on different metadata or decoded values. [S172]
3. Validate optimized output against a simpler reference query, including boundary values, missing fields and unusual distributions. An execution-plan label indicates a selected mechanism, not its correctness for an undocumented data contract.

■ 43.4.6 WORDS — remember these

1. **Predicate:** a condition used to select records — an expression whose truth determines query eligibility under the relevant semantics. **Predicate pushdown:** evaluating selection closer to storage — moving supported filtering work earlier to reduce later processing. **Candidate row:** a record not yet ruled out — input retained until the necessary conditions can be evaluated correctly.

43.5 Statistics and skipping

■ 43.5.1 PLAIN — in simple words

1. A block may carry a summary such as its smallest and largest value. A query can skip the block when that summary proves that no value can match.
2. The word “proves” matters. If the summary merely suggests that matching values are unlikely, skipping could lose valid results.
3. Statistics can be missing or imprecise. A safe reader then reads more data rather than inventing a stronger exclusion claim than the metadata supports.

■ 43.5.2 PLAIN — a picture in your head

1. A box labelled “order numbers 100 through 199” cannot contain order 250 when the label is reliable and uses the same numbering system.
2. A box labelled “mostly old orders” does not justify skipping it while looking for a recent order. One exception would invalidate that shortcut.
3. **Where the comparison breaks:** binary formats have type-specific comparison and null rules. A text prefix bound, a floating-point special value or a missing statistic needs more care than a simple integer label.

■ 43.5.3 PLAIN — a worked example

1. Three toy blocks have integer ranges [0, 9], [10, 19] and [20, 29]. A query selects values from 12 through 16 inclusive.
2. The first block can be skipped because its maximum 9 is below 12. The third can be skipped because its minimum 20 is above 16. The middle remains a candidate and its actual values must be checked.
3. A range of [10, 19] does not prove that 12, 13, 14, 15 or 16 actually occurs. It only fails to rule them out. Candidate selection and confirmed matches are different stages.
4. If the middle block has no statistics, read it. If its bounds are corrupt, treating them as authoritative can lose rows; metadata integrity and validation remain part of the storage system's responsibility.

■ 43.5.4 PLAIN — what is really happening inside

1. Writers compute summaries while creating files or pages. Readers compare the query bounds with those summaries before fetching unnecessary payloads.
2. Clustering similar values can make block ranges narrower and improve skipping. Randomly mixing every date into every block may leave each block spanning the whole report interval.
3. Null counts, bloom filters and additional indexes can support other decisions, but each has a defined scope. A bloom filter's possible match still requires verification; absence is useful only when the filter is valid and correctly interpreted.

■ 43.5.5 TECHNICAL — the engineer's version

1. Parquet's optional page index separates value-bound information from page offsets so readers can skip suitable pages without first reading every page header. The index is not a general secondary B-tree index. [S172]
2. Conservative interval exclusion for an inclusive query [low, high] is $\text{block_max} < \text{low}$ or $\text{block_min} > \text{high}$, assuming trustworthy comparable integer bounds. Equality at a boundary must not be excluded.
3. The lab tests boundary, missing-statistic and overlapping-range cases against direct value filtering. It does not validate every Parquet logical type, truncated string bound, NaN convention or encrypted-file metadata path.

■ 43.5.6 WORDS — remember these

1. **Zone map:** summary bounds for a data region — metadata such as minimum and maximum used for conservative elimination. **Conservative pruning:** skipping only what cannot match — a rule that may retain extra candidates but must not discard valid results. **Data clustering:** placing similar values near one another — a layout choice that can tighten summaries and improve selective access.

43.6 CPU, memory and input-output trade-offs

■ 43.6.1 PLAIN — in simple words

1. Saving storage reads may require more CPU work to decode values. Saving disk space may increase memory needed for dictionaries or decompressed blocks.
2. The best balance depends on what is scarce and what the query actually does. A CPU-bound report and a slow-storage scan can favour different settings.

3. Measure the entire question under a stated environment. A compression ratio, a byte count and a latency figure describe different aspects of the system.

■ 43.6.2 PLAIN — a picture in your head

1. Mira can carry fewer tightly packed bundles from the storeroom, but Dev may spend longer unpacking them at his desk.
2. If walking is the slow part, tight packing helps. If unpacking already keeps Dev busy while the storeroom is next door, another packing method might be preferable.
3. **Where the comparison breaks:** real engines can overlap I/O and computation, use vectorized decoding and cache data. Adding isolated stage times does not always predict end-to-end elapsed time.

■ 43.6.3 PLAIN — a worked example

1. Suppose reading raw input takes an assumed 8 seconds and processing takes 2 seconds without overlap. A compressed alternative takes 2 seconds to read and 5 seconds to decompress and process.
2. Under this deliberately serial model, the first path takes 10 seconds and the second 7. Compression saves time despite increasing CPU work.
3. Change only the raw-read time to 1 second on a much faster storage path. The uncompressed model now totals 3 seconds; the compressed path is not automatically preferable. Real measurements would need updated compressed-read time and overlap behaviour too.
4. The companion report therefore records generated row count, exact layout lengths, compressed lengths, round-trip checks and equal query results separately. It does not convert these into a production capacity claim.

■ 43.6.4 PLAIN — what is really happening inside

1. Execution can process batches of values efficiently, reducing per-value dispatch and improving locality. The benefit depends on operators, representation and available hardware.
2. Large dictionaries, many concurrent scans and decompressed buffers can consume substantial memory. A query that fits alone may spill or slow under concurrent load.
3. Updates and small writes introduce another cost. An analytical file arrangement may require writing replacement files, maintaining delete information or compacting many small files before queries remain efficient.

■ 43.6.5 TECHNICAL — the engineer's version

1. Report workload and measurement scope: cold or warm data, projected columns, selected fraction, ordering, compression settings, concurrency and result consumption. Avoid comparing an in-memory decoded array with a disk-backed compressed file as though layout were the only variable.
2. Distinguish logical payload, encoded size, compressed size, bytes transferred and CPU/memory use. A query can save one of these while increasing another.
3. The file-format and encoding specifications explain available mechanisms; they do not guarantee a workload-specific speedup. Verify correctness first, then measure the selected implementation under representative conditions. [S170] [S171] [S102]

■ 43.6.6 WORDS — remember these

1. **Vectorized execution:** processing batches of values together — an implementation approach reducing overhead and exploiting suitable data layout. **Decode cost:** work to reconstruct usable values — CPU and memory effort introduced by an encoding or compression layer. **End-to-end measurement:** timing the complete defined operation — evidence that includes all stages required to produce and consume the intended result.

43.97 Practice and worked answers

1. **Question:** How much raw field payload is needed for two of fifty 8-byte columns across ten million rows? **Answer:** 160,000,000 bytes, excluding metadata and compression.
2. **Question:** Why must columns preserve row alignment? **Answer:** A quantity and price at corresponding positions must still belong to the same record. Independently sorting columns destroys that relationship.
3. **Question:** When does the toy run-length encoding become larger? **Answer:** Alternating codes produce one five-byte run per value, so 1,000 values become 5,000 run bytes instead of 1,000 one-byte codes.
4. **Question:** Does a 250 kB compressed block allow an 8-byte physical read for one value? **Answer:** Not generally. The reader may need the whole compressed block and substantial decoded data.
5. **Question:** Can a block with bounds 10 through 19 be skipped for a query 12 through 16? **Answer:** No. The bounds overlap the query, even though they do not prove an actual match.
6. **Question:** What should a safe reader do when useful statistics are absent? **Answer:** Retain the block as a candidate and inspect it rather than inventing an exclusion.
7. **Question:** Is a smaller compressed file necessarily faster to query? **Answer:** No. Decoding, memory, access granularity and the workload affect the outcome.
8. **Question:** What does the companion format establish about Parquet interoperability? **Answer:** Nothing. It is a labelled original byte-layout exercise, not a Parquet implementation.

43.98 Common wrong ideas

1. **Wrong:** Columnar storage changes the logical rows. **Right:** It changes representation while preserving their relationships.
2. **Wrong:** Projection savings directly predict equal latency savings. **Right:** Other stages still contribute to time.
3. **Wrong:** Dictionary encoding always compresses data. **Right:** High cardinality and metadata can reduce or reverse its benefit.
4. **Wrong:** More compression is always better. **Right:** Decode and access costs can dominate.
5. **Wrong:** Pushdown means the whole predicate was decided without reading values. **Right:** Some conditions only reduce candidates.
6. **Wrong:** An overlapping min/max range proves a match. **Right:** It only prevents safe exclusion.
7. **Wrong:** Missing statistics justify guessing. **Right:** Safe pruning becomes less aggressive.
8. **Wrong:** A toy byte experiment is a production engine benchmark. **Right:** Its evidence applies only to the stated representation and operations.

43.99 Chapter summary in 20 lines

1. Row layouts group a record's fields together.

2. Column layouts group values of one field together.
3. Analytical projections can avoid unrelated columns.
4. Row alignment must survive every representation change.
5. Row groups bound columnar processing units.
6. Encodings exploit repeated or predictable value patterns.
7. Dictionary codes need their dictionary and framing.
8. Run-length encoding benefits consecutive repetitions.
9. Delta encoding benefits suitable differences between values.
10. Compression must be evaluated separately from value encoding.
11. Blocks determine the minimum decoding work for some reads.
12. Pushdown moves supported filtering closer to storage.
13. Semantics must agree across execution layers.
14. Metadata can rule out blocks conservatively.
15. Overlapping bounds identify candidates, not confirmed matches.
16. Missing statistics should reduce pruning, not correctness.
17. Clustering can improve the usefulness of block summaries.
18. CPU, memory and I/O costs can move in opposite directions.
19. Compare equal results under equal workload assumptions.
20. A format specification supplies mechanisms, not guaranteed speedups.

Full-Text Search and Relevance

44.0 What this chapter gives you

1. Looking up an exact order ID and finding a useful document are different tasks. Text search must decide how to split language, which expressions match and how to order several plausible results.
2. You will build an inverted index by hand, distinguish word presence from phrases, inspect a ranking rule and evaluate results against explicit information needs. You will also see why a search result is not proof that the retrieved statement is correct.
3. The companion examples use a small invented English corpus and basic SQLite FTS5 where the runtime supports it. They do not claim multilingual search quality, a complete search service or the live KedByte website's search implementation.

44.1 Words and documents

■ 44.1.1 PLAIN — in simple words

1. A document is the unit a search system returns or indexes: a page, chapter, product description or smaller passage. Choosing that unit changes what the search result means.
2. A query expresses an information need, but the words typed may only approximate it. Someone asking about a pen repair may not benefit from every product page containing the word pen.
3. Search begins by defining the searchable collection and who may see each document. A relevant private result is still inappropriate for a reader who lacks permission to access it.

■ 44.1.2 PLAIN — a picture in your head

1. Mira labels six cards and lets Dev search their descriptions. Each result points to a particular card, not to an unexplained sentence detached from its source.
2. If a long manual is split into passages, each passage should still identify the manual, section and edition so the reader can recover context.
3. **Where the comparison breaks:** digital systems can index many overlapping fragments automatically. Without stable document and passage identity, updates and deletions can leave stale or duplicate search results.

■ 44.1.3 PLAIN — a worked example

1. Our synthetic corpus contains D1 blue pen, D2 blue notebook, D3 red pen, D4 pen repair guide, D5 notebook backup guide and D6 pen blue.
2. A query for documents containing both blue and pen should return D1 and D6 under a simple word-presence rule. A query for the exact phrase blue pen should return only D1.

3. A person asking how to repair a pen has a different need. In this deliberately tiny labelled example, D4 is the relevant answer even though several other cards contain pen.
4. These cards are invented teaching text. They are not claims about actual products, repair procedures or the content of any published KedByte chapter.

■ 44.1.4 PLAIN — what is really happening inside

1. The indexing pipeline chooses document boundaries, extracts text and attaches metadata such as title, language, source version and access scope.
2. Retrieval identifies candidates according to the query model. Ranking then orders those candidates using a selected scoring rule.
3. The display layer should expose enough source context to help the reader judge usefulness. Search snippets are shortcuts into documents, not independent verification of their claims.

■ 44.1.5 TECHNICAL — the engineer's version

1. Full-text retrieval operates over a defined document collection and query representation. PostgreSQL separates document normalization into tsvector values and query representation into tsquery values. [S174]
2. Document identity, passage boundaries and versioning are application decisions. Record them so indexing updates can replace or remove the intended material without leaving ambiguous fragments.
3. Apply access restrictions to all exposed information, including result counts, titles and snippets where those reveal protected content. A later click-time authorization check does not undo an earlier disclosure in search results.

■ 44.1.6 WORDS — remember these

1. **Document:** the searchable unit — an identified body of content, potentially a whole page or a bounded passage. **Information need:** what the reader is trying to learn — the task that a query attempts to express. **Candidate retrieval:** finding possible answers — selecting documents for further scoring and verification under a query model.

44.2 Tokenisation

■ 44.2.1 PLAIN — in simple words

1. A tokenizer decides which pieces of text count as searchable units. Splitting on spaces is one possible rule, but it is not sufficient for every language or identifier.
2. Normalization decides which differences to ignore. Case, accents, punctuation and word endings can help or hinder matching depending on the task.
3. The query and documents must be interpreted compatibly. Searching a normalized query against differently normalized content can hide documents that a reader reasonably expects to find.

■ 44.2.2 PLAIN — a picture in your head

1. Dev cuts each card into word labels before sorting them. If he treats blue-pen as one label in documents but two labels in queries, some matches disappear.
2. If he removes every mark from an order code, two distinct codes can become indistinguishable. A rule useful for ordinary prose may be unsafe for exact identifiers.

3. **Where the comparison breaks:** natural languages differ in writing systems, morphology and segmentation. One English classroom tokenizer is not a universal language processor.

■ 44.2.3 PLAIN — a worked example

1. The toy tokenizer lowercases the six ASCII-English cards and extracts runs of letters or digits. `Blue pen` and `blue PEN` therefore produce the same two tokens.
2. `blue-pen` becomes `blue` and `pen` under this declared rule. An exact identifier search for `0-1042`, however, should use its own identifier-aware path rather than assume prose tokenization preserves the full key.
3. Removing accents might make some users' queries easier to match, but it can also collapse distinctions. A system should state whether it preserves or removes those differences and test the languages it actually serves.
4. A stemmer might map related English endings to a common form. That can increase matches, but it may also combine words whose meanings differ in a particular technical context.

■ 44.2.4 PLAIN — what is really happening inside

1. Text extraction and Unicode normalization occur before or alongside tokenization. The pipeline then may apply case folding, stop-word removal, stemming or language-specific dictionaries.
2. Position information records where tokens occur. Removing tokens or changing token boundaries affects phrase matching and snippets, so these operations must follow the engine's documented conventions.
3. Language detection is itself fallible. Mixed-language documents, code examples and product identifiers may need different analyzers or retained original text for exact matching.

■ 44.2.5 TECHNICAL — the engineer's version

1. SQLite FTS5 provides tokenizer choices such as `unicode61`, `ascii` and the Porter wrapper, with documented token-character and diacritic behaviour. The exercise uses simple English inputs and does not validate every tokenizer feature. [S173]
2. PostgreSQL text-search configurations connect parsers and dictionaries to lexeme normalization. Query construction functions differ in how they interpret operators, phrases and ordinary text. [S175]
3. Maintain analyzer/version metadata with the index. Changing tokenization without rebuilding or reconciling indexed content can mix incompatible representations and make result differences difficult to explain.

■ 44.2.6 WORDS — remember these

1. **Tokenisation:** splitting text into searchable units — applying a specified segmentation rule to content and queries. **Lexeme:** a normalized searchable form — a dictionary or analyzer output that may combine several written word forms. **Analyzer:** the text interpretation pipeline — extraction, tokenization and normalization rules used for indexing and querying.

44.3 Inverted indexes

■ 44.3.1 PLAIN — in simple words

1. An inverted index turns “document to words” into “word to documents.” Instead of reading every card, the searcher opens the list for the requested word.

2. Several word lists can be combined. AND keeps documents appearing in every required list; OR keeps documents appearing in at least one.
3. Word membership alone cannot prove a phrase. For that, the index or a later check needs the words' order and positions.

■ 44.3.2 PLAIN — a picture in your head

1. Mira makes a drawer for each word. Inside the blue drawer are references to cards D1, D2 and D6. Inside pen are D1, D3, D4 and D6.
2. Dev intersects the two lists to find D1 and D6. He then checks positions to distinguish blue pen from pen blue.
3. **Where the comparison breaks:** real posting lists are compressed and may include frequencies, positions or other metadata. Efficient query execution can skip portions rather than literally opening every reference in order.

■ 44.3.3 PLAIN — a worked example

1. The blue posting set is {D1, D2, D6}. The pen posting set is {D1, D3, D4, D6}. Their intersection is {D1, D6} and their union contains five documents.
2. Using zero-based token positions, D1 has blue at 0 and pen at 1. D6 has pen at 0 and blue at 1. Requiring pen immediately after blue selects D1 only.
3. For the word notebook, the posting set is {D2, D5}. Intersecting notebook with pen gives an empty set in this corpus, a legitimate result rather than an index failure.
4. Deleting D1 requires removing its eligibility from the search result. A stale posting pointing to a deleted document must not be treated as proof that the document still exists or may be displayed.

■ 44.3.4 PLAIN — what is really happening inside

1. Index construction collects token occurrences by document. Query evaluation merges or intersects the relevant postings and may verify phrases or other conditions against positions or source content.
2. Updates can create new index segments that are later merged. Readers need a consistent view of which document versions and deletion markers are effective.
3. Some index structures return possible matches that need rechecking. This is acceptable when the recheck removes false candidates before the final result, not when approximate candidates are presented as exact matches.

■ 44.3.5 TECHNICAL — the engineer's version

1. PostgreSQL GIN text-search indexes store lexemes with matching locations; GiST uses signatures that can require rechecking false matches. The access method's exactness and stored metadata affect execution. [S184]
2. The toy index records token positions and compares its AND, OR and phrase results with direct scans of the same tokenized documents. It is not a compressed posting-list engine or a concurrent index implementation.
3. In FTS5 external-content configurations, applications must keep content and index updates consistent using the documented mechanisms. An index definition alone does not automatically repair a previously inconsistent external-content history. [S173]

■ 44.3.6 WORDS — remember these

1. **Inverted index:** a word-to-document map — an access structure locating documents containing searchable terms. **Posting list:** occurrences associated with one term — document references, often accompanied by frequency or position information. **Positional index:** an index retaining term locations — metadata supporting phrases and proximity conditions beyond word presence.

44.4 Scoring and ranking

■ 44.4.1 PLAIN — in simple words

1. A search score orders candidates according to a rule. It is not a probability that a document is true, and scores from different queries or engines may not be directly comparable.
2. Useful rules can consider how often a term occurs, how common it is across documents and whether it appears in a title or a long body.
3. Repetition should not always grow the score without limit. A page repeating pen a thousand times is not necessarily more useful than a clear repair guide using the word twice.

■ 44.4.2 PLAIN — a picture in your head

1. Dev gives some weight to words that distinguish a card from the collection. A rare technical term can be more informative than a word printed on nearly every card.
2. He also notices where the match occurs. A title devoted to repair may be a stronger clue for a repair question than a passing reference in a long unrelated document.
3. **Where the comparison breaks:** relevance depends on the reader's task and context. A numerical formula can rank clues, but it cannot guarantee that the top document answers the actual question correctly.

■ 44.4.3 PLAIN — a worked example

1. In our six-document corpus, pen appears in four documents and blue in three. A simple inverse-frequency weight $\ln(N / df)$ gives pen about 0.405 and blue about 0.693.
2. A presence-only score for the two query terms gives D1 and D6 about 1.099, D2 about 0.693, and D3/D4 about 0.405. This is a toy rule, not SQLite's BM25 output.
3. A term-frequency saturation factor $(k + 1) * f / (k + f)$ with $k = 1.2$ gives 1 when frequency f is 1 and about 1.571 when f is 3. Tripling repetition therefore does not triple this contribution.
4. D1 and D6 still tie under the presence rule. A deterministic document-ID tie-break makes output repeatable, but does not turn the tie-break into a claim of greater relevance.

■ 44.4.4 PLAIN — what is really happening inside

1. Ranking computes features over the candidate set and collection statistics. Document length, term frequency and field weighting can affect the result.
2. Changing the corpus can change scores even when a particular document stays unchanged, because collection statistics such as document frequency change.
3. Ranking parameters should be evaluated on representative queries and separate development/test sets. Adjusting a rule until it succeeds on one demonstration can overfit that demonstration.

■ 44.4.5 TECHNICAL — the engineer's version

1. BM25 combines term weighting with frequency saturation and document-length normalization. The exact formula and parameter choices should be stated for the implementation being used. [S182]
2. SQLite FTS5 negates its BM25 score so numerically smaller values sort first for better matches under that implementation. A generic assumption that every search score should be sorted descending is therefore wrong. [S173]
3. Bound SQL values with parameters, but remember that a full-text query string may still have its own operators and resource costs. Define whether the interface accepts literal words or an advanced query language, and handle invalid expressions without exposing internal details.

■ 44.4.6 WORDS — remember these

1. **Document frequency:** how many documents contain a term — a collection statistic used to distinguish common from rarer terms. **BM25:** a family of lexical ranking rules — scoring that combines term weighting, frequency saturation and document-length effects. **Tie-break:** a rule ordering equal-scored results — a deterministic convention that does not itself establish relevance.

44.5 Language and spelling

■ 44.5.1 PLAIN — in simple words

1. Readers may use spelling variants, abbreviations or different words for the same concept. Search can help by recognizing selected alternatives, but every expansion can also introduce unwanted matches.
2. Language-specific processing matters. An English stemming rule should not be presented as a solution for Hindi, Japanese or mixed-language technical writing.
3. Exact identifiers deserve special treatment. Correcting a misspelled ordinary word can be helpful; silently changing a receipt ID can retrieve the wrong person's record.

■ 44.5.2 PLAIN — a picture in your head

1. Mira knows that a customer asking for a writing instrument may be looking for a pen. She can suggest that connection while still allowing the customer to inspect the original query.
2. She would not silently change order O-1042 to O-1043 because the second card happened to be easier to find.
3. **Where the comparison breaks:** automatic synonym and spelling systems cannot rely on human situational understanding. They need bounded rules, confidence policies and evaluation across the intended users' language patterns.

■ 44.5.3 PLAIN — a worked example

1. A query `notebok` differs from `notebook` by one missing `o`. An edit-distance suggestion can propose `notebook`, but the interface should distinguish the suggestion from an exact match.
2. Expanding `notebook` to `computer` might help a device-shopping query while damaging a stationery query. The corpus and task determine whether that synonym is useful.
3. In our corpus, `notebook backup guide` is intentionally ambiguous: `notebook` could refer to a computer or written notes. Word overlap does not resolve the intended sense by itself.

4. A bilingual book reader should evaluate actual language examples and technical terms. The small English lab is evidence about its declared tokens only, not proof that every language's segmentation and spelling needs are covered.

■ 44.5.4 PLAIN — what is really happening inside

1. Query expansion can add synonyms, spelling candidates or related forms before retrieval. This often increases the candidate set and can reduce precision while improving recall.
2. The expansion history should be observable when it changes the meaning of the query. A reader can then distinguish what they typed from what the system searched.
3. Search snippets also need safe rendering. Source text can contain markup or code; highlighting must not turn untrusted content into executable browser instructions.

■ 44.5.5 TECHNICAL — the engineer's version

1. PostgreSQL's text-search configurations and dictionaries are language- and task-dependent. Normalization choices are not interchangeable, and appropriate configuration must accompany stored and queried text. [S174] [S175]
2. Highlighting functions can return text containing markup or source characters that need safe output handling. Treat a snippet as untrusted content and use a deliberate escaping/rendering strategy rather than inserting it blindly as HTML. [S175]
3. Keep exact-key retrieval separate from fuzzy natural-language search when identity is consequential. A suggestion is not authorization to replace the requested key or to expose nearby private records.

■ 44.5.6 WORDS — remember these

1. **Query expansion:** adding related search terms — a recall-oriented technique whose extra candidates can change precision and meaning. **Edit distance:** the number of selected character edits between strings — a spelling similarity measure, not a proof of intended identity. **Snippet:** a displayed excerpt from a result — contextual text that requires source attribution and safe rendering.

44.6 Evaluating search quality

■ 44.6.1 PLAIN — in simple words

1. Search quality must be checked against actual information needs. Finding a word is not the same as helping the reader answer a question.
2. Precision asks how many returned results are relevant. Recall asks how much of the known relevant material was returned. The labels and the chosen result cutoff matter.
3. A useful evaluation includes queries with no answer, rare terms, spelling problems and access restrictions. A few successful showcase queries do not establish general quality.

■ 44.6.2 PLAIN — a picture in your head

1. Mira gives Dev a list of questions and independently marks which cards help answer each one. She then compares his results with those judgements.
2. If Dev returns every card for every question, he finds all relevant cards but also burdens the reader with irrelevant material.

3. **Where the comparison breaks:** relevance judgements can be incomplete or disputed. Evaluation should preserve who labelled what, the intended task and the uncertainty in the labels rather than treating them as infallible ground truth.

■ 44.6.3 PLAIN — a worked example

1. For the labelled query “how to repair a pen,” suppose only D4 is considered relevant. A hypothetical result order is D1, D3, D4.
2. Precision at three is 1 relevant result divided by 3 returned, or one third. Recall at three is 1 found divided by 1 known relevant, or 1.
3. The first useful result is at rank 3, so reciprocal rank is 1/3. Moving D4 first improves this navigation measure without changing the three-result set’s precision or recall.
4. If there are no known relevant documents for another query, recall’s denominator is zero. Record the chosen evaluation convention or report it as undefined; do not silently count an arbitrary 100% success.

■ 44.6.4 PLAIN — what is really happening inside

1. Evaluation stores a corpus version, queries, relevance judgements, retrieval configuration and result lists. Repeatability requires all of those, not only the scoring formula.
2. Candidate generation and ranking can fail separately. If a useful document never enters the candidate set, a perfect reranker cannot recover it from nothing.
3. Operational quality includes response time, index freshness, deletion propagation and authorization. A relevant answer drawn from an outdated or unauthorized source can still fail the product’s requirements.

■ 44.6.5 TECHNICAL — the engineer’s version

1. Precision and recall compare retrieved documents with judged relevance; they focus on useful retrieval rather than accuracy dominated by a large irrelevant population. [S181]
2. Use stable tie-breaking and avoid duplicate document IDs when computing set-based metrics. State whether the unit is a whole document, passage or source, because overlapping passages can inflate apparent coverage.
3. The companion checks establish posting-set correctness, phrase behaviour, selected FTS5 queries and metric arithmetic on a tiny labelled corpus. Independent user evaluation and multilingual, accessibility and production-load testing remain separate tasks.

■ 44.6.6 WORDS — remember these

1. **Precision:** how much of the returned material is relevant — relevant retrieved items divided by retrieved items under a stated cutoff and unit. **Recall:** how much relevant material was found — relevant retrieved items divided by the known relevant population. **Reciprocal rank:** a measure of how soon the first useful answer appears — one divided by its rank, with an explicit no-answer convention.

44.97 Practice and worked answers

1. **Question:** Which cards contain both blue and pen? **Answer:** D1 and D6. Only D1 contains the phrase blue immediately followed by pen.

2. **Question:** Why does tokenization belong in the index's version record? **Answer:** A changed analyzer can alter boundaries and matches even when the source text is unchanged.
3. **Question:** What does a posting list contain? **Answer:** References to documents containing a term, potentially with frequencies and positions.
4. **Question:** Why is a search score not a truth probability? **Answer:** It measures the selected ranking features, not the correctness of every statement in a document.
5. **Question:** Which direction sorts SQLite FTS5 BM25 better matches first? **Answer:** Ascending, because that implementation returns numerically smaller scores for better matches.
6. **Question:** May spelling correction silently replace an exact order ID? **Answer:** No. A suggested nearby identifier is not the same request and may expose unrelated information.
7. **Question:** What are precision and recall for one relevant result among three returned when only one relevant document is known? **Answer:** Precision 1/3 and recall 1.
8. **Question:** Can ranking recover a useful document omitted by candidate retrieval? **Answer:** Not unless another retrieval stage introduces it. Ranking only orders what it receives.

44.98 Common wrong ideas

1. **Wrong:** A word match proves a useful answer. **Right:** Relevance depends on the information need and context.
2. **Wrong:** Space splitting works for every language. **Right:** Tokenization is language- and task-dependent.
3. **Wrong:** AND word matching proves an exact phrase. **Right:** Phrase matching needs order and position information.
4. **Wrong:** More repetitions always mean a better result. **Right:** Ranking rules often limit repetition's contribution.
5. **Wrong:** Every score should be sorted descending. **Right:** Read the implementation's score convention.
6. **Wrong:** Fuzzy search is appropriate for every identifier. **Right:** Exact identity needs a separate contract.
7. **Wrong:** Returning everything is excellent search because recall is high. **Right:** Precision and reader effort also matter.
8. **Wrong:** Search-result permission can wait until the click. **Right:** Titles, counts and snippets can already disclose protected information.

44.99 Chapter summary in 20 lines

1. Choose the document unit and collection boundary.
2. A query approximates a reader's information need.
3. Retrieval and ranking are separate stages.
4. Tokenization defines the searchable units.
5. Normalization changes which distinctions are retained.
6. Query and document analyzers must be compatible.
7. Inverted indexes map terms to documents.
8. Posting intersections implement simple AND conditions.
9. Phrase matching requires order and positions.

10. Index updates must track document versions and deletions.
11. Ranking scores reflect a selected rule, not truth.
12. Rare terms can contribute different evidence from common ones.
13. Frequency saturation limits repeated-word influence.
14. Score direction and tie-breaking are implementation details.
15. Language and spelling expansion need task-specific evaluation.
16. Exact identifiers should not be silently fuzzy-matched.
17. Snippets require context, permission and safe rendering.
18. Precision and recall have different denominators.
19. Candidate omissions cannot be repaired by ordering alone.
20. Evaluate usefulness, freshness, access and operational behaviour together.

Vector Search: Similarity Is Not Truth

45.0 What this chapter gives you

1. Vector search represents items as lists of numbers and retrieves items that are close under a chosen mathematical rule. It can find useful similarities even when the exact query words do not appear in the result.
2. You will calculate distances, compare exact and approximate retrieval, inspect index trade-offs and test permission filters. The central boundary is simple: closeness in a representation is not proof of factual correctness, identity or authorization.
3. The companion exercises use small explicit vectors and direct Python arithmetic. They do not train an embedding model, implement HNSW or run Faiss, and they do not claim production semantic-search quality.

45.1 Representing items as vectors

■ 45.1.1 PLAIN — in simple words

1. A vector is an ordered list of numbers. For search, a procedure turns each document, image or other item into such a list so similar items may occupy nearby positions.
2. The numbers gain meaning from the procedure that created them. Two lists of the same length are not automatically comparable if they came from different models or preprocessing rules.
3. A vector is a representation, not the original item. Keep a reliable link to the source and its version so a result can be inspected rather than trusted because a number looks impressive.

■ 45.1.2 PLAIN — a picture in your head

1. Mira makes a map where cards about similar topics tend to sit near one another. A new question is placed on the same map, and she inspects nearby cards.
2. The map helps find candidates, but a nearby card might be outdated, misleading or about the wrong sense of a word.
3. **Where the comparison breaks:** learned vectors usually have many dimensions that are not individually labelled like streets. A coordinate does not necessarily correspond to a human-readable topic, and the geometry depends on the training objective.

■ 45.1.3 PLAIN — a worked example

1. Begin with an invented two-dimensional representation rather than a trained model. Let the query be $Q = (1, 0)$, and items $A = (1, 0)$, $B = (0.8, 0.6)$, $C = (0, 1)$, $D = (-1, 0)$ and $E = (2, 0)$.
2. These coordinates are teaching data. No assertion is made that a real document about pens naturally receives any of these numbers.

3. A search algorithm can rank the items once a metric is chosen. It cannot decide whether A's underlying text is correct merely because A's vector equals the query vector.
4. If a different encoder maps the same source into another coordinate system, combining its output with the old index without compatibility checks can make the ranking meaningless.

■ 45.1.4 PLAIN — what is really happening inside

1. An embedding pipeline extracts and preprocesses content, runs an encoder and stores the resulting vector with source identity and metadata.
2. Query encoding must use a compatible procedure. Some systems use different query and document encoders trained to work together; compatibility is therefore more specific than simply using identical software names.
3. Content updates, chunking changes or encoder revisions can require re-embedding. An index containing mixed generations needs an explicit compatibility policy and evaluation rather than accidental coexistence.

■ 45.1.5 TECHNICAL — the engineer's version

1. Store encoder identity/version, vector dimension, numerical type, normalization policy, source revision and chunk boundaries. These attributes are part of the retrieval contract.
2. Validate dimensions and finite numeric values before distance computation. NaN, infinity or a zero vector can invalidate a selected comparison, especially cosine normalization.
3. Faiss exposes several index and metric choices rather than one universal meaning of vector proximity. Its documented metric conventions must be matched to the representation and task. [S176] [S177]

■ 45.1.6 WORDS — remember these

1. **Embedding**: an item's numerical representation — an ordered vector produced by a defined encoder and preprocessing pipeline. **Embedding space**: the coordinate system used for comparison — a representation whose geometry depends on its model and training objective. **Representation compatibility**: vectors can be compared meaningfully — agreement between encoder, dimension, normalization and metric contracts.

45.2 Distance and similarity

■ 45.2.1 PLAIN — in simple words

1. Euclidean distance measures straight-line separation. Inner product multiplies corresponding coordinates and adds the results. Cosine similarity compares direction after accounting for vector length.
2. These rules can rank the same items differently. Choosing one is part of defining what "similar" means for the application.
3. A smaller distance is usually better under a distance rule, while a larger similarity is usually better under a similarity rule. Always check the actual API's returned quantity and direction.

■ 45.2.2 PLAIN — a picture in your head

1. Imagine arrows from the centre of a map. Two arrows can point in the same direction but have different lengths.

2. Cosine treats direction as important and ignores that length difference. Euclidean distance still sees the separation between the arrow tips.
3. **Where the comparison breaks:** the choice of geometry is not a universal model of human meaning. A trained representation may rely on length, direction or another scoring rule, so changing the metric can damage retrieval.

■ 45.2.3 PLAIN — a worked example

1. For $Q = (1, 0)$ and $B = (0.8, 0.6)$, squared Euclidean distance is $(1 - 0.8)^2 + (0 - 0.6)^2 = 0.4$. The distance itself is the square root, about 0.632.
2. The squared distances from Q to A, B, E, C and D are respectively 0, 0.4, 1, 2 and 4. Exact Euclidean ranking therefore places them in that order, with a defined tie-break if needed.
3. Inner products with Q are $A = 1, B = 0.8, C = 0, D = -1$ and $E = 2$. Maximum-inner-product search prefers E to A in this example.
4. Cosine similarities are $A = 1, B = 0.8, C = 0, D = -1$ and $E = 1$. A and E point in the same direction despite their different lengths. A zero vector has no defined cosine direction and is rejected by our model.

■ 45.2.4 PLAIN — what is really happening inside

1. Distance evaluation performs arithmetic across dimensions. Exact search repeats this comparison for every eligible stored vector unless another exact structure can safely eliminate candidates.
2. Squared Euclidean distance preserves Euclidean ordering because the square root is monotonic for non-negative inputs. It does not preserve the numerical unit of distance itself.
3. Normalizing vectors changes their length to one. For unit vectors, squared Euclidean distance equals $2 - 2 \times \text{inner_product}$, connecting cosine and Euclidean rankings under those assumptions.

■ 45.2.5 TECHNICAL — the engineer's version

1. Faiss reports squared L2 distance for its L2 metric. Its inner-product search is not automatically cosine similarity; cosine comparison requires suitable normalization of both queries and stored vectors. [S176]
2. The local functions check equal dimensions, finite values and nonzero norms where required. They use deterministic ID tie-breaking so repeated results can be compared without interpreting tie order as evidence of relevance.
3. A score threshold must be evaluated for the chosen encoder, metric and task. A value such as 0.8 has no universal meaning of “80% factually correct” or “the same person.”

■ 45.2.6 WORDS — remember these

1. **Euclidean distance:** straight-line separation — the square root of summed squared coordinate differences. **Inner product:** coordinate products added together — a similarity score affected by stored-vector magnitudes as well as direction. **Cosine similarity:** directional agreement — inner product divided by the product of nonzero vector norms.

45.3 Exact and approximate search

■ 45.3.1 PLAIN — in simple words

1. Exact nearest-neighbour search returns the closest eligible items under the declared metric. It says nothing about whether that metric captures the reader's real need perfectly.
2. Approximate search saves work by examining a selected subset or a compressed representation. It can miss a mathematically nearer item.
3. Approximation is useful when its measured cost-quality trade-off fits the application. Calling it approximate should lead to evaluation, not to assuming that errors are either impossible or harmless.

■ 45.3.2 PLAIN — a picture in your head

1. Mira can measure the distance from a question to every card on the map, or search only promising neighbourhoods first.
2. The neighbourhood shortcut is faster when it avoids much work, but the closest card might be just outside the searched area.
3. **Where the comparison breaks:** different indexes choose candidates using graphs, partitions, quantized codes or other structures. The map picture does not establish the recall or runtime of any particular algorithm.

■ 45.3.3 PLAIN — a worked example

1. In a separate fixture, $Q = (0, 0)$, $A = (0.1, 0)$, $B = (0.2, 0)$, $C = (5, 5)$ and $D = (5.1, 5)$. The exact top two under Euclidean distance are A and B.
2. Suppose a deliberately limited candidate stage returns only A and C. Reranking those candidates exactly still returns A and C; it cannot recover B because B was never considered.
3. The overlap with the exact top-two set is one item out of two, so neighbour recall at two is 0.5. This is approximation recall against the metric's exact answer, not human relevance recall.
4. Expanding the candidate set to A, B and C recovers the exact top two in this tiny example. It does not prove that a fixed candidate budget will achieve the same recall on a different collection.

■ 45.3.4 PLAIN — what is really happening inside

1. Candidate generation reduces the number of full comparisons. Approximate indexes attempt to locate useful regions or neighbours without exhaustively scanning all vectors.
2. Reranking can improve ordering within the candidate set, especially when the first stage used compressed scores. Its maximum possible recovery remains limited by candidate coverage.
3. An exact baseline is valuable for measuring approximation loss on representative queries. Human-labelled relevance is another baseline, because exact geometric neighbours can still be poor answers.

■ 45.3.5 TECHNICAL — the engineer's version

1. Faiss distinguishes exhaustive flat indexes from non-exhaustive structures such as IVF and HNSW. Index parameters affect the balance between search work and recovered neighbours. [S177]
2. Define recall@k against a fixed exact result set, including tie handling and eligible filters. A candidate stage and a reranker should be measured separately when diagnosing omissions.
3. The companion approximation example intentionally restricts a candidate list. It demonstrates the coverage limit, not the internal behaviour or performance of an ANN library.

■ 45.3.6 WORDS — remember these

1. **Nearest neighbour:** an item closest under a metric — a geometric result whose usefulness depends on the representation. **Approximate nearest-neighbour search:** a shortcut to likely close items — retrieval that may miss exact neighbours to reduce cost. **Candidate recall:** how much required material reaches the later stage — coverage that limits what reranking can ultimately recover.

45.4 Index trade-offs

■ 45.4.1 PLAIN — in simple words

1. A vector index organizes stored vectors so a query can search more selectively. Building and maintaining that organization costs memory, computation and update work.
2. Some designs search connected neighbours in a graph. Others first choose coarse groups and search selected groups. Compression can reduce stored vector size while introducing approximation.
3. There is no single best setting independent of data, queries and operating requirements. A setting that works well on one demonstration may behave differently after growth or distribution changes.

■ 45.4.2 PLAIN — a picture in your head

1. Mira can connect each card to nearby cards and follow those links, or divide the map into districts and inspect selected districts.
2. More links or more inspected districts can improve the chance of finding the right card, but consume additional space or query work.
3. **Where the comparison breaks:** graph construction and partition training have precise algorithms and implementation details. A human map does not prove search complexity, deletion support or behaviour under concurrent updates.

■ 45.4.3 PLAIN — a worked example

1. One million vectors with 768 coordinates stored as 32-bit floats require $1,000,000 \times 768 \times 4 = 3,072,000,000$ raw vector bytes, or 3.072 decimal GB.
2. That excludes identifiers, metadata, graph links, allocator overhead and replicated copies. Quoting only the raw vector size as the service's total memory requirement would be incomplete.
3. A hypothetical product-quantized representation using 96 one-byte subvector codes has 96 code bytes per vector instead of 3,072 float bytes. Codebooks, IDs and optional original vectors still add storage, and distance estimates may become approximate.
4. A coarse-group index that probes 10 groups instead of 2 usually performs more candidate work. The actual improvement in recall must be measured; the group count alone does not guarantee a fixed accuracy level.

■ 45.4.4 PLAIN — what is really happening inside

1. HNSW builds a hierarchy of neighbour graphs and navigates from sparse upper layers toward a more detailed lower layer. Its search effort is controlled by algorithm and implementation parameters.
2. Inverted-file vector indexes assign vectors to coarse regions. A query selects regions to probe and evaluates candidates within them, potentially using compressed codes before exact reranking.

3. Updates, deletions and model changes require maintenance policies. Some index forms rebuild, mark deletions or support limited mutation patterns; the application must not assume every library supports the same lifecycle.

■ 45.4.5 TECHNICAL — the engineer's version

1. The original HNSW research describes hierarchical graph navigation for approximate neighbour retrieval. Its empirical results and design do not constitute a universal latency or recall guarantee for every modern implementation. [S178]
2. Faiss documents distinct storage formulas and supported operations for its index classes. Use the selected class's actual behaviour and measured overhead when planning capacity or update handling. [S177]
3. Benchmark build time, index memory, query latency distributions, recall, filtered retrieval and update/deletion behaviour. Report the exact encoder, metric, data distribution, parameters and hardware before comparing results.

■ 45.4.6 WORDS — remember these

1. **HNSW**: hierarchical neighbour-graph search — an approximate indexing approach navigating graphs at several levels of detail. **IVF index**: search through selected coarse groups — an inverted-file organization assigning vectors to regions before candidate evaluation. **Product quantization**: compact codes for vector parts — a representation replacing subvectors with codebook references for storage and distance estimation.

45.5 Filtering and evaluation

■ 45.5.1 PLAIN — in simple words

1. Search may need to consider only a permitted tenant, language, document type or time range. Those filters define the eligible population before usefulness is judged.
2. Taking the global top few and filtering afterward can return too few eligible results even when good permitted results exist nearby.
3. Permission filtering is not merely a quality adjustment. Protected content, scores, titles and snippets must not leak through intermediate or fallback paths.

■ 45.5.2 PLAIN — a picture in your head

1. Mira has separate cabinets for two clients. A client asks for the nearest two cards in their own cabinet.
2. Dev first chooses the nearest two from both cabinets, then removes the other client's cards. He may return nothing while two suitable cards remain in the correct cabinet.
3. **Where the comparison breaks**: actual indexes support filters in different ways, sometimes before search, during graph traversal or after candidate generation. Correct access enforcement and recall under filtering both require verification.

■ 45.5.3 PLAIN — a worked example

1. Let $Q = (0, 0)$. Documents V1 and V2 belong to tenant A and lie at distances 0.1 and 0.2. V3 and V4 belong to tenant B and lie at distances 0.3 and 0.4.
2. A tenant-B request for two results should consider V3 and V4. Searching the global top two first yields V1 and V2; filtering afterward leaves zero results.

3. Searching the eligible tenant-B set exactly returns V3 and V4. Overfetching more global candidates can help this example, but a fixed overfetch count does not guarantee sufficient eligible results for every distribution.
4. The tenant scope in the lab is supplied by an explicit trusted test context. It is not accepted as arbitrary proof from a caller, and the exercise does not implement a production login or authorization provider.

■ 45.5.4 PLAIN — what is really happening inside

1. Metadata filters interact with index traversal and candidate limits. The eligible population can be much smaller or differently distributed than the overall corpus.
2. Evaluation should measure recall after the actual filters, not only on unfiltered queries. Rare tenants or restrictive time ranges can expose failures hidden by aggregate results.
3. Empty-result and fallback behaviour matters. Relaxing a permission filter to produce an answer is not an acceptable relevance improvement; changing an optional user preference requires a visible policy.

■ 45.5.5 TECHNICAL — the engineer's version

1. Distinguish mandatory authorization predicates from optional relevance filters. The former must remain enforced through candidate retrieval, reranking, caching and display.
2. Report exact-neighbour recall and human relevance metrics separately. The first measures approximation relative to geometry; the second measures whether the retrieved source helps with the labelled task. [S181]
3. Test filter-before-limit behaviour, cross-tenant requests, stale metadata, deleted sources and empty eligible sets. Passing vector-distance tests alone does not establish these boundaries.

■ 45.5.6 WORDS — remember these

1. **Eligible set:** items permitted by the request's complete conditions — the population within which retrieval should answer the question. **Post-filtering:** removing candidates after retrieval — an approach that can reduce returned coverage when the candidate budget was spent on ineligible items. **Overfetching:** retrieving extra candidates before later selection — a mitigation that must be evaluated rather than assumed to guarantee filtered recall.

45.6 Retrieval limits

■ 45.6.1 PLAIN — in simple words

1. A nearby document may be wrong, obsolete or about a subtly different issue. Similarity is a route to evidence, not a replacement for reading that evidence.
2. When a retrieved passage is supplied to a text-generating system, the generated answer adds another stage that can misinterpret, omit or invent information.
3. Good retrieval should preserve source identity, version, context and permission. A plausible answer with no trustworthy connection to its sources is difficult to verify.

■ 45.6.2 PLAIN — a picture in your head

1. Mira finds a card near the question on her topic map. She opens the full source before using it to make a decision.

2. A helper who writes a fluent summary can still misunderstand the card. The map and the summary each need their own checks.
3. **Where the comparison breaks:** retrieved documents can contain untrusted instructions as well as facts. A system using external text must not grant that text authority to change access rules or execute operations.

■ 45.6.3 PLAIN — a worked example

1. Suppose a passage about backup creation is geometrically close to a question about restore verification. It may discuss related vocabulary but omit the steps needed to establish that a restore actually works.
2. A retrieval score of 0.92 does not fill that gap. The answer must distinguish what the passage supports from what still needs another source or an experiment.
3. If the retrieved document belongs to an older software version, its commands may not apply to the current environment. Source version and consultation date are relevant evidence, not decorative metadata.
4. A system can correctly retrieve all three passages selected by its metric and still answer the wrong question. Evaluate retrieval, source interpretation and final factual support as separate stages.

■ 45.6.4 PLAIN — what is really happening inside

1. Retrieval pipelines often combine lexical and vector candidates, merge them and rerank. Each stage can improve one failure mode while introducing another, such as duplicate passages crowding out diverse sources.
2. A source-grounded answer must link claims to actual supporting text and retain relevant limitations. A citation to a merely related passage is not adequate support.
3. Updating or deleting a source requires corresponding treatment of embeddings, indexes, caches and retained outputs. Removing the original file alone does not establish that every derived copy has disappeared.

■ 45.6.5 TECHNICAL — the engineer's version

1. Separate four contracts: encoder semantics, approximate retrieval quality, access eligibility and answer support. Success at one layer does not establish the others.
2. Treat retrieved text as data from its actual trust level. It must not override system instructions, authorize external actions or bypass tenant restrictions merely because it was returned by an index.
3. The laboratory proves selected geometry, candidate-coverage and filtering examples under declared finite inputs. It makes no claim that vector proximity establishes identity, truth, safety or universal semantic understanding.

■ 45.6.6 WORDS — remember these

1. **Hybrid retrieval:** combining different candidate methods — often lexical and vector search followed by merging or reranking. **Grounded answer:** a response supported by inspected evidence — claims linked to sources that actually establish them, with limitations retained. **Retrieval boundary:** what finding a source does and does not prove — the distinction between candidate similarity and verified, authorized factual support.

45.97 Practice and worked answers

1. **Question:** Can vectors from two unrelated encoders be compared merely because both have 768 coordinates? **Answer:** No. Dimension equality does not establish representation compatibility.
2. **Question:** What is squared distance from (1, 0) to (0.8, 0.6)? **Answer:** 0.4. The Euclidean distance is approximately 0.632.
3. **Question:** Why do $A = (1, 0)$ and $E = (2, 0)$ tie under cosine with $Q = (1, 0)$? **Answer:** They point in the same direction; cosine removes their length difference.
4. **Question:** Why reject a zero vector for cosine in the lab? **Answer:** Its norm is zero, so the normalization denominator and directional comparison are undefined.
5. **Question:** Exact top two are A/B, but candidates contain A/C. What is neighbour recall at two? **Answer:** One recovered exact neighbour divided by two, or 0.5.
6. **Question:** What does the raw 3.072 GB estimate omit? **Answer:** IDs, metadata, index structures, memory overhead, replicas and any additional stored representations.
7. **Question:** Why can global top-two retrieval followed by tenant filtering return nothing? **Answer:** The candidate budget may be filled entirely by ineligible documents before filtering.
8. **Question:** Does similarity 0.92 mean a passage is 92% true? **Answer:** No. It is a metric-dependent score, not a calibrated factual-correctness probability.

45.98 Common wrong ideas

1. **Wrong:** A vector contains an item's complete meaning. **Right:** It is a representation learned or defined for particular objectives.
2. **Wrong:** Equal dimensions make models interchangeable. **Right:** Encoder and preprocessing compatibility must be established.
3. **Wrong:** Inner product is always cosine. **Right:** Normalization and nonzero norms matter.
4. **Wrong:** Exact vector search guarantees a relevant answer. **Right:** It guarantees the selected geometric result, not human usefulness.
5. **Wrong:** Reranking can repair every candidate omission. **Right:** Missing candidates remain unavailable.
6. **Wrong:** An index's raw-vector size is total service memory. **Right:** Index and operating overhead can be substantial.
7. **Wrong:** Permission filters may be relaxed to improve recall. **Right:** Authorization defines the allowed population.
8. **Wrong:** A high-scoring citation proves the generated answer. **Right:** The cited text must actually support each claim.

45.99 Chapter summary in 20 lines

1. Vectors are ordered numerical representations.
2. Their meaning depends on the encoder and preprocessing.
3. Preserve source identity and representation version.
4. Validate dimensions and finite values.
5. Euclidean distance measures separation.
6. Squared distance preserves order but changes the returned quantity.
7. Inner product depends on vector magnitudes.
8. Cosine compares nonzero vector directions.

9. Exact search is exact only under its declared metric.
10. Approximate retrieval can omit closer items.
11. Reranking cannot recover absent candidates.
12. Measure geometric recall separately from human relevance.
13. Graphs and coarse groups trade work for candidate coverage.
14. Compression saves space with additional accuracy and decoding trade-offs.
15. Capacity includes more than raw vector bytes.
16. Filters define the eligible retrieval population.
17. Post-filtering after a small global limit can lose useful results.
18. Retrieved content remains subject to access and trust boundaries.
19. Similarity is not truth, identity or authorization.
20. Inspect the source and verify what the answer actually supports.

Metrics, Bias and Honest Analysis

46.0 What this chapter gives you

1. A metric is a precisely defined question expressed as a number. The calculation can be flawless while the definition, population or interpretation is wrong.
2. You will inspect denominators, missing observations, selection effects and causal claims. You will also calculate an uncertainty interval while keeping clear what the interval's assumptions do and do not cover.
3. All operational comparisons in this chapter are invented teaching examples. They are not observations of KedByte customers, published business performance or evidence that a particular real process causes an improvement.

46.1 A metric is a definition

■ 46.1.1 PLAIN — in simple words

1. “Sales,” “success” and “active user” are labels, not complete calculations. A useful metric specifies which events count, which period they belong to and what is excluded.
2. A number should carry its unit and grain. Orders, order lines, items, money and customers are different quantities even when they come from the same database.
3. Changing a definition can change the result without any change in the underlying business. Such a change should be recorded, not presented as an unexplained improvement or decline.

■ 46.1.2 PLAIN — a picture in your head

1. Mira asks how busy the shop was. Dev can count customers, orders, items or hours of queueing. Each answer describes a different part of “busy.”
2. Before comparing two weeks, they agree to count the same thing under the same rules.
3. **Where the comparison breaks:** digital dashboards can hide definitions behind attractive labels and automatically refreshed charts. The visible number needs a traceable specification, not just a tooltip saying it is accurate.

■ 46.1.3 PLAIN — a worked example

1. The canonical fixture contains two orders, four order lines, six items and 28,650 paise of agreed line amount. None of those counts can be substituted for another without changing the question.
2. Average agreed amount per order is $28,650 / 2 = 14,325$ paise. Per line it is $28,650 / 4 = 7,162.5$ paise. Per item it is $28,650 / 6 = 4,775$ paise.
3. Calling 28,650 “cash received” would assert information the fixture does not contain. The database stores agreed order facts, not evidence of payment settlement.

4. A metric card should therefore state: agreed line amount, INR paise, these four canonical lines, no taxes/discounts/refunds/shipping included, and the selected source and transformation versions.

■ 46.1.4 PLAIN — what is really happening inside

1. A query operationalizes a definition using filters, joins and aggregates. Every step can change the population or multiplicity of contributing records.
2. A metric registry connects a human definition to the query, owner, version and test cases. It helps prevent two teams from using one label for incompatible calculations.
3. Reconciliation should link the displayed value back to its inputs. A dashboard total is more trustworthy when its contributing keys and exclusions can be inspected under appropriate permissions.

■ 46.1.5 TECHNICAL — the engineer's version

1. Specify numerator, denominator, eligibility, time basis, unit, grain, exclusions, missingness policy and aggregation level. Preserve the distinction between an observation and an inference.
2. SQL aggregates operate on the rows produced by prior relational operations. A wrong join or filter can create a coherent but unintended statistic. [S58] [S80]
3. Version metric definitions and record restatements. Comparing an old definition's historical series with a new definition's current value requires an explicit bridge or a recomputed comparable series.

■ 46.1.6 WORDS — remember these

1. **Metric definition:** the exact question behind a number — rules for population, units, timing, inclusion and calculation. **Operationalization:** turning an idea into a measurable procedure — implementing a definition through observable records and calculations. **Restated series:** historical results recalculated under a new rule — a versioned view distinct from what was previously published.

46.2 Denominators and populations

■ 46.2.1 PLAIN — in simple words

1. A percentage needs a denominator: successful out of which opportunities? Changing that population can change the rate substantially.
2. Combining percentages requires attention to group sizes. The average of two group percentages is not generally the percentage across both groups.
3. A comparison also needs comparable populations. If one process handles mostly simple cases and another mostly difficult cases, their overall rates can reflect the case mix rather than the process itself.

■ 46.2.2 PLAIN — a picture in your head

1. Mira compares two counters. One handles ninety simple enquiries and ten difficult ones; the other handles twenty simple and eighty difficult ones.
2. Looking only at each counter's overall completion rate can hide the difference in work assigned to them.

3. Where the comparison breaks: “simple” and “difficult” must be defined before the outcome and applied consistently. Inventing categories afterward can create another misleading comparison.

■ 46.2.3 PLAIN — a worked example

1. In a synthetic two-method example, method A completes 81 of 90 easy cases and 1 of 10 hard cases. Its subgroup rates are 90% and 10%, and its overall rate is $82/100 = 82\%$.
2. Method B completes 19 of 20 easy cases and 16 of 80 hard cases. Its subgroup rates are 95% and 20%, while its overall rate is $35/100 = 35\%$.
3. B’s observed rate is higher within both defined subgroups, but A’s overall rate is higher because A handled a much larger share of easy cases. This is a constructed aggregation reversal, often called Simpson’s paradox.
4. Under a hypothetical common mix of half easy and half hard, the standardized rates are 50% for A and 57.5% for B. These are reweighted descriptive calculations, not proof that changing methods would cause those outcomes.

■ 46.2.4 PLAIN — what is really happening inside

1. Overall rates weight subgroup rates by each subgroup’s share of the denominator. When those shares differ, aggregate comparisons answer a different question from within-group comparisons.
2. Standardization applies a chosen common population mix. It can clarify one source of difference, but the selected weights and subgroup definitions must be disclosed.
3. Not every variable should automatically be adjusted for. A variable affected by the process or a selection rule can change the causal question. Descriptive reweighting is not a substitute for a justified causal design.

■ 46.2.5 TECHNICAL — the engineer’s version

1. For subgroup counts, compute a pooled rate as $\text{sum}(\text{successes}) / \text{sum}(\text{eligible_count})$, not an unweighted mean of rates unless equal group weighting is the actual question.
2. Preserve numerator and denominator in stored aggregates so later regrouping remains possible. An average without its count may be insufficient to compute a correct combined average.
3. The Simpson example is an original arithmetic construction. It demonstrates sensitivity to population composition; it does not estimate a treatment effect or validate the easy/hard classifier for real operations.

■ 46.2.6 WORDS — remember these

1. **Denominator:** the population a rate is out of — the eligible count or exposure giving a numerator its meaning. **Case mix:** the composition of the observed population — subgroup proportions that can influence aggregate comparisons. **Standardization:** applying a common comparison population — reweighting subgroup results under explicitly chosen weights.

46.3 Missingness

■ 46.3.1 PLAIN — in simple words

1. Missing information is not automatically zero, failure or success. It means the required value is unavailable under the current observation process.
2. Ignoring missing records changes the denominator. That may be reasonable for a clearly labelled observed-only statistic, but it does not automatically describe the entire eligible population.

3. The reason information is missing matters. If difficult cases are less likely to be recorded, the observed subset can systematically look better than the full population.

■ 46.3.2 PLAIN — a picture in your head

1. Mira receives delivery-status slips for only ninety of one hundred eligible deliveries. Nine of the returned slips say late and eighty-one say on time.
2. The ten absent slips could change the full-population rate. They cannot be treated as on time merely because no complaint was filed.
3. **Where the comparison breaks:** missingness can arise at several technical stages: capture, transmission, parsing, joins or permission filters. An apparently empty cell does not reveal which stage failed.

■ 46.3.3 PLAIN — a worked example

1. There are 100 eligible deliveries, 81 known on time, 9 known late and 10 with unknown status. The observed-only rate is $81 / 90 = 90\%$.
2. If all unknown statuses are late, the full-population on-time rate is 81%. If all are on time, it is 91%. The data alone therefore bounds the rate between 81% and 91% under the stated binary-status assumption.
3. This range is not a statistical confidence interval. It is a deterministic sensitivity bound over the unresolved records.
4. A dashboard can report “90% among 90 observed statuses; 10 of 100 statuses missing.” That statement preserves both the useful observed statistic and its limitation.

■ 46.3.4 PLAIN — what is really happening inside

1. SQL aggregates often ignore NULL values in specific ways. `COUNT(*)` counts rows, while counting a nullable expression counts its non-null values. The denominator must match the definition.
2. Joins can create or hide missingness. An inner join can silently remove records lacking a matching status row; an outer join retains them so a policy can be applied explicitly.
3. Imputation fills missing values under a model or rule. It can be useful, but the filled values are inferred, not newly observed facts, and their uncertainty should not disappear from the analysis.

■ 46.3.5 TECHNICAL — the engineer’s version

1. Separate unknown, not applicable, not yet observed, invalid and deliberately withheld states when the workflow needs those distinctions. A single NULL field may be insufficient without status metadata.
2. Missingness assumptions affect inference. An observed-only rate represents the observed subset directly; extending it to unobserved cases requires justification beyond the division itself.
3. The companion checks compare observed-only rates with explicit best/worst bounds and reject impossible counts. They do not implement a general missing-data model or assert that an imputation rule recovers the true values. [S80]

■ 46.3.6 WORDS — remember these

1. **Missingness:** required information is unavailable — a property of the observation process whose causes affect interpretation. **Observed-only statistic:** a result using records with available values — a measure whose population excludes unresolved observations. **Sensitivity bound:** a range un-

der alternative assumptions — an explicit calculation showing how unknown information could change a conclusion.

46.4 Selection effects

■ 46.4.1 PLAIN — in simple words

1. The records available for analysis may not represent the population we care about. People, systems or filters can determine which cases become visible.
2. Studying only successful requests cannot reveal the experience of requests that failed before logging. Studying only customers who stayed cannot fully explain those who left.
3. More records do not automatically remove this problem. A huge systematically selected dataset can support a very precise answer to the wrong population question.

■ 46.4.2 PLAIN — a picture in your head

1. Mira asks only customers who return next week whether they liked last week's service. Customers who decided never to return are absent from the survey.
2. The collected opinions are real, but they describe a selected group rather than all customers served.
3. **Where the comparison breaks:** selection can be implemented invisibly by query filters, telemetry sampling and retention rules. The analyst may not notice it by reading the final table alone.

■ 46.4.3 PLAIN — a worked example

1. Suppose 1,000 requests are attempted. The application logs only the 900 that reach its success handler. Of those, 891 complete within the chosen time bound.
2. The logged-only rate is $891 / 900 = 99\%$. Treating all 1,000 attempts as the required denominator gives $891 / 1,000 = 89.1\%$ known timely successes, with the other outcomes needing classification.
3. A dashboard based only on the success table can therefore look excellent while hiding 100 attempts. The arithmetic is not the defect; the observation and eligibility boundaries differ.
4. The remedy is to instrument the complete request lifecycle and reconcile attempts, successes, failures and unknown outcomes under a consistent identity scheme, not simply to rename 99% as reliability.

■ 46.4.4 PLAIN — what is really happening inside

1. Data reaches an analytical table through capture, sampling, filtering and retention. Each stage can select cases based on properties related to the outcome.
2. Duplicate logs can bias the other direction by giving some events extra weight. The intended statistical unit might be a request, user or day rather than one stored log row.
3. Evaluation data can also be contaminated by information unavailable at the intended decision time. A predictor using a field written after completion may appear accurate while being unusable when the real decision must be made.

■ 46.4.5 TECHNICAL — the engineer's version

1. Document the sampling frame and all inclusion/exclusion stages. Trace a record's route into the analysis and inspect cases lost before the final query.

2. For temporal evaluation, preserve the feature-availability boundary: the model or rule should use only information available at the relevant prediction time. Keep development and evaluation decisions separate to reduce overfitting.
3. Repeated observations from one subject or time series can be dependent. NIST's autocorrelation discussion illustrates why order and dependence matter; treating every row as an independent draw can overstate information. [S179]

■ 46.4.6 WORDS — remember these

1. **Selection bias:** the observed group systematically differs from the target — distortion caused by how cases enter or remain in the dataset. **Sampling frame:** the cases available for selection — the actual population from which observations can be drawn. **Information leakage:** using knowledge unavailable at the intended decision point — contamination that can make evaluation results unrealistically strong.

46.5 Association versus cause

■ 46.5.1 PLAIN — in simple words

1. Two quantities changing together establishes an observed association, not automatically that one caused the other.
2. Another factor can influence both, or the direction can run the other way. The data-collection process can also create an apparent relationship.
3. A causal question asks what would change under an intervention. Answering it requires a justified design and assumptions, not just a chart with a sloping line.

■ 46.5.2 PLAIN — a picture in your head

1. Mira notices that busy days have longer queues. She should not conclude that deliberately making people queue will create more demand.
2. Demand may contribute to queue length, while staffing, promotions or holidays influence both. Several stories can fit the same observed pattern.
3. **Where the comparison breaks:** causal reasoning can be formalized with experiments and explicit models. The point is not that causes are unknowable, but that the evidence must distinguish plausible explanations.

■ 46.5.3 PLAIN — a worked example

1. Consider a proposed new checkout workflow. Comparing this month's rate with last month's rate also changes season, staffing, customer mix and perhaps measurement rules.
2. Randomly assigning eligible cases to old and new workflows can help separate assignment from those pre-existing differences, provided the experiment is properly designed and implemented.
3. Randomization does not guarantee identical finite groups, and shared queues can create interference between them. Missing outcomes, noncompliance and changed logging can still damage the interpretation.
4. The synthetic A/B case-mix table in section 46.2 was not randomized. Its subgroup and standardized percentages remain descriptive calculations, not experimentally established causal effects.

■ 46.5.4 PLAIN — what is really happening inside

1. Observational data records what happened under existing selection and assignment processes. Counterfactual outcomes under another action are not directly present in the same record.
2. Experiments define interventions, assignment, outcomes and analysis rules. Observational causal studies require additional assumptions about confounding, timing and selection.
3. Good reporting separates the observed difference from the causal interpretation and states the design limitations. It does not dismiss useful associations, but it does not promote them into stronger claims without evidence.

■ 46.5.5 TECHNICAL — the engineer's version

1. A scatter plot can reveal patterns and possible relationships but does not establish cause and effect. NIST explicitly distinguishes this diagnostic use from causal proof. [S180]
2. Define the estimand: the population, intervention, outcome and contrast being sought. Choosing covariates or subgroups should follow that causal question rather than automatically adjusting for every available field.
3. The book does not estimate a real treatment effect. Its examples teach the separation between arithmetic, descriptive association and causal inference, leaving a real study's assumptions and review to its actual context.

■ 46.5.6 WORDS — remember these

1. **Association:** quantities vary together in observations — a statistical relationship that does not by itself establish intervention effects. **Confounder:** a factor affecting the causal comparison — a variable related to both assignment/exposure and outcome in a relevant causal structure. **Estimand:** the precise quantity a study seeks — a defined population-level contrast under specified interventions or conditions.

46.6 Communicating uncertainty

■ 46.6.1 PLAIN — in simple words

1. A reported number can be uncertain because only a sample was observed, measurements are imperfect or records are missing. These are different sources of uncertainty.
2. An interval calculated from a sampling model covers only the uncertainty represented by that model. It does not repair biased selection, incorrect labels or a changing process.
3. Explain the result with its denominator, assumptions and practical limits. Extra decimal places should not create the appearance of knowledge the evidence does not contain.

■ 46.6.2 PLAIN — a picture in your head

1. Mira checks a sample of packages rather than every package. She can calculate how much the observed proportion might vary under repeated comparable sampling.
2. If she checked only the easiest packages to reach, a narrow interval around that selected sample does not solve the selection problem.
3. **Where the comparison breaks:** statistical coverage is a property of a procedure under assumptions. It is not a promise that every particular interval contains the fixed unknown quantity.

■ 46.6.3 PLAIN — a worked example

1. In a separate idealized sample, 90 of 100 independent, comparably sampled cases satisfy a binary condition. The observed proportion is 0.9.
2. Using a Wilson interval with $z = 1.96$ gives an approximate two-sided 95% interval from 0.826 to 0.945. This calculation assumes a suitable binomial sampling model and is not the same as the missing-status bound from section 46.3.
3. The Wilson centre is $(p + z^2/(2n)) / (1 + z^2/n)$. Its half-width is $z \times \sqrt{(p(1-p)/n + z^2/(4n^2)) / (1 + z^2/n)}$. Substituting $p = 0.9$ and $n = 100$ produces the stated rounded limits.
4. The companion code checks the arithmetic, edge cases and valid range. It does not establish that real operational observations satisfy independence, stable probability or representative sampling.

■ 46.6.4 PLAIN — what is really happening inside

1. Sampling intervals describe repeated-sampling performance under a model. Measurement uncertainty instead concerns how values are attributed from instruments and observations; missingness bounds explore unresolved cases.
2. Dependence reduces the information obtained from repeated observations compared with an ideal independent sample of the same row count. A thousand correlated readings are not automatically a thousand independent pieces of evidence.
3. Report uncertainty together with decision relevance. A small numerical difference may be operationally unimportant, while a modest uncertainty range near a safety or capacity limit may matter greatly.

■ 46.6.5 TECHNICAL — the engineer's version

1. NIST presents the Wilson interval and discusses alternatives for small samples or few failures. Avoid the naive interpretation that a confidence interval assigns a 95% posterior probability to a fixed parameter without a Bayesian model. [S183]
2. A confidence procedure's nominal coverage is conditional on its statistical assumptions and can be approximate. Selection, dependence, repeated unplanned testing and definition changes can invalidate a simplistic interpretation. [S179]
3. Preserve raw counts, calculation method, source version and rounding policy. Label unresolved measurement and data-quality limitations separately instead of folding them into an unjustified single "confidence" percentage.

■ 46.6.6 WORDS — remember these

1. **Confidence interval:** a range from a sampling procedure — an interval method with stated repeated-sampling coverage under assumptions. **Wilson interval:** a binomial-proportion interval — a score-based method whose limits remain within the valid probability range. **Statistical unit:** the entity providing an observation — the request, person, item or time unit whose independence and weighting need definition.

46.97 Practice and worked answers

1. **Question:** What does 28,650 paise mean in the canonical fixture? **Answer:** The agreed amount of the four included order lines, not verified cash collection.
2. **Question:** Why does averaging subgroup percentages often fail? **Answer:** It ignores subgroup denominator sizes unless equal group weighting is the intended question.

3. **Question:** What explains the A/B aggregate reversal? **Answer:** Different easy/hard case mixtures. The constructed arithmetic does not establish a causal effect.
4. **Question:** With 81 on time, 9 late and 10 unknown out of 100, what is observed-only performance? **Answer:** 90% among the 90 observed statuses; the full-population binary bound is 81% to 91%.
5. **Question:** Is that 81%–91% range a confidence interval? **Answer:** No. It is a sensitivity bound over missing outcomes.
6. **Question:** Why can success-only logs mislead? **Answer:** They omit attempts that never reached the success handler, changing the denominator and population.
7. **Question:** Does a sloping scatter plot establish cause? **Answer:** No. It shows association; causal interpretation needs additional design and assumptions.
8. **Question:** What does the Wilson calculation fail to verify? **Answer:** Whether the real observations satisfy the sampling, independence, measurement and stability assumptions required for its interpretation.

46.98 Common wrong ideas

1. **Wrong:** A familiar metric name is a complete definition. **Right:** Population, timing, units and exclusions must be explicit.
2. **Wrong:** A correct formula guarantees a correct conclusion. **Right:** The formula may answer the wrong question.
3. **Wrong:** Group percentages can always be averaged directly. **Right:** Denominator weights matter.
4. **Wrong:** Missing means failure or zero. **Right:** Missingness needs a declared interpretation and may require separate status.
5. **Wrong:** More rows automatically remove bias. **Right:** Systematic selection can persist at any scale.
6. **Wrong:** Association is evidence enough for intervention effects. **Right:** Causal conclusions require stronger assumptions or design.
7. **Wrong:** Every repeated measurement is independent. **Right:** Dependence can reduce effective information.
8. **Wrong:** A confidence interval covers every source of uncertainty. **Right:** It covers only what the selected statistical model represents.

46.99 Chapter summary in 20 lines

1. A metric is a defined question, not only a label.
2. Preserve the distinction between orders, lines, items and money.
3. State the numerator, denominator and eligible population.
4. Keep units and time basis explicit.
5. Version definition changes and historical restatements.
6. Pool counts before calculating a pooled rate.
7. Case mix can reverse aggregate comparisons.
8. Standardization is a declared reweighting, not automatic causal proof.
9. Missing observations do not become known outcomes by convenience.
10. Report observed-only statistics with their coverage.
11. Sensitivity bounds and confidence intervals answer different questions.
12. Selection can occur before data reaches the final table.

13. Success-only logs can hide unsuccessful attempts.
14. Statistical units are not always individual stored rows.
15. Association does not by itself establish cause.
16. Define the intervention and estimand for causal questions.
17. Sampling intervals rely on explicit assumptions.
18. Dependence and selection are not cured by extra decimal places.
19. Preserve counts, methods and unresolved limitations.
20. Communicate the strongest conclusion the evidence actually supports.



PART H

Running Data Systems

Operate, change, protect and retire a data system, then assemble the complete case study.

Permissions, Secrets and Tenant Boundaries

47.0 What this chapter gives you

1. Finding a record and being allowed to read it are different questions. A correct query can return information to the wrong person. This chapter adds an explicit permission decision to the data machinery developed so far.
2. You will distinguish identity verification, action permission, database privileges and tenant separation. You will trace a request across the application, database, cache, export and background worker rather than treating the login screen as the whole boundary.
3. The examples extend the fictional shop into a teaching service with two tenants, T-A and T-B. They are independent organisations in this example, not automatically two branches of the same organisation. A branch restriction may be an additional rule inside a tenant.
4. The companion capstone trusts a small principal object supplied by its tests. It demonstrates scoped operations in an isolated SQLite database. It does not implement login, session verification, PostgreSQL row security or a production security perimeter.

47.1 Who may do what

■ 47.1.1 PLAIN — in simple words

1. Authentication asks who is making a request. Authorisation asks whether that person or program may perform this particular action on this particular information now. Knowing a person's name does not settle the second question.
2. An order may be readable by a shop assistant but not changeable after acceptance. An accountant may see totals without seeing delivery notes. A support operator may investigate a failed job without downloading every customer's records.
3. Write permissions as specific actions. "Can access orders" is incomplete: reading one order, listing all orders, exporting them and changing their prices have different consequences.
4. A safe decision needs a trustworthy identity, the requested action, the target resource and relevant context. When essential context is absent or invalid, do not invent a permissive interpretation.

■ 47.1.2 PLAIN — a picture in your head

1. Imagine a building receptionist who recognises Dev. Recognition lets the receptionist establish who arrived; it does not give Dev the keys to every room.
2. A separate assignment says which rooms Dev may enter and what work he may perform there. The assignment can expire even though Dev remains the same person.

3. **Where the comparison breaks:** software has many entrances that do not look like doors: APIs, exports, scheduled jobs, search endpoints and cached responses. Hiding a button blocks none of those routes by itself.

■ 47.1.3 PLAIN — a worked example

1. Give the synthetic principal dev-a membership in T-A and the actions order.read and order.create. Give reviewer-a order.read only. Neither principal belongs to T-B.
2. dev-a reading T-A order CAP-001 is permitted if the order exists and any additional branch rule passes. reviewer-a creating a new order is denied even inside T-A. dev-a reading T-B CAP-001 is also denied, despite an identical local order identifier.
3. Now revoke dev-a's membership. The next operation must use the service's declared revocation policy. A cached membership valid for another hour would create a one-hour revocation delay; that delay must not be described as immediate revocation.
4. A compact decision table should include a valid same-tenant action, a forbidden action, a wrong tenant, an absent principal, an expired assignment and an unknown resource. These are different cases, not six names for one test.

■ 47.1.4 PLAIN — what is really happening inside

1. The application receives a credential or authenticated session, resolves a principal and evaluates a policy. It then performs the action using only the allowed scope. The client-supplied URL or form is not authoritative evidence of that scope.
2. A policy can depend on role, ownership, membership, branch, lifecycle state or an explicit grant. Each fact has an origin and a freshness requirement. Conflicting or stale facts need a defined decision, not whichever answer is convenient.
3. Decisions and actions can be separated by time. If the target or assignment changes between them, the earlier decision may no longer justify the later action. Where that matters, recheck relevant conditions at the protected operation or bind them to a versioned transaction boundary.

■ 47.1.5 TECHNICAL — the engineer's version

1. OWASP distinguishes authentication from authorisation and recommends default denial, permission checks on every request and tests for access-control logic. Those principles motivate, but do not fully specify, our original permission matrix. [S185]
2. Model a decision as allow(principal, action, resource, context). A successful decision is not transferable to a different resource, operation or tenant unless the policy explicitly grants that wider scope.
3. The local capstone receives a trusted Principal value directly from its test harness. Constructing that object is not authentication. A real adapter must validate its session or token, issuer, audience, expiry and relevant membership before it invokes the operation.

■ 47.1.6 WORDS — remember these

1. **Principal:** the person or program asking — an authenticated or otherwise explicitly established security identity used in policy evaluation. **Authorisation:** permission for a particular act — evaluation of a principal's rights over a resource in a defined context. **Revocation delay:** time before a withdrawn permission stops working — the exposure window introduced by cached or long-lived authority.

47.2 Least privilege

■ 47.2.1 PLAIN — in simple words

1. Give each person and program the permissions needed for its job, not every permission that would make development easier. A reporting task should not need to rewrite orders merely to read totals.
2. Separate routine work from exceptional maintenance. The program serving customer requests and the operator changing table definitions have different responsibilities and should not automatically share one powerful account.
3. A permission can spread through membership in another group. Removing one direct grant does not necessarily remove the same permission inherited elsewhere.
4. Least privilege is a continuing maintenance task. Old integrations and temporary repair accounts can accumulate privileges after their original purpose has ended.

■ 47.2.2 PLAIN — a picture in your head

1. Mira gives the cleaner a key to the storage room, not authority to alter the shop's agreements. The locksmith needs broader access for a repair, but that access is temporary and supervised.
2. Keeping the locksmith's master key beside the till would erase the benefit of carefully issued smaller keys.
3. **Where the comparison breaks:** software privileges can be inherited through roles, functions and deployment tools. The visible account name may conceal more authority than its direct permissions suggest.

■ 47.2.3 PLAIN — a worked example

1. Define three illustrative database identities: `app_runtime`, `report_reader` and `schema_operator`. `app_runtime` performs approved application changes; `report_reader` runs scoped reads; `schema_operator` performs reviewed schema work outside ordinary request handling.
2. Suppose `report_reader` belongs to a broad legacy group with `UPDATE` privileges. Revoking its direct `UPDATE` grant does not demonstrate that updates are impossible. Inspect and test the effective privilege path, including inherited rights.
3. A migration that creates a new table must also decide its owner and grants. Testing the old tables says nothing about the new table until its effective permissions have been checked.
4. An operator review records the identity, permitted operations, scope, expiry where appropriate and a successful denied-operation test. A screenshot of a role name alone proves little.

■ 47.2.4 PLAIN — what is really happening inside

1. Database privileges govern operations on objects such as schemas, tables, columns and functions. Ownership and role membership can supply additional authority beyond a single visible grant.
2. Applications often connect using a shared database identity. In that arrangement, the database does not automatically know which human is behind each request. The application must preserve that distinction or establish a trustworthy database-level context.
3. Powerful maintenance code can legitimately cross normal boundaries. Its input handling, caller checks and execution identity therefore deserve separate review; calling it a helper does not make its authority harmless.

■ 47.2.5 TECHNICAL — the engineer's version

1. PostgreSQL 17 documents privileges, object ownership and GRANT/REVOKE semantics. Object privileges and row-level policy are complementary: the former controls allowed operations on objects; the latter can restrict rows for applicable commands. [S186]
2. A shared-table design requires an audit of effective runtime authority, including ownership and roles that bypass row policies. Do not assess the request path using only a superuser test session.
3. Capability-style designs can pass narrower handles instead of a global unrestricted connection. That is an application design choice, not a feature automatically supplied by naming a Python variable `read_only`.

■ 47.2.6 WORDS — remember these

1. **Least privilege:** only the authority needed — minimising effective permissions and their duration for a defined responsibility. **Effective privilege:** what the identity can actually do — the result of direct grants, memberships, ownership and special execution rules. **Runtime identity:** the account serving ordinary work — the database or service identity used by the request-handling process.

47.3 Database and application checks

■ 47.3.1 PLAIN — in simple words

1. An application can check a rule before asking the database to act. A database can also enforce selected rules when the action reaches it. These checks protect different parts of the path.
2. Repeating the same mistaken assumption in two places is not independent protection. Both layers must use trustworthy context and agree about the intended rule.
3. Database checks are especially valuable against forgotten application predicates. They still do not protect an export already copied elsewhere or an unrestricted administrator acting outside the normal path.

■ 47.3.2 PLAIN — a picture in your head

1. The receptionist checks a visitor's assignment, and a room door checks a badge. A mistaken receptionist decision need not open the door if the badge genuinely lacks permission.
2. But if the receptionist can print any badge without restriction, the second check offers less separation than it first appears to.
3. **Where the comparison breaks:** a database policy can inspect a session setting established by the application. A caller able to change that setting may be able to impersonate another scope unless the surrounding boundary prevents it.

■ 47.3.3 PLAIN — a worked example

1. Consider this original PostgreSQL teaching sketch, not an executed server configuration. `trusted_tenant()` represents a separately reviewed, non-spoofable context mechanism; it is deliberately not defined as "whatever the request says."

```
ALTER TABLE scoped_orders ENABLE ROW LEVEL SECURITY;
ALTER TABLE scoped_orders FORCE ROW LEVEL SECURITY;
CREATE POLICY scoped_order_policy ON scoped_orders
  USING (tenant_id = trusted_tenant())
  WITH CHECK (tenant_id = trusted_tenant());
```


2. USING governs which existing rows pass the policy for applicable operations. WITH CHECK constrains proposed row values. A write must not be allowed to move a row into another tenant merely because the original row was visible.
3. Test through the real intended runtime role and connection pool. Also test omitted context, reused connections and an attempted tenant change. Running the sketch as an unrestricted superuser does not establish tenant isolation.
4. This policy sketch is incomplete as a deployable configuration. Roles, table privileges, context construction, function execution rights and all other policies must be reviewed together.

■ 47.3.4 PLAIN — what is really happening inside

1. A policy predicate participates in query processing. It may filter rows or reject a proposed modification, depending on the command and policy definition.
2. Application checks add business meaning: whether an order is editable, whether a refund needs another approval, or whether the chosen export purpose is permitted. Those meanings are not inferred from a tenant column alone.
3. Connection pooling creates a lifecycle problem. Any per-request context must be established and cleared in a scope that prevents one request from inheriting another's identity, including error and rollback paths.

■ 47.3.5 TECHNICAL — the engineer's version

1. In PostgreSQL 17, enabled row security without an applicable policy defaults to denying row access. Superusers and BYPASSRLS roles bypass it; owners normally bypass it unless FORCE is used. Whole-table operations and integrity checks have additional documented exceptions. [S187]
2. Permissive policies combine with OR and restrictive policies with AND, within the documented applicability rules. An accidentally broad permissive policy can therefore defeat a narrow policy's intended result. [S187]
3. The SQLite capstone uses explicit application predicates instead of PostgreSQL row policies. Its tests establish behaviour through that helper, not safety against someone holding an unrestricted database connection.

■ 47.3.6 WORDS — remember these

1. **Row-level security:** a filter or rule on individual rows — database policy enforcement beyond ordinary object privileges. **Connection context:** identity-related state attached to database work — information whose establishment, reuse and clearing form part of the security boundary. **Policy bypass:** an execution path exempt from a rule — special authority that must be accounted for when testing enforcement.

47.4 Tenant separation

■ 47.4.1 PLAIN — in simple words

1. A tenant is a separately scoped organisation or customer group in a shared service. The service may share machinery, but one tenant must not automatically inherit another's records or authority.
2. Scope belongs in more places than a SELECT statement. It affects record identities, references, duplicate-request keys, caches, object names, exports and background jobs.
3. Two tenants may both have an order called CAP-001. Treating CAP-001 alone as globally unique can mix their work, even when each tenant's own data is internally consistent.

4. A tenant boundary and a branch boundary answer different questions. A manager allowed across branches inside T-A is not thereby allowed into independent tenant T-B.

■ 47.4.2 PLAIN — a picture in your head

1. Two shops rent separate rooms in one warehouse. Each room has a shelf labelled 17. “Get the item from shelf 17” is incomplete without the room.
2. A delivery note must carry the room as well as the shelf. Otherwise the warehouse worker can follow the note faithfully and still take the wrong item.
3. **Where the comparison breaks:** shared services can duplicate and transform records into caches or reports. The tenant scope must survive those transformations; it is not a physical wall that stays attached automatically.

■ 47.4.3 PLAIN — a worked example

1. The capstone uses keys such as (tenant_id, order_id) and (tenant_id, request_id). A request called R-1 in T-A is not a replay of R-1 in T-B.
2. A child line references its parent’s complete scoped key. Otherwise a syntactically valid child could point to a parent in another tenant.
3. A cache key might include an explicit tuple of tenant, resource identity and representation version. For user-specific responses it may need additional permission context, or the design should avoid sharing that cached representation.
4. Searching across all tenants, selecting the top ten matches and then removing forbidden results can also change the answer. Apply authorised eligibility before the result limit when that is the intended query contract, and test empty and narrow scopes.

■ 47.4.4 PLAIN — what is really happening inside

1. Tenant context is established at the trusted boundary and travels with the operation. A job producer records it; a worker validates that the job is allowed; every derived resource is named and queried within that scope.
2. Database constraints can prevent some cross-tenant associations through composite keys. They cannot decide whether the supplied tenant truly belongs to the authenticated principal without an authority mechanism.
3. Separate databases offer a different isolation boundary from shared tables. They also change maintenance and reporting costs. Neither architecture is automatically secure if administrative credentials, exports or backup access are shared carelessly.

■ 47.4.5 TECHNICAL — the engineer’s version

1. The original shared-table capstone includes tenant in relevant primary and foreign keys and in application predicates. Its boundary is the trusted helper interface, not arbitrary SQL.
2. OWASP’s multi-tenant guidance identifies tenant context and separation across data access and shared infrastructure as concerns beyond authentication alone. Our branch/tenant fixtures are original examples of that distinction. [S188]
3. For each store, document the isolation unit, identity source, access mechanism and privileged bypass path. This inventory is more informative than a blanket claim that the architecture is tenant-safe.

■ 47.4.6 WORDS — remember these

1. **Tenant:** one separately scoped customer organisation — an isolation domain within a service that may share infrastructure. **Scoped key:** an identifier plus its domain — a composite identity such as `tenant_id` and `order_id`. **Confused deputy:** a privileged helper misused through supplied instructions — an authority-bearing component acting for the wrong caller or scope.

47.5 Secret handling

■ 47.5.1 PLAIN — in simple words

1. A secret is information whose possession grants a capability or exposes protected data: a database password, signing key or service token, for example. A username or public key need not be secret merely because it looks technical.
2. Keep secrets out of books, sample files, public repositories, logs and screenshots. A placeholder should be unmistakably a placeholder, not a copied working credential.
3. Restrict who can obtain each secret, which systems may use it and how long it remains valid. Replacing a leaked string in the source file does not necessarily invalidate copies already made.
4. Plan rotation and revocation before an incident. A service that cannot replace its credentials without losing all availability has a serious operational dependency.

■ 47.5.2 PLAIN — a picture in your head

1. A safe combination should not be taped to the safe. Moving the note into a drawer helps only if everyone who could see the tape can no longer open the drawer.
2. If the combination was photographed, shredding the original note does not make the photograph stop working. The combination itself needs changing.
3. **Where the comparison breaks:** rotating an encryption key is not always equivalent to changing a password. Existing ciphertext may still require an older key or a carefully managed re-encryption process.

■ 47.5.3 PLAIN — a worked example

1. The production design proposal gives the reporting job a short-lived, read-scoped credential obtained from a protected service identity. The book supplies no such credential and makes no call to a secret manager.
2. A rotation rehearsal records when the replacement became usable, which consumers moved to it and when the old authority was revoked. A successful new connection does not by itself demonstrate that old access is disabled.
3. Suppose a log captured an authorization header. Treat log storage and its readers as part of the exposure, revoke the affected credential under the incident procedure and review why redaction failed. Merely deleting one log line is not a complete response.
4. The lab uses synthetic principal names only. Its absence of passwords is not proof of a secure login implementation; login is outside the exercise.

■ 47.5.4 PLAIN — what is really happening inside

1. A secret has a lifecycle: creation, distribution, use, rotation, revocation and disposal. Each step has permissions, evidence and failure cases.

2. Applications should log operational facts without recording the capability itself. Error handlers, tracing middleware and debug dumps deserve attention because they can expose values the normal path hides.
3. A deployment system can inject secrets without placing them in source, but the runtime still has some means of using them. Restricting process access, crash dumps and administrative inspection remains relevant.

■ 47.5.5 TECHNICAL — the engineer's version

1. OWASP's secrets-management guidance covers centralised handling, lifecycle controls, access restriction and rotation. A specific deployment must still establish how workload identity reaches that mechanism and how a failed rotation behaves. [S189]
2. Distinguish authentication credentials from encryption keys and signing keys. Their consumers, compromise consequences and retirement procedures differ; one generic rotate_secret function does not settle all three.
3. The public companion code contains no production credentials. Its tests do not validate an external secret manager, hardware security module or production key-retirement procedure.

■ 47.5.6 WORDS — remember these

1. **Secret:** protected capability-bearing information — a credential or key requiring controlled disclosure and use. **Rotation:** replacing active secret material — transitioning consumers to new authority under an explicit lifecycle. **Revocation:** withdrawing previously valid authority — disabling future accepted use, distinct from deleting one stored copy.

47.6 Adversarial boundary tests

■ 47.6.1 PLAIN — in simple words

1. A test that an authorised user succeeds is only half the story. Also test that a nearby unauthorised request fails without changing or revealing protected data.
2. Change one important input at a time: tenant, object identifier, action, membership, version or lifecycle state. That reveals which boundary caused the decision.
3. Repeat the tests through every supported path. A secure detail page does not prove a secure export, bulk endpoint or background worker.
4. A rejected response is not enough if the operation already changed data. Inspect the protected state as well as the returned error.

■ 47.6.2 PLAIN — a picture in your head

1. Mira tests a locked room by entering with the correct key, then trying a wrong key, an expired pass and an open side entrance.
2. She also checks whether the attempted entry changed the lock or left the side door open. Hearing "access denied" from a guard does not establish what happened elsewhere.
3. **Where the comparison breaks:** timing, error text and aggregate counts can reveal information without returning a whole record. A test suite has to name the observable channels it actually checks.

■ 47.6.3 PLAIN — a worked example

1. Create T-A CAP-001 and T-B CAP-001 with deliberately different synthetic contents. Read using an authorised T-A principal and assert that only T-A's contents appear.
2. Attempt a T-B read and write with that same principal. Assert denial and compare both tenants' records before and after. A cross-tenant write must not quietly become a successful zero-row business operation.
3. Reuse R-1 independently in both tenants. Assert that each tenant receives its own result. Reuse R-1 inside T-A with a changed payload and assert a conflict without a second order.
4. Supply a principal missing the required action and inspect that no request/outbox record was created. These controls make the capstone's scoped helper behaviour observable, while leaving real identity-provider tests to a deployment-specific adapter.

■ 47.6.4 PLAIN — what is really happening inside

1. The test constructs known protected data, invokes one path under known authority and compares outputs and state with explicit expectations. Its evidence concerns that configuration and path.
2. A bypass test deliberately invokes a lower layer to show the boundary. The SQLite connection can execute unrestricted SQL, so possession of that connection is outside the helper's tenant guarantee.
3. Release evidence should identify the tested revision and runtime. A passing result from last week's code does not automatically describe today's additional export endpoint.

■ 47.6.5 TECHNICAL — the engineer's version

1. Organise tests by principal × action × resource scope × state. Add negative and boundary cases rather than only generating many random inputs without a policy oracle.
2. Record denied-operation effects, error-shape expectations and tested interfaces. Avoid describing a narrow unit suite as penetration testing or an independent security audit.
3. The final companion inventory identifies the executable checks. The production checklist separately calls for authentication integration, pooled-context lifecycle tests, real database role tests, export/worker coverage and review of privileged paths. [S185]

■ 47.6.6 WORDS — remember these

1. **Negative control:** a case that should be refused — a deliberately unauthorised or invalid operation with a checked outcome. **Policy oracle:** the expected permission decision — the independently stated rule used to judge a test result. **Boundary coverage:** which protected paths were exercised — an inventory of tested interfaces, scopes and states, not a universal guarantee.

47.97 Practice and worked answers

1. **Question:** dev-a belongs to T-A. Does a caller-supplied tenant_id of T-B change that membership? **Answer:** No. Scope comes from trusted authority, not from the target supplied in a request.
2. **Question:** Why is (tenant_id, request_id) preferable to request_id alone in this shared-table example? **Answer:** Independent tenants can legitimately choose the same request identifier; their replay histories must not collide.
3. **Question:** What is missing from a successful PostgreSQL test run as superuser? **Answer:** Evidence about the intended restricted role, since superuser bypasses ordinary row-policy enforcement.

4. **Question:** Does hiding the export button prevent an unauthorised export request? **Answer:** No. The server-side export path needs its own enforced permission boundary.
5. **Question:** Does replacing a leaked password in a configuration file revoke existing copies? **Answer:** Not necessarily. The credential's authority must be invalidated and affected consumers updated under the incident procedure.
6. **Question:** What does a zero-row update establish? **Answer:** Only that no row matched the applied operation. It is not automatically a successful business update or a sufficiently private denial.
7. **Question:** Can a branch manager in T-A read all of T-B? **Answer:** Not under the example's policy. Branch scope within one tenant does not grant another tenant's authority.
8. **Question:** What is the strongest claim from the capstone's access tests? **Answer:** The stated scoped helper accepts and rejects the supplied synthetic cases in its tested runtime. It does not implement or certify a complete security perimeter.

47.98 Common wrong ideas

1. **Wrong:** Logged in means allowed. **Right:** Each protected action still needs a permission decision.
2. **Wrong:** A difficult-to-guess identifier is access control. **Right:** Knowledge of an identifier is not permission to use it.
3. **Wrong:** Every row-policy test should use the administrator account. **Right:** Test the actual runtime role and its exceptions.
4. **Wrong:** Tenant filtering belongs only in the main table. **Right:** Derived stores and background paths need scoped identities too.
5. **Wrong:** Revoking a direct grant removes every route to authority. **Right:** Inheritance, ownership and special execution paths may remain.
6. **Wrong:** A secret removed from a repository is no longer exposed. **Right:** Existing copies and remaining authority require separate treatment.
7. **Wrong:** An error proves that nothing changed. **Right:** Inspect effects and transaction outcomes.
8. **Wrong:** Passing unit tests means the application is secure. **Right:** Evidence is bounded by the paths, roles, configuration and threats actually tested.

47.99 Chapter summary in 20 lines

1. Authentication and authorisation answer different questions.
2. Name the principal, action, resource and relevant context.
3. Establish tenant scope at a trusted boundary.
4. Deny operations whose required authority is absent.
5. Separate reading, writing, exporting and administrative actions.
6. Treat revocation freshness as a declared operational property.
7. Minimise effective privileges, not only visible direct grants.
8. Separate runtime and schema-maintenance identities.
9. Application and database checks protect different layers.
10. Test database policies using the intended restricted role.
11. Account for ownership and bypass exceptions.
12. Prevent pooled connections from retaining another request's context.
13. Include tenant scope in relevant keys and references.

14. Preserve that scope through caches, jobs, search and exports.
15. Keep branch restrictions distinct from tenant boundaries.
16. Give secrets a complete controlled lifecycle.
17. Rotation and revocation are not the same as deleting a file.
18. Test refused operations and their effects on stored state.
19. State exactly which interfaces and threats a test covers.
20. A bounded teaching helper is not a complete production security system.

Retention, Deletion and Data Lineage

48.0 What this chapter gives you

1. Information has a useful lifetime. Keeping it forever can create unnecessary exposure; deleting it carelessly can destroy evidence, break references or make recovery impossible. The job is to define and implement a lifecycle rather than choose one slogan.
2. You will separate hiding a record, deleting a logical row, removing derived copies and sanitising storage media. You will also learn why an empty query result is not a universal certificate of erasure.
3. The policies and dates in this chapter are synthetic engineering examples. They are not legal retention periods, advice about a jurisdiction or a statement that a particular real organisation may delete a record. Actual legal, contractual and operational requirements need their own assessment.
4. The continuing shop's agreed order facts are not silently erased or rewritten here. The deletion exercise uses a separately introduced optional contact note, NOTE-7, and a finite inventory of stores. This keeps lifecycle learning separate from the canonical order totals.

48.1 Purpose and record classes

■ 48.1.1 PLAIN — in simple words

1. Start by asking why each kind of information exists. An order line, temporary import file, optional delivery note and access log serve different purposes and need not have the same lifetime.
2. A field can be useful at one stage and unnecessary later. Keeping a temporary instruction inside every downstream report can spread information beyond the work it was collected to support.
3. Decide the unit of lifecycle control. Sometimes an entire record is retired; sometimes an optional annotation is removed while a separately justified transaction record remains.
4. The purpose must describe actual use. Calling everything “for analytics” does not explain which question requires which fields, who may use them or when that use ends.

■ 48.1.2 PLAIN — a picture in your head

1. Mira keeps an agreed order form, a packing checklist and a sticky note saying where to leave a parcel. The order form may support one responsibility, while the sticky note serves only a particular delivery.
2. Copying the sticky note into every monthly sales report does not make the report more correct. It creates another copy that someone must control.

3. **Where the comparison breaks:** digital transformations can preserve identifying information indirectly through keys, small groups or joins. Removing a visible name does not automatically make the remaining data anonymous.

■ 48.1.3 PLAIN — a worked example

1. Introduce three fictional record classes: `agreed_order_line`, `optional_contact_note` and `failed_import_payload`. Give each an owner, stated purpose, access scope and lifecycle rule. No real legal period is assigned by the book.
2. NOTE-7 belongs to tenant T-A and a synthetic contact. Its purpose is one temporary follow-up. It is not required to calculate O-1042 or O-1043, whose agreed amounts remain 17,100 and 11,550 paise.
3. If NOTE-7 is copied into a search index and a support export, those are additional lifecycle responsibilities. The source table alone is no longer the complete inventory.
4. A record-class card can state: class, purpose, owner, collected fields, allowed uses, authorised readers, lifecycle trigger, exceptions, stores and evidence required for closure. A missing owner is itself an unresolved operational question.

■ 48.1.4 PLAIN — what is really happening inside

1. Schemas and pipelines decide where a field travels. A lineage description connects original inputs, transformations and derived outputs so lifecycle work can follow those routes.
2. Separating optional annotations from durable agreement facts can make independent lifecycle actions easier. The split must preserve valid references and document what an order means after an annotation disappears.
3. Minimisation can occur before storage as well as afterward. A transformation that never copies an unnecessary field avoids a future deletion problem in that destination.

■ 48.1.5 TECHNICAL — the engineer's version

1. Treat classification as an application policy, not a property automatically inferred from a database type. TEXT can contain harmless labels or sensitive free-form notes; the engine does not determine their purpose.
2. OpenLineage's object model distinguishes jobs, runs and datasets, providing a vocabulary for recording input/output relationships. It does not itself prove that every copy has been discovered or that a deletion was executed. [S193]
3. The chapter's class cards and deletion inventory are original teaching designs. Their adequacy depends on the declared system boundary and the requirements supplied by its owners.

■ 48.1.6 WORDS — remember these

1. **Record class:** a group with a common responsibility — data categorised by purpose, access and lifecycle requirements rather than file extension alone. **Purpose limitation:** using information for a defined need — an explicit boundary on collection and downstream use. **Data minimisation:** keeping only what the task needs — reducing unnecessary fields, copies or lifetime under a stated requirement.

48.2 Retention policies

■ 48.2.1 PLAIN — in simple words

1. A retention rule needs a start event, a duration or end condition, and an action. “Delete after thirty days” is ambiguous until we know thirty days after what.
2. Creation time, completion time and last legitimate activity may be different dates. Choosing the wrong clock can retire active work or keep abandoned information indefinitely.
3. Exceptions need an explicit basis, owner and review point. A permanent unexplained hold can turn a time-limited policy into “keep forever” without anyone making that decision openly.
4. A scheduled job is an implementation of a policy, not the policy itself. The organisation must still decide what should happen and verify that the job does it.

■ 48.2.2 PLAIN — a picture in your head

1. A library loan lasts fourteen days after checkout, not fourteen days after the book was printed. The event starting the clock changes everything.
2. A renewal changes the due date under a rule. Someone rubbing out the date on the card is not a valid renewal process.
3. **Where the comparison breaks:** a digital object can be referenced by several workflows with different completion conditions. One clock may not describe all legitimate uses of shared information.

■ 48.2.3 PLAIN — a worked example

1. For NOTE-7 only, invent a policy of expiry thirty elapsed days after its follow-up is closed. Let the synthetic closure instant be 1 September 2026 at 12:00 UTC. The resulting expiry is 1 October 2026 at 12:00 UTC.
2. This is elapsed-time arithmetic. A policy based on local calendar dates could produce a different boundary and must specify its time zone and treatment of daylight-saving transitions where relevant.
3. A hold on NOTE-7 suspends the deletion action under the chosen policy, but the hold needs a reason code, owner and review deadline. The automated system should not invent the reason.
4. Test one instant before expiry, exactly at expiry and after expiry. Also test an absent closure time and an active hold. In our example, unknown timing means review required, not an invented old date.

■ 48.2.4 PLAIN — what is really happening inside

1. A policy evaluator maps current record facts and policy version to an eligibility decision. The execution worker then attempts the corresponding action within a bounded batch.
2. Eligibility can change between selection and action. A newly recorded hold, for example, must not be ignored simply because the row appeared in yesterday’s export. Recheck conditions at the operation boundary where the design requires it.
3. Record the policy version used. When a policy changes, previously scheduled work may need reevaluation rather than blindly executing an obsolete list.

■ 48.2.5 TECHNICAL — the engineer’s version

1. Use explicit aware instants for elapsed-time examples and retain the original event-time semantics. A timestamp’s storage format does not decide whether the policy is elapsed-time or calendar-based.

2. Our companion evaluator rejects invalid types and negative durations, treats a missing closure instant as unresolved and makes the equality boundary explicit. Those checks test the arithmetic policy only, not a legal obligation.
3. Batch deletion should be restartable with bounded work and an auditable selection predicate. A large unqualified DELETE is not a lifecycle programme merely because it runs every night.

■ 48.2.6 WORDS — remember these

1. **Retention trigger:** the event starting a lifecycle rule — a defined transition such as closure, not an assumed timestamp. **Hold:** an explicit suspension of an action — a governed exception with reason, authority and review conditions. **Policy version:** which rule was applied — an identity connecting a lifecycle decision to the exact governing definition.

48.3 Deletion across copies

■ 48.3.1 PLAIN — in simple words

1. Hiding a row from a screen, marking it inactive and deleting it from a table are different actions. None automatically reaches every copy made earlier.
2. A search index, cache, export, replica, analytics table or backup may retain information after the original screen stops showing it.
3. A lifecycle workflow therefore needs a store inventory and a clear action for each store. Unknown destinations remain unknown; they cannot become completed tasks by omission.
4. The strongest honest completion statement names its boundary: which stores, which record identities, which actions and which checks were completed.

■ 48.3.2 PLAIN — a picture in your head

1. Mira removes a note from the counter. Dev still has a photocopy, and a courier has a photograph. The empty counter says nothing about those copies.
2. A checklist helps only if it includes the places the note actually travelled. Ticking every item on an incomplete checklist does not prove that no other copy exists.
3. **Where the comparison breaks:** storage engines can retain old physical representations that ordinary queries cannot read. Logical visibility and recoverability from media are separate layers.

■ 48.3.3 PLAIN — a worked example

1. The synthetic inventory for NOTE-7 contains an active table, search projection, short-lived cache and controlled export. Each task has a scoped key and a status: pending, completed, failed, blocked or unknown.
2. Complete the active-table deletion and search removal. Suppose cache invalidation fails and the export owner has not confirmed its action. The overall status is not complete, despite two successful tasks.
3. Retrying a completed task should be safe under the chosen operation contract. However, “record not found” is meaningful only after the worker has confirmed the correct tenant, store and identifier.
4. The companion inventory model checks these status transitions and refuses an empty inventory as proof of completion. It does not contact a cache, delete a real export or inspect an operator’s computer.

■ 48.3.4 PLAIN — what is really happening inside

1. Database deletion changes logical visibility under the engine's rules. Physical storage reclamation may occur later and can depend on old readers, logging, compaction and maintenance.
2. PostgreSQL describes how deleted or superseded row versions remain until vacuum can reclaim their space. Reclaiming space is not the same claim as sanitising all historical copies or media. [S190]
3. SQLite's `secure_delete` setting concerns overwriting certain deleted content in database pages, with documented limits including some virtual-table arrangements. It is not a promise about every journal, backup, filesystem snapshot or external copy. [S191]

■ 48.3.5 TECHNICAL — the engineer's version

1. Separate logical deletion, application inaccessibility, engine-level reclamation and media sanitisation in both requirements and evidence. A single Boolean deleted column cannot represent every layer honestly.
2. NIST SP 800-88 Revision 2 describes media sanitisation in terms of making target data access infeasible at a stated effort level. It is a media-management reference, not proof that an application row deletion has fulfilled every lifecycle requirement. [S192]
3. A deletion ledger can itself become sensitive. Keep only the identifiers and evidence necessary for its purpose, restrict access and give that ledger a lifecycle too.

■ 48.3.6 WORDS — remember these

1. **Logical deletion:** removal from ordinary active data — a change to the database's visible record state. **Sanitisation:** rendering target media data infeasible to recover under a stated standard — a different layer from an empty application query. **Store inventory:** the declared destinations to address — an explicit set whose completeness must be assessed, not assumed.

48.4 Derived data and lineage

■ 48.4.1 PLAIN — in simple words

1. Derived information is built from other information: a daily total, search document, recommendation feature or cleaned export. Understanding its origins helps us correct or retire it.
2. Some transformations can be updated by removing one contribution. Others need recomputation or a carefully defined approximation. The transformation determines the work.
3. A derived result can remain identifying or sensitive even after obvious names disappear. Its meaning, granularity and joinability matter.
4. Lineage is a map of relationships, not evidence that the mapped action has already happened.

■ 48.4.2 PLAIN — a picture in your head

1. A recipe card lists ingredients used in a cake. It tells Mira which batches might contain one recalled ingredient.
2. The card does not remove the ingredient from a baked cake. The appropriate action might be withdrawing or replacing the whole batch rather than trying to subtract it.
3. **Where the comparison breaks:** some digital aggregates can be adjusted exactly from sufficient retained contributions, while others cannot. A count, median and trained model have different update mechanics.

■ 48.4.3 PLAIN — a worked example

1. A synthetic report stores $\text{sum}=300$ and $\text{count}=3$ for contributions 50, 100 and 150. Removing the 100 contribution gives $\text{sum}=200$ and $\text{count}=2$, so the mean remains 100. Keeping only the original mean would not generally permit this update.
2. A median stored without its contributing values cannot generally be corrected by subtracting one number. The system needs enough retained state or a recomputation path under the applicable lifecycle policy.
3. A search projection that embeds NOTE-7 must be rebuilt or removed under its own index update rules. A later successful source query does not demonstrate that the searchable projection is gone.
4. A lineage record should identify source versions, transformation version and output identity. “Produced by yesterday’s job” is insufficient when the job was rerun with different inputs.

■ 48.4.4 PLAIN — what is really happening inside

1. A transformation may combine many records and create new identities. Lifecycle propagation requires a mapping from the affected source scope to the corresponding outputs or a method of rebuilding all affected outputs.
2. Shared outputs raise policy questions. Retiring one person’s optional note need not delete unrelated aggregate facts, but the retained result needs an independently justified meaning and risk assessment.
3. Data written outside instrumented pipelines may be absent from automated lineage. Manual exports and debug copies must not disappear from the inventory simply because a lineage tool did not observe them.

■ 48.4.5 TECHNICAL — the engineer’s version

1. OpenLineage run and dataset metadata can document transformation relationships. The record must still be populated correctly, and incomplete instrumentation yields incomplete observations. [S193]
2. Distinguish exact incremental maintenance from approximate correction. An exact sum/count representation supports the stated example; a sketch or model requires its own update and error contract.
3. Do not infer model unlearning, anonymisation or comprehensive deletion from a successful DELETE statement. Those claims need methods and evidence specific to the retained representation.

■ 48.4.6 WORDS — remember these

1. **Lineage:** where a result came from — recorded relationships among inputs, transformations, executions and outputs. **Incremental maintenance:** updating a result from a change — applying a justified delta instead of recomputing everything. **Recomputation:** rebuilding from an authorised source set — creating a new derived result under a recorded transformation version.

48.5 Backups and exceptions

■ 48.5.1 PLAIN — in simple words

1. Backups preserve older states for recovery. That purpose can conflict with an expectation that an active-data deletion instantly reaches every historical snapshot.

2. State the policy for retained backups explicitly: access restrictions, expiry, permitted recovery use and what happens to previously deleted information after restoration.
3. Restoring a backup can reintroduce information removed since the backup was made. Recovery therefore needs a lifecycle reconciliation step, not only an integrity check.
4. An exception is not permission for arbitrary reuse. A restricted retained copy should not quietly become a normal source for analytics or customer support.

■ 48.5.2 PLAIN — a picture in your head

1. Mira seals an old notebook in a recovery box. It contains a note later removed from the working notebook.
2. Opening the box after a flood restores useful history, but also restores that old note. The recovery procedure must know which later lifecycle decisions to reapply.
3. **Where the comparison breaks:** encrypted backups, key escrow, incremental chains and storage snapshots complicate selective removal. A practical policy must match the actual storage architecture rather than assume every copy is independently editable.

■ 48.5.3 PLAIN — a worked example

1. A synthetic backup is taken at checkpoint B-20. NOTE-7 is retired afterward under lifecycle action D-21. Restoring B-20 without considering D-21 can bring the note back into active use.
2. In the exercise design, the recovery process restores into isolation, loads the authorised post-backup lifecycle decision set, reapplies applicable actions and checks the result before serving traffic.
3. The lifecycle decision record must contain enough information to prevent reintroduction while not preserving unnecessary original note content. Its access and retention need separate rules.
4. If the recovery team lacks the required decision set, it records that gap and keeps the affected restored data out of ordinary use until resolved. It does not mark the lifecycle work complete because the database starts successfully.

■ 48.5.4 PLAIN — what is really happening inside

1. Backup policies describe retained states and recovery paths; lifecycle policies describe permitted uses and actions. The two have to be designed together.
2. Deleting one encrypted key can make selected ciphertext unavailable only under strict assumptions about key copies, derivation, plaintext copies and scope. Destroying a shared key can also make unrelated necessary records unrecoverable.
3. Record exceptions with their scope, owner, reason and review conditions. The system should distinguish retained-but-restricted from active-and-permitted, rather than hiding both behind one deleted flag.

■ 48.5.5 TECHNICAL — the engineer's version

1. Cryptographic erasure is a key-management and sanitisation claim with prerequisites, not a generic synonym for deleting a row. NIST's sanitisation guidance provides the relevant media/key context; it does not validate our application example. [S192]
2. The companion's restore test uses a fresh SQLite destination and synthetic data. It demonstrates logical state comparison and a subsequent lifecycle decision; it does not test cloud retention locks, key destruction or physical-media recovery.

3. Evidence should identify restored backup, recovery time boundary, applied lifecycle actions and unresolved retained copies. A broad “everything deleted” certificate would exceed those observations.

■ 48.5.6 WORDS — remember these

1. **Reintroduction:** retired information returning through recovery — restoration of an older state without later lifecycle decisions. **Restricted retention:** kept for a bounded exception — retained data whose access and use differ from ordinary active records. **Cryptographic erasure:** making protected ciphertext unusable by sanitising necessary key material — a claim dependent on key scope and remaining copies.

48.6 Evidence of completion and remaining limits

■ 48.6.1 PLAIN — in simple words

1. Completion evidence should describe what was attempted, what succeeded and what remains unresolved. A reassuring message without those details is not a reliable record.
2. Each action needs a target identity, policy basis, execution outcome and suitable check. A failed worker and a completed worker are not interchangeable because both stopped running.
3. Keep the certificate narrower than the evidence. “Removed from these three active stores” can be supported while “no copy exists anywhere” cannot.
4. When new destinations are discovered, the inventory and outstanding work change. A prior completion result may remain historically true for its old scope without being sufficient for the expanded scope.

■ 48.6.2 PLAIN — a picture in your head

1. Mira signs a checklist saying she inspected rooms A, B and C. The statement should not say she inspected the entire building if she never entered room D.
2. If room D is discovered later, she adds a new inspection task rather than erasing the history of what she actually checked.
3. **Where the comparison breaks:** absence is often harder to observe in distributed stores than presence. Search results may be stale, access-limited or scoped incorrectly, so the verification method matters.

■ 48.6.3 PLAIN — a worked example

1. A synthetic completion record for NOTE-7 states tenant T-A, policy version RP-3, inventory version INV-2 and three completed active-store tasks. It separately lists one backup retained under a restricted exception until its configured expiry.
2. That record supports “active-store actions completed under INV-2; restricted backup remains.” It does not support “all copies permanently erased.”
3. A task marked unknown prevents the inventory model from reporting complete. A task marked blocked requires its exception to remain visible. Empty task sets are not treated as successful comprehensive action.
4. The companion tests those status rules and checks their deterministic outputs. It cannot verify an unobserved third-party store; its evidence is deliberately limited to the model and local exercise.

■ 48.6.4 PLAIN — what is really happening inside

1. Lifecycle work is a state machine with evidence attached to transitions. Repeated delivery and retries should preserve action identity without manufacturing additional authority.
2. An append-only audit trail can retain action outcomes while the original content disappears. The trail's identifiers, actor information and reasons still need access and retention controls.
3. Operational review compares expected destinations with observed executions, reconciles gaps and assigns follow-up responsibilities. The useful result is a bounded, inspectable claim rather than an absolute adjective.

■ 48.6.5 TECHNICAL — the engineer's version

1. A completion predicate can require a nonempty declared task set and successful verification for every required task, while reporting exceptions separately. The task set's completeness remains an external premise.
2. Store the evidence method, not merely a Boolean: query scope, target version, execution result and timestamp, with sensitive values minimised. Confirm that the check itself ran under authority capable of seeing the intended target.
3. Publication of a deletion receipt does not create proof of independently controlled copies. Separate controllable scope, verified scope, restricted retention and unknown scope in the final statement.

■ 48.6.6 WORDS — remember these

1. **Completion predicate:** the exact rule for saying an action is done — a condition over declared tasks, evidence and exceptions. **Verification scope:** what a check could actually observe — the stores, identities, versions and authority included in the observation. **Residual copy:** information remaining after selected actions — a retained, failed, blocked or unknown destination requiring explicit treatment.

48.97 Practice and worked answers

1. **Question:** Does hiding NOTE-7 from a screen remove its search projection? **Answer:** Not unless the projection's own update or deletion path is completed and checked.
2. **Question:** Is the example's thirty-day period a legal recommendation? **Answer:** No. It is invented elapsed-time arithmetic for teaching policy implementation.
3. **Question:** Why specify the retention trigger? **Answer:** Creation, closure and last activity can produce different expiry dates and different permitted actions.
4. **Question:** Can an unchanged aggregate mean prove that no contribution was removed? **Answer:** No. In the sum/count example, removing 100 from 50,100,150 leaves the mean at 100 while changing the population.
5. **Question:** Does an empty deletion-task list establish complete erasure? **Answer:** No. The inventory may be missing; the model rejects that vacuous conclusion.
6. **Question:** What risk follows restoring B-20 after action D-21? **Answer:** An older copy can reintroduce retired information unless the recovery workflow reapplies relevant later lifecycle decisions.
7. **Question:** Does vacuum establish media sanitisation? **Answer:** No. Space reclamation and physical recoverability are different claims.
8. **Question:** What should a completion record say when a restricted backup remains? **Answer:** Name the completed active scope and disclose the retained backup exception and its governing conditions.

48.98 Common wrong ideas

1. **Wrong:** Retention is one period for the whole database. **Right:** Different record classes can have different purposes and rules.
2. **Wrong:** Missing timing means the record is old enough to delete. **Right:** Unknown timing requires a declared unresolved-data policy.
3. **Wrong:** Deleting a row reaches every copy. **Right:** Derived and historical stores require explicit treatment.
4. **Wrong:** A removed name makes every remaining record anonymous. **Right:** Joinability, granularity and context can preserve identifying meaning.
5. **Wrong:** A lineage map executes deletion. **Right:** It records relationships; execution and verification are separate.
6. **Wrong:** A backup restore is complete when the engine starts. **Right:** Later lifecycle and access decisions may need reconciliation.
7. **Wrong:** A deletion ledger can safely keep everything forever. **Right:** The ledger also has purpose, access and retention requirements.
8. **Wrong:** An absolute erasure claim follows from a few successful queries. **Right:** State the stores and observation boundary actually verified.

48.99 Chapter summary in 20 lines

1. Give each record class a purpose and an owner.
2. Separate optional annotations from independently required facts.
3. Minimise unnecessary fields and downstream copies.
4. Define the event that starts a retention clock.
5. Distinguish elapsed durations from calendar rules.
6. Govern exceptions with reasons and review conditions.
7. Recheck changing eligibility at the action boundary.
8. Hiding, logical deletion and sanitisation are different actions.
9. Inventory caches, indexes, exports, analytics and backups.
10. Unknown destinations do not count as completed work.
11. Record the transformation history of derived outputs.
12. Preserve sufficient state for justified incremental corrections.
13. Recompute when a simple subtraction is not valid.
14. Do not infer anonymity or model unlearning from a removed field.
15. Make backup retention and permitted use explicit.
16. Prevent restored historical data from bypassing later lifecycle decisions.
17. Treat key destruction as a scoped, prerequisite-dependent claim.
18. Give lifecycle evidence its own access and retention policy.
19. Report completed, restricted and unresolved scope separately.
20. A completion statement must not exceed its actual evidence.

Schema Changes Without Surprises

49.0 What this chapter gives you

1. A schema change alters the agreement between stored records and the programs using them. The challenge is not merely to make a new table definition valid; it is to keep old data and overlapping application versions meaningful while the change takes place.
2. You will plan compatibility windows, separate expansion from removal, design bounded back-fills and define evidence for cutover. You will also distinguish undoing a deployment from reconstructing information already lost.
3. The worked change adds an explicit currency field to a separate copy of the shop's INR-only teaching data. It does not change the agreed amounts or pretend that a real multi-currency checkout has been implemented.
4. All executable changes in the companion use newly created, disposable databases. No instruction in this chapter authorises running a migration against an existing user or production database.

49.1 Compatibility windows

■ 49.1.1 PLAIN — in simple words

1. During an update, not every program changes at the same instant. An old web process, new worker, reporting job and delayed mobile request may all use the database during the transition.
2. A safe plan asks which combinations must work. A new program with the old schema and an old program with the new schema can fail in different ways.
3. Compatibility includes meaning, not only syntax. A field still called amount can become dangerous if one version reads rupees while another writes paise.
4. List readers and writers before changing the contract. A forgotten export script is still a consumer even if it has no place on the architecture diagram.

■ 49.1.2 PLAIN — a picture in your head

1. Mira changes an order form while some printed pads with the old design remain at the tills. The office has to interpret both forms for a while.
2. Removing a box from the new form does not remove it from old pads. Changing the meaning of an existing box without a label can be worse than adding a new one.
3. **Where the comparison breaks:** software can keep running old code for hours, replay old messages later and cache schema assumptions. The overlap is not limited to what is visibly on the counter today.

■ 49.1.3 PLAIN — a worked example

1. The original agreed prices are integer paise and the teaching contract says INR. A proposed schema adds `currency_code` so future readers no longer need that implicit assumption.
2. Version V1 knows only `amount_paise`. Version V2 knows `amount_paise` and `currency_code`. During the overlap, a V1 insert may omit the new field; a V2 reader must not falsely claim that every missing value in every dataset means INR.
3. For this specific copied fixture, documented provenance establishes that all old records are INR. A compatibility adapter may use that provenance while the backfill runs. Imported records without the same provenance remain unresolved.
4. The compatibility matrix therefore includes V1 read/write before and after expansion, V2 read/write during partial backfill and V2-only work after cutover. It also records when V1 writers are no longer allowed.

■ 49.1.4 PLAIN — what is really happening inside

1. Readers depend on field presence, type and meaning. Writers depend on accepted inputs, defaults and constraints. Schema and application releases change those dependencies at different times.
2. A message or API contract can outlive the program that originally created it. Replaying old payloads must follow a declared version adapter or rejection policy, not accidental interpretation by current code.
3. Database locks and resource use are a second compatibility dimension. A logically compatible DDL statement can still block ordinary work for longer than the service permits.

■ 49.1.5 TECHNICAL — the engineer's version

1. Model compatibility as a relation among application version, schema version, payload version and data state. “Backward compatible” is incomplete unless the direction and supported combinations are stated.
2. PostgreSQL’s ALTER TABLE operations have operation-specific lock and rewrite behaviours. A change plan must consult the exact operation and version rather than assume every ADD COLUMN is operationally identical. [S194]
3. The capstone migration experiment tests a small local subset. It is not an online production migration or evidence of acceptable lock duration under concurrent load.

■ 49.1.6 WORDS — remember these

1. **Compatibility window:** the overlap period for contracts — the time during which specified old and new consumers must coexist. **Consumer:** anything relying on the data contract — a reader, writer, job, export, API adapter or replay process. **Semantic compatibility:** preserving meaning across versions — agreement about units and interpretation, not only accepted syntax.

49.2 Expand and contract

■ 49.2.1 PLAIN — in simple words

1. Expansion adds the new representation while preserving the old path. Contraction removes the old path only after the remaining dependencies have been checked.
2. Separating these steps gives programs time to move. It also creates temporary complexity, which needs an owner and a removal plan.

3. Two writable copies of the same fact can disagree. Decide which one is authoritative during each phase and how disagreement is detected.
4. Do not make the destructive final step the first step merely because it produces a cleaner schema diagram.

■ 49.2.2 PLAIN — a picture in your head

1. To replace a bridge while traffic continues, first build the new crossing, direct traffic onto it and only then remove the old one.
2. Building the new bridge is not proof that every road sign now points to it. The switch requires observation and coordination.
3. **Where the comparison breaks:** two schema representations can both accept writes. Unlike passive bridges, they can create conflicting versions of the same fact unless the transition protocol controls them.

■ 49.2.3 PLAIN — a worked example

1. Phase E1 adds nullable `currency_code` to the copied exercise table. Existing reads remain possible, and no historical value is guessed during the DDL step itself.
2. Phase E2 deploys a reviewed writer that supplies INR for this INR-only workflow and a reader that handles the recorded transition state. Old-writer behaviour is explicitly tracked.
3. Phase E3 backfills eligible historical rows using their known provenance. Phase E4 validates completeness and allowed values, confirms that old writers are gone and strengthens the constraint.
4. Only a later contraction removes compatibility fallback logic. That removal is a distinct decision with evidence, not an automatic side effect of the backfill completing once.

■ 49.2.4 PLAIN — what is really happening inside

1. An additive schema permits both representations for a limited period. Application releases then change which representation is read or written.
2. Dual writes should normally be kept within one appropriate atomic boundary when they describe one fact. Separate best-effort writes can leave one representation stale after a failure.
3. Monitoring must distinguish expected transitional gaps from defects. An alert cannot interpret missing `currency_code` correctly without knowing the current phase and its deadline.

■ 49.2.5 TECHNICAL — the engineer's version

1. The expand/backfill/validate/contract sequence here is an original deployment plan. Its safety depends on the stated writer controls and provenance assumptions; it is not a universal recipe guaranteeing zero downtime.
2. PostgreSQL 17 supports adding certain CHECK and foreign-key constraints as NOT VALID and validating them later. The feature does not mean “ignore all checks”: new changes remain subject to the documented enforcement rules. [S194]
3. SQLite's supported ALTER TABLE operations differ from PostgreSQL's, and more involved changes may require the documented table-rebuild procedure. Do not copy PostgreSQL syntax into the SQLite lab and infer equivalence. [S196]

■ 49.2.6 WORDS — remember these

1. **Expand:** add a compatible new path — introducing representation or capability before removing the old one. **Contract:** retire the old path — removing obsolete fields, adapters or access after dependencies are checked. **Authoritative representation:** the selected source of meaning — the version whose value governs while multiple copies coexist.

49.3 Backfills

■ 49.3.1 PLAIN — in simple words

1. A backfill transforms existing records into a new representation. It should have explicit eligibility, ordering, batch limits and rules for partial progress.
2. A stopped backfill should be resumable without silently duplicating work or overwriting a more recent legitimate change.
3. A default is not evidence. Filling every missing currency with INR is justified only where the record's origin or governing contract establishes INR.
4. Progress must represent completed work. Advancing a cursor before the data change is committed can skip records after a failure.

■ 49.3.2 PLAIN — a picture in your head

1. Dev adds a currency label to old order cards. He knows one labelled box contains only INR orders, but a mixed box from an external source has no such guarantee.
2. He completes and records one small batch before moving his bookmark. If interrupted, the bookmark tells him which completed batch to resume after.
3. **Where the comparison breaks:** records can change while a backfill is running. A database bookmark and a person's static pile of cards do not have identical concurrency behaviour.

■ 49.3.3 PLAIN — a worked example

1. Copy five synthetic rows with ordered identifiers 1 through 5 and a provenance flag. Rows 1–4 have the documented INR-only origin. Row 5 has unknown origin.
2. Process at most two eligible rows per transaction. The first committed batch fills rows 1 and 2. The next fills 3 and 4. Row 5 remains unresolved rather than receiving an invented label.
3. Retry the second batch. Its conditional predicate requires `currency_code IS NULL` and the known origin; already-filled rows are unchanged. Idempotence here means repeating the permitted transformation does not invent additional changes.
4. Inject an exception before the batch commit. Both data changes and progress metadata owned by that transaction must roll back. A progress record outside the transaction would need its own recovery protocol.

■ 49.3.4 PLAIN — what is really happening inside

1. A worker selects a bounded set under a stable ordering and applies a conditional update. Keyset progress can avoid some problems of changing offset-based lists, but its correct use still depends on the key and eligibility rules.
2. A competing writer can make a row ineligible or change its version. Recheck relevant conditions at update time; do not treat an earlier selection as permanent permission to overwrite.

3. Resource budgets matter. A backfill uses I/O, logs, locks and cache capacity that ordinary traffic also needs. Throttle or pause based on defined service signals rather than forcing maximum speed throughout.

■ 49.3.5 TECHNICAL — the engineer's version

1. Specify a transformation precondition and postcondition. For the exercise: source provenance is `INR_ONLY`, destination is `NULL`, and the only permitted new value is `INR`. Unknown provenance is not a successful transformation.
2. Keep progress and transformed data atomic where possible, or design replay-safe reconciliation where they cross a boundary. Preserve counts for selected, changed, already-complete, conflicting and unresolved rows.
3. The companion backfill is sequential and in memory. Its rollback and replay tests do not establish correctness under every production writer schedule or acceptable WAL growth.

■ 49.3.6 WORDS — remember these

1. **Backfill:** transforming already-stored records — population of a new representation under explicit source assumptions. **Keyset progress:** resuming from a stable ordered key — a traversal technique whose eligibility and concurrent-change boundaries still need design. **Transformation precondition:** what must be true before changing a row — the provenance, version or state justifying the operation.

49.4 Validation and cutover

■ 49.4.1 PLAIN — in simple words

1. Validation asks whether the new representation satisfies the intended contract. A backfill process exiting successfully does not answer every validation question.
2. Compare counts, keys, values, exclusions and meaning. Equal totals can hide two offsetting mistakes.
3. Cutover changes which path the service trusts or serves. It should occur after the relevant evidence is available and while a clearly defined recovery option still exists.
4. The validation itself needs a time boundary. A clean check followed by an uncontrolled old writer can make the new state incomplete again.

■ 49.4.2 PLAIN — a picture in your head

1. Mira compares old and new order forms before telling every till to use the new version. She checks actual identifiers and line values, not only the number of pages.
2. She also removes the old blank pads. Otherwise the next order can recreate the ambiguity she just resolved.
3. **Where the comparison breaks:** software versions can remain in worker pools, clients or retry queues after a deployment dashboard says the main service is updated.

■ 49.4.3 PLAIN — a worked example

1. Validate the copied canonical fixture: four lines, six units, two order identities and 28,650 paise. Compare each line's scoped key, product, quantity, agreed price and currency.
2. Confirm O-1042 remains 17,100 paise and O-1043 remains 11,550. A currency addition is not permission to replace their agreed prices with current catalogue prices.

3. Count unresolved origins separately. If the five-row backfill example still contains row 5 with unknown currency, it cannot support the statement “all rows satisfy the new required currency contract.”
4. Before cutover, stop or adapt every old writer, rerun the applicable validation and establish the new constraint. The order matters because uncontrolled writes between checking and enforcing can reopen the gap.

■ 49.4.4 PLAIN — what is really happening inside

1. Validation queries observe a snapshot or sequence of snapshots according to the database and transaction settings. Their scope should be recorded so discrepancies can be interpreted.
2. A database constraint can maintain selected properties after successful validation. It does not prove that the chosen currency value reflects the real transaction unless the input provenance is sound.
3. Index creation and schema locking may require special operational treatment. Even a carefully chosen concurrent index build has failure states that a deployment script must inspect.

■ 49.4.5 TECHNICAL — the engineer’s version

1. PostgreSQL documents CREATE INDEX CONCURRENTLY restrictions and the possibility of an invalid index after failure. “The command was started” is not evidence of a usable index or successful schema cutover. [S195]
2. Bind validation evidence to source revision, schema state, record scope and runtime. A row count from one database and a constraint report from another do not compose into one verified result.
3. The companion validates a deliberately small copied fixture. Production cutover needs representative traffic, real role/constraint tests and a deployment-specific concurrency and recovery plan.

■ 49.4.6 WORDS — remember these

1. **Cutover:** switching the authoritative path — a controlled transition from old to new representation or service behaviour. **Reconciliation:** comparing meaningfully corresponding records — checking identities, values and declared exclusions rather than one headline total. **Validation boundary:** the exact state a check describes — its database, snapshot or time, schema and input scope.

49.5 Rollback versus roll-forward

■ 49.5.1 PLAIN — in simple words

1. Rolling back an application release can restore old code. It does not automatically restore deleted information or reverse every data change made by the new code.
2. Some changes preserve enough information to undo them. Others lose distinctions: merging fields, rounding values or deleting columns can make exact reconstruction impossible without another source.
3. A roll-forward repair applies a new corrective change instead of pretending the earlier state can simply be restored. The choice depends on evidence, compatibility and the remaining recovery paths.
4. A backup is useful only if it can restore the needed state under the allowed time and data-loss constraints. Restoring an old backup can also discard legitimate work performed afterward.

■ 49.5.2 PLAIN — a picture in your head

1. Putting the old form back on the counter does not recover notes shredded during the update. The paper design and the recorded information are different things.
2. Sometimes the correct response is to retain the new form and repair its unclear label, rather than force every completed order back into a structure that no longer fits.
3. **Where the comparison breaks:** databases can support transactional rollback for some DDL and data changes, but behaviour depends on the engine, operation and commit boundary. Do not generalise from one product's example.

■ 49.5.3 PLAIN — a worked example

1. Adding nullable `currency_code` and filling justified INR values preserves the old `amount_paise` field. Reverting V2 to V1 during the compatible phase may be possible under the declared reader/writer rules.
2. By contrast, replacing all exact paise with rounded rupee amounts loses information. Both 7,550 and 7,549 paise could round to 75 under one whole-rupee rounding policy. The old exact value cannot be inferred from that rounded value alone.
3. If new legitimate orders were created after a backup, restoring that backup to undo the migration may remove those orders too. The recovery plan must address the intervening work, not call it irrelevant because rollback is urgent.
4. A repair record should state which values were reconstructed from retained evidence, which were corrected by new authorised input and which remain unresolved.

■ 49.5.4 PLAIN — what is really happening inside

1. A rollback boundary is a point before or after durable commitment. Within a transaction, supported changes can be discarded. After commitment, recovery requires another operation with its own risks.
2. Reversibility is a property of the transformation and retained evidence. An invertible encoding can be reversed; an aggregation or destructive overwrite may not be.
3. Deployment teams should rehearse the intended recovery procedure on representative disposable copies. Discovering that the old application cannot read newly accepted values during an incident is too late.

■ 49.5.5 TECHNICAL — the engineer's version

1. Record a point of no return for each destructive step and the evidence required before crossing it. Keep application rollback, schema rollback and data restoration as separate entries.
2. SQLite's documented generalised ALTER TABLE procedure includes explicit copying, rebuilding and checks. A home-made variant that changes ordering or skips dependencies is not validated merely because the resulting CREATE TABLE looks right. [S196]
3. The companion demonstrates rollback inside its owned transaction and explicitly labels transformations that lose information. It does not claim an automated production rollback facility.

■ 49.5.6 WORDS — remember these

1. **Rollback:** returning within a defined reversible boundary — undoing a transaction or deployment, with scope that must be stated. **Roll-forward:** correcting through a subsequent change — a new reviewed transformation rather than automatic restoration of the past. **Irreversible trans-**

formation: one that loses needed distinctions — a mapping whose original inputs cannot be recovered from the retained output alone.

49.6 Staged evidence

■ 49.6.1 PLAIN — in simple words

1. A good migration record separates proposed work, reviewed work, executed work and verified outcomes. Those stages are related but are not the same event.
2. Preparing a script proves that a script exists. Running it proves that execution occurred. A successful exit does not prove every intended business property.
3. Preserve failure evidence. Replacing a failed report with a later green result without recording the change hides what was learned and which state was actually tested.
4. The final decision belongs to the authorised owner. A technical check supplies evidence; it does not grant itself permission to change a live system.

■ 49.6.2 PLAIN — a picture in your head

1. A bridge drawing, an approved construction plan, a completed bridge and an inspected bridge are four different things.
2. A stamp saying “plan approved” cannot be moved onto a photograph and treated as a structural inspection of the finished work.
3. **Where the comparison breaks:** software artefacts can change with one byte, and the difference may be invisible in a screenshot. Evidence needs a reliable identity for the actual candidate tested.

■ 49.6.3 PLAIN — a worked example

1. The proposed currency migration record lists the schema precondition, eligible provenance, maximum batch size, transaction owner, stop conditions, validation queries and recovery plan.
2. A disposable rehearsal records the exact script digest and input fixture identity, then reports changed rows, unresolved rows and rollback-control results. Its scope is the rehearsal environment.
3. A later live execution would require separate authority and live preconditions. The rehearsal’s four correct rows do not establish that a larger external dataset has only four eligible rows or no exceptions.
4. After execution, verify the intended schema, row-level reconciliation and the behaviours of the actual readers and writers. Retain the before/after record and unresolved findings.

■ 49.6.4 PLAIN — what is really happening inside

1. Evidence connects an artefact identity to an environment, an action and an observation. Losing any of those links weakens the conclusion.
2. A migration runner can automate ordering and record applied identifiers. It cannot infer a business requirement omitted from the migration design.
3. A staged process limits the size of an uncertain step. It also makes a failure easier to diagnose because the observed change is bounded and the previous state is recorded.

■ 49.6.5 TECHNICAL — the engineer's version

1. Use a review template containing preimages, change identity, schema version, engine/driver versions, permissions, transaction boundaries, reconciliation outputs and recovery conditions. The template is an original engineering aid, not certification.
2. Distinguish idempotent transformation logic from a migration registry that simply refuses a repeated migration ID. They address different retry situations.
3. The capstone's final checks establish only the listed disposable-example outcomes. They neither access the user's database nor grant a release decision for an unrelated application.

■ 49.6.6 WORDS — remember these

1. **Precondition:** what must be true before a change — the checked state and authority on which safe execution depends. **Artefact identity:** which exact candidate was used — a stable reference, often a digest, linking execution to reviewed bytes. **Rehearsal:** a bounded execution before live work — an experiment whose representativeness and limitations must be stated.

49.97 Practice and worked answers

1. **Question:** Why test an old writer after adding a required concept? **Answer:** It may omit the new field or supply values with old semantics during the compatibility window.
2. **Question:** Can every NULL currency become INR? **Answer:** Only where authoritative provenance establishes that meaning; unknown origins remain unresolved.
3. **Question:** What must be atomic in the backfill example? **Answer:** The owned batch changes and its committed progress record, or an alternative replay-safe protocol must handle their separation.
4. **Question:** Does 28,650 paise after migration prove every line is unchanged? **Answer:** No. Compare keys and individual values too; offsetting errors can preserve a total.
5. **Question:** Why remove old writers before final validation/enforcement? **Answer:** Otherwise they can recreate incompatible data after an earlier clean check.
6. **Question:** Does restoring old code recover rounded-away paise? **Answer:** No. Lost information needs another trustworthy source or remains unknown.
7. **Question:** What does a successful rehearsal establish? **Answer:** The recorded candidate behaved as observed on the stated disposable input and environment, not every production condition.
8. **Question:** Does a green technical report authorise a live migration? **Answer:** No. Authority and technical evidence are separate requirements.

49.98 Common wrong ideas

1. **Wrong:** A schema is used only by the newest application. **Right:** Old workers, exports and replayed payloads can remain active.
2. **Wrong:** Adding a default supplies missing historical evidence. **Right:** Defaults are rules, not recovered facts.
3. **Wrong:** Dual writing guarantees agreement. **Right:** It needs an atomic or reconciled transition protocol.
4. **Wrong:** A backfill should always run as fast as possible. **Right:** Ordinary service capacity and failure budgets still matter.
5. **Wrong:** Equal totals prove a correct migration. **Right:** Keys, values, exclusions and semantics also need comparison.

6. **Wrong:** Reverting code undoes every data change. **Right:** Committed and lossy transformations may need separate recovery.
7. **Wrong:** A migration ID proves safe repeated execution. **Right:** Registry identity and transformation idempotence are different properties.
8. **Wrong:** A prepared script is an executed and verified result. **Right:** Preparation, execution, verification and authority are separate stages.

49.99 Chapter summary in 20 lines

1. A schema is a contract with multiple consumers.
2. List old and new readers and writers.
3. Preserve units and meaning across the compatibility window.
4. Inspect operation-specific lock and rewrite behaviour.
5. Expand before removing the old path.
6. State which representation is authoritative during transition.
7. Give temporary compatibility code an owner and retirement condition.
8. Backfill only where provenance justifies the new value.
9. Use bounded batches and explicit ordering.
10. Keep committed progress aligned with committed changes.
11. Recheck update conditions when competing writers can act.
12. Preserve unresolved rows instead of guessing.
13. Validate identities and values, not only totals.
14. Prevent old writers from reopening a validated gap.
15. Treat cutover as a distinct controlled event.
16. Separate application rollback from data restoration.
17. Identify transformations that lose information.
18. Rehearse the actual recovery procedure on disposable data.
19. Bind evidence to exact artefacts and environments.
20. Technical success does not manufacture live-change authority.

Observability, Capacity and Cost

50.0 What this chapter gives you

1. A running system produces signals about its behaviour. Useful observability connects those signals to questions: Are users completing their work? Where is time being spent? Which resource is approaching a limit? What changed before the failure?
2. You will distinguish logs, metrics and traces; calculate a capacity forecast with explicit assumptions; and separate workload measurements from guesses based on server labels.
3. Every rate, storage size and cost in the worked examples is invented for arithmetic. None is a measurement of KedByte's live products, a current hosting price or a capacity promise for a particular machine.
4. This chapter connects Chapter 20's measurement discipline with Chapter 46's metric definitions. A precise dashboard can still answer the wrong question when it misses failed attempts or mixes incompatible populations.

50.1 Signals tied to user outcomes

■ 50.1.1 PLAIN — in simple words

1. Start with the work the service is supposed to complete. A healthy-looking processor graph does not prove that an order was recorded correctly or that a customer can retrieve it.
2. Define success, failure and unknown outcomes. A request that timed out may have committed or may not have reached the database; the timeout alone does not settle the stored result.
3. Measure the experience at a useful boundary. Server processing time excludes some network and client delays, while a browser measurement may include them.
4. An objective is a target, not an observation. The system must collect evidence to determine whether the target was met.

■ 50.1.2 PLAIN — a picture in your head

1. Mira's delivery van has a working engine and a full fuel tank. Those facts are useful, but customers care whether the correct parcels arrive on time.
2. Engine temperature helps explain a delay; delivery completion tells her whether the service actually fulfilled its purpose.
3. **Where the comparison breaks:** digital work may complete invisibly after a timeout. Outcomes need stable identities and reconciliation, not just a single caller's immediate impression.

■ 50.1.3 PLAIN — a worked example

1. Define a synthetic request objective: at least 99.5% of 20,000 eligible read requests return an authorised, valid response within a stated threshold during one observation window.
2. Under that definition, the allowance for nonconforming requests is $0.5\% \times 20,000 = 100$. If 60 are nonconforming, the measured success proportion is $19,940 / 20,000 = 99.7\%$, with 40 requests of the illustrative allowance remaining.
3. These calculations depend on the eligibility and observation rules. Excluding failed requests because they lack a successful response record would change the denominator and could make the result misleading.
4. Record unknown outcomes separately. A claim of 99.7% cannot silently treat unobserved attempts as successful merely because no error reached the dashboard.

■ 50.1.4 PLAIN — what is really happening inside

1. A service-level indicator operationalises a user-facing property through measured events or values. A service-level objective selects a target for that indicator over a defined window.
2. Collection occurs at a particular boundary. Load balancers, application handlers and database drivers may each see different populations and durations.
3. Reconcile those populations when interpreting a failure. An attempt can disappear before application logging, and a database commit can occur before its acknowledgement is lost.

■ 50.1.5 TECHNICAL — the engineer's version

1. Google's SRE text distinguishes indicators, objectives and agreements and emphasises choosing indicators that represent useful service behaviour. The book's 20,000-request example is our own arithmetic, not Google's reliability target. [S197]
2. Document event eligibility, correctness checks, measurement boundary, threshold, window, missingness and aggregation. Include the specification version in the metric definition.
3. Error-budget arithmetic does not permit violating a data-integrity invariant. A latency objective and a rule against cross-tenant disclosure are different kinds of requirement, not interchangeable percentages.

■ 50.1.6 WORDS — remember these

1. **Service-level indicator:** a measured service property — a defined observation of availability, latency, correctness or another relevant outcome. **Service-level objective:** the target for an indicator — a specified level over a stated population and window. **Error budget:** the allowed nonconformance under an objective — an arithmetic planning quantity whose definition does not override other requirements.

50.2 Logs, metrics and traces

■ 50.2.1 PLAIN — in simple words

1. A log records an event with details. A metric summarises quantities over time or categories. A trace connects steps belonging to a particular operation across components.
2. Each helps answer different questions. A count shows how often something happened; a trace can show where one request waited; a log can explain a particular rejection.

3. Collect enough context to connect evidence without copying unnecessary private data. Request identifiers are often more useful than complete request bodies.
4. Missing signals need explanation. An absent log can mean nothing happened, collection failed, the event was sampled out or the retention window expired.

■ 50.2.2 PLAIN — a picture in your head

1. Mira has a daily counter of orders, a diary of unusual incidents and a route sheet following one parcel through packing and delivery.
2. The counter cannot explain one damaged parcel. The route sheet cannot by itself show the whole month's failure rate. The tools complement each other.
3. **Where the comparison breaks:** digital signals can be sampled, buffered, delayed and dropped. Their apparent timing and completeness depend on the collection system.

■ 50.2.3 PLAIN — a worked example

1. A synthetic request takes 240 milliseconds end to end. Its trace records 20 milliseconds before the application, 40 milliseconds in a queue, 130 milliseconds in database work and 50 milliseconds in other application/response work. The stated non-overlapping intervals sum to 240.
2. Real spans can overlap, so blindly adding every displayed span duration can double-count time. The example's disjoint intervals are an explicit teaching assumption.
3. The associated metric counts one eligible request and its outcome. A rejection log stores the scoped request identifier and reason category, not the secret credential or full customer note.
4. If only one in ten traces is retained, the retained traces cannot automatically supply an exact count of all requests. Use a suitable unsampled counter or a documented estimator instead.

■ 50.2.4 PLAIN — what is really happening inside

1. Instrumentation creates events and measurements in running code. Collectors transport and process them; storage and query systems retain and expose them.
2. Correlation identifiers let investigators connect related signals. Their propagation needs care at trust boundaries, because external callers can supply misleading identifiers unless the system distinguishes trusted and untrusted context.
3. Labels determine how metrics are divided. Adding every order ID as a metric label can create a huge number of distinct time series; detailed per-order evidence may belong in a different store with appropriate access and retention.

■ 50.2.5 TECHNICAL — the engineer's version

1. OpenTelemetry describes signals including traces, metrics and logs and their associated data models. Adopting its vocabulary or library does not automatically establish complete instrumentation or causal diagnosis. [S198]
2. Record sampling, buffering, export failure and clock assumptions. A timestamp difference across unsynchronised hosts is not necessarily a precise duration; use suitable local duration measurements and trace semantics.
3. Google's monitoring guidance identifies latency, traffic, errors and saturation as useful general signals. Our observation design must still connect those signals to the shop's actual operations and declared failure conditions. [S103]

■ 50.2.6 WORDS — remember these

1. **Log:** a recorded event — timestamped or otherwise ordered details describing a particular occurrence. **Metric:** a measured quantity — an aggregated or sampled numeric signal with units and dimensions. **Trace:** connected steps of an operation — spans and context describing work across a request path. **Cardinality:** how many distinct labelled combinations exist — a property that can drive metric storage and query cost.

50.3 Storage and growth

■ 50.3.1 PLAIN — in simple words

1. Storage demand depends on how much arrives, how large each retained representation is and how long it remains. Indexes, logs, copies and temporary work add to the primary records.
2. A count of students, shops or users does not determine storage by itself. One user can create thousands of records, large attachments or almost no activity.
3. Distinguish logical payload size from physical occupied storage. Compression, row overhead, fragmentation and retention can change the latter substantially.
4. A forecast should expose its assumptions and define when measurements will replace them. A neat multiplication is not a measured capacity result.

■ 50.3.2 PLAIN — a picture in your head

1. Mira estimates shelf space from boxes arriving per day, the space per box and the days each box remains. Counting suppliers alone is not enough.
2. She also needs room for labels, aisles, returns and temporary packing. Those are not new sales, but they still occupy the building.
3. **Where the comparison breaks:** database compression and indexes do not scale exactly like cardboard boxes. Updates and maintenance can create temporary or retained representations whose size depends on engine behaviour.

■ 50.3.3 PLAIN — a worked example

1. Invent a workload of 12,000 records per day, each with 600 bytes of logical payload, retained for 365 days. The payload-only estimate is $12,000 \times 600 \times 365 = 2,628,000,000$ bytes, or 2.628 decimal GB.
2. Under a deliberately hypothetical 1.8× combined table/index overhead multiplier, the active representation estimate becomes 4.7304 GB. Three active copies would total 14.1912 GB under the same simplifying assumptions.
3. This does not include backups, logs, exports, attachments, temporary sort space or free capacity for maintenance. Do not add them through an unexplained universal percentage; estimate their mechanisms separately.
4. If a host has 100 GB usable and 70 GB is the chosen intervention threshold, starting at 20 GB with 2 GB/day net growth reaches that threshold in $(70-20)/2 = 25$ days. The 70 GB threshold is an example policy, not a universal safe limit.

■ 50.3.4 PLAIN — what is really happening inside

1. Physical storage grows through inserts, updates, index maintenance and retained history. Deletion may make space reusable without immediately shrinking the file.

2. Copies have different lifetimes. A follower, nightly backup and exported report cannot be modelled accurately as three identical continuously updated files.
3. Measure representative data and maintenance cycles. A fresh empty database can understate steady-state overhead; an unusual bulk load can overstate typical daily growth.

■ 50.3.5 TECHNICAL — the engineer's version

1. Keep byte units explicit: decimal GB is 10^9 bytes and GiB is 2^{30} bytes. The forecast above uses decimal units throughout.
2. PostgreSQL's statistics views expose specific counters and activity information under documented visibility and timing rules. They are observations with semantics, not a direct prediction of next year's data size. [S200]
3. A useful forecast records measured baseline, growth distribution, retention action, replication factor, backup policy and uncertainty. Update it when the workload mix changes rather than treating the first spreadsheet as permanent truth.

■ 50.3.6 WORDS — remember these

1. **Logical payload:** the data's application representation — the meaningful values before engine and operational overhead. **Net growth:** increase after relevant additions and removals — a rate whose measurement window and storage boundary must be stated. **Intervention threshold:** the point to act before exhaustion — a chosen policy leaving time and resources for a controlled response.

50.4 Saturation and headroom

■ 50.4.1 PLAIN — in simple words

1. A system is saturated when demand pushes a relevant resource toward the point where more work mostly creates waiting, rejection or failure.
2. The bottleneck might be CPU, storage latency, memory, a connection pool, a lock or a slow external dependency. Buying more of a different resource may not help.
3. Headroom is spare usable capacity under the chosen workload and failure assumptions. It is not simply the percentage of CPU currently idle.
4. Repeated retries can increase demand while the system is already struggling. A recovery policy needs limits, not an instruction to retry everything immediately forever.

■ 50.4.2 PLAIN — a picture in your head

1. A counter can serve customers at a finite rate. Once arrivals exceed that rate, the queue grows even though the assistant is working continuously.
2. Asking waiting customers to rejoin the queue every few seconds makes the crowd larger without producing more completed orders.
3. **Where the comparison breaks:** software can parallelise some work and has multiple coupled queues. A single service-rate number is a model, not a universal description of the whole architecture.

■ 50.4.3 PLAIN — a worked example

1. A synthetic worker completes 100 jobs per second, while 120 arrive per second for 30 seconds. Ignoring variability and starting empty, the queue grows by $(120-100) \times 30 = 600$ jobs.

2. After arrivals stop, draining those 600 jobs at 100 per second takes 6 seconds under the same assumptions. If arrivals instead continue at 90 per second, net drain is only 10 per second and the backlog takes 60 seconds.
3. For a stable observation window, Little's-law arithmetic relates average work in the system L , throughput λ and average time W as $L=\lambda W$, under the relevant steady-state conditions. At 80 jobs/second and 0.25 seconds average residence, $L=20$ jobs. This is not a p99 latency estimate.
4. If retries add 50 attempts per second to the original 120 while capacity stays 100, the backlog grows faster. Backoff, retry budgets and admission control address this feedback, but each needs an explicit business failure policy.

■ 50.4.4 PLAIN — what is really happening inside

1. Requests wait when a needed resource is unavailable. Queueing changes latency distributions and can consume additional memory or connection capacity.
2. Rejecting or shedding some work can protect the rest, but the service must decide which operations may be rejected and how callers learn the outcome. An order with unknown commit status cannot be retried as a brand-new order without considering identity.
3. Capacity during one-node failure can be lower than normal capacity. Headroom calculations should include the failure state the service claims to tolerate.

■ 50.4.5 TECHNICAL — the engineer's version

1. Google's overload discussion covers load shedding and mechanisms for avoiding overload amplification. The queue arithmetic here is an original deterministic model, not an experiment on Google's systems. [S199]
2. Throughput and latency should be measured together under a specified arrival process. Closed-loop clients that wait for responses can reduce their offered load as the server slows, hiding a production arrival-rate problem.
3. Report accepted, completed, rejected, timed-out and unknown attempts separately. A benchmark that silently discards failed requests may make overload appear faster rather than less reliable.

■ 50.4.6 WORDS — remember these

1. **Saturation:** demand meeting a limiting resource — a condition producing increased waiting, rejection or reduced useful throughput. **Headroom:** usable spare capacity — margin under stated workload and failure assumptions, not a generic machine percentage. **Admission control:** deciding which work to accept — protection of bounded resources through explicit entry policies. **Retry amplification:** repeated attempts adding load during trouble — feedback that can worsen the condition callers are trying to recover from.

50.5 Cost attribution

■ 50.5.1 PLAIN — in simple words

1. Cost belongs to resources and work over time. A monthly server price alone does not describe the full cost of a data service.
2. Include storage, backups, transfer, monitoring, support and operational effort where they matter. State what is excluded rather than hiding it in an apparently precise cost per user.
3. Shared infrastructure needs an allocation rule. Equal division, usage-based division and capacity-reservation division answer different questions.

4. A cheaper architecture is not necessarily cheaper for the required outcome. A design that saves storage but makes restoration too slow may fail its purpose.

■ 50.5.2 PLAIN — a picture in your head

1. Mira shares a delivery van with another shop. Dividing the bill equally is simple, but one shop may use most of the route time.
2. Charging by parcel count can still be misleading when one shop's parcels are much larger. The allocation method needs to match the decision being made.
3. **Where the comparison breaks:** computing has fixed commitments, burst usage, tiered prices and shared bottlenecks. A single per-request charge may not represent marginal cost or reserved capacity.

■ 50.5.3 PLAIN — a worked example

1. Invent monthly amounts of 6,000 currency units for compute, 1,000 for storage, 800 for backups and 1,200 for observability. The sum is 9,000 before any omitted transfer, tax, support or labour costs.
2. With 300,000 eligible completed operations, the allocated average is $9,000 / 300,000 = 0.03$ units per completed operation under this chosen boundary.
3. If only 200,000 operations complete while the fixed bill remains 9,000, the same allocated measure becomes 0.045. That does not mean each additional operation physically costs 0.045; average and marginal cost differ.
4. A report should state the currency, accounting window, included charges, denominator and allocation method. These invented values are not quotes from a cloud provider or estimates for the user's website.

■ 50.5.4 PLAIN — what is really happening inside

1. Resource meters measure selected consumption. Billing rules convert it into charges, and an attribution model allocates those charges to tenants, workflows or products.
2. A shared fixed cost cannot be uniquely divided by arithmetic alone; an allocation policy is required. The policy should be recorded so comparisons remain consistent.
3. Optimisation should preserve required outcomes. Removing all backups lowers one bill line while removing the recovery property the system was supposed to provide.

■ 50.5.5 TECHNICAL — the engineer's version

1. Separate invoice reconciliation, allocated unit cost, marginal cost and opportunity cost. They answer different management questions and can legitimately produce different numbers.
2. Evaluate changes using a fixed workload definition and correctness constraints. A query that becomes cheap by excluding required records is not an optimisation of the original question.
3. The companion cost function checks nonnegative finite amounts and an explicit nonzero denominator. It does not connect to billing accounts or supply current market prices.

■ 50.5.6 WORDS — remember these

1. **Allocated cost:** a share assigned under a rule — a reporting quantity based on an explicit apportionment policy. **Marginal cost:** the change caused by an additional unit of work — distinct from

dividing the total bill by current volume. **Cost boundary:** which charges are counted — the stated inclusions and exclusions behind a reported amount.

50.6 Operational reviews

■ 50.6.1 PLAIN — in simple words

1. Review a data service as an ongoing responsibility, not a machine purchased once. Workload, dependencies, retention and permissions change.
2. A useful review compares actual outcomes with objectives, checks forecasts against measurements and assigns concrete actions for gaps.
3. Alerts need an owner and a response. A dashboard containing hundreds of red boxes but no decision path is not a complete operating method.
4. Keep a record of what changed and what followed. An incident is a chance to improve the system's evidence and controls, not an excuse to replace uncertainty with a confident story.

■ 50.6.2 PLAIN — a picture in your head

1. Mira checks the van's service record, delivery outcomes and next month's expected workload. She does not wait for the engine to fail before discovering that no replacement driver is available.
2. When a delivery goes wrong, she records the observed sequence and tests plausible explanations rather than blaming the nearest person automatically.
3. **Where the comparison breaks:** distributed incidents can have several interacting causes. One graph changing first does not automatically establish it as the root cause.

■ 50.6.3 PLAIN — a worked example

1. A synthetic weekly review contains: request success under its metric definition, p50/p95/p99 latency with sample counts, oldest queued job age, restore rehearsal age, growth versus forecast and unresolved lifecycle tasks.
2. A storage forecast says the intervention threshold is 25 days away. Procurement or migration takes 14 days and the team reserves 7 days for uncertainty. That leaves only 4 days before the planned action should begin under these assumptions.
3. A backup exists, but the last verified restore is old and uses a different application schema. The review records a new rehearsal task; it does not treat backup presence as current recovery proof.
4. An alert for growing outbox age includes a runbook: inspect worker health and failures, check destination availability, preserve message identities and avoid blind replay with new keys.

■ 50.6.4 PLAIN — what is really happening inside

1. A review closes the loop between measurements and decisions. Forecast errors change the next forecast; recurring incidents change tests, capacity or architecture.
2. Operational evidence should retain its context: revision, environment, workload and observation time. Comparing two dashboards without those details can confuse a workload change with an implementation improvement.
3. A runbook proposes bounded observations and safe actions. Destructive changes and broad privilege escalation need their own authority, not an implicit permission hidden in an alert.

■ 50.6.5 TECHNICAL — the engineer's version

1. Tie each alert to an actionable condition and owner. Separate symptoms affecting users from diagnostic signals that help explain them; both are valuable but serve different roles. [S103]
2. Maintain explicit unknowns: untested failover, incomplete telemetry, obsolete restore evidence or unidentified growth sources. Assigning an owner makes an unknown manageable; renaming it green does not.
3. The capstone's operational worksheet uses original synthetic arithmetic. Its production extension requires measurements from the actual service rather than estimates inferred from user count alone.

■ 50.6.6 WORDS — remember these

1. **Runbook:** a documented response path — observations, decisions, bounded actions and escalation conditions for an operational situation. **Forecast error:** the difference between prediction and observation — evidence used to update assumptions rather than hide uncertainty. **Operational owner:** the accountable responder — the person or team responsible for a signal, policy or unresolved task.

50.97 Practice and worked answers

1. **Question:** What is the nonconformance allowance at 99.5% over 20,000 eligible requests? **Answer:** 100 under the example's exact definition.
2. **Question:** Can all trace span durations simply be added? **Answer:** Not if spans overlap; the worked 240 ms decomposition explicitly uses disjoint intervals.
3. **Question:** How much payload do 12,000 daily records of 600 bytes create over 365 days? **Answer:** 2,628,000,000 bytes before physical and operational overhead.
4. **Question:** At 20 GB used, 2 GB/day growth and a 70 GB intervention threshold, how much time remains? **Answer:** 25 days under constant net growth.
5. **Question:** How quickly does a 600-job backlog drain with capacity 100/s and ongoing arrivals 90/s? **Answer:** In 60 seconds under the deterministic assumptions, not 6 seconds.
6. **Question:** Does $L=\lambda W$ estimate p99 latency? **Answer:** No. It relates suitable averages under steady-state conditions.
7. **Question:** What is the allocated average of 9,000 units over 300,000 completed operations? **Answer:** 0.03 units per operation within the stated cost boundary.
8. **Question:** Does an existing backup establish current recoverability? **Answer:** No. A relevant restore procedure must be tested and its outcome checked.

50.98 Common wrong ideas

1. **Wrong:** Low CPU proves healthy service. **Right:** User outcomes and other bottlenecks may disagree.
2. **Wrong:** Missing errors mean all requests succeeded. **Right:** Observation coverage and unknown outcomes must be checked.
3. **Wrong:** Logs, metrics and traces are interchangeable. **Right:** They preserve different kinds of evidence.
4. **Wrong:** Number of users determines capacity. **Right:** Workload, data size, concurrency and objectives determine the relevant demand.

5. **Wrong:** More retries always improve reliability. **Right:** Retries can amplify overload and duplicate effects.
6. **Wrong:** A cost per operation is its marginal cost. **Right:** Allocated averages and incremental resource costs differ.
7. **Wrong:** A forecast is a specification. **Right:** It is an assumption-dependent prediction that needs updating.
8. **Wrong:** A dashboard replaces an operating team. **Right:** Signals need ownership, interpretation and bounded response procedures.

50.99 Chapter summary in 20 lines

1. Start observability with the user's intended outcome.
2. Define success, failure and unknown states.
3. Keep indicator, objective and agreement distinct.
4. Preserve the eligibility denominator and observation boundary.
5. Use logs, metrics and traces for their different strengths.
6. Correlate operations without copying unnecessary secrets or personal data.
7. Account for sampling, delayed export and missing signals.
8. Avoid unbounded metric-label cardinality.
9. Forecast storage from rates, representation sizes and lifetimes.
10. Separate payload from indexes, logs, copies and temporary work.
11. Keep GB and GiB explicit.
12. Set intervention thresholds early enough to act.
13. Identify the actual bottleneck before buying resources.
14. Measure throughput and latency under a declared arrival process.
15. Bound retries and protect the service from overload amplification.
16. State the failure assumptions behind headroom.
17. Make cost inclusions and allocation rules explicit.
18. Distinguish average cost from marginal cost.
19. Turn forecast errors and incidents into revised actions and tests.
20. Give every important unresolved operational condition an owner.

Capstone: From One Shop to a Multi-Branch Service

51.0 What this chapter gives you

1. We began with a person recording a sale. We can now name the contracts, representations, queries, transactions, copies and operational responsibilities that make a larger service understandable.
2. This capstone assembles a small executable teaching system rather than adding a new layer of unexplained technology. Its purpose is to make selected correctness claims observable, including refused operations and recovery after injected failures.
3. The original Mira's Corner fixture remains unchanged: O-1042 is 17,100 paise, O-1043 is 11,550, and four agreed lines represent six units and 28,650 paise. Those are agreed line amounts, not proof of payment settlement.
4. The capstone introduces separate synthetic identifiers: tenants T-A/T-B, branch B-1, products NOTE/PEN, requests R-1/R-2 and orders CAP-001/CAP-002. These are new test fixtures, not explanations of the earlier unresolved stock discrepancy.
5. The implementation is an isolated Python/SQLite learning lab. It does not include a production login, payment processor, carrier, web server, replicated database, physical-failure test or legal compliance determination.

51.1 Requirements and boundaries

■ 51.1.1 PLAIN — in simple words

1. Before choosing machinery, define what the small service must do. It should accept a permitted order, record the agreed prices, reduce the selected stock and remember the result of a repeated request.
2. Related changes must succeed together or leave no accepted order. A partially written order with fully deducted stock is not the intended outcome.
3. Each tenant's work must remain separate through the trusted helper interface. The same request or order identifier may exist independently in two tenants.
4. The service also records an intention to create a local follow-up task. That intention must not disappear because an acknowledgement is lost, and repeated delivery must not create repeated local tasks.
5. Not every real shop requirement belongs in the miniature model. Exclusions are stated so readers do not mistake a useful teaching implementation for a finished commercial checkout.

■ 51.1.2 PLAIN — a picture in your head

1. Mira writes an order, marks the shelf count and leaves a task for packing. The three records describe different responsibilities but belong to one accepted operation.
2. Dev repeating the same request after the answer was lost should recover the earlier result, not pretend another customer bought the same goods.
3. **Where the comparison breaks:** the lab's packing task is another database row. Creating that row does not physically pack a parcel, and it cannot make an uncontrolled external carrier perform an action exactly once.

■ 51.1.3 PLAIN — a worked example

1. Initialise T-A/B-1 with ten notebooks and fifty pens. The catalogue prices are 7,550 and 2,000 paise respectively. The separate T-B fixture has its own records with the same local product codes.
2. Submit R-1 for T-A order CAP-001 with two NOTE and one PEN. Its accepted total is $2 \times 7,550 + 1 \times 2,000 = 17,100$ paise. Remaining stock is eight notebooks and forty-nine pens in that branch.
3. Submit the identical R-1 again. The result remains CAP-001 at 17,100; stock remains eight and forty-nine; the outbox still contains one intention for that request.
4. Submit R-1 with three notebooks instead. That is a different command under the same request identity and is rejected as a conflict. The earlier result is not overwritten.
5. Submit a T-B target using a T-A-only principal. The helper rejects it before returning or changing protected T-B records. This does not test whether a real token was authenticated correctly; the principal is supplied by the trusted test harness.

■ 51.1.4 PLAIN — what is really happening inside

1. Input validation establishes the command's shape. Authorisation establishes its permitted scope. A database transaction then binds the accepted result, stock change, request record and outbox intention.
2. A request identity connects retries to the same intended operation. The stored command representation permits comparison with later uses of that identity.
3. Database constraints reject selected impossible states. Application code handles workflow decisions the schema does not express. Tests inspect both the successful path and the places where protection is intentionally absent.

■ 51.1.5 TECHNICAL — the engineer's version

1. The contract accepts bounded nonempty identifiers, one branch, a nonempty cart, unique product codes and strictly positive integer quantities. Python booleans are rejected as quantities even though `bool` is a subclass of `int`.
2. Prices are copied from the lab's catalogue inside the owned transaction. Request replay returns the persisted accepted result instead of repricing against a later catalogue. The example does not implement quote expiry, discounts or a separate price-approval workflow.
3. The helper requires an idle connection and owns `BEGIN IMMEDIATE`, `COMMIT` and `ROLLBACK`. SQLite transaction and driver behaviour are explained by their documentation; the selected workflow and limits are original. [S25] [S86]

■ 51.1.6 WORDS — remember these

1. **Accepted operation:** a committed result under a declared contract — the business outcome represented by the transaction, not merely a received HTTP request. **Scope boundary:** where the guarantee applies — the trusted helper, tenant, database and supported operations included in the claim. **Exclusion:** a deliberately unimplemented responsibility — an explicit limit such as payment settlement or physical dispatch.

51.2 Schema and workflows

■ 51.2.1 PLAIN — in simple words

1. Store each fact where its meaning is clear. Catalogue price describes what is offered now; an order-line price describes what was accepted for that order.
2. Stock belongs to a tenant, branch and product. An order belongs to a tenant and has lines referring to products in that same tenant.
3. Request records describe replay identity. Outbox records describe intended follow-up. Inbox records describe locally accepted deliveries. These are not three copies of an order total with identical purposes.
4. A separate schema for these responsibilities makes inconsistencies easier to state and test. It does not remove the need to decide which changes must share a transaction.

■ 51.2.2 PLAIN — a picture in your head

1. Mira keeps a price board, an order ledger, a shelf register and a packing-task tray. The price board changes without rewriting yesterday's signed order.
2. A note in the tray refers to an order rather than asking the packer to reconstruct the order from today's prices.
3. **Where the comparison breaks:** database references can be enforced mechanically, while paper references depend on people. Neither a foreign key nor a paper number proves that the real goods moved.

■ 51.2.3 PLAIN — a worked example

1. The lab schema uses these logical grains. A scoped key includes tenant wherever two tenants can legitimately reuse the local identity.

Relation	One row means	Important key or boundary
tenants	One synthetic organisation	tenant_id
catalogue	Current offer for one product	tenant_id, sku
stock	One branch's available quantity	tenant_id, branch_id, sku
orders	One accepted order	tenant_id, order_id
order_lines	One accepted product line	tenant_id, order_id, line_no
requests	One successful request result	tenant_id, request_id
outbox	One intended task notification	tenant_id, event_id
inbox	One locally processed delivery identity	tenant_id, event_id
tasks	One local follow-up task	tenant_id, event_id

2. CAP-001 has two order lines. Its total is derived from the accepted line quantities and prices, not recalculated from the current catalogue on every read.
3. Reprice NOTE in a separate test after acceptance. Reading CAP-001 still yields 17,100 paise. A genuinely new order can use the new catalogue price under the lab's contract.
4. The schema rejects a child whose scoped parent does not exist. Test that the child cannot borrow an order identity from another tenant by omitting part of the reference.

■ 51.2.4 PLAIN — what is really happening inside

1. The `accept_order` helper validates and authorises, starts its transaction, checks request identity, resolves product prices, conditionally deducts stock and inserts the related records.
2. The stock UPDATE includes a sufficient-quantity predicate. A zero-row result is treated as a failure of the intended reservation, not as an accepted order with missing stock work.
3. The accepted prices are protected from routine helper edits because no supported helper operation rewrites order lines. That is not the same as database-level immutability against an unrestricted connection.
4. All writes use bound values. Parameterisation prevents values from being interpreted as SQL syntax; it does not establish whether the caller is allowed to make the request.

■ 51.2.5 TECHNICAL — the engineer's version

1. SQLite STRICT tables, explicit NOT NULL requirements, CHECK predicates and composite foreign keys form the declarative part of the model. Foreign-key enforcement is enabled and verified on helper-created connections. [S60] [S59]
2. A transaction serialises this lab's write operation under SQLite's locking model. The result is not a proof that a corresponding PostgreSQL implementation behaves identically at every isolation level.
3. The implementation accepts one owned transaction at a time per connection and fails rather than silently committing a caller's unrelated pending work. This transaction-ownership boundary is part of its API contract.

■ 51.2.6 WORDS — remember these

1. **Row grain:** what one row asserts — the identity and meaning that determine valid joins and constraints. **Agreed price snapshot:** the price stored at acceptance — an order fact distinct from the mutable catalogue. **Transaction ownership:** responsibility for completion — the component entitled to commit or roll back the particular unit of work.

51.3 Correctness under retries

■ 51.3.1 PLAIN — in simple words

1. A retry is ambiguous unless it identifies the operation being repeated. The service needs to tell another delivery of the same command from a new command that happens to look similar.
2. Store the request identity with its successful result in the same transaction as the effect. Otherwise a failure can leave either an effect with no replay record or a replay record with no effect.
3. An outbox records follow-up intent atomically with the accepted order. Delivery can still be repeated, so the receiver needs its own duplicate policy.

4. Exactly one local database task is a narrower claim than exactly one email, payment or physical shipment. Keep that distinction when crossing external boundaries.

■ 51.3.2 PLAIN — a picture in your head

1. Dev sends a numbered instruction and does not hear back. He sends the same number again. Mira checks the completed-instruction register and returns the stored answer.
2. Mira's packing notification can also be delivered twice. The packer checks that notification's number before creating a second task.
3. **Where the comparison breaks:** the two registers may live in different databases and commit independently. A message acknowledgement cannot magically make two independent commits one indivisible event.

■ 51.3.3 PLAIN — a worked example

1. Run `accept_order` for R-1 and deliberately ignore the returned result. The committed state still contains the order, stock reduction, request result and outbox intention.
2. Retry R-1 with the same canonical command. The helper returns the stored result without a second deduction. Change its cart under the same key and the comparison rejects the conflict.
3. Deliver the outbox event to the local consumer. The consumer commits an inbox identity and one task together. Now simulate a lost acknowledgement before the producer records delivery completion.
4. Redeliver that event. The consumer sees the already-recorded identical event and does not create another task. The producer can then record a successful acknowledgement. The example observes two deliveries but one local task.
5. Inject an exception after task insertion but before its consumer transaction commits. Both task and inbox insert roll back. A later delivery can therefore create the task once instead of being falsely suppressed by an inbox-only residue.

■ 51.3.4 PLAIN — what is really happening inside

1. Idempotency requires an identity scope, a payload comparison, an effect boundary and a retention rule. A unique request key alone does not explain what happens when the payload changes or the old record expires.
2. The lab's command representation has sorted unique product lines and explicit branch/order identifiers. It compares canonical text, not only a hash, so digest equality is not treated as proof that arbitrary payloads are equal.
3. Producer completion and consumer effect are distinct observations. A lost acknowledgement leaves the producer uncertain even though the consumer may already have committed.
4. Retiring replay/inbox records would change the protection window. The exercise does not silently garbage-collect them and continue claiming permanent duplicate suppression.

■ 51.3.5 TECHNICAL — the engineer's version

1. Uniqueness and related effect writes share an owned transaction. Duplicate detection is a database-backed protocol, not a process-local set that disappears when the application restarts.
2. The consumer stores the event payload with its identity and checks conflicting redelivery. An identity reused for a different task is not accepted as a harmless duplicate.

3. The demonstration uses local SQLite transactions and synthetic delivery calls. It does not exercise a broker, network partition or external provider's idempotency contract. Chapters 26 and 39 explain why those boundaries need their own design.

■ 51.3.6 WORDS — remember these

1. **Idempotency scope:** where a replay identity is meaningful — the tenant, operation and retention boundary governing duplicate handling. **Outbox:** atomically recorded delivery intent — a local record that survives with the business change but does not guarantee external completion. **Inbox:** the consumer's delivery record — an identity committed with the local effect to support repeat handling. **Acknowledgement loss:** the result exists but the sender did not learn it — a source of uncertainty rather than proof that the action failed.

51.4 Reporting and reconciliation

■ 51.4.1 PLAIN — in simple words

1. A useful report states what its rows and totals mean. The same data can answer questions about orders, lines, units or agreed money, but those are not interchangeable measures.
2. Keep the original teaching fixture separate from newly accepted capstone orders. Combining them without labels changes the population and makes earlier totals appear wrong.
3. Reconcile at the level where mistakes can occur: identities, quantities, prices, stock changes and task counts. One matching grand total cannot detect every error.
4. Report generation should preserve authorisation scope. A correct sum over the wrong tenant is still the wrong response.

■ 51.4.2 PLAIN — a picture in your head

1. Mira compares the order ledger, shelf changes and packing tasks at closing time. Each record answers a different part of what happened.
2. Two wrong prices can cancel in the total. Checking the individual lines reveals the defect the headline number hides.
3. **Where the comparison breaks:** a database reconciliation compares stored claims, not the physical shelf itself. A physical count remains an independent observation that can disagree for several reasons.

■ 51.4.3 PLAIN — a worked example

1. Recreate the earlier canonical fixture using its unchanged lab. Assert two orders, four lines, six units and 28,650 paise. The expected-stock-10/observed-stock-9 example remains unresolved.
2. In a separate capstone database, accept CAP-001 only. Assert its two lines, three units, 17,100 paise and one outbox event. Do not add this deliberately similar example to the canonical total.
3. After two deliveries of the same capstone event, assert one inbox row and one task for that scoped identity. Count delivery attempts separately from completed local effects.
4. If an independent T-B order has the same local order ID, a T-A report must still include only T-A's authorised records. Test the filter using distinguishable values, not two identical fixtures that would hide a mix-up.

■ 51.4.4 PLAIN — what is really happening inside

1. A query produces a new relation whose grain follows its joins and grouping. Joining one order to multiple lines and multiple task records can multiply contributions unless the query controls each relationship.
2. Stored totals can be checked against line sums; stock changes can be compared with accepted reservations; task identities can be compared with recorded outbox events. Each reconciliation has a time and scope boundary.
3. External outcomes remain separate. An order total is not a payment settlement, and a task row is not a dispatched parcel. The schema would need additional evidence-bearing records for those claims.

■ 51.4.5 TECHNICAL — the engineer's version

1. The companion tests use explicit ORDER BY where result ordering matters and compare complete keys and values. SQL output order is not inferred from insertion order.
2. Keep aggregates at their intended grain, using preaggregation or EXISTS where appropriate. Chapter 13's fan-out counterexample remains a reminder that syntactically valid SQL can answer the wrong question. [S58] [S80]
3. A reconciliation report records its definition and exclusions. It does not quietly turn a sum of agreed line amounts into revenue recognition, tax calculation or cash accounting.

■ 51.4.6 WORDS — remember these

1. **Reconciliation population:** the records intended for comparison — a defined set that prevents mixing separate fixtures or tenants. **Effect count:** how many accepted changes occurred — distinct from how many attempts or deliveries were observed. **External outcome:** an event beyond the local transaction — payment, delivery or another result needing separate evidence.

51.5 Recovery and access tests

■ 51.5.1 PLAIN — in simple words

1. Test interrupted work, not only successful work. A failure halfway through a multi-step change should leave a state the contract can explain.
2. Test recovery into a fresh destination and compare meaningful records. The presence of a backup file is not a successful restore result.
3. Test forbidden scope before and after errors. A rollback should not leave reused request context pointing at another tenant.
4. Name the failure being simulated. A raised Python exception is not a power cut, and an in-memory restore is not an off-site disaster-recovery exercise.

■ 51.5.2 PLAIN — a picture in your head

1. Mira rehearses an interrupted order by stopping after writing one page and checking that the temporary work is discarded consistently.
2. She rehearses recovery by opening a copied ledger at a different desk and reading actual entries. Looking at the closed recovery box is not enough.

3. **Where the comparison breaks:** software failures occur at several layers. Exception handling exercises the application path; physical durability depends on storage, operating-system and device promises not exercised by that same test.

■ 51.5.3 PLAIN — a worked example

1. Inject an exception after stock deduction and before accepted-order completion. Assert that stock, orders, lines, request record and outbox return to the pre-operation state under the owned transaction.
2. Retry after removing the injected failure. Assert a single accepted result and the expected remaining stock. A failed attempt did not consume the successful request key permanently in this contract.
3. Use SQLite's backup API to copy the synthetic database into a fresh destination. Re-enable connection-level foreign-key enforcement as required and compare scoped orders, line values, stock and replay records.
4. Run integrity and foreign-key checks for their distinct purposes. An integrity result is not a substitute for checking reference violations or business totals.
5. Attempt cross-tenant reads/writes and a forbidden action. Assert that outputs and stored state respect the trusted-helper contract. Then explicitly note that unrestricted SQL access lies outside that contract.

■ 51.5.4 PLAIN — what is really happening inside

1. Rollback restores the transaction's database changes. It cannot undo an external side effect already performed, which is why the lab records intent instead of dispatching an email inside the transaction.
2. A backup API coordinates copying according to the engine's rules. The copied database still needs validation and appropriate application configuration before it is used.
3. The test harness supplies known state and forced interruption points. It checks the resulting state against an expected contract rather than inferring success from an exception name alone.

■ 51.5.5 TECHNICAL — the engineer's version

1. Python's sqlite3 API exposes backup and explicit transaction management; SQLite documents the distinct integrity and foreign-key checks. The final test record identifies the actual Python/SQLite runtime used. [S55] [S59] [S86]
2. The earlier two-connection temporary-file locking exercise demonstrates one bounded real SQLite writer-contention scenario. The capstone's deterministic models and in-memory tests do not expand that into a multi-host concurrency claim.
3. Production evidence would additionally need genuine authentication/role integration, representative concurrent workloads, restore objectives, failure-domain tests, secret handling and lifecycle verification across actual copies. Those are a deployment programme, not an unreported result of this lab.

■ 51.5.6 WORDS — remember these

1. **Fault injection:** deliberately interrupting a known path — a test technique whose simulated failure must be named precisely. **Restore rehearsal:** recovering and checking a separate destination — evidence about a specific recovery procedure and input. **Integrity check:** inspection of de-

clared structural properties — one layer of validation, distinct from complete business or security correctness.

51.6 Review and release evidence

■ 51.6.1 PLAIN — in simple words

1. A finished teaching package should let another reader see which examples were implemented, which tests ran and which discussions remain conceptual.
2. Keep manuscript, code and evidence aligned. A paragraph claiming a particular lab exists is misleading if the download contains no such exercise.
3. Test counts measure cases executed, not overall truth. A test can correctly demonstrate missing protection, and a large suite can still omit an important requirement.
4. A public learning book is not a production-service certificate. Readers should be able to learn from the evidence without inheriting an unsupported guarantee.

■ 51.6.2 PLAIN — a picture in your head

1. Mira hands Dev a recipe, the ingredients actually used and the notes from a trial batch. He can distinguish what was cooked from what was only proposed.
2. If the trial used one small oven, the notes should not claim that a factory production line was tested.
3. **Where the comparison breaks:** reproducibility also depends on software versions and exact input bytes. A familiar filename alone may not identify the same candidate.

■ 51.6.3 PLAIN — a worked example

1. Extract the companion code into a fresh folder and read its README before execution. Run the supplied unittest discovery command with the supported Python version. The output identifies successes, failures and any explicit skips.
2. Read the lab inventory. The foundations modules cover Chapters 1–12; `advanced_lab` contains bounded models and `query/representation` exercises for later topics; `capstone_lab` assembles the scoped local order workflow and Part H calculations.
3. Compare the included QA report with the package manifest. The report identifies the actual test run and generated artefacts; the manifest supports checking whether a later copy has changed.
4. A skipped optional capability is not a passed test. An illustrative PostgreSQL statement remains documented rather than executed unless the evidence explicitly identifies a real PostgreSQL run.
5. The package's website is the book reader, not a deployed checkout application. Publishing the reader makes educational text and downloads available; it does not install the capstone as a commercial service.

■ 51.6.4 PLAIN — what is really happening inside

1. A test suite supplies executable assertions over known inputs. A build pipeline converts one manuscript into PDF, volume and browser representations. Structural checks then compare chapter coverage, identifiers and resource targets.
2. Visual inspection catches a different class of problem: a clipped formula, unreadable table or misplaced heading may survive a text-only check.

3. The release record should distinguish automatic checks, visual review, source research and any independent review. Do not invent a reviewer or declare that one authoring pass is independent of itself.

■ 51.6.5 TECHNICAL — the engineer's version

1. Python unittest's result model distinguishes failures, errors and skips. The supplied runner records the runtime and actual counts rather than hardcoding a flattering number in the manuscript. [S201]
2. Stable chapter and section identifiers support cross-edition reference even when a standalone volume has different page numbers. Each volume's contents refer to that volume; its chapter numbers match the complete edition.
3. A package manifest is an integrity aid, not authentication of its publisher by itself. A trustworthy distribution channel and an independently obtained expected digest are needed for stronger origin claims.
4. The practical conclusion is not "never trust a system." It is to define the required claim, inspect the mechanism, run a suitable observation and keep the conclusion within the evidence.

■ 51.6.6 WORDS — remember these

1. **Executable assertion:** a machine-checkable expectation — a predicate comparing an observed result with a declared outcome. **Release manifest:** an inventory of delivered bytes — filenames, sizes and digests that support integrity checks. **Evidence boundary:** the limit of a justified conclusion — the tested scope, assumptions and unresolved properties attached to a result.

51.97 Practice and worked answers

1. **Question:** What stock remains after CAP-001 and its identical retry? **Answer:** Eight notebooks and forty-nine pens in T-A/B-1; the retry returns the accepted result without a second deduction.
2. **Question:** What happens if R-1 returns with a different quantity? **Answer:** A conflicting command under the same scoped identity is refused; the old result is not overwritten.
3. **Question:** Does repricing NOTE change an accepted CAP-001? **Answer:** No. Its line prices remain the accepted snapshot.
4. **Question:** What does two deliveries and one task establish? **Answer:** The tested local inbox/task protocol suppresses an identical repeated delivery. It does not establish exactly-once external dispatch.
5. **Question:** Why compare both tenants' state after a denied write? **Answer:** A denied response alone does not prove that no protected state changed.
6. **Question:** Does this capstone explain the original stock mismatch? **Answer:** No. Its new fixtures are separate; the original expected-ten/observed-nine discrepancy still has no asserted cause.
7. **Question:** What does the backup test not establish? **Answer:** Off-site recovery, device power-loss durability, PostgreSQL recovery and production objectives remain outside its scope.
8. **Question:** Is the website reader a production checkout deployment? **Answer:** No. It publishes educational content and downloads, not a live order-processing service.

51.98 Common wrong ideas

1. **Wrong:** A capstone must implement every commercial requirement. **Right:** A useful educational system has explicit included claims and exclusions.

2. **Wrong:** A request key alone creates idempotency. **Right:** Scope, payload comparison, atomic effects and retention also matter.
3. **Wrong:** The current catalogue is the historical agreement. **Right:** Accepted line prices are separate facts.
4. **Wrong:** An outbox guarantees exactly one external action. **Right:** It records intent; receiver and external-effect boundaries remain.
5. **Wrong:** A cross-tenant SELECT test proves all access paths are safe. **Right:** Exports, workers, mutations and privileged interfaces need their own coverage.
6. **Wrong:** Matching totals reconcile everything. **Right:** Compare scoped identities and values as well.
7. **Wrong:** Exception injection tests physical durability. **Right:** It exercises only the injected software path and transaction outcome.
8. **Wrong:** Many passing tests make the publication infallible. **Right:** Test coverage and factual/editorial review remain distinct forms of evidence.

51.99 Chapter summary in 20 lines

1. Begin the capstone with explicit requirements and exclusions.
2. Keep new fixtures separate from the original canonical records.
3. Establish principal and tenant scope before protected operations.
4. Validate bounded command shapes and integer quantities.
5. Capture agreed prices separately from mutable catalogue prices.
6. Give every table a clear row grain and scoped identity.
7. Own one transaction for each accepted local operation.
8. Keep stock, order, request result and outbox intent atomic.
9. Compare repeated payloads under the same request identity.
10. Recover a lost response without creating another order.
11. Commit consumer inbox and local task together.
12. Distinguish redelivery from repeated local effect.
13. Do not extend local deduplication to uncontrolled external actions.
14. Reconcile keys, quantities, prices and scoped totals.
15. Test refused operations and their stored effects.
16. Inject bounded failures and inspect rollback.
17. Restore into a fresh destination and check useful records.
18. Identify the actual runtime, assertions and skips.
19. Keep text, code, downloads and release records consistent.
20. Carry the evidence discipline from one shop to every larger system.

The Reference Desk: Glossary and Diagnostic Guides

52.0 What this chapter gives you

1. This chapter is a place to return when a term, query or failure needs a precise next step. It is not a substitute for the explanations earlier in the book; it connects their vocabulary and methods to reusable reference material.
2. The complete A–Z glossary follows the chapter in the full edition and is available in the browser reader. Each entry points back to its teaching section. The sources register records document titles, versions, URLs and consultation dates.
3. Stable references such as 26.4 and 47.3 mean the same section across the complete PDF, volume editions and website. Printed page numbers may differ between editions, so chapter/section identifiers are the reliable cross-edition route.
4. The diagnostic paths below begin with observation and scope. They do not give blanket permission to run destructive commands, broaden access or modify an unfamiliar live database.

52.1 Plain-to-technical glossary

■ 52.1.1 PLAIN — in simple words

1. A useful definition connects an everyday idea to a precise technical meaning. It also marks where that meaning changes with context or implementation.
2. Similar words do not always mean the same thing. A database, database engine, database server and database file refer to different parts of the system.
3. Begin with the plain meaning, then use the linked section to inspect assumptions and mechanisms. Memorising a term without its boundary can make a mistaken explanation sound more convincing.
4. The A–Z register collects the vocabulary blocks from the manuscript. Where a term appears in several contexts, the relevant definitions and locations remain visible rather than being silently merged into one misleading sentence.

■ 52.1.2 PLAIN — a picture in your head

1. A city map tells you where a street is; it does not describe every room in every building. The glossary locates an idea, and the chapter supplies the detailed route.
2. Two streets can share a familiar name in different towns. In the same way, “consistency” can refer to several distinct properties unless the surrounding discussion names the model.

3. **Where the comparison breaks:** technical vocabulary is not governed by one universal naming authority. Standards, product documentation and informal usage can use related words differently.

■ 52.1.3 PLAIN — a worked example

1. Look up transaction. The everyday meaning is a group of related changes treated as one controlled unit. The technical discussion adds commit, rollback, isolation, durability and the boundary around external effects.
2. Look up idempotency. The plain idea is that repeating the same intended operation need not produce another effect. Chapter 26 and the capstone add scoped identity, payload comparison, transaction placement and retention.
3. Look up consistency before using it in an argument. A business invariant, a replica freshness requirement and linearizability are different claims. A system can satisfy one and fail another.
4. A glossary answer becomes useful when it leads to the relevant question: Which changes? Which identity? Which observation order? Which failure? Which version of the product?

■ 52.1.4 PLAIN — what is really happening inside

1. The glossary builder extracts named terms and their definitions from the WORDS blocks and associates them with chapter/section identifiers.
2. The website groups entries alphabetically and offers text filtering. Chapter search uses titles, section terms and chapter text rather than pretending a keyword match establishes technical correctness.
3. A repeated term can have several entries or locations. That is preferable to discarding context simply to make a smaller count look tidier.

■ 52.1.5 TECHNICAL — the engineer's version

1. Use the source register when a definition is standard- or product-specific. The glossary's role is navigation; it does not supersede the cited specification.
2. Treat identifiers in a reference system as stable keys. A title can be edited while a chapter number and section identity preserve links across formats.
3. Generated vocabulary is checked for resolvable source locations. Mechanical extraction does not independently verify every definition; factual accuracy still depends on the manuscript and its cited evidence.

■ 52.1.6 WORDS — remember these

1. **Glossary:** a vocabulary reference — definitions connected to their contexts and teaching locations. **Cross-reference:** a route to related material — a stable chapter, section or source identifier rather than a guessed page number. **Implementation boundary:** where a claim stops generalising — the product, version and configuration for which a detail is established.

52.2 SQL and data-type reference

■ 52.2.1 PLAIN — in simple words

1. Read a query as a precise question about a chosen population. First understand what one input row means, then what joins, filters and grouping do to that population.

2. Choose a type by the value's meaning. Counts, money, names, dates, instants and missing values should not become interchangeable strings merely because strings are easy to store.
3. Use explicit ordering when order matters and bound values when inputs come from outside the SQL text.
4. A correct query still needs appropriate authority, transaction context and an honest interpretation of its result.

■ 52.2.2 PLAIN — a picture in your head

1. Mira asks Dev to take particular folders, match related pages, discard ineligible entries, group the remaining facts and sort the answer.
2. Each instruction changes the question. Accidentally matching each order page to every packing slip can multiply the answer even when Dev adds the numbers correctly.
3. **Where the comparison breaks:** a database optimiser can execute a logically equivalent plan in a different physical order. The readable conceptual sequence is not a claim about the engine's exact execution schedule.

■ 52.2.3 PLAIN — a worked example

1. The canonical agreed-line calculation is quantity multiplied by agreed unit price in paise, summed over the intended four lines. Its result is 28,650 paise. It is not a sum of today's catalogue prices.
2. This small reference table connects common intentions to important cautions. It is not a complete SQL grammar.

Intention	Common SQL form	Check before trusting the result
Choose values	SELECT expressions FROM relation	What does one input row mean?
Select a population	WHERE predicate	How are NULL and excluded rows treated?
Connect facts	JOIN ... ON complete_key	Can either side multiply the other?
Preserve unmatched rows	LEFT JOIN	Does a later WHERE remove them again?
Summarise groups	GROUP BY, SUM, COUNT	What is the output grain and denominator?
Retain detail with a calculation	OVER (...)	What partition, ordering and frame apply?
Specify order	ORDER BY, explicit tie-breaker	Are ties and missing values defined?
Change selected rows	UPDATE ... WHERE ...	Is the complete key and expected version included?
Bind input values	Driver parameters	Values are not SQL identifiers or permissions
Complete a unit of work	COMMIT or ROLLBACK	Which component owns the transaction?

3. COUNT(*) and COUNT(nullable_column) can differ because they count different things. Preserve the denominator required by the question rather than choosing whichever expression gives the more attractive number.
4. SUM(DISTINCT amount) does not generally repair a multiplied join. Two legitimate lines can have equal amounts and both should count. Fix the grain and relationship, not the coincidence of values.

■ 52.2.4 PLAIN — what is really happening inside

1. Parsing establishes a valid statement; name/type resolution connects it to objects and operations; planning chooses access paths; execution produces the result under the database's rules.
2. Representation decisions affect correctness. An integer amount needs a unit and range; a decimal needs scale and rounding policy; a timestamp needs a meaning and time basis.
3. A constraint can reject selected invalid values. It cannot determine whether a supplied measurement, currency or identity matches the outside world without a trusted observation or policy.

■ 52.2.5 TECHNICAL — the engineer's version

1. PostgreSQL and SQLite are not interchangeable implementations of every type, function or transaction behaviour. Use the product/version source attached to the example, especially around NULL, typing, locking and schema changes. [S58] [S80] [S86]
2. A compact representation checklist follows. It describes design questions rather than claiming one universally best physical type.

Value meaning	Representation decision	Required surrounding contract
Count	Integer with a suitable range	Unit, allowed sign, overflow behaviour
Money	Exact decimal or integer minor units	Currency, scale, rounding and agreement time
Identifier	Text, integer or structured key	Scope, stability, generation and reuse policy
Name	Unicode text	Comparison, normalisation and display rules
Instant	Aware time representation	Time scale, zone/offset interpretation and precision
Local date	Calendar date	Calendar and policy context; not an instant by itself
Unknown value	NULL or explicit status model	Unknown versus not applicable versus invalid
Measured quantity	Value plus unit and context	Method, precision, uncertainty and observation time

3. Keep SQL syntax examples and executable files aligned. A code block labelled as a sketch should not be treated as a complete application merely because it resembles runnable SQL.

■ 52.2.6 WORDS — remember these

1. **Query grain:** what one result row means — the level created by joins, projection and grouping. **Tie-breaker:** an additional ordering key — a rule making otherwise tied results deterministic where required. **Representation contract:** meaning attached to stored values — units, range, missingness and interpretation beyond the physical type.

52.3 Diagnostic decision guides

■ 52.3.1 PLAIN — in simple words

1. A symptom is the starting observation, not the conclusion. “The total is wrong” and “the database is broken” are different statements.

2. Choose a small next observation that can distinguish plausible explanations. Broad changes made before preserving evidence can destroy the information needed to understand the failure.
3. Keep read-only inspection separate from repair. A diagnostic query does not authorise deleting the rows it happens to find.
4. Record the database, tenant, time, revision and exact query or operation involved. A result from the wrong environment can be perfectly accurate and irrelevant.

■ 52.3.2 PLAIN — a picture in your head

1. A shop light does not turn on. Mira first checks whether one bulb, one room or the whole building is affected. She does not immediately replace every wire.
2. The scope of the symptom determines the next useful check. A systematic route reduces guessing and avoids damaging evidence.
3. **Where the comparison breaks:** data-system failures can be partial and concurrent. The same query can legitimately observe different states under different snapshots or replicas.

■ 52.3.3 PLAIN — a worked example

1. Use this wrong-total route on a disposable copy or through authorised read-only inspection:

```
Unexpected total
-> confirm metric definition, unit and population
-> confirm tenant, time basis and source version
-> compare contributing primary keys
-> inspect join multiplicity before aggregation
-> inspect NULL, filters and unmatched rows
-> compare agreed values with mutable catalogue values
-> reconcile raw contributions and declared exclusions
-> preserve evidence; propose a bounded correction
```

2. Use this slow-query route without assuming an index is the answer:

```
Slow request
-> locate where the time is spent
-> separate queueing, locks, database work and network
-> capture query, parameters, plan and observation scope
-> compare estimates with observed work where safe
-> check workload, selectivity, memory and storage waits
-> test one justified change on representative data
-> compare equal results and full latency distributions
```

3. Use this timeout route before retrying a write:

```
Caller did not receive a result
-> preserve the original request identity
-> ask whether the operation's status can be queried
-> replay only under its documented same-command policy
-> distinguish committed, rejected and still-unknown outcomes
-> do not create a new operation merely to make uncertainty disappear
```

4. Each route chooses an observation. None says that a symptom alone proves data corruption, failed commit or insufficient hardware.

■ 52.3.4 PLAIN — what is really happening inside

1. Diagnosis compares hypotheses with observations. A good next check eliminates or narrows explanations while changing as little relevant state as possible.

2. Evidence can be time-sensitive. Query plans, lock waits and replica lag can change; record their observation window rather than presenting them as permanent properties.
3. A repair should have its own preconditions, scope, validation and recovery plan. The urgency of a symptom does not remove those responsibilities.

■ 52.3.5 TECHNICAL — the engineer's version

1. For stale reads, check the requested consistency contract, routing, snapshot and observed replication position. A follower being behind is not automatically corruption.
2. For missing records, check authorisation filters, tenant scope, transaction outcome, lifecycle action and environment before attempting restoration. A correctly hidden row and a lost row require different responses.
3. For suspected corruption, preserve the original evidence and use engine-specific checks on an appropriate copy. Structural checks, checksums and business reconciliation establish different properties; consult Chapters 31–32 before repair.

■ 52.3.6 WORDS — remember these

1. **Symptom:** what was observed — a result needing explanation, not an already-proven cause. **Hypothesis:** a candidate explanation — a claim that should be distinguished from alternatives through suitable evidence. **Bounded observation:** a deliberately limited check — a query or measurement with named scope and minimal unintended change.

52.4 Design-review questions

■ 52.4.1 PLAIN — in simple words

1. A design review should make hidden assumptions visible. Ask what each record means, who may act and what happens when requests overlap or answers are lost.
2. Review the whole path, not only the database diagram. Inputs, workers, caches, exports and recovery procedures can invalidate an otherwise careful schema.
3. Require a counterexample to each important guarantee. Asking “How could this fail?” helps turn an attractive description into a testable contract.
4. A reviewer can identify evidence gaps without claiming the design is impossible. The next step is a targeted test or clarification, not an invented green verdict.

■ 52.4.2 PLAIN — a picture in your head

1. Before building a new storeroom, Mira asks what it must hold, who gets a key, how deliveries enter and what happens if the only door is blocked.
2. A beautiful floor plan is incomplete until those operating questions have answers.
3. **Where the comparison breaks:** software can change very quickly. A review of one version should not automatically cover a later edit to a constraint, permission or retry policy.

■ 52.4.3 PLAIN — a worked example

1. Apply these questions to the capstone before extending it:

Area	Review question	Example evidence
Meaning	What does one row assert?	Grain, units, provenance and exclusions
Identity	Where is each key unique?	Scoped key and reuse counterexamples
Authority	Who may perform each action?	Allow/deny matrix and effective-role checks
Change	Which facts must commit together?	Interrupted-operation state comparisons
Concurrency	Which overlapping schedule would break the rule?	A failing schedule and a tested protocol
Replay	What is the same intended request?	Identity, payload comparison and retention
Copies	Which reads may be stale?	Routing and declared consistency model
Recovery	What can actually be restored?	Isolated restore with checked records
Lifecycle	Which copies remain after deletion?	Inventory and explicit residual scope
Operations	What objective and capacity assumptions apply?	Workload, measurements and response owners

2. A new payment integration changes the external-effect boundary. The local task's once-only test does not become proof of once-only charging.
3. A new tenant export changes access coverage. The original detail-query test is not enough; the export's scope and content need separate tests.
4. A new retention policy changes replay guarantees if it removes request identities. The design must state what happens after that protection window ends.

■ 52.4.4 PLAIN — what is really happening inside

1. Reviews compare the proposed mechanisms with requirements and counterexamples. They expose which promises rely on unverified assumptions.
2. A small set of well-chosen adversarial cases can be more informative than a large set of near-identical successes. The goal is evidence coverage, not a decorative test count.
3. Review findings should retain severity, scope, proposed response and verification status. Author approval of a plan does not by itself close an implementation defect.

■ 52.4.5 TECHNICAL — the engineer's version

1. Separate safety properties, progress properties and operational objectives. "Never create two accepted orders for one scoped request" differs from "eventually deliver every valid notification" and from "respond within 200 ms."
2. Identify failure assumptions: process crash, device loss, network partition, byzantine behaviour or accidental operator changes. A mechanism designed for one fault model should not inherit another model's guarantee silently.
3. The review worksheet is an original reference aid. It records questions, not an automatic approval algorithm for every possible data system.

■ 52.4.6 WORDS — remember these

1. **Safety property:** a prohibited outcome never occurs — a correctness claim such as no duplicate accepted effect under the stated model. **Progress property:** required work can eventually advance — a liveness claim dependent on assumptions about failures and resources. **Fault model:** the failures considered — the boundary within which a mechanism's reasoning and tests are intended to apply.

52.5 Test and recovery templates

■ 52.5.1 PLAIN — in simple words

1. A test record should let another person repeat the important steps and understand the result. "It worked" omits the input, environment, operation and expected outcome.
2. A recovery record needs a specific source, destination and time boundary. Restoring the wrong backup correctly is still the wrong recovery.
3. Failed and skipped steps should remain visible. They are not automatically covered by the successes around them.
4. Templates organise evidence; filling the boxes does not manufacture observations that were never made.

■ 52.5.2 PLAIN — a picture in your head

1. A recipe notebook records ingredients, oven setting and actual result. Leaving out the temperature makes a successful trial hard to repeat.
2. A recovery checklist records which box was opened and which pages were compared. Checking only that a box exists cannot fill the result column.
3. **Where the comparison breaks:** execution environments can differ invisibly through library versions, configuration and concurrent work. Reproducibility requires more than copying visible commands.

■ 52.5.3 PLAIN — a worked example

1. Use this original bounded-test template for a new exercise:

```

Claim:
Candidate identity:
Environment and versions:
Input identities and scope:
Assumptions and excluded failures:
Operation and authority:
Expected output and state:
Actual output and state:
Negative or interruption control:
Result: pass / fail / error / skipped / unresolved
Evidence location:
What this result does not establish:

```

2. Use this recovery template before treating a backup as useful evidence:

```

Recovery purpose and authorised scope:
Backup identity, time and dependencies:
Required keys/configuration, without embedding secrets:
Fresh isolated destination:
Target recovery point and declared objectives:
Procedure and stop conditions:

```


Structural checks:
 Key/value and application reconciliation:
 Post-backup lifecycle decisions to reapply:
 Access-boundary checks:
 Actual duration and unresolved loss interval:
 Approval required before serving restored data:

3. For a replay test, record both attempts with the same scoped identity and compare effects after each. A response that merely repeats the same text is not enough if stock changed twice.
4. For a denied-operation test, inspect state and returned information. The error message is only one observable result.

■ 52.5.4 PLAIN — what is really happening inside

1. A test harness constructs controlled conditions, invokes the candidate and checks assertions. A recovery rehearsal applies a procedure to a selected historical state and validates the resulting destination.
2. Both need a reproducible identity for the tested material. Digests support byte comparison, while version and environment records explain how those bytes were used.
3. A result can be narrow and still valuable. The discipline is to keep the useful finding without converting it into a wider unsupported guarantee.

■ 52.5.5 TECHNICAL — the engineer's version

1. Use machine-readable reports alongside human interpretation. Record failures, errors and skips distinctly rather than treating every completed process as a successful test. [S201]
2. Restore evidence should compare business-level identities and values in addition to engine checks. A structurally valid database can still contain the wrong dataset.
3. Keep credentials out of the evidence bundle. Record how authority was established without copying the secret material needed to exercise it.

■ 52.5.6 WORDS — remember these

1. **Test fixture:** the controlled input state — synthetic or appropriately authorised data used for a specified check. **Reproducibility record:** enough context to repeat a result — candidate, inputs, environment, operations and observations. **Skipped check:** a check not executed under current conditions — neither a pass nor proof of failure of the underlying feature.

52.6 Source and version register

■ 52.6.1 PLAIN — in simple words

1. Technical behaviour changes across products and versions. The reference beside a claim tells you where its documented meaning came from and which scope was consulted.
2. A standard, a vendor implementation and a teaching model are different sources of authority. An original shop example can demonstrate arithmetic without becoming an industry standard.
3. Read sources when applying a mechanism to real work. The book provides a path to them, not a promise that every future release preserves every detail.
4. The final source register combines the earlier source identifiers with the added references. Old consultation dates remain historical; they are not relabelled as newly rechecked merely because a new PDF was built.

■ 52.6.2 PLAIN — a picture in your head

1. Mira keeps the instruction manual for the exact machine in the shop. A manual for another model may look similar but describe different buttons and limits.
2. A handwritten experiment note is useful too, provided it says what was tried and does not pretend to be the manufacturer's specification.
3. **Where the comparison breaks:** online documentation can change at an unchanged address. A URL alone is weaker evidence than a versioned document plus a consultation date and recorded scope.

■ 52.6.3 PLAIN — a worked example

1. A paragraph marked [S86] refers to Python's version-3.13 sqlite3 documentation. It does not establish which exact Python or SQLite build a reader's computer uses; the executable report records the actual authoring runtime separately.
2. A paragraph about PostgreSQL 17 row policies marked [S187] describes that documented feature and its exceptions. The local SQLite lab does not become a PostgreSQL test because the two appear in one chapter.
3. An invented queue with 120 arrivals/s and 100 completions/s is an original arithmetic model. Its result follows from its assumptions, not from a measurement of a hosting plan.
4. For an applied design, record the selected source version, relevant configuration, local experiment and remaining uncertainty together. No single reference establishes every layer of the final system.

■ 52.6.4 PLAIN — what is really happening inside

1. The publication builder resolves source identifiers to entries containing author or organisation, title, URL, consultation date and a short description of supported scope.
2. Manuscript references are checked for missing identifiers. That mechanical check prevents dangling citations; it does not prove that every citation is sufficient or that every interpretation is correct.
3. Readers should recheck version-specific details before deployment. When a source changes, preserve the earlier statement's historical context rather than pretending it always described the new version.

■ 52.6.5 TECHNICAL — the engineer's version

1. The final package includes canonical Markdown, structured chapter metadata, glossary/source registers, executable exercises and build/QA records. Derived formats use the same chapter numbering and headings.
2. The cited materials include official database and language documentation, standards, original research and author-hosted technical texts. The synthetic shop, diagrams, chosen protocols and worked arithmetic are original teaching material unless explicitly attributed otherwise.
3. No part of the source register claims independent technical certification of the book. Source support, local testing, visual review and publication approval remain distinct kinds of evidence.
4. The final lesson is the method used throughout: define meaning, name the boundary, inspect the mechanism, test the relevant case and state only what follows. That method is transferable beyond the particular tools in these pages.

■ 52.6.6 WORDS — remember these

1. **Primary source:** the originating documentation or research — evidence closer to the specified behaviour than a second-hand summary. **Consultation date:** when a source was checked — a historical marker that does not turn a moving page into an immutable specification. **Version register:** the declared implementation/document context — a record distinguishing documented versions from the runtime actually tested.

52.97 Practice and worked answers

1. **Question:** Which reference survives a change in volume pagination? **Answer:** The stable chapter and section identifier, such as 47.3.
2. **Question:** What is the first check for an unexpected total? **Answer:** Confirm the metric definition, units, population and authorised scope before proposing a repair.
3. **Question:** Why is SUM(DISTINCT amount) not a general join repair? **Answer:** Equal legitimate values can be distinct contributions; the relationship grain must be corrected.
4. **Question:** What identity should be preserved after a write timeout? **Answer:** The original scoped request identity under its documented replay contract.
5. **Question:** What distinguishes a safety property from a latency target? **Answer:** Safety forbids a defined outcome; latency targets quantify service performance over a stated population and window.
6. **Question:** Does a skipped test count as passed? **Answer:** No. It records a check not executed under the current conditions.
7. **Question:** Does a valid citation ID establish independent accuracy review? **Answer:** No. It establishes a resolvable reference, not universal correctness or independence.
8. **Question:** What is the book's most reusable skill? **Answer:** Connecting a precisely defined claim to a mechanism and suitable bounded evidence, while preserving what remains unknown.

52.98 Common wrong ideas

1. **Wrong:** Knowing a term means knowing its guarantees. **Right:** Read the definition's context, assumptions and implementation boundary.
2. **Wrong:** Page numbers are stable across every edition. **Right:** Chapter and section identities are the stable cross-edition route.
3. **Wrong:** A symptom is already a diagnosis. **Right:** Use observations to distinguish competing explanations.
4. **Wrong:** A faster query is necessarily the same question. **Right:** Compare results, population and semantics as well as time.
5. **Wrong:** A complete template supplies missing evidence. **Right:** Fields must contain actual observations or explicit unresolved status.
6. **Wrong:** A cited product's example tests another product. **Right:** Documentation and runtime scope must remain distinct.
7. **Wrong:** A file hash proves its publisher's identity by itself. **Right:** Integrity comparison needs a trusted expected value or distribution context.
8. **Wrong:** A reference book can remove every uncertainty. **Right:** It can teach readers to identify, bound and investigate uncertainty accurately.

52.99 Chapter summary in 20 lines

1. Use the reference desk to locate precise ideas and next steps.
2. Connect plain meanings to technical definitions and their limits.
3. Prefer stable chapter and section identifiers across editions.
4. Preserve context when one term has several meanings.
5. Read SQL as a question about a declared population.
6. Check grain before joins and aggregation.
7. Choose representations by meaning, units and range.
8. Treat NULL and missing status deliberately.
9. Specify ordering and tie-breakers where needed.
10. Preserve request identity after an uncertain write outcome.
11. Diagnose from symptoms rather than assuming a cause.
12. Choose bounded observations before repair.
13. Separate safety, progress and performance requirements.
14. Ask for counterexamples to important guarantees.
15. Record exact candidates, inputs and environments.
16. Distinguish pass, fail, error, skip and unresolved states.
17. Restore into isolation and compare useful records.
18. Keep primary sources and version context accessible.
19. Do not turn local checks into unearned universal certification.
20. Meaning, mechanisms and evidence belong together in every data system.

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
<code>foundations_lab.py</code>	1–3: canonical orders, number/text/time representations and NULL behavior	Pure functions and a fresh SQLite fixture.
<code>history_lab.py</code>	4–6: measurements, scoped identity, duplicates and reconstructed history	Small explicit state machines; not a production event store.
<code>data_bridge_lab.py</code>	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.
<code>schema_sql_lab.py</code>	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
<code>advanced_lab.py</code>	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.
<code>capstone_lab.py</code>	47–52: permission fixtures, tenant-scoped orders, outbox/inbox tasks, retries, local restore, retention status, resumable backfill and capacity/cost arithmetic	Trusted Principal objects, in-memory SQLite and bounded functions; not authentication, a public API or a deployed service.

The advanced models

The B-tree exercise implements ordered separator selection and a single overflowing-leaf split. It does not implement a complete balanced persistent B-tree. The log example replays committed toy transactions from a supplied durable prefix; it does not parse an engine’s actual WAL or simulate torn storage sectors. The compaction example can discard tombstones only under its stated complete-history, latest-only assumptions.

The replica model tracks contiguous applied positions. The fencing model has the resource check a monotonically increasing authority token. The consensus exercises enumerate majorities, compare last-term/last-index pairs and test the current-term direct-commit condition. These are selected rules, not a full Raft implementation or proof of progress under a real network.

The transaction participant and compensation classes show explicit state transitions and repeated-decision handling. The cache and manifest classes model generation/version checks in one process; they do not implement shared-memory atomicity or a distributed metadata service.

The stream examples distinguish fixed windows, sliding windows, session overlap and late corrections. The columnar exercise packs signed 64-bit integers and uses zlib to compare representations. It is not a Parquet writer or a disk-I/O benchmark. Text search uses a tiny tokenized corpus, with an additional SQLite FTS5 check when that feature is present. Vector examples calculate exact distances and filter by tenant before selecting results; they do not train an embedding model or implement a production approximate-nearest-neighbor index.

The Wilson interval, missing-status bounds and nearest-rank percentiles check arithmetic under explicit assumptions. They do not establish representative sampling, independence or causal identification in real business records.

The capstone boundary

The capstone accepts a canonicalized cart, reads current catalogue prices during acceptance, conditionally reduces stock and stores the agreed order, request identity and outbox event in one owned transaction. An identical scoped replay returns the committed result without repricing or reducing stock again. A conflicting payload using the same scoped request identity is rejected.

The consumer stores one local task and its inbox identity atomically. An injected lost acknowledgement occurs after that local consumer commit; redelivery does not create another task. This is not proof that an external courier, email provider or payment processor performs its work exactly once.

The local restore uses SQLite's backup API, enables foreign-key checking on the target, checks database integrity and references, compares logical records and continues a synthetic operation. It does not test power loss, cloud loss, independent failure domains or an off-site recovery objective.

The Principal objects are trusted fixtures constructed by the test. They are not login tokens. Holding the raw SQLite connection bypasses the helper authorization boundary; a test explicitly demonstrates this limit. The model does not enforce immutable agreements against an administrator or implement a complete checkout lifecycle.

Reading the test record

The release test record gives the exact test count, failures, errors, skips, runtime versions and hashes of the executable files. A skipped feature check is not a pass for that feature. Test names and assertions are the evidence; the number of tests is only a count.

Exercises elsewhere in the book can ask you to draw a schedule, inspect a plan, choose a workload or design a real restore procedure. Not every such task is a ready-made automated implementation. PostgreSQL deployments, actual network faults, device flush behavior, complete tenant security and live schema migrations remain tasks for an appropriately isolated environment with explicit authorization.

A–Z Glossary

Definitions are retained with their teaching context. Some familiar terms have several related meanings. Chapter and section identifiers are stable across editions.

A

ABA pattern

a value changes away and later returns to its earlier value — an intervening history that a comparison of only the final value cannot detect.

Read in context: 12.6.6

ABA problem

a value changes and returns, hiding intervening work — a limitation of equality-only checks that do not identify the relevant state history.

Read in context: 25.5.6

Accepted operation

a committed result under a declared contract — the business outcome represented by the transaction, not merely a received HTTP request.

Read in context: 51.1.6

Access method

the machinery behind a route — an engine-defined algorithm and operator interface used to organise and search index entries.

Read in context: 15.1.6

Accuracy

closeness to what is actually being measured; technically, closeness of agreement between a measured value and a true value of the measurand, not a numerical quantity in VIM usage.

Read in context: 4.3.6

Acknowledgement boundary

what a positive receipt actually confirms — the defined acceptance point represented by a queue, consumer or provider response.

Read in context: 26.5.6

Acknowledgement chain

the sequence of accepted handovers — responses from application, database, operating system and storage layers with distinct meanings.

Read in context: 30.1.6

Acknowledgement loss

the result exists but the sender did not learn it — a source of uncertainty rather than proof that the action failed.

Read in context: 51.3.6

Acquisition order

the sequence in which resources are requested — a shared ordering protocol intended to prevent cyclic waits.

Read in context: 24.4.6

Admission control

deciding which work to accept — protection of bounded resources through explicit entry policies.

Read in context: 50.4.6

Adversarial input

data chosen to exploit assumptions — inputs deliberately selected to cause incorrectness, excessive resource use or other unwanted behaviour.

Read in context: 17.5.6

Advisory lock

a coordination agreement writers must choose to follow — an application-defined locking facility not automatically tied to every relevant data mutation.

Read in context: 24.1.6

Affected-row check

verify how many target records changed — inspection of the statement result against an expected cardinality.

Read in context: 22.3.6

After-value

the state an accepted update should leave — a logical replacement used by this teaching recovery model.

Read in context: 28.5.6

Aggregate state

information accumulated for a group — working values sufficient to compute an aggregate such as a sum and count.

Read in context: 19.1.6

Aggregation

combining several records into an answer; technically, a calculation over a defined group of inputs.

Read in context: 2.3.6

Agreed line price

the price attached to a particular historical agreement — a transaction-specific fact distinct from a product's current catalogue price.

Read in context: 9.6.6

Agreed price snapshot

the price stored at acceptance — an order fact distinct from the mutable catalogue.

Read in context: 51.2.6

Alias

a name attached to an output expression — a result-column label that does not itself alter the source table's schema.

Read in context: 12.4.6

Alignment

place values at required byte boundaries — padding and layout rules supporting an implementation's representation or access requirements.

Read in context: 27.3.6

Allocated cost

a share assigned under a rule — a reporting quantity based on an explicit apportionment policy.

Read in context: 50.5.6

Allowed history

a sequence the contract permits — an operation trace used to evaluate a consistency or correctness claim.

Read in context: 36.6.6

Allowed lateness

the accepted revision interval — a configured policy governing how long late window data may affect retained computation.

Read in context: 41.5.6

Allowlisted query choice

selecting among explicitly permitted statement structures — a validated mapping from a caller's option to trusted fixed SQL.

Read in context: 12.6.6

Analytical workload

queries that study populations of records — work often involving scans, grouping and historical comparison.

Read in context: 42.1.6

Analyzer

the text interpretation pipeline — extraction, tokenization and normalization rules used for indexing and querying.

Read in context: 44.2.6

Append-only

ordinary updates add rather than replace — a mutation policy whose enforcement and exceptions must be specified.

Read in context: 6.6.6

Application

the program people use; technically, the software that implements a workflow and requests operations from other components.

Read in context: 1.1.6

Application consistency

related service components agree — correctness of references and workflows beyond one engine's internal file validity.

Read in context: 31.2.6

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

Approximate nearest-neighbour search

a shortcut to likely close items — retrieval that may miss exact neighbours to reduce cost.

Read in context: 45.3.6

Architecture trade-off

a benefit obtained with another cost or constraint — a choice evaluated against the complete task rather than one isolated metric.

Read in context: 42.6.6

Archive coverage

the recovery history actually retained and retrievable — verified continuity of required log material over a stated range.

Read in context: 31.3.6

Arithmetic bound

the largest or smallest result an operation may produce — a range derived from operand limits and the operation's cardinality.

Read in context: 10.4.6

Artefact identity

which exact candidate was used — a stable reference, often a digest, linking execution to reviewed bytes.

Read in context: 49.6.6

As-of time

when the available information was evaluated — the observation or snapshot boundary attached to a report.

Read in context: 19.4.6

Association

quantities vary together in observations — a statistical relationship that does not by itself establish intervention effects.

Read in context: 46.5.6

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 commit

acknowledge before full local log persistence — a configured contract that can trade recent-transaction durability for lower latency.

Read in context: 28.2.6

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

At-least-once delivery

retries may deliver a message repeatedly — a delivery contract that prioritizes eventual attempts while permitting duplicates under its stated assumptions.

Read in context: 26.5.6

Atomic commitment

one compatible commit-or-abort decision — a property distinct from simultaneous visibility or complete transaction isolation.

Read in context: 38.1.6

Atomic intent recording

accept the business change and its follow-up instruction together — committing state and outbox intent in one local transaction.

Read in context: 26.4.6

Atomic metadata publication

one accepted version replaces its expected predecessor — a concurrency boundary preventing incompatible partial or stale updates.

Read in context: 42.4.6

Atomic publication

output becomes accepted as one local change — committing records and their completion evidence within one transaction boundary.

Read in context: 40.5.6

Atomic visibility

observers do not see an intermediate replacement state — a concurrency property distinct from surviving a crash or power loss.

Read in context: 30.3.6

Atomicity

the grouped changes take effect together or not at all — a transaction property at a specified resource boundary.

the selected changes take effect together — all-or-nothing transactional acceptance of the covered database work.

Read in context: 8.4.6, 21.3.6

Attempt amplification

extra work caused by retries — the ratio or count of transaction attempts relative to completed business operations.

Read in context: 23.6.6

Attribute closure

everything the starting fields let you determine — the set of attributes implied by repeatedly applying the specified functional dependencies.

Read in context: 13.4.6

Audit trail

information used to explain actions; technically, a record of relevant operations and context, with its own integrity and retention requirements.

Read in context: 2.6.6

Authentication

establish a source or actor under a security protocol — a property not supplied merely by an unkeyed matching checksum.

Read in context: 32.2.6

Authorisation

permission to perform an action — an access decision separate from whether the action is a valid domain transition.

permission for a particular act — evaluation of a principal's rights over a resource in a defined context.

Read in context: 6.2.6, 47.1.6

Authoritative counter

a stored total used to make decisions — state whose agreement with the underlying events or allocations needs its own maintained invariant.

Read in context: 11.4.6

Authoritative field source

the record allowed to decide a disputed value — a defined authority for a particular fact, not necessarily one universal source for every field.

Read in context: 32.4.6

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

Authoritative representation

the record a decision is defined to rely on — a source-of-truth role assigned by the architecture.

the selected source of meaning — the version whose value governs while multiple copies coexist.

Read in context: 8.6.6, 49.2.6

B

Backfill

transforming already-stored records — population of a new representation under explicit source assumptions.

Read in context: 49.3.6

Backoff

wait longer before another attempt — a retry-spacing strategy that reduces repeated contention or overload.

Read in context: 24.5.6

Backpressure

make upstream work respect downstream limits — a mechanism that slows or bounds production when consumers cannot keep up.

downstream limits affect upstream admission — a mechanism preventing work from accumulating without bound.

Read in context: 20.3.6, 39.5.6

Backup

retained material intended for restoration — a copy or representation with a defined consistency and recovery scope.

a copy retained for a defined recovery purpose — data and metadata acquired under a procedure that supports a stated restoration objective.

Read in context: 8.5.6, 31.1.6

Backup retention

how long recovery copies remain available — a policy governing restore-point preservation and eventual retirement.

Read in context: 31.6.6

Backup verification

check the retained input against defined expectations — structural, manifest or checksum validation that may precede a full restore.

Read in context: 31.5.6

Baseline

the saved comparison point — a defined workload, configuration and observed result before an intervention.

Read in context: 18.6.6

Benchmark contract

the rules of the measurement — a specification of inputs, operations, environment, timing boundaries and success criteria.

Read in context: 20.1.6

Binary format

a byte layout read under explicit rules — a representation not restricted to a plain-text grammar.

Read in context: 7.5.6

Birthday effect

collisions become plausible before the space is full — pair growth causes collision probability to rise around the square root of a uniform output-space size.

Read in context: 17.5.6

Blocking

waiting for conflicting access to be released — a resource wait that need not contain a dependency cycle.

progress waits for unavailable coordination — a limitation distinct from a participant process being physically stopped.

Read in context: 24.3.6, 38.3.6

Blocking operation

input must accumulate before output can proceed — a stage such as a full sort under its particular execution strategy.

Read in context: 18.4.6

BM25

a family of lexical ranking rules — scoring that combines term weighting, frequency saturation and document-length effects.

Read in context: 44.4.6

Bound parameter

a value supplied separately from query text — data passed through a driver's parameter interface rather than inserted into SQL by string concatenation.

a value supplied separately from SQL structure — driver-mediated binding to a placeholder without interpreting the supplied value as new statement syntax.

Read in context: 8.2.6, 12.6.6

Boundary coverage

which protected paths were exercised — an inventory of tested interfaces, scopes and states, not a universal guarantee.

Read in context: 47.6.6

Boundary evidence

proof about a particular stage — an acknowledgement or record whose scope must not be exaggerated.

Read in context: 39.6.6

Boundary test

a test at an acceptance limit — a selected case at, immediately below or immediately above a declared domain boundary.

Read in context: 11.1.6

Bounded observation

a deliberately limited check — a query or measurement with named scope and minimal unintended change.

Read in context: 52.3.6

Buffer hit

a needed page was already in the relevant cache — an engine-level observation, not a statement that all possible lower-level work vanished.

Read in context: 18.5.6

Buffer pool

the database's in-memory page workspace — a managed cache of pages and metadata used by the storage engine.

Read in context: 27.2.6

Business domain

the values that make sense for a particular purpose — a semantic acceptance set including units, bounds, missingness and interpretation.

Read in context: 10.4.6

Business precondition

what must be true before an operation is accepted — an application rule that may require checking affected rows or queried state.

Read in context: 21.1.6

Business reconciliation

compare representations that should agree — an identity- and meaning-aware check of records, totals and state transitions.

Read in context: 32.4.6

Business revision

the edit state named by the application — an explicit version value used for conflict detection or documented history.

Read in context: 25.1.6

Busy outcome

the system cannot complete the requested access immediately — a contention result that requires an explicit wait, retry or failure policy.

Read in context: 8.1.6

Byte order

which end of a multi-byte number comes first — the endianness used to encode a numeric value.

Read in context: 7.5.6

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: 27.2.6, 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

Calibration

comparing indications with suitable references; technically, establishing the specified relationship between reference values and indications with their associated uncertainties.

Read in context: 4.3.6

Candidate

a record ready for the next acceptance stage — a validated, non-conflicting input that has not necessarily been published.

Read in context: 7.6.6

Candidate eligibility

the conditions a replacement must satisfy — requirements covering authority, history and readiness rather than mere reachability.

Read in context: 34.3.6

Candidate key

a possible way to identify each row uniquely — a minimal identifying attribute set under the model's constraints.

Read in context: 11.2.6

Candidate match

a pair worth investigating; technically, a proposed association that has not yet satisfied the final decision policy.

Read in context: 5.5.6

Candidate recall

how much required material reaches the later stage — coverage that limits what reranking can ultimately recover.

Read in context: 45.3.6

Candidate retrieval

finding possible answers — selecting documents for further scoring and verification under a query model.

Read in context: 44.1.6

Candidate row

a record not yet ruled out — input retained until the necessary conditions can be evaluated correctly.

Read in context: 43.4.6

Canonical ID

the receiving system's chosen reference; technically, an identifier used for the entity representation treated as the current reference under a stated mapping policy.

Read in context: 5.6.6

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

Capture boundary

the point at which this exercise obtains its input — the boundary after which one immutable byte string is used for parsing and diagnostics.

Read in context: 7.1.6

Cardinality

how many; here, the number of rows in a dataset or intermediate result.

how many related rows are permitted or required — the minimum and maximum counts on each side of a relationship.

how many records or matches there are — a count or relationship multiplicity whose context must be stated.

how many distinct labelled combinations exist — a property that can drive metric storage and query cost.

Read in context: 1.4.6, 9.3.6, 13.2.6, 50.2.6

Cardinality estimate

a predicted row count — the planner's expected output size for an operation under its available statistics and model.

Read in context: 18.2.6

CASE expression

an explicit choice among result alternatives — a SQL conditional expression selecting an output according to its stated conditions.

Read in context: 12.4.6

Case mix

the composition of the observed population — subgroup proportions that can influence aggregate comparisons.

Read in context: 46.2.6

Catalogue

organized metadata about data assets — a service or representation locating tables and describing their definitions and versions.

Read in context: 42.3.6

Catalogue price

the current listed price in the product model — a mutable reference value that must not silently replace historical agreement values.

Read in context: 9.6.6

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 order

one event preceding another through a dependency — the order induced by the model's local actions and communications.

Read in context: 6.3.6

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

Censored observation

a boundary rather than an exact value; technically, partial information restricting a value to a range, such as an unobserved completion time known to exceed a deadline.

completion time is only partly known — an observation limited by a timeout or stopping rule rather than a measured final duration.

Read in context: 4.5.6, 20.2.6

Change data capture

following source mutations — obtaining inserts, updates and deletions for downstream processing.

Read in context: 40.3.6

CHECK constraint

a declared predicate on acceptable values — an engine-enforced condition whose NULL semantics must also be understood.

Read in context: 9.2.6

Check predicate

an expression used to reject prohibited row values — a condition evaluated under an engine's CHECK semantics.

Read in context: 11.1.6

Check-then-insert race

two requests act on the same observed absence — an unsafe gap between an existence query and an unprotected insertion decision.

Read in context: 22.4.6

Checker scope

the objects and properties actually examined — the boundary limiting what a successful diagnostic result establishes.

Read in context: 32.3.6

Checkpoint

how far processing has reached — a recorded boundary of incorporated source input.

a recorded starting boundary for recovery — coordinated data and metadata progress that limits required redo under an engine's protocol.

recorded safe progress — a recovery marker bound to preserved state and explicitly scoped effects.

Read in context: 6.4.6, 28.4.6, 40.5.6

Checkpoint pressure

current work caused by reaching that boundary — I/O and related overhead generated by checkpoint activity.

Read in context: 28.4.6

Checksum coverage

the bytes included in an integrity calculation — the exact range or file set a checksum can help verify.

Read in context: 32.2.6

Child row

a row carrying a reference — a record constrained to refer to an allowed parent under the declared semantics.

Read in context: 9.3.6

Choreography

participants react to events — an arrangement distributing workflow reactions while still requiring explicit correctness and observability.

Read in context: 38.5.6

Cleanup horizon

the boundary beyond which old state can be reclaimed — the visibility or retention limit constraining safe version removal.

Read in context: 25.3.6

Client-server database

an engine operating as a separate service — an arrangement in which clients exchange requests and results with a database server.

Read in context: 8.3.6

Closed-loop load

clients wait before issuing their next request — a workload in which completion controls later arrivals.

Read in context: 20.1.6

COALESCE

choosing the first supplied value — an expression returning the first non-null argument, not a recovery of missing truth.

Read in context: 12.4.6

Code point

an assigned position in the Unicode code space; technically, a numeric value used in the character model, not a guarantee of one visible glyph.

Read in context: 3.4.6

Coercion

automatic conversion; technically, a transformation between types that may preserve or change intended meaning.

converting a value into another representation or type — an engine or application behaviour that must be deliberate when it changes interpretation.

Read in context: 2.2.6, 9.2.6

Collision

different inputs share one hash result — equality of hash outputs without equality of original inputs.

Read in context: 17.2.6

Collision resistance

hard to deliberately find two matching fingerprints — computational difficulty of finding distinct inputs with equal digests under the algorithm's security assumptions.

Read in context: 17.4.6

Column

a named role within each row — an attribute position with declared type and associated meaning.

Read in context: 9.2.6

Column-oriented format

a layout that groups values by column within its storage structure — an organisation suited to some selective and analytical reads, not a universal performance guarantee.

Read in context: 7.5.6

Columnar storage

values grouped by field — a layout that can reduce irrelevant reads for analytical projections.

Read in context: 43.1.6

Combined standard uncertainty

the uncertainty after input effects are combined; technically, the standard uncertainty of the result obtained from its input uncertainty model, including relevant dependence.

Read in context: 4.6.6

Command

a request to do something — an intention evaluated against current state and applicable rules.

Read in context: 6.2.6

Commit

accepting the transaction; technically, the transaction-completion operation under the engine's documented configuration.

accepting the transaction's database changes — successful completion under the engine's stated persistence and visibility semantics.

accept a transaction's database changes — the completion operation whose acknowledgement has engine- and configuration-specific guarantees.

Read in context: 1.5.6, 8.4.6, 21.2.6

Commit eligibility

evidence satisfying a commitment rule — the conditions permitting a particular log position to become committed.

Read in context: 37.6.6

Commit-time failure

rejection when trying to complete a transaction — an error arising at the commit boundary rather than necessarily during an earlier statement.

Read in context: 11.5.6

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

Commutative operation

an operation whose result is unchanged by swapping order — a property that does not automatically extend to validation rules or external effects.

Read in context: 6.3.6

Compaction

merge and reorganize stored runs — background work that installs a new representation and discards only eligible obsolete state.

Read in context: 29.5.6

Compaction backlog

reorganization work waiting to be completed — pending maintenance that can increase read cost, space use or write stalls.

Read in context: 29.6.6

Compare-and-set

write only when the expectation still matches — an atomic conditional mutation based on an expected value, revision or generation.

Read in context: 25.5.6

Compatibility

the ability of specified participants to exchange usable data — a tested relationship between writer, reader and semantic contracts.

Read in context: 7.5.6

Compatibility window

the overlap period for contracts — the time during which specified old and new consumers must coexist.

Read in context: 49.1.6

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: 6.5.6, 21.6.6, 38.4.6

Complete scope

all identity dimensions needed to select the intended record — the tenant, branch and local key components required by the data model.

Read in context: 22.3.6

Completion order

the sequence in which effects finish — an order that may differ from publishing or delivery order.

Read in context: 39.4.6

Completion predicate

the exact rule for saying an action is done — a condition over declared tasks, evidence and exceptions.

Read in context: 48.6.6

Composite index

one index built from several fields — an access structure over tuples of expressions in a specified order.

Read in context: 15.3.6

Composite key

an identifier made from several parts; technically, a key consisting of more than one attribute.

several fields identifying a row together; technically, a key consisting of more than one attribute.

an identity made from several values together — a unique column combination whose scope is not supplied by any one component alone.

several fields identify one thing together — an identifying attribute set whose combined values are unique in its declared scope.

Read in context: 2.4.6, 5.3.6, 9.4.6, 13.2.6

Composite reference

a link made from more than one value — a tuple of child columns identifying a parent within the corresponding scope.

Read in context: 11.3.6

Composite uniqueness

a combination must be distinct — a constraint enforcing uniqueness over a tuple of columns.

Read in context: 22.4.6

Compression block

a unit compressed and decoded together — a boundary affecting storage efficiency and selective-read cost.

Read in context: 43.3.6

Concurrency

work that overlaps in time; technically, the execution of operations whose interleaving can affect their observed results.

Read in context: 1.3.6

Conditional write

change a record only when its requirement still holds — a write guarded by a predicate evaluated under the database's concurrency rules.

Read in context: 22.3.6

Confidence interval

a range from a sampling procedure — an interval method with stated repeated-sampling coverage under assumptions.

Read in context: 46.6.6

Conflict group

records claiming one identity with incompatible content — a set for which this batch policy refuses to choose a winner.

Read in context: 7.6.6

Conflict policy

what to do when a rule prevents a write — an explicit decision to reject, reconcile, return an equivalent result or apply a defined update.

Read in context: 22.4.6

Conflict resolution

decide what to retain after incompatible changes — a policy for rejection, reviewed replacement, safe merge or recomputation.

Read in context: 25.6.6

Confounder

a factor affecting the causal comparison — a variable related to both assignment/exposure and outcome in a relevant causal structure.

Read in context: 46.5.6

Confused deputy

a privileged helper misused through supplied instructions — an authority-bearing component acting for the wrong caller or scope.

Read in context: 47.4.6

Connection

a session through which an application interacts with an engine — a context carrying database access and potentially transactional state.

Read in context: 8.3.6

Connection context

identity-related state attached to database work — information whose establishment, reuse and clearing form part of the security boundary.

Read in context: 47.3.6

Connection pool

a managed set of reusable connections — a bounded resource-sharing mechanism, not unlimited server capacity.

Read in context: 8.3.6

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

Conservative pruning

skipping only what cannot match — a rule that may retain extra candidates but must not discard valid results.

Read in context: 43.5.6

Consistency

accepted states obey the specified rules — validity relative to actual constraints and invariants, not an assurance that every stored claim is true.

Read in context: 21.3.6

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 backup boundary

the state the recovery procedure can validly reconstruct — a snapshot or log-supported acquisition point with matching metadata.

Read in context: 31.2.6

Consistent cut

observations that fit one permitted source history — a boundary that avoids incompatible mixtures of concurrent states.

Read in context: 40.2.6

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

Constraint

a rule enforced at a defined database boundary — a declared condition whose violation prevents a relevant operation from succeeding under the engine's semantics.

Read in context: 11.1.6

Constraint validation

checking records against a declared rule — an operation whose data scope and timing must be distinguished from future enforcement.

Read in context: 11.6.6

Consultation date

when a source was checked — a historical marker that does not turn a moving page into an immutable specification.

Read in context: 52.6.6

Consumer

anything relying on the data contract — a reader, writer, job, export, API adapter or replay process.

Read in context: 49.1.6

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

Containment

limit further harm while investigating — an action that can precede full diagnosis or repair.

Read in context: 32.6.6

Content addressing

name bytes by a content-derived identifier — object identification based on a specified representation and digest scheme.

Read in context: 17.6.6

Content digest

a compact fingerprint of captured bytes — the result of applying a specified hash function to an exact byte sequence.

Read in context: 7.1.6

Contention

operations competing for a shared resource — overlapping demand that can require waiting, rejection or coordination.

Read in context: 8.6.6

Contiguous progress

every required earlier position is included — a stronger condition than merely having seen a large position number.

Read in context: 40.6.6

Contract

what participants can rely on; technically, specified inputs, outputs, invariants, errors and relevant guarantees.

retire the old path — removing obsolete fields, adapters or access after dependencies are checked.

Read in context: 1.6.6, 49.2.6

Contribution identity

which source fact a result uses — a stable scoped identifier for a counted or summed input contribution.

Read in context: 19.6.6

Control plane

the machinery choosing and configuring behaviour — coordination mechanisms such as role assignment and membership management.

Read in context: 34.1.6

Conversion factor

the multiplier between compatible units; technically, the ratio used to transform numerical values while preserving the represented quantity.

Read in context: 4.2.6

Coordination boundary

the interval or operation within which competing work is controlled — the scope of a lock, version condition or transactional mechanism.

the operations that must be ordered or made indivisible together — the scope within which an invariant's read, decision and write are protected.

Read in context: 8.1.6, 11.4.6

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

Correction

a change to repair a stated defect; technically, an authorised transition whose scope, evidence and consequences should be recorded.

Read in context: 2.6.6

Correction target

the particular entry being repaired — an explicit reference needed to validate and explain a correction.

Read in context: 6.5.6

Correctness oracle

a separate expected-answer check — a mechanism that rejects a fast but semantically wrong variant.

Read in context: 20.5.6

Correlated failure

one cause affects several copies — a failure pattern that invalidates an assumption of independent redundancy.

Read in context: 30.5.6

Correlation

values vary together — a relationship that can invalidate an assumption that conditions are independent.

Read in context: 18.2.6

Cosine similarity

directional agreement — inner product divided by the product of nonzero vector norms.

Read in context: 45.2.6

Cost boundary

which charges are counted — the stated inclusions and exclusions behind a reported amount.

Read in context: 50.5.6

Counterbalancing

vary which option runs first — an experimental ordering technique intended to reduce systematic sequence effects.

Read in context: 20.4.6

Counterexample

a concrete case that disproves an overbroad statement — an input or sequence for which a claimed invariant or interpretation does not hold.

Read in context: 10.6.6

Covariance

how two quantities vary together; technically, a measure of joint variation that contributes to propagation when input estimates are dependent.

Read in context: 4.6.6

Covering index

the guide carries the values this question needs — an index containing sufficient query attributes for an eligible index-only or covering access path.

Read in context: 15.4.6

Crash trace

the ordered evidence surrounding an interruption — a record of persisted state, surviving log material and recovery decisions.

Read in context: 28.6.6

Cross-reference

a route to related material — a stable chapter, section or source identifier rather than a guessed page number.

Read in context: 52.1.6

Cross-row invariant

a rule about several records together — a condition involving relationships, aggregates or coordinated state beyond one proposed row.

a rule about several records together — a correctness condition involving a set of rows rather than one row's fields.

Read in context: 11.4.6, 22.5.6

Crosswalk

a mapping between identifier systems; technically, a relation connecting scoped source identifiers with identifiers in another model.

Read in context: 5.6.6

Cryptographic erasure

making protected ciphertext unusable by sanitising necessary key material — a claim dependent on key scope and remaining copies.

Read in context: 48.5.6

CSV

a text representation of tabular records — comma-separated values with a specified dialect and quoting rules.

Read in context: 7.3.6

Current attribute

what a record says about the present state — a value whose intended temporal interpretation follows the entity's current representation.

Read in context: 10.3.6

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

Cursor

the driver's interface to a statement and its results — an object through which rows and relevant execution information are retrieved.

Read in context: 12.1.6

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, 49.4.6

D

Data clustering

placing similar values near one another — a layout choice that can tighten summaries and improve selective access.

Read in context: 43.5.6

Data contract

the agreement about incoming information — explicit structure, semantics, identity and compatibility rules for a data interface.

Read in context: 40.1.6

Data dictionary

the reference explaining what fields mean — maintained metadata describing grain, domains, units, source, lifecycle and interpretation.

Read in context: 10.5.6

Data lineage

how outputs depend on inputs — a recorded graph of source versions, transformations and derived assets.

Read in context: 42.5.6

Data minimisation

avoiding unnecessary stored detail — collecting and retaining only the information justified for the defined purpose.

keeping only what the task needs — reducing unnecessary fields, copies or lifetime under a stated requirement.

Read in context: 6.6.6, 48.1.6

Data minimization

limiting information to a justified purpose — a design practice reducing unnecessary collection, copying and exposure.

Read in context: 42.5.6

Data plane

where requests produce actual effects — the operational path that must enforce the control plane's decisions.

Read in context: 34.1.6

Database

the organised records; technically, a managed collection of data interpreted under a model and its rules.

Read in context: 1.1.6

Database engine

the software managing stored data operations — the component that executes supported queries, changes and integrity mechanisms.

Read in context: 8.2.6

Database page

a unit in the engine's physical layout — a fixed-size storage block containing records, index entries or other engine structures.

Read in context: 27.1.6

Date

a calendar position; technically, a calendar day without necessarily specifying a time of day or an instant.

Read in context: 3.5.6

DBMS

the record-management machinery; technically, the engine providing database operations and its documented guarantees.

the database-management software — facilities for defining, accessing and administering databases.

Read in context: 1.1.6, 8.2.6

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

Deadlock

a cycle of participants waiting on one another — a condition requiring some participant to release, abort or otherwise break the cycle.

Read in context: 24.3.6

Deadlock victim

the transaction selected to break a cycle — a participant aborted by the engine so other work can proceed.

Read in context: 24.4.6

Decimal arithmetic

arithmetic based on decimal representation; technically, operations over decimal values under a precision and rounding context.

Read in context: 3.3.6

Decision record

retained evidence of commit or abort — information needed to resolve participants consistently after interruption.

Read in context: 38.2.6

Declarative query

a request that describes the required result — a statement whose logical meaning does not prescribe one physical execution plan.

Read in context: 12.1.6

Decode cost

work to reconstruct usable values — CPU and memory effort introduced by an encoding or compression layer.

Read in context: 43.6.6

Decoding

turning bytes into characters — interpreting a byte sequence under a specified character encoding.

Read in context: 7.2.6

Deduplication

recognising repeated representations of the same thing; technically, a policy-driven process whose identity assumptions and conflict handling must be explicit.

recognising a repeated identity; technically, detecting and handling duplicate deliveries or records under an explicitly defined equivalence and retention policy.

Read in context: 2.4.6, 5.4.6

Deduplication horizon

how long repeated intent remains recognizable — the retention interval and protocol conditions under which replay records are available.

Read in context: 26.3.6

Deduplication key

what identifies one contribution — the scoped fields used to recognise a replay rather than a separate event.

Read in context: 19.5.6

Deferred constraint

a rule whose check is postponed to a declared later boundary — an eligible constraint allowed to be temporarily unsatisfied within a transaction but required by successful final checking.

Read in context: 11.5.6

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

DELETE

removing selected stored rows — a SQL operation whose effects are constrained by referential rules and transaction boundaries.

Read in context: 12.5.6

Delimiter

the mark used to separate fields — a structural character interpreted according to the parser's current state.

Read in context: 7.3.6

Delivery attempt

one effort to transport a message; technically, a transmission attempt that may repeat the same logical event.

Read in context: 5.4.6

Delivery gap

uncertainty between an external action and its recorded acknowledgement — a failure window that can require replay or reconciliation.

Read in context: 24.6.6

Delta encoding

storing changes from earlier values — a representation whose savings depend on the size and regularity of differences.

Read in context: 43.2.6

Denominator

what the division is over; technically, the quantity that defines the base of a ratio or average.

what the total is divided by — the explicitly defined count or weight underlying a ratio.

the population a rate is out of — the eligible count or exposure giving a numerator its meaning.

Read in context: 3.6.6, 19.2.6, 46.2.6

Denormalisation

deliberately keeping an extra representation — controlled redundancy introduced for stated requirements with explicit maintenance obligations.

Read in context: 13.5.6

Derived data

an answer made from other data; technically, a representation produced by a transformation or calculation over inputs.

Read in context: 1.4.6

Deserialisation

rebuilding values from a stored representation — decoding and interpreting data into runtime objects under a specific format.

Read in context: 7.4.6

Design trade-off

a choice that improves some requirements at a cost to others — an explicit comparison of behaviour and responsibility under stated assumptions.

Read in context: 8.6.6

Destination capacity rule

an extra bound imposed by this storage adapter — a limitation distinct from parsing and the earlier domain contract.

Read in context: 9.2.6

Detection versus prevention

finding a violation versus stopping it from being committed — distinct responsibilities requiring different evidence.

Read in context: 11.4.6

Deterministic replay

the same retained inputs and rules give the same state — reconstruction without uncontrolled dependencies on current external values.

Read in context: 6.4.6

Deterministic result order

a sequence fixed by the stated query and data assumptions — an ordering whose relevant ties are resolved under chosen comparison rules.

Read in context: 12.3.6

Deterministic transformation

the same complete inputs yield the same output — a rule without uncontrolled time, randomness or external-state dependencies.

Read in context: 40.4.6

Dialect

the particular set of parsing conventions — settings governing delimiters, quotation marks, escaping and related behaviour.

Read in context: 7.3.6

Dictionary encoding

replacing values with reusable codes — a representation containing a distinct-value dictionary and an occurrence-code sequence.

Read in context: 43.2.6

Digest

a compact fingerprint of bytes — a fixed-length output of a hash algorithm, whose meaning depends on the exact input bytes and algorithm.

Read in context: 17.1.6

Dimension

the physical kind of a quantity; technically, its expression in powers of the chosen base quantities, such as length or time.

descriptive context for grouping facts — attributes connected to facts under a current or historical interpretation.

Read in context: 4.2.6, 42.2.6

Dimensional analysis

checking a calculation through its units; technically, checking compatibility of dimensions across an equation while recognising that semantics require further checks.

Read in context: 4.2.6

Directory durability

preserve the file's naming relationship — persistence of directory metadata needed to find the intended file after failure.

Read in context: 30.3.6

Dirty page

a changed cached page awaiting corresponding writeback — a buffer whose managed contents differ from its lower stored representation.

Read in context: 27.4.6

Dirty read

seeing a change before it commits — observation of another transaction's uncommitted data.

Read in context: 23.1.6

Disagreement

two observations or expectations do not match — a finding that requires diagnosis rather than automatically proving storage damage.

Read in context: 32.1.6

Dispatcher

the worker attempting recorded external tasks — a process that reads delivery intent and manages sends and acknowledgements.

Read in context: 26.4.6

Display name

the label shown to a person; technically, a presentation attribute that need not be unique or stable.

Read in context: 5.1.6

DISTINCT

removing repeated output rows — a SQL operation over selected result values, not automatic repair of a data model.

Read in context: 9.1.6

Distinct count

count different values under a rule — cardinality of the included value set after the specified equality and NULL treatment.

Read in context: 19.5.6

Divergent history

copies contain incompatible continuations — a situation requiring authority and recovery analysis rather than simple row copying.

Read in context: 34.6.6

Document

the searchable unit — an identified body of content, potentially a whole page or a bounded passage.

Read in context: 44.1.6

Document frequency

how many documents contain a term — a collection statistic used to distinguish common from rarer terms.

Read in context: 44.4.6

Domain

the allowed kind of value; technically, the set of permitted values and associated meaning for an attribute.

the values and meaning permitted for a role — a business-level contract potentially narrower than a storage type.

Read in context: 2.1.6, 9.2.6

Domain type

the business meaning of the value; technically, a model-specific type whose rules may exceed the underlying storage type.

Read in context: 3.1.6

Downstream impact

outputs and decisions depending on disputed data — the propagation scope that must be reviewed after a correction.

Read in context: 32.6.6

Draft record

unfinished work that remains open to revision — a lifecycle-specific representation with explicit incompleteness rules and no implied finalisation.

Read in context: 10.3.6

Dual-write gap

one system changes while another does not — a failure window between separately committed operations in different resources.

Read in context: 26.4.6

Duplicate member

a name repeated within one object — an interoperability hazard rejected by this book's input profile.

Read in context: 7.4.6

Durability

surviving specified failures; technically, the persistence guarantee for accepted changes under stated assumptions.

the promised survival of committed work — persistence under defined failures, engine configuration and storage assumptions.

acknowledged work survives specified failures — persistence under a named configuration, recovery model and failure boundary.

Read in context: 1.5.6, 8.4.6, 21.4.6

Durability contract

what accepted data is promised to survive — a guarantee defined against specified failures and storage assumptions.

Read in context: 30.1.6

Durability evidence

observations supporting a bounded persistence claim — a test record tied to configuration, fault model and verification procedure.

Read in context: 30.6.6

Durable prefix

the surviving accepted beginning of a log — the valid ordered region known to have reached the required persistence boundary.

Read in context: 28.3.6

Duration

elapsed amount of time; technically, a measured or specified interval whose units and arithmetic semantics must be stated.

Read in context: 3.5.6

E**Edit distance**

the number of selected character edits between strings — a spelling similarity measure, not a proof of intended identity.

Read in context: 44.5.6

Edit precondition

the state an update expects to replace — a condition checked at the write boundary to detect intervening changes.

Read in context: 25.2.6

Effect count

how many accepted changes occurred — distinct from how many attempts or deliveries were observed.

Read in context: 51.4.6

Effective privilege

what the identity can actually do — the result of direct grants, memberships, ownership and special execution rules.

Read in context: 47.2.6

Elapsed time

how long the chosen boundary took on a clock — wall-clock duration including waits within that boundary.

Read in context: 14.6.6

Election

selecting a coordinator under protocol rules — an authority decision with explicit membership and eligibility conditions.

Read in context: 34.3.6

Eligible access path

a route that can implement the required semantics — an index or scan option permitted by the query, operators and engine rules.

Read in context: 15.6.6

Eligible set

items permitted by the request's complete conditions — the population within which retrieval should answer the question.

Read in context: 45.5.6

Embedded database

an engine included in an application process — database functionality supplied through library calls rather than a separate server process.

Read in context: 8.3.6

Embedding

an item's numerical representation — an ordered vector produced by a defined encoder and preprocessing pipeline.

Read in context: 45.1.6

Embedding space

the coordinate system used for comparison — a representation whose geometry depends on its model and training objective.

Read in context: 45.1.6

Empty string

text containing no characters; technically, a zero-length string, distinct from SQL NULL in the engines discussed here.

Read in context: 3.6.6

Encoding

the mapping into bytes; technically, a defined representation of a character sequence for storage or interchange.

Read in context: 3.4.6

End-to-end boundary

the complete chain whose result is claimed — all participating systems and effects required for a stated business outcome.

Read in context: 26.6.6

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

End-to-end measurement

timing the complete defined operation — evidence that includes all stages required to produce and consume the intended result.

Read in context: 43.6.6

Enforcement gap

a declared rule without sufficient protection — a condition the implementation can violate despite the intended policy.

Read in context: 10.6.6

Entity

the thing the model is about; technically, a distinguishable object, concept or occurrence represented within a domain model.

Read in context: 5.1.6

Entity resolution

deciding which records concern the same thing; technically, inference over records that may lack a shared reliable identifier.

Read in context: 5.5.6

Envelope

the wrapper around a payload — metadata and framing interpreted before the enclosed message.

Read in context: 7.5.6

Epoch

a generation of authority or configuration — a value distinguishing successive valid ownership periods under a defined protocol.

Read in context: 34.4.6

Equality contract

the rule for deciding that two keys are the same — a domain-specific equivalence relation that hashing and comparison must respect.

Read in context: 17.3.6

Error budget

the allowed nonconformance under an objective — an arithmetic planning quantity whose definition does not override other requirements.

Read in context: 50.1.6

Estimand

the exact quantity a study seeks — the defined performance or outcome measure to which observations are intended to speak.

the precise quantity a study seeks — a defined population-level contrast under specified interventions or conditions.

Read in context: 20.5.6, 46.5.6

Euclidean distance

straight-line separation — the square root of summed squared coordinate differences.

Read in context: 45.2.6

Event

a record of something relevant happening — an occurrence interpreted under a stated event schema.

Read in context: 6.1.6

Event identity

the name of one accepted occurrence or change; technically, the identifier combination that distinguishes an event under the producer's contract.

Read in context: 5.4.6

Event sourcing

keeping the accepted event history as the principal record — an architecture that reconstructs state from authoritative events.

Read in context: 6.1.6

Event time

when the described occurrence happened; technically, the timestamp assigned to the event under its source's semantics.

when the described event is said to have happened — a timestamp assigned under the source and pipeline's semantic contract.

Read in context: 2.5.6, 41.2.6

Eviction

remove a cached item to make room — replacement subject to usage, dirty-state and recovery constraints.

Read in context: 27.2.6

Evidence

what supports a claim; technically, information whose relevance and limitations must be assessed for that claim.

Read in context: 1.2.6

Evidence boundary

the limit of a justified conclusion — the tested scope, assumptions and unresolved properties attached to a result.

Read in context: 51.6.6

Evidence gap

something the records cannot establish — missing information or verification needed to support a claim about the world.

Read in context: 10.6.6

Evidence-preserving repair

fix a problem without concealing its original state — a procedure retaining provenance and disclosing reconstructed or lost information.

Read in context: 32.5.6

Exclusion

a deliberately unimplemented responsibility — an explicit limit such as payment settlement or physical dispatch.

Read in context: 51.1.6

Exclusion register

explain what did not contribute — recorded counts and reasons for records outside the report's accepted population.

Read in context: 19.6.6

Executable assertion

a machine-checkable expectation — a predicate comparing an observed result with a declared outcome.

Read in context: 51.6.6

Execution plan

the chosen route to the answer — a structured set of access, join, calculation and ordering operations used by the executor.

Read in context: 18.1.6

Existence precheck

asking whether a record is present before attempting a write — a read that does not by itself prevent a concurrent conflicting insert.

Read in context: 11.2.6

Existence test

asking whether at least one match is present — a condition that need not reproduce every matching row in the output.

Read in context: 9.4.6

Expand

add a compatible new path — introducing representation or capability before removing the old one.

Read in context: 49.2.6

Expanded uncertainty

a stated multiple of standard uncertainty; technically, $U = k \times u_{\text{combined}}$, accompanied by the coverage factor and justified interpretation.

Read in context: 4.6.6

Expected complexity

cost under specified distribution assumptions — an average or probabilistic bound, not a guarantee for every input.

Read in context: 17.3.6

Expected version

the history position a request relies on — a concurrency precondition used to detect stale writes.

Read in context: 6.2.6

Extended error code

a machine-readable refinement of a failure category — an implementation-specific identifier that distinguishes more precise error causes.

Read in context: 11.6.6

External effect

something changes beyond the local transaction — an action in another system or the physical world not automatically reversed by database rollback.

a change outside the local transaction — an email, print, remote API action or other operation not undone by ordinary database rollback.

Read in context: 21.6.6, 24.6.6

External outcome

an event beyond the local transaction — payment, delivery or another result needing separate evidence.

Read in context: 51.4.6

External validity

whether a result transfers beyond its test — the justified scope of applying observations to other workloads, environments or populations.

Read in context: 20.6.6

Extraction boundary

exactly which source state or records were selected — the snapshot, interval or immutable object defining a batch's coverage.

Read in context: 40.2.6

F**Fact table**

measurable observations at a declared grain — analytical records whose keys and units define what may be aggregated.

Read in context: 42.2.6

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: 21.4.6, 30.5.6, 33.1.6

Failure injection

deliberately create a specified fault — a controlled experiment against an authorized disposable system.

Read in context: 30.6.6

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 positive

a match accepted when the entities differ; technically, a positive decision for a negatively labelled case in the stated evaluation.

a possible match that is not actually present — a positive candidate result rejected by a later exact check.

Read in context: 5.5.6, 17.6.6

False suspicion

a healthy participant is treated as failed — a classification possible under delays, partitions or process pauses.

Read in context: 34.2.6

Fan-out

one source row producing several matched result rows — multiplicity introduced by a one-to-many match in a query.

one record becomes several joined contributions — multiplication of rows through a one-to-many or many-to-many relationship.

how many next regions one guide can select — the number of child references in an internal tree node.

Read in context: 9.4.6, 13.6.6, 16.2.6

Fault injection

deliberately interrupting a known path — a test technique whose simulated failure must be named precisely.

Read in context: 51.5.6

Fault model

the failures an experiment assumes or injects — the specific process, operating-system, device or communication conditions being tested.

the failures considered — the boundary within which a mechanism's reasoning and tests are intended to apply.

Read in context: 28.6.6, 52.4.6

Fencing

preventing obsolete authority from acting — enforcement at a protected resource or infrastructure boundary that rejects stale owners.

Read in context: 34.4.6

Field

one named part of a record; technically, an attribute position with an agreed interpretation.

Read in context: 2.1.6

Field ID

stable column identity independent of a display name — metadata used to preserve meaning through supported schema changes.

Read in context: 42.4.6

File extension

the familiar suffix on a name — a naming convention that does not independently validate the represented format.

Read in context: 7.1.6

File format

how bytes are arranged; technically, the syntax and interpretation rules for a representation.

Read in context: 1.3.6

File synchronization

request persistence of pending file state — an operating-system operation with documented data, metadata and completion semantics.

Read in context: 30.3.6

File-level error

a failure affecting interpretation of the input as a whole — an outcome that withholds publication when trustworthy complete parsing is unavailable.

Read in context: 7.6.6

Filter false positive

a filter suggests a candidate that is absent — extra verification work without a false final membership claim.

Read in context: 29.4.6

Flattened export

related information copied into a single rectangular result — a representation that may repeat parent attributes at a finer output grain.

Read in context: 10.2.6

Floating point

a number representation with a movable scale; technically, a finite significand and exponent representation with defined rounding behaviour.

Read in context: 3.3.6

Flush

push pending changes toward a specified lower layer — an operation whose persistence guarantee depends on the API and storage contract.

Read in context: 27.4.6

Forecast error

the difference between prediction and observation — evidence used to update assumptions rather than hide uncertainty.

Read in context: 50.6.6

Foreign key

a checked reference to another stored key; technically, a constraint requiring referencing values to correspond to an eligible referenced key, subject to the database's rules.

a declared reference to another allowed key — a referential-integrity constraint linking child values to parent values.

a stored link required to match an eligible parent — a referential constraint on child values with defined missingness, actions and checking time.

Read in context: 5.3.6, 9.3.6, 11.3.6

Fragmentation

useful space is distributed inconveniently — a layout condition whose performance effect depends on the engine and access pattern.

Read in context: 27.5.6

Framing

marking where messages or records begin and end — a boundary convention independent of the meaning of each payload.

make field boundaries unambiguous — an encoding convention that preserves where components begin, end and differ in type or length.

Read in context: 7.2.6, 17.1.6

Freelist

storage recorded as available for reuse — an engine-managed collection of free pages or blocks.

Read in context: 27.6.6

Frequent value

one value appears in many records — a high-frequency member of a distribution relevant to selectivity estimation.

Read in context: 15.5.6

Freshness objective

how current a result must be — an explicit bound connecting publication to its required source state.

Read in context: 42.6.6

Full-transaction retry

repeat the decision from fresh protected reads — a new attempt that does not reuse an invalidated transaction's conclusions blindly.

Read in context: 23.6.6

Functional dependency

one part determines another; technically, a constraint under which equal determining attributes imply equal dependent attributes.

one set of fields fixes another under a rule — $X \rightarrow Y$ means equality on X implies equality on Y in every legal relation instance.

Read in context: 2.3.6, 13.4.6

G

Generation identifier

distinguish separate lifetimes of a reused key — an immutable identity component that changes when a resource is recreated.

Read in context: 25.4.6

Global invariant

a rule spanning several owners — a correctness condition not established by independent local checks alone.

Read in context: 35.5.6

Glossary

a vocabulary reference — definitions connected to their contexts and teaching locations.

Read in context: 52.1.6

Grain

what one record stands for; technically, the observation or business unit represented by each row.

what one row represents; technically, the unit of observation or business fact represented at the dataset's chosen level.

Read in context: 1.4.6, 2.3.6

Grain contract

a precise statement of what one row represents — documentation tying a record's unit, key, multiplicity and temporal meaning together.

Read in context: 10.2.6

Grapheme cluster

a unit closer to what a reader perceives as a character; technically, a text-segmentation unit defined by applicable Unicode rules and possible tailoring.

Read in context: 3.4.6

Gross mass

goods plus the included packaging; technically, mass under a definition that includes the stated container and packing materials.

Read in context: 4.1.6

Grounded answer

a response supported by inspected evidence — claims linked to sources that actually establish them, with limitations retained.

Read in context: 45.6.6

Group commit

one synchronization supports several commits — batching log persistence while meeting each transaction's selected acknowledgement condition.

Read in context: 28.2.6

Grouping key

the fields defining each collection — expressions whose equal grouping values identify one aggregate group.

Read in context: 19.1.6

Guarded update

a mutation conditional on identity and expected state — a write whose predicate tests the specific prior condition required by the operation.

Read in context: 12.6.6

H**Half-even**

choosing the even final digit at a tie; technically, rounding halfway cases to the nearest value whose retained least significant digit is even.

Read in context: 4.4.6

Half-open interval

include the start but not the end — an interval $[a,b)$ useful for adjoining non-overlapping ranges.

Read in context: 16.5.6

Half-specified reference

only part of a multi-component link is supplied — an incomplete child tuple that needs an explicit acceptance policy.

Read in context: 11.3.6

Harness overhead

work caused by the measurement setup — time and resources consumed by generation, instrumentation and reporting rather than the target operation alone.

Read in context: 20.5.6

Hash function

a rule making an organising or checking value — a deterministic mapping under specified parameters and input representation.

Read in context: 17.1.6

Hash join

match through a temporary hashed key structure — a build-and-probe strategy that still checks equality and preserves required duplicates.

Read in context: 18.3.6

Hash 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

Hash partitioning

choose a data region from a key's hash — routing through a versioned mapping from derived values to partitions.

Read in context: 17.6.6

Headroom

usable spare capacity — margin under stated workload and failure assumptions, not a generic machine percentage.

Read in context: 50.4.6

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

Histogram

a summary of how values are distributed — a bucketed representation used for approximate frequency or range reasoning.

Read in context: 18.2.6

Historical recovery point

a selected earlier usable state — a state reconstructible from compatible retained recovery material.

Read in context: 33.6.6

Historical snapshot

a saved description of an earlier state — data retained under an explicit capture time, purpose and correction policy.

Read in context: 13.1.6

Historical snapshot attribute

a value copied to describe an earlier event or agreement — a stored attribute with event-specific temporal semantics, not automatic immutability.

Read in context: 10.3.6

Historical value

a value tied to an earlier event or agreement; technically, information whose interpretation must not be replaced by an unrelated current-state value.

Read in context: 2.3.6

HNSW

hierarchical neighbour-graph search — an approximate indexing approach navigating graphs at several levels of detail.

Read in context: 45.4.6

Hold

an explicit suspension of an action — a governed exception with reason, authority and review conditions.

Read in context: 48.2.6

Hot partition

one piece receives disproportionate work — a localized throughput, latency or resource bottleneck.

Read in context: 35.3.6

Hybrid retrieval

combining different candidate methods — often lexical and vector search followed by merging or reranking.

Read in context: 45.6.6

Hypothesis

a candidate explanation — a claim that should be distinguished from alternatives through suitable evidence.

Read in context: 52.3.6

Hypothesis-driven tuning

change what evidence suggests — performance investigation linking an explicit suspected cause to a controlled test.

Read in context: 18.6.6

I

Idempotency

repeating the operation without repeating its intended effect; technically, stability of the declared effect under repeated application within the operation's identity and failure model.

Read in context: 5.4.6

Idempotency scope

where a replay identity is meaningful — the tenant, operation and retention boundary governing duplicate handling.

Read in context: 51.3.6

Identifier

a reference to a particular thing; technically, a value or tuple interpreted within a defined namespace and lifecycle.

a value used to refer to something; technically, a designation interpreted within an issuing namespace and its lifecycle rules.

Read in context: 2.4.6, 5.1.6

Identity boundary

the scope inside which a label identifies something — the domain over which a key or composite key is interpreted and uniqueness is enforced.

Read in context: 10.2.6

Identity conflict

one ID attached to incompatible content; technically, a violation of the mapping from an identifier to the immutable or otherwise controlled claim it is supposed to name.

Read in context: 5.4.6

Identity history

the record of how associations changed; technically, retained versions or events describing identifier mappings and their applicable decision boundaries.

Read in context: 5.6.6

Identity scope

the domain in which a key is unique — the tenant, operation and other components defining a request key's namespace.

Read in context: 26.2.6

Idle in transaction

a transaction remains open without active useful work — a state that can retain locks and snapshots while the application waits elsewhere.

Read in context: 24.2.6

Idle input

a source currently producing no records — a condition requiring a policy, not proof that earlier events can never return.

Read in context: 41.4.6

Immediate constraint

a rule checked at its ordinary early boundary — an invariant enforced without the supported transaction-level postponement.

Read in context: 11.5.6

Immutable table file

a sorted file replaced rather than updated in place — an installed run whose ordinary contents do not change after creation.

Read in context: 29.2.6

Implementation boundary

where a claim stops generalising — the product, version and configuration for which a detail is established.

Read in context: 52.1.6

Implementation detail

how a particular system works; technically, behaviour that must not be assumed to apply to every conforming or competing system.

Read in context: 1.6.6

Import profile

the exact agreement for an exchange — a bounded specification combining encoding, syntax, fields and application rules.

Read in context: 7.3.6

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

the consumer's delivery record — an identity committed with the local effect to support repeat handling.

Read in context: 51.3.6

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: 26.5.6, 39.3.6

Included column

payload carried alongside search keys — a non-key attribute stored in an index by an engine supporting that feature.

Read in context: 15.4.6

Inclusive interval

a range containing its endpoints — the boundary interpretation used by BETWEEN in the illustrated SQL conditions.

Read in context: 12.2.6

Incompatible key reuse

one identity is presented with different meaning — a conflict requiring rejection or explicit reconciliation, not silent replacement.

Read in context: 26.3.6

Incremental extraction

reading changes since recorded progress — a method whose cursor must account for the source's actual change semantics.

Read in context: 40.2.6

Incremental maintenance

updating a result from a change — applying a justified delta instead of recomputing everything.

Read in context: 48.4.6

Indeterminate outcome

the caller cannot yet tell whether it completed — uncertainty commonly caused by losing communication around a commit or external effect.

Read in context: 21.2.6

Index

an extra route to selected records — a maintained data structure supporting specified search, ordering or constraint operations.

Read in context: 15.1.6

Index maintenance

keeping the extra route correct — updates, cleanup and related work required as the underlying data changes.

Read in context: 15.2.6

Indication

what the instrument shows; technically, the quantity value supplied by a measuring instrument or system.

Read in context: 4.1.6

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

Information leakage

using knowledge unavailable at the intended decision point — contamination that can make evaluation results unrealistically strong.

Read in context: 46.4.6

Information need

what the reader is trying to learn — the task that a query attempts to express.

Read in context: 44.1.6

Ingestion time

when a system accepted the input — a separately useful observation that does not replace the original event time.

Read in context: 41.2.6

Inner join

returning matching row combinations — a join that omits rows without a match in the joined relation.

keep successful pairings — a join that contributes only row combinations satisfying the join condition.

Read in context: 9.5.6, 13.3.6

Inner product

coordinate products added together — a similarity score affected by stored-vector magnitudes as well as direction.

Read in context: 45.2.6

INSERT

adding records — a SQL write operation that proposes new rows under the target table's active rules.

Read in context: 12.5.6

Instant

one point on a timeline; technically, a temporal position distinguishable from its local display representation.

Read in context: 3.5.6

Instrumentation

measurements added to execution — observation code collecting timings, counts or resource evidence, potentially with its own overhead.

Read in context: 18.5.6

Integer

a whole number; technically, a value in the set of signed whole numbers, represented with implementation-specific limits.

Read in context: 3.2.6

Integer division

division evaluated using integer operands and rules — an operation that can discard the fractional component rather than produce a decimal quotient.

Read in context: 12.4.6

Integrity check

a check that retained content agrees with a reference — evidence of consistency, not automatic evidence of real-world truth.

inspection of declared structural properties — one layer of validation, distinct from complete business or security correctness.

Read in context: 6.6.6, 51.5.6

Interface

how one component asks another to do something; technically, the exposed operations and their input, output and error contracts.

Read in context: 1.3.6

Interleaving

the steps of different requests mixed together — an operation ordering that preserves each participant's required local order.

Read in context: 22.1.6

Intermediate state

a visible stage before final completion — a condition that a saga-based application must interpret correctly.

Read in context: 38.6.6

Intervention threshold

the point to act before exhaustion — a chosen policy leaving time and resources for a controlled response.

Read in context: 50.3.6

Invalidation

mark a cached result unusable — a signal or state transition preventing further eligible reuse.

Read in context: 39.2.6

Invariant

a rule that must keep holding; technically, a predicate required of accepted states or permitted transitions.

a rule that must remain true — a property the chosen model preserves across accepted operations.

a condition the system is meant to preserve — a stated property that every accepted state transition must maintain.

a condition that must keep holding — a predicate preserved across the operations within its specified scope.

a rule every accepted state must preserve — a condition maintained by constraints and operation protocols across permitted transitions.

a rule accepted operations must preserve — a predicate over the relevant system state that the protocol is designed to maintain.

Read in context: 1.1.6, 6.2.6, 8.1.6, 11.4.6, 21.3.6, 26.1.6

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

Inverted index

a word-to-document map — an access structure locating documents containing searchable terms.

Read in context: 44.3.6

Irreversible transformation

one that loses needed distinctions — a mapping whose original inputs cannot be recovered from the retained output alone.

Read in context: 49.5.6

IS NULL

a direct test for missingness — a SQL predicate that checks whether an expression yields the NULL marker.

Read in context: 12.2.6

Isolation

rules for concurrent work seeing and affecting data — the semantics governing interaction between overlapping transactions.

rules for overlapping work — guarantees governing visibility and interaction among concurrent transactions.

Read in context: 8.4.6, 21.4.6

Isolation level

the chosen rules for overlapping transactions — a named contract restricting visibility and permitted concurrency anomalies.

Read in context: 23.1.6

IVF index

search through selected coarse groups — an inverted-file organization assigning vectors to regions before candidate evaluation.

Read in context: 45.4.6

J

Jitter

deliberately vary the waiting time — bounded randomness used to avoid synchronized retries.

Read in context: 24.5.6

Join

combining rows through a stated condition — an operation whose result can contain multiple matches for one source row.

Read in context: 9.4.6

Join predicate

the rule for pairing records — a Boolean condition controlling which row combinations contribute to a joined relation.

Read in context: 13.2.6

JSON array

a sequence of values — an ordered collection whose elements may themselves be structured.

Read in context: 7.4.6

JSON object

a group of named values — a collection of name/value members within a JSON representation.

Read in context: 7.4.6

Junction table

a table of links — an associative relation whose rows represent a defined pairing or richer relationship.

Read in context: 9.4.6

K

Key collision

two attempted records using the same constrained identity — a uniqueness conflict whose business interpretation requires the surrounding policy.

Read in context: 11.2.6

Key conservation

no entry disappears or is invented — equality between the intended inserted entries and those represented after restructuring.

Read in context: 16.4.6

Key dependency

recovery requires particular cryptographic material — an access and availability requirement for decrypting retained data.

Read in context: 31.6.6

Keyset progress

resuming from a stable ordered key — a traversal technique whose eligibility and concurrent-change boundaries still need design.

Read in context: 49.3.6

Known subtotal

a total over available values only — a partial calculation that must not be labelled complete when required observations are missing.

Read in context: 19.2.6

L

Last-write-wins policy

accept the selected later replacement — a conflict rule that can discard competing state and requires a defined ordering basis.

Read in context: 25.6.6

Latch

short-lived protection of an internal structure — a synchronisation mechanism distinct from a transaction's logical locking policy.

Read in context: 16.6.6

Late arrival

a record received after its event period — an observation whose processing time occurs later than the event-time window it describes.

Read in context: 19.4.6

Late data

a record behind the accepted progress frontier — input whose event time precedes the relevant watermark or closed-window boundary.

Read in context: 41.4.6

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

Leading key

the first part of an ordered index key — the prefix that groups entries before later components are compared.

Read in context: 15.3.6

Leaf

the bottom searchable page — a node containing data entries rather than further child pages in this teaching B+ tree.

Read in context: 16.1.6

Least privilege

only the authority needed — minimising effective permissions and their duration for a defined responsibility.

Read in context: 47.2.6

Left join

preserving the left input when no right match exists — an outer join that supplies NULL right-side values for unmatched left rows.

Read in context: 9.5.6

Left outer join

keep every left record, even without a match — a join adding a null-extended row for each unmatched left input row.

Read in context: 13.3.6

Length prefix

a size written before the content — a field giving the amount of following data in a stated unit.

Read in context: 7.2.6

Lexeme

a normalized searchable form — a dictionary or analyzer output that may combine several written word forms.

Read in context: 44.2.6

Lexicographic order

compare the first difference — tuple ordering determined by the earliest component on which two tuples differ.

Read in context: 15.3.6

LIMIT

a cap on rows returned by a query — an output selection clause that does not by itself bound all execution work.

Read in context: 12.3.6

Lineage

how one result came from other data; technically, the dependency path between inputs, transformations and outputs.

where a result came from — recorded relationships among inputs, transformations, executions and outputs.

Read in context: 2.5.6, 48.4.6

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

Load factor

how full the hash structure is — entries divided by the relevant bucket or slot count under a stated convention.

Read in context: 17.2.6

Locality

useful records are close in the accessed representation — spatial or temporal concentration that can reduce repeated movement or improve cache reuse.

related work stays close together — placement or access behaviour reducing the number of independent resources an operation must involve.

Read in context: 15.5.6, 35.1.6

Lock compatibility

which access modes can coexist — the engine's conflict relation between requested and held lock modes.

Read in context: 24.2.6

Lock holder

the participant currently retaining protected access — the session or transaction owning a granted lock.

Read in context: 24.1.6

Lock lifetime

how long protected access remains held — the release boundary determined by transaction, statement, session or explicit protocol.

Read in context: 24.2.6

Lock scope

the resource access being coordinated — the row, table, key or other engine/application object covered by a lock.

Read in context: 24.1.6

Lock upgrade

requesting stronger access while already holding weaker access — a mode transition that can introduce additional conflicts.

Read in context: 24.4.6

Log

a recorded event — timestamped or otherwise ordered details describing a particular occurrence.

Read in context: 50.2.6

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

Log sequence number

a position in a named recovery stream — an engine-defined offset used to identify and compare WAL progress.

Read in context: 28.3.6

Log tail

the final portion of the recorded sequence — a region that can be incomplete or require validity checks after failure.

Read in context: 28.1.6

Logical backup

reconstruct objects from their logical definitions and values — an export such as SQL or another supported logical representation.

Read in context: 31.1.6

Logical clock

a counter that helps preserve event order — a timestamping mechanism that need not represent physical time.

Read in context: 6.3.6

Logical deletion

removal from ordinary active data — a change to the database's visible record state.

Read in context: 48.3.6

Logical inconsistency

records or calculations violate their intended meaning — a mismatch that can exist even when all files are structurally readable.

Read in context: 32.1.6

Logical payload

the data's application representation — the meaningful values before engine and operational overhead.

Read in context: 50.3.6

Logical record

one complete parsed record — a structure that may span more than one physical text line.

Read in context: 7.3.6

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

Loop count

how often a node is invoked — an execution multiplicity that must be considered alongside per-loop measurements.

Read in context: 18.5.6

Lossless coercion

changing representation without losing the represented value — an allowed conversion such as a supported textual integer into integer storage.

Read in context: 10.4.6

Lossless compression

a smaller representation with exact recovery — byte encoding whose decoder reproduces the original input.

Read in context: 43.3.6

Lossless decomposition

splitting without inventing or losing combinations on reconstruction — decomposition whose natural join recovers every legal original relation instance under the stated dependencies.

Read in context: 13.4.6

Lossy conversion

a change that forgets information; technically, a mapping from which the original value cannot always be reconstructed.

Read in context: 3.1.6

Lossy index entry

a representation that does not retain all distinguishing information — an entry requiring rechecking against fuller data to establish a match.

Read in context: 17.3.6

Lossy transformation

some source detail is discarded — a mapping that cannot generally be inverted to recover the original records.

Read in context: 40.4.6

Lost update

one accepted contribution disappearing beneath another write — a concurrency anomaly caused by insufficient coordination of read-modify-write operations.

one accepted change is overwritten by another — a write anomaly in which a later write fails to preserve an effect that should remain represented.

Read in context: 8.1.6, 22.2.6

LSM tree

organize writes through memory and merged sorted runs — a log-structured merge family of storage designs with deferred reorganization.

Read in context: 29.1.6

M

Magic marker

an initial clue about the representation — a fixed signature used to select or validate a format boundary, not authenticate a producer.

Read in context: 7.5.6

Majority

more than half of the configured group — $\text{floor}(N/2)+1$ participants for a fixed N-member configuration.

Read in context: 37.3.6

Manifest

the record of which storage files belong to the current state — metadata coordinating installed runs and their lifecycle.

an explicit inventory for a data version — metadata naming the files or entries that belong to a committed state.

Read in context: 29.2.6, 42.3.6

Many-to-many

several records on either side can relate — an association that needs an explicit representation and participation rules.

Read in context: 9.4.6

Marginal cost

the change caused by an additional unit of work — distinct from dividing the total bill by current volume.

Read in context: 50.5.6

Materialisation

save an intermediate result for reuse — creation of a stored working representation rather than re-computing or streaming every use.

Read in context: 18.1.6

Materialised projection

an answer stored for later reading — a persisted derived view rather than a query result recomputed on every request.

Read in context: 6.4.6

Materialized view

a stored query result — derived data whose refresh policy determines how current it is.

Read in context: 42.1.6

Measurand

the quantity you intend to measure; technically, the quantity defined as the target of a measurement, with the necessary object and condition information.

Read in context: 4.1.6

Measurement boundary

what the timer includes — the explicitly chosen start and completion conditions of an observed operation.

Read in context: 20.6.6

Membership check

confirm that the target actually exists — equality verification after locating a candidate or insertion position.

Read in context: 16.3.6

Membership configuration

the participants counted by the protocol — a versioned set whose changes require safety-preserving rules.

Read in context: 36.3.6

Memtable

the searchable workspace for recent writes — an in-memory structure holding entries before or alongside persistent sorted files.

Read in context: 29.2.6

Merge decision

an explicit association of records; technically, a versioned decision to treat selected representations as referring to one entity under stated evidence and rules.

Read in context: 5.6.6

Merge fan-in

how many ordered streams are combined at once — the number of input runs a merge step can consume under its buffer and resource limits.

Read in context: 14.4.6

Merge granularity

the unit at which changes are compared — a document, row, field or operation boundary used by the conflict protocol.

Read in context: 25.6.6

Merge iterator

read several ordered runs as one ordered sequence — an iterator that combines candidates while resolving key and version rules.

Read in context: 29.3.6

Merge join

match while advancing through ordered inputs — a strategy using compatible sorted join keys and correct handling of equal-key groups.

Read in context: 18.3.6

Mergeable summary

subgroup evidence can be combined correctly — a summary with a defined merge operation preserving the intended metric or error model.

Read in context: 19.5.6

Message authentication code

a keyed check on a message — an integrity and authenticity mechanism for parties sharing a key, not automatically a public signature.

Read in context: 17.4.6

Metadata

information describing other managed information — structural definitions such as columns, constraints and indexes in this context.

Read in context: 8.2.6

Metric

a measured quantity — an aggregated or sampled numeric signal with units and dimensions.

Read in context: 50.2.6

Metric definition

the exact question behind a number — rules for population, units, timing, inclusion and calculation.

Read in context: 46.1.6

Metric owner

the accountable interpreter of a reported measure — a role responsible for definition changes, exclusions and disputes.

Read in context: 42.5.6

Migration

a controlled evolution of structure or data — a versioned change with preconditions and a recovery strategy.

Read in context: 8.5.6

Missingness

required information is unavailable — a property of the observation process whose causes affect interpretation.

Read in context: 46.3.6

Missingness policy

what absence means and how it is handled; technically, the domain rules for rejecting, preserving, classifying or analysing unavailable values.

Read in context: 3.6.6

Mixed-snapshot calculation

combining facts observed at different boundaries — a derived result that may correspond to no single database state.

Read in context: 23.3.6

Model

a deliberately limited representation; technically, a set of concepts and rules chosen to answer particular questions.

Read in context: 1.6.6

Model defect

a representation that cannot express the intended work correctly — a structural or semantic mismatch with valid domain cases.

Read in context: 10.6.6

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

Multiset

a collection that can contain repeated values — the bag-style semantics relevant to many ordinary SQL results.

Read in context: 9.1.6

Mutation path

a way records can change — an application route, job, import, administrator action or other writer that must respect the invariant.

Read in context: 22.5.6

Mutation policy

the rules for changing a record — a specification of permitted transitions, authorization, evidence and failure handling.

Read in context: 10.3.6

MVCC

let readers use suitable versions while changes occur — multi-version concurrency control using versioned data and transactional visibility rules.

Read in context: 25.1.6

N

Namespace

where a name has its meaning; technically, the scope within which identifiers are interpreted and uniqueness is defined.

the context in which an identifier has meaning; technically, an issuing or interpretive scope within which names are assigned and resolved.

a named container used to distinguish object names — in PostgreSQL, a schema that can contain tables and other database objects.

Read in context: 2.4.6, 5.3.6, 10.5.6

Natural key

a key drawn from the work's own attributes; technically, an identifying set of domain attributes rather than a record-only identifier introduced by the receiving model.

Read in context: 5.2.6

Nearest neighbour

an item closest under a metric — a geometric result whose usefulness depends on the representation.

Read in context: 45.3.6

Negative control

a case that should be refused — a deliberately unauthorised or invalid operation with a checked outcome.

Read in context: 47.6.6

Negative lookup

a search that establishes absence — a lookup returning no visible matching key after checking relevant structures.

Read in context: 29.4.6

Nested-loop join

repeat matching work for each outer record — a join strategy whose inner access can range from a full scan to a selective lookup.

Read in context: 18.3.6

Net growth

increase after relevant additions and removals — a rate whose measurement window and storage boundary must be stated.

Read in context: 50.3.6

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

Non-repeatable read

rereading a row yields a later committed value — a permitted observation change under some isolation contracts.

Read in context: 23.3.6

Normal form

a named condition for placing relational facts — a dependency-based schema property such as 3NF or BCNF.

Read in context: 13.5.6

Normalisation

a specified way to standardise equivalent text sequences; technically, conversion to a Unicode normalisation form, not general identity verification.

Read in context: 3.4.6

NOT NULL

a requirement that a database value be present — a constraint rejecting SQL NULL for a column.

Read in context: 9.2.6

Not-null rule

a requirement that a value be present — a column condition excluding SQL NULL, distinct from nonempty text or positive numeric value.

Read in context: 11.1.6

NULL

a specific SQL missing/unknown marker; technically, a marker with defined behaviour in comparisons, logic, constraints and aggregates.

Read in context: 3.6.6

Null extension

empty right-hand fields for an unmatched record — synthetic NULL-valued attributes introduced by an outer join, not a stored right-hand record.

Read in context: 13.3.6

Nullable foreign key

a reference column that can contain SQL NULL — an optional association under the engine's stated matching semantics.

Read in context: 9.5.6

O

Observation

something someone or something noticed; technically, an acquisition of information under a stated method and context.

something recorded as seen or measured — evidence that need not itself authorise or cause a state-changing business action.

Read in context: 1.2.6, 6.1.6

Observation contract

the stated point or interval a result describes — the temporal and visibility meaning assigned to a query or report.

Read in context: 23.3.6

Observed-only statistic

a result using records with available values — a measure whose population excludes unresolved observations.

Read in context: 46.3.6

Obsolete version

an earlier stored state no longer needed by relevant readers — a version eligible for reclamation under the engine's visibility and maintenance rules.

Read in context: 25.3.6

Occupancy

how full a page is — the used share of its usable entry capacity under the actual layout.

Read in context: 16.2.6

Offsetting errors

wrong values cancel in an aggregate — discrepancies that preserve a total while corrupting individual records.

Read in context: 32.4.6

One-to-many

one parent can have several children — a relationship whose child references need not be unique across the child table.

Read in context: 9.3.6

One-to-one

no more than one related row at each relevant side — a relationship whose optionality must still be specified separately.

Read in context: 9.3.6

Open-loop load

arrivals follow an independent schedule — a workload whose offered demand does not automatically slow with each response.

Read in context: 20.1.6

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

Operational owner

the person or team responsible for running and recovering the system — a named responsibility beyond writing its initial code.

the accountable responder — the person or team responsible for a signal, policy or unresolved task.

Read in context: 8.6.6, 50.6.6

Operational workload

queries and changes that run the business — work often focused on small, current, constrained transactions.

Read in context: 42.1.6

Operationalization

turning an idea into a measurable procedure — implementing a definition through observable records and calculations.

Read in context: 46.1.6

Operator precedence

the default grouping rules for an expression — syntax rules that determine binding unless parentheses explicitly override the grouping.

Read in context: 12.2.6

Operator state

remembered information used by a computation — partial aggregates, identities, timers and metadata needed to process later input.

Read in context: 41.6.6

Optimistic concurrency

detect a conflict when saving rather than holding a long edit lock — a protocol that validates an earlier observation at the write boundary.

Read in context: 25.5.6

Optimistic concurrency check

accepting a change only when relevant expected state still matches — a conflict-detection technique whose adequacy depends on the chosen predicate and version-maintenance policy.

Read in context: 12.6.6

Optional relationship

a permitted link that need not be present — a reference with a minimum participation count of zero.

Read in context: 9.5.6

Oracle

an independent expected-result mechanism — a simpler model used to compare the behaviour of a tested implementation.

Read in context: 16.6.6

Orchestration

a coordinator directs workflow steps — an arrangement that centralizes sequencing and recovery decisions.

Read in context: 38.5.6

ORDER BY

the requested ordering of a query result — a clause specifying sort expressions, directions and applicable ordering semantics.

Read in context: 12.3.6

Orphan file

a stored object not referenced by an accepted table state — prepared or abandoned data requiring a deliberate retention or cleanup policy.

Read in context: 42.3.6

Orphan row

a child with no required matching parent — an existing record violating the intended referential relationship.

Read in context: 11.3.6

Out-of-line value

a large value stored separately from its main row — payload reached through a reference rather than fully embedded in the ordinary tuple.

Read in context: 27.3.6

Outbox

committed work carries a delivery instruction — a transactional record of an event or message to be delivered by a separate process.

a durable list of external work implied by accepted changes — message-intent records committed atomically with local business state.

atomically recorded delivery intent — a local record that survives with the business change but does not guarantee external completion.

Read in context: 21.6.6, 24.6.6, 51.3.6

Outcome reconciliation

resolve uncertainty using recorded identity and state — a procedure for determining what a prior attempt actually accepted.

Read in context: 26.6.6

Outer transaction

the enclosing acceptance boundary — the transaction whose final commit or rollback governs changes retained from inner savepoint work.

Read in context: 21.5.6

Output grain

one resulting row means one what — the level at which a transformed record states a fact.

Read in context: 40.4.6

Overfetching

retrieving extra candidates before later selection — a mitigation that must be evaluated rather than assumed to guarantee filtered recall.

Read in context: 45.5.6

Overflow

exceeding a representation's capacity; technically, an operation whose mathematical result lies outside the representable range.

Read in context: 3.2.6

P

Page cache

operating-system memory holding file data — a cache between application I/O and lower storage devices.

Read in context: 30.2.6

Page occupancy

how much of a page is usefully filled — the proportion consumed by payload and required structures under a stated layout.

Read in context: 27.1.6

Page split

divide an overfull region — redistribution into two nodes with updated parent boundaries and relevant sibling links.

Read in context: 16.4.6

Pagination

splitting a result into separately retrieved pages — a reading pattern needing ordering and cross-request consistency decisions.

Read in context: 12.3.6

Parent row

the row being referenced — a record whose key is the target of a particular foreign-key relationship.

Read in context: 9.3.6

Parser

a shape reader; technically, software that recognises a representation according to its grammar.

Read in context: 2.2.6

Parsing

finding the structure in a representation — recognising records, fields and values under a grammar.

Read in context: 7.2.6

Partial aggregation

summarize locally before combining — retaining sufficient intermediate state for the intended final calculation.

Read in context: 35.5.6

Partial index

a guide for a defined subset — an index whose entries include only rows satisfying its predicate.

Read in context: 15.4.6

Partial operation

only part of a logically related change becoming effective — an incomplete business action whose pieces were not protected by the needed boundary.

Read in context: 8.1.6

Partial order

an order that does not compare every pair — a relation under which some distinct events remain unordered.

Read in context: 6.3.6

Partial rollback

discard only a selected later portion — rollback to a savepoint rather than abandonment of the entire outer transaction.

Read in context: 21.5.6

Partial write

fewer bytes were accepted than requested — an I/O result requiring explicit handling rather than an assumption of complete transfer.

Read in context: 30.2.6

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

Participation minimum

the least number of required related records — a cardinality condition such as “every completed order has at least one line”.

Read in context: 10.2.6

Partition key

the value deciding placement — a field or expression mapping a record into a defined partition.

Read in context: 35.1.6

Partition progress

how far one input segment has advanced — a scoped position that must not be confused with universal completion.

Read in context: 41.1.6

Path

instructions for locating something in a filesystem — a name interpreted under platform-specific pathname-resolution rules.

Read in context: 7.1.6

Path property

useful information about an intermediate route — characteristics such as ordering, parameter dependence and estimated size used in planning.

Read in context: 18.4.6

Payload

the useful represented content — bytes under a stated boundary, excluding or including overhead only as explicitly defined.

Read in context: 14.2.6

Payload equivalence

when two attempts mean the same thing — an explicit comparison contract over validated request semantics.

Read in context: 26.2.6

Peer group

rows tied under a window's ordering — entries equal on all ordering expressions used to define peers.

Read in context: 19.3.6

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

Percentile

a position in an ordered population or sample — a quantile defined under a stated convention, such as nearest rank.

Read in context: 20.2.6

Phantom

a repeated condition matches a changed set — a predicate-query membership change caused by another transaction's committed work.

Read in context: 23.1.6

Physical backup

preserve an engine's storage representation — a copy requiring the matching consistency, version and recovery procedure.

Read in context: 31.1.6

Physical corruption

stored representation is damaged or invalid — a failure involving bytes or engine structures rather than merely an undesirable business value.

Read in context: 32.1.6

Physical replication

following the storage engine's changes — replication tied to physical representation and recovery information.

Read in context: 33.2.6

Physical row location

where one stored version currently resides — engine metadata such as PostgreSQL's `ctid`, not an enduring logical identity.

Read in context: 25.1.6

Placeholder

a marked position for a bound value — syntax such as `?` in the demonstrated Python/SQLite interface.

Read in context: 12.6.6

Placement invariant

a rule about where records belong — a condition linking keys, layout versions and valid owners.

Read in context: 35.6.6

Plan regression

a changed execution choice worsens a defined workload — an observed performance deterioration, not merely a different plan diagram.

Read in context: 15.6.6

Plan selection

choosing among legal routes — the optimiser's decision based on estimated costs and available information.

Read in context: 15.6.6

Planner cost

an estimate for comparing execution choices — a model-specific quantity derived from statistics and configured operation costs.

Read in context: 14.6.6

Point-in-time recovery

restore a supported earlier state — replay from a matching base to a chosen engine-supported recovery target.

Read in context: 31.3.6

Policy bypass

an execution path exempt from a rule — special authority that must be accounted for when testing enforcement.

Read in context: 47.3.6

Policy oracle

the expected permission decision — the independently stated rule used to judge a test result.

Read in context: 47.6.6

Policy version

which rule was applied — an identity connecting a lifecycle decision to the exact governing definition.

Read in context: 48.2.6

Portability

the ability to move a defined workload between environments — compatibility of the used semantics and operations, not merely similar syntax.

Read in context: 8.6.6

Positional index

an index retaining term locations — metadata supporting phrases and proximity conditions beyond word presence.

Read in context: 44.3.6

Post-filtering

removing candidates after retrieval — an approach that can reduce returned coverage when the candidate budget was spent on ineligible items.

Read in context: 45.5.6

Postcondition

something promised after an action — a predicate the completed operation is intended to establish or preserve.

Read in context: 10.1.6

Posting list

occurrences associated with one term — document references, often accompanied by frequency or position information.

Read in context: 44.3.6

Power-loss protection

preserve specified device state during power interruption — a hardware capability whose scope and health must be verified.

Read in context: 30.4.6

Pre-aggregation

calculate a total before joining it to other detail — reducing a relation to the required grouping grain before a later join.

Read in context: 13.6.6

Pre-binding validation

checking input before the driver sends it — validation while original runtime distinctions remain available to application code.

Read in context: 10.4.6

Precision

repeated readings agreeing closely; technically, closeness among replicate indications or measured values under specified conditions.

how much of the returned material is relevant — relevant retrieved items divided by retrieved items under a stated cutoff and unit.

Read in context: 4.3.6, 44.6.6

Precision, matching

the share of accepted matches that are correct; technically, true positives divided by all positive predictions, not measurement precision from Chapter 4.

Read in context: 5.5.6

Precondition

something that must hold before an action — a predicate required at an operation's entry boundary.
what must be true before a change — the checked state and authority on which safe execution depends.

Read in context: 10.1.6, 49.6.6

Predicate

a condition used to choose records — an expression whose truth controls qualification in a query operation.

a condition used to select records — an expression whose truth determines query eligibility under the relevant semantics.

Read in context: 14.1.6, 43.4.6

Predicate pushdown

evaluating selection closer to storage — moving supported filtering work earlier to reduce later processing.

Read in context: 43.4.6

Predicate tracking

remember the read conditions that matter — implementation metadata used to detect dependencies involving rows or ranges a transaction read.

Read in context: 23.5.6

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

Preimage

the exact state expected before a repair — captured values or bytes used to verify that a proposed change targets the reviewed input.

Read in context: 32.5.6

Preimage resistance

hard to work backwards from a digest — computational difficulty of finding an input producing a specified hash output.

Read in context: 17.4.6

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

Primary key

the table's chosen identifying key; technically, a selected set of columns whose combined values uniquely identify each row and cannot be null under the relevant database rules.

the chosen required identifier for a table — the designated key used as a principal row identity in the current schema.

Read in context: 5.1.6, 11.2.6

Primary source

the originating documentation or research — evidence closer to the specified behaviour than a second-hand summary.

Read in context: 52.6.6

Prime attribute

a field belonging to at least one minimal key — an attribute contained in some candidate key, not necessarily the selected primary key.

Read in context: 13.5.6

Principal

the person or program asking — an authenticated or otherwise explicitly established security identity used in policy evaluation.

Read in context: 47.1.6

Probe sequence

alternative places checked after a collision — a defined traversal of slots in an open-addressed hash table.

Read in context: 17.2.6

Process boundary

a separation between running programs — an isolation and communication boundary that changes failure and resource handling.

Read in context: 8.3.6

Processing time

when a processor handles a record — time measured by the executing system, potentially different on replay.

Read in context: 41.2.6

Product quantization

compact codes for vector parts — a representation replacing subvectors with codebook references for storage and distance estimation.

Read in context: 45.4.6

Progress property

required work can eventually advance — a liveness claim dependent on assumptions about failures and resources.

Read in context: 52.4.6

Projection

an answer built from other records — a derived representation produced by applying transformation rules to source data.

choose the fields needed — a relational or SQL operation producing selected expressions rather than every available attribute.

Read in context: 6.1.6, 14.2.6

Projection pruning

avoiding unneeded fields — selecting only the columns required to compute the query result.

Read in context: 43.1.6

Projection version

which interpretation produced the answer — an identifier for the relevant transformation or state representation.

Read in context: 6.4.6

Projection, in a query

choosing which values to display — an operation that selects output columns and can hide source identity.

Read in context: 9.1.6

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

Provenance

the history behind a record; technically, information about entities, activities, agents and derivations involved in producing it.

where information came from and how it was handled — recorded source, capture and transformation context, not automatic proof of truth.

the recorded origin and processing history — information linking source entities, transformations and resulting data.

Read in context: 2.5.6, 7.1.6, 40.1.6

Publication boundary

the point where accepted work becomes available in the destination — a separately controlled step beyond decoding and input validation.

Read in context: 7.6.6

Publication unit

the selected set of records accepted together — the transaction scope for this destination, distinct from an entire source file.

Read in context: 9.6.6

Publisher confirm

the broker acknowledges publication — evidence about the publisher-to-broker boundary rather than consumer completion.

Read in context: 39.3.6

Purpose limitation

using information for a defined need — an explicit boundary on collection and downstream use.

Read in context: 48.1.6

Q

Qualified name

a name that includes its containing scope — an identifier such as `sales.orders` rather than an unqualified table name.

Read in context: 10.5.6

Quantisation

fitting values to allowed steps; technically, mapping values to a discrete set of representable or permitted outputs.

replacing a continuous range with steps; technically, mapping input values to discrete output levels.

Read in context: 3.3.6, 4.4.6

Quarantine

retaining rejected material for controlled review — a handling state with separately specified access and retention rules.

Read in context: 7.6.6

Query

a precise question; technically, an operation expressing a requested result over data.

a request for a result from data — an expression evaluated under a data model and its language semantics.

Read in context: 1.4.6, 8.2.6

Query expansion

adding related search terms — a recall-oriented technique whose extra candidates can change precision and meaning.

Read in context: 44.5.6

Query grain

what one result row means — the level created by joins, projection and grouping.

Read in context: 52.2.6

Query plan

the engine's proposed route to an answer — an execution structure using operations such as scans, filters, joins and aggregation.

Read in context: 8.2.6

Query rewrite

a semantics-preserving change in representation — transformation of a query before or during planning under the engine's rules.

Read in context: 18.1.6

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

Queueing delay

time waiting before work can proceed — delay caused by contention or scheduling rather than the operation's own active computation.

Read in context: 14.6.6

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

Quoting

marking a region as one field — a representation mechanism that allows otherwise structural characters inside a value.

Read in context: 7.3.6

R

Race condition

an answer that depends on an unsafe overlap — behaviour whose correctness changes with the relative execution order of competing operations.

Read in context: 22.1.6

Range

the smallest through largest supported values; technically, the domain bounds of a representation or declared type.

Read in context: 3.2.6

Range partition

a piece owning an ordered interval — a partition defined by lower and upper key bounds.

Read in context: 35.2.6

Range scan

walk entries between boundaries — ordered traversal beginning near a lower bound and stopping when an upper condition fails.

Read in context: 16.5.6

Raw sample

an individual retained observation — an unaggregated measurement from which summaries can be recomputed.

Read in context: 20.6.6

Read amplification

extra storage work for a logical read — a defined ratio or probe count measured at a specified layer.

reading more bytes than the requested logical data — additional work caused by layout, blocks, meta-data or access patterns.

Read in context: 29.4.6, 43.3.6

Read dependency

a decision relies on an observed fact — a relationship between a transaction's read and another transaction's relevant write.

Read in context: 23.4.6

Read scaling

sharing question-answering work — distributing eligible read operations across resources without assuming identical freshness.

Read in context: 33.1.6

Read-modify-write

read a value, compute a replacement, then store it — a multi-step sequence that needs an appropriate concurrency rule.

Read in context: 8.1.6

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

Rebuild

write a reorganized physical representation — maintenance that can improve layout while requiring I/O, temporary capacity and coordination.

Read in context: 27.5.6

Recall

the share of true matches found; technically, true positives divided by the actual positive cases under the evaluation labels.

how much relevant material was found — relevant retrieved items divided by the known relevant population.

Read in context: 5.5.6, 44.6.6

Reciprocal rank

a measure of how soon the first useful answer appears — one divided by its rank, with an explicit no-answer convention.

Read in context: 44.6.6

Recomputation

rebuilding from an authorised source set — creating a new derived result under a recorded transformation version.

Read in context: 48.4.6

Reconciliation

comparing related accounts of the same work; technically, identifying and resolving or explicitly retaining discrepancies under a defined procedure.

checking whether related records agree as expected — a comparison of identities, relationships and totals rather than a single balancing sum.

explaining how the input relates to the output — accounting for candidates, repeats, rejections and file-level limits without silently losing material.

compare defined representations of the same work — an evidence-based check of populations, keys, amounts and explained differences.

compare related records to resolve disagreement — a procedure that checks quantities, identities and outcomes against stated invariants and evidence.

comparing expected and observed information — a set of coverage, identity, value and invariant checks across representations.

comparing meaningfully corresponding records — checking identities, values and declared exclusions rather than one headline total.

Read in context: 1.2.6, 6.5.6, 7.6.6, 13.6.6, 22.6.6, 40.6.6, 49.4.6

Reconciliation identity

an equation connecting related records — a check that quantities and recorded state changes agree under stated inclusions and exclusions.

Read in context: 26.1.6

Reconciliation population

the records intended for comparison — a defined set that prevents mixing separate fixtures or tenants.

Read in context: 51.4.6

Reconstruction boundary

how far an answer can be rebuilt — the scope supported by retained inputs, snapshots and interpretation rules.

Read in context: 6.6.6

Record class

a group with a common responsibility — data categorised by purpose, access and lifecycle requirements rather than file extension alone.

Read in context: 48.1.6

Record locator

information leading from an entry to its record — an engine-specific reference or identifying value used to retrieve associated data.

Read in context: 16.3.6

Recording time

when a defined system boundary recorded the information; technically, a timestamp whose acquisition point must be specified.

Read in context: 2.5.6

Recovery base

the known state from which replay begins — a checkpoint or backup identified together with its matching log boundary.

Read in context: 28.5.6

Recovery dependency

something required to make restored data usable — keys, configuration, software, access and other prerequisites beyond database bytes.

Read in context: 30.5.6

Recovery log

information needed to reconstruct a valid stored state — engine records supporting recovery after incomplete persistence or interruption.

Read in context: 28.1.6

Recovery oracle

independent evidence of the expected result — identities and values used to judge what should remain after the tested failure.

Read in context: 30.6.6

Recovery 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 point objective

how much recent work the plan may lose — a stated acceptable recovery-point gap, not an achieved measurement by itself.

Read in context: 8.5.6

Recovery protocol

the rules for interpreting surviving evidence — a specified procedure connecting log validity, transaction status and stored data.

Read in context: 28.1.6

Recovery 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

Recovery target

the intended state or boundary to restore — a chosen time, log position or other engine-supported objective with matching evidence.

Read in context: 28.6.6

Recovery time objective

how long recovery may take — a target for restoring a defined service level under a stated scenario.

Read in context: 8.5.6

Recovery timeline

a named branch of physical recovery history — metadata distinguishing histories after recovery or promotion under the engine's rules.

Read in context: 31.3.6

Recovery unit

the things the restoration claim includes — an explicit scope such as one database or a complete application service.

Read in context: 8.5.6

Redelivery

a message is delivered again — a transport event that must be distinguished from a new business event.

Read in context: 39.4.6

Redo

reconstruct an accepted change from recovery information — replay that restores data not fully reflected in the main data files before failure.

Read in context: 27.4.6

Redo horizon

the earliest recovery position still needed — the log boundary from which reconstruction must begin for a particular purpose.

Read in context: 28.4.6

Redundancy

several representations of a fact — duplication whose consistency depends on maintaining a specified relationship between representations.

Read in context: 13.1.6

Referential action

what happens to a relationship when a parent changes — a declared response such as restriction, cascading deletion or setting a reference to NULL.

Read in context: 11.3.6

Referential check

verify relationships between records — a test that child references resolve according to the schema's key rules.

Read in context: 32.3.6

Referential integrity

references remaining valid under declared rules — the property enforced by active foreign-key constraints within their semantics.

Read in context: 9.6.6

Regression envelope

cases checked for unintended deterioration — representative workloads beyond the one case an intervention aims to improve.

Read in context: 18.6.6

Regression fixture

data and steps reproducing a past failure — a test case intended to prevent recurrence of the specific defect.

Read in context: 32.6.6

Regression test

a repeatable check preserving an intended behaviour — evidence that a known property still holds for the tested inputs after a change.

Read in context: 9.6.6

Rehearsal

a bounded execution before live work — an experiment whose representativeness and limitations must be stated.

Read in context: 49.6.6

Reinitialization

rebuilding a participant from valid source state — a controlled procedure restoring a compatible replica and its role.

Read in context: 34.6.6

Reintroduction

retired information returning through recovery — restoration of an older state without later lifecycle decisions.

Read in context: 48.5.6

Relative path

a location described from a starting point — a pathname whose interpretation depends on a working directory or another supplied base.

Read in context: 7.1.6

Relative update

describe the change rather than an old replacement — an expression such as `value = value - amount` evaluated by the database.

Read in context: 22.2.6

Release manifest

an inventory of delivered bytes — filenames, sizes and digests that support integrity checks.

Read in context: 51.6.6

Repair branch

a deliberate resolution of incomplete transactional work — an explicitly permitted continuation that restores the failed condition before a later commit.

Read in context: 11.5.6

Repair manifest

the record of a bounded correction — evidence, scope, changes, checks and authority supporting an intervention.

Read in context: 32.5.6

Replacement entry

the newly accepted corrected representation — a record distinguished from the earlier entry it supersedes or repairs.

Read in context: 6.5.6

Replacement result

a new complete answer for the same key — output that supersedes an earlier revision rather than adding to it.

Read in context: 41.5.6

Replay

another attempt carrying the same defined intent — a repeated request that should resolve through its existing identity and outcome.

Read in context: 26.3.6

Replay position

how far recovery has applied the stream — progress measured under the engine's interpretation of log records.

Read in context: 28.3.6

Replay review

inspect historical pending work before reactivation — reconciliation of restored tasks with effects that may already exist outside the backup boundary.

Read in context: 31.5.6

Replay-safe delivery

repetition does not duplicate the intended effect — handling a repeated identity consistently while detecting conflicting payloads.

Read in context: 40.5.6

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

Replication slot

retained source progress for a consumer — a PostgreSQL mechanism whose log retention and restart behaviour require monitoring.

Read in context: 40.3.6

Report grain

what one result row describes — the precise unit represented after selection, joining and aggregation.

Read in context: 19.1.6

Representation compatibility

vectors can be compared meaningfully — agreement between encoder, dimension, normalization and metric contracts.

Read in context: 45.1.6

Representation contract

meaning attached to stored values — units, range, missingness and interpretation beyond the physical type.

Read in context: 52.2.6

Repricing

calculating using a different price basis — recomputation with current or alternative prices rather than the stored agreed values.

Read in context: 10.3.6

Reprocessing

applying rules again to retained inputs — a controlled rebuild whose interpretation depends on input and transformation versions.

Read in context: 40.6.6

Reproducibility record

enough context to repeat a result — candidate, inputs, environment, operations and observations.

Read in context: 52.5.6

Reproducible example

an example another reader can check; technically, one supplied with sufficient inputs, procedure and environment information to repeat its stated observation.

Read in context: 1.6.6

Request identity

the stable name of one intended operation — a scoped identifier linking repeated attempts to one business decision.

Read in context: 26.2.6

Reservation

a recorded allocation under defined rules — a business state transition that reduces available capacity while linking the allocation to a request.

Read in context: 22.6.6

Reset scope

what was actually returned to a starting condition — the exact layers or resources reinitialised before a measurement.

Read in context: 20.4.6

Residual copy

information remaining after selected actions — a retained, failed, blocked or unknown destination requiring explicit treatment.

Read in context: 48.6.6

Residual filter

a check made after finding candidates — a predicate evaluated after an access path has already visited candidate entries or rows.

Read in context: 14.5.6

Resolution

the smallest distinguishable change; technically, the change in the measured quantity that causes a perceptible change in the corresponding indication under the stated conditions.

Read in context: 4.3.6

Restated series

historical results recalculated under a new rule — a versioned view distinct from what was previously published.

Read in context: 46.1.6

Restatement

a later version of an earlier report — a deliberate correction whose relationship to previously published results remains visible.

Read in context: 41.6.6

Restoration

bringing back a stored copy — a recovery operation that must account for the copy's age and subsequent required changes.

Read in context: 6.6.6

Restore acceptance

decide whether the recovered state meets its objective — a documented result based on actual recovery and relevant application checks.

Read in context: 31.5.6

Restore isolation

keep the recovered copy from affecting live systems — a controlled environment with outbound effects and production access disabled during validation.

Read in context: 31.2.6

Restore rehearsal

actually exercise the recovery procedure — a controlled test measuring whether the required state and service can be recovered.

recovering and checking a separate destination — evidence about a specific recovery procedure and input.

Read in context: 31.4.6, 51.5.6

Restore test

trying the recovery procedure and inspecting its result — evidence about a specified backup and environment rather than a promise about all future failures.

Read in context: 8.5.6

Restore-time suppression

prevent retired data from re-entering ordinary use — reapplication of approved deletion or access decisions after historical recovery.

Read in context: 31.6.6

Restricted retention

kept for a bounded exception — retained data whose access and use differ from ordinary active records.

Read in context: 48.5.6

Result cardinality

how many output rows there are — the row count at a specified stage, distinct from records examined or bytes read.

Read in context: 14.1.6

Result contract

the meaning promised by a query's output — a specification of columns, units, grain, missingness and ordering where relevant.

Read in context: 10.5.6

Result population

the records or combinations included in an answer — the scope after joins and filters, which must match the report's intended question.

Read in context: 9.5.6

Resurrection

an old deleted value becomes visible again — a correctness failure caused by losing the deletion information while older state remains reachable.

Read in context: 29.5.6

Retention

how long and why information is kept; technically, a policy governing continued storage and eventual disposition for defined record classes.

Read in context: 2.6.6

Retention boundary

which history remains available — a stated limit on kept records, detail, time or access.

Read in context: 6.6.6

Retention trigger

the event starting a lifecycle rule — a defined transition such as closure, not an assumed timestamp.

Read in context: 48.2.6

Retrieval boundary

what finding a source does and does not prove — the distinction between candidate similarity and verified, authorized factual support.

Read in context: 45.6.6

Retry amplification

repeated attempts adding load during trouble — feedback that can worsen the condition callers are trying to recover from.

Read in context: 50.4.6

Retry budget

the allowed additional effort — limits on attempts, elapsed time or resources spent resolving transient failures.

Read in context: 24.5.6

Retry deadline

the point beyond which another attempt is not allowed — an operation-level bound on time spent resolving transient failures.

Read in context: 23.6.6

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

Reusable space

allocated storage available for later records — free capacity inside an existing database structure rather than necessarily returned to the filesystem.

Read in context: 27.5.6

Reversal

an opposite recorded effect tied to an earlier entry — a constrained correction operation whose scope must be defined.

Read in context: 6.5.6

Revision

a particular state after a change; technically, an identifiable version of a record or artefact.

Read in context: 2.6.6

Revision exhaustion

no unused value remains in the chosen domain — a boundary requiring an explicit policy rather than silent counter reuse.

Read in context: 25.4.6

Revocation

withdrawing previously valid authority — disabling future accepted use, distinct from deleting one stored copy.

Read in context: 47.5.6

Revocation delay

time before a withdrawn permission stops working — the exposure window introduced by cached or long-lived authority.

Read in context: 47.1.6

Roll-forward

correcting through a subsequent change — a new reviewed transformation rather than automatic restoration of the past.

Read in context: 49.5.6

Rollback

cancelling uncommitted work; technically, discarding the transaction changes within the requested rollback scope.

abandoning uncommitted database work — restoration of the transaction's prior database state, not reversal of unrelated external actions.

abandon uncommitted transactional work — restoration of the transaction's database effects under its supported semantics.

returning within a defined reversible boundary — undoing a transaction or deployment, with scope that must be stated.

Read in context: 1.5.6, 8.4.6, 21.2.6, 49.5.6

Root

the starting guide — the top node from which every lookup begins.

Read in context: 16.1.6

Rotation

replacing active secret material — transitioning consumers to new authority under an explicit lifecycle.

Read in context: 47.5.6

Rounding

reporting a nearby value on a chosen scale; technically, mapping a value to a specified representable set under a tie-breaking rule.

Read in context: 4.4.6

Rounding policy

the chosen way to handle non-exact results; technically, the rule and stage used to select an allowed output value.

Read in context: 3.3.6

Row grain

what one row stands for — the unit of representation against which counts and aggregates must be interpreted.

what one row asserts — the identity and meaning that determine valid joins and constraints.

Read in context: 9.1.6, 51.2.6

Row group

a horizontal subset stored in column pieces — a unit connecting corresponding column chunks within a columnar file.

Read in context: 43.1.6

Row header

metadata around the visible field values — implementation information needed to locate, interpret or manage a stored row version.

Read in context: 27.3.6

Row width

the amount carried by one row representation — estimated or measured bytes per row at a specified physical or logical stage.

Read in context: 14.2.6

Row-level security

a filter or rule on individual rows — database policy enforcement beyond ordinary object privileges.

Read in context: 47.3.6

RPO

how much recent accepted work may be lost — a recovery-point requirement defined against a specified incident and acceptance boundary.

Read in context: 31.4.6

RTO

how long required service restoration may take — a recovery-time requirement with explicit start and completion events.

Read in context: 31.4.6

Run overlap

several files cover some of the same key range — a layout property affecting how many candidates a lookup may need to inspect.

Read in context: 29.3.6

Run-length encoding

storing repetitions as a value and count — an encoding effective when identical values are consecutive.

Read in context: 43.2.6

Runbook

a documented response path — observations, decisions, bounded actions and escalation conditions for an operational situation.

Read in context: 50.6.6

Runtime identity

the account serving ordinary work — the database or service identity used by the request-handling process.

Read in context: 47.2.6

Runtime provenance

which software actually produced the result — the recorded engine and interpreter versions and relevant configuration of an experiment.

Read in context: 27.6.6

S

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

Safety property

a prohibited outcome never occurs — a correctness claim such as no duplicate accepted effect under the stated model.

Read in context: 52.4.6

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

Sample standard deviation

the observed spread of a sample; technically, the square root of the sum of squared deviations from its mean divided by $n - 1$.

Read in context: 4.3.6

Sampling frame

the set from which observations can be selected; technically, the operational list or mechanism used to reach candidate sample units.

the cases available for selection — the actual population from which observations can be drawn.

Read in context: 4.5.6, 46.4.6

Sanitisation

rendering target media data infeasible to recover under a stated standard — a different layer from an empty application query.

Read in context: 48.3.6

Saturation

a limiting resource is at or near effective capacity — a condition in which additional demand tends to create waiting, rejection or deterioration rather than proportional useful work.

demand meeting a limiting resource — a condition producing increased waiting, rejection or reduced useful throughput.

Read in context: 20.3.6, 50.4.6

Savepoint

a local return point inside transaction work — a named boundary to which later transactional changes can be rolled back.

Read in context: 21.5.6

Scale

what one stored step represents; here, the fixed relationship between an integer amount and its displayed unit.

Read in context: 3.2.6

Scan

visit data through an access path — sequential or otherwise structured traversal of records or index entries to produce candidates.

Read in context: 14.1.6

Scatter-gather

ask several pieces and combine their replies — a distributed query pattern with completeness and coordination requirements.

Read in context: 35.5.6

Schedule evidence

the recorded sequence that actually ran — observations of operations, waits, errors and outcomes under a stated concurrency configuration.

Read in context: 22.1.6

Schema

the declared structure; technically, a definition of fields, types, relationships and applicable structural rules.

Read in context: 2.2.6

Schema boundary

what the declared database model does and does not enforce — the limit beyond which application workflow or other mechanisms remain necessary.

Read in context: 9.6.6

Schema design

the plan for organising records and their rules — a model of structures, domains, relationships and invariants implemented by a database and its surrounding system.

Read in context: 10.1.6

Schema migration

a controlled change to database structure or representation — a transition requiring data compatibility, verification and a defined failure disposition.

Read in context: 11.6.6

Schema version

which contract applies; technically, an identifier for a defined revision of the schema and its interpretation.

the named edition of a record contract — an identifier for the supported structure and meaning, distinct from a software release.

Read in context: 2.2.6, 7.4.6

Scope boundary

where the guarantee applies — the trusted helper, tenant, database and supported operations included in the claim.

Read in context: 51.1.6

Scoped guarantee

a promise with explicit limits — a correctness claim tied to defined effects, participants, failures and retention assumptions.

Read in context: 26.6.6

Scoped key

an identifier plus its domain — a composite identity such as `tenant_id` and `order_id`.

Read in context: 47.4.6

Search invariant

the condition kept true at every step — the target, if present, remains inside the selected subtree's permitted range.

Read in context: 16.3.6

Secondary index

an additional searchable organisation — an access path distinct from the primary physical or identifying organisation of the table.

Read in context: 15.1.6

Secret

protected capability-bearing information — a credential or key requiring controlled disclosure and use.

Read in context: 47.5.6

Select list

the values requested for each output row — the column references or expressions following `SELECT`.

Read in context: 12.1.6

SELECT statement

a description of an answer to read — a SQL query specifying output expressions and the operations needed to define a result.

Read in context: 12.1.6

Selection bias

the selection process distorting the intended picture; technically, systematic differences caused by how observations enter the analysed sample relative to the target population.

the observed group systematically differs from the target — distortion caused by how cases enter or remain in the dataset.

Read in context: 4.5.6, 46.4.6

Selectivity

the fraction that passes a condition — qualifying rows divided by a stated candidate population, either estimated or observed.

Read in context: 14.5.6

Semantic compatibility

preserving meaning across versions — agreement about units and interpretation, not only accepted syntax.

Read in context: 49.1.6

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

Semantics

what a representation means; technically, the interpretation and permitted consequences assigned to it.

Read in context: 2.1.6

Sensitivity bound

a range under alternative assumptions — an explicit calculation showing how unknown information could change a conclusion.

Read in context: 46.3.6

Sentinel entity

a special record used to stand for a designated case — a modelling choice that must not fabricate identity or conceal unknown meaning.

Read in context: 9.5.6

Separator key

a boundary directing the next step — an internal key dividing ordered child ranges under a defined comparison convention.

Read in context: 16.1.6

Sequence gap

an expected predecessor is missing — a discontinuity that prevents this strict projector from claiming a complete prefix.

Read in context: 6.3.6

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

Serializability

concurrent work has a valid serial explanation — equivalence of committed transaction effects to an execution in some serial order.

Read in context: 23.5.6

Serializable outcome

a result explainable by a legal one-at-a-time execution — an outcome equivalent to some serial order of the participating transactions.

Read in context: 23.4.6

Serialization failure

a transaction is refused to preserve the isolation guarantee — an error requiring an appropriate full-operation retry or failure response.

Read in context: 23.5.6

Serverless, in SQLite

no separate database server is required — SQLite's embedded-engine meaning, distinct from some cloud-service uses of the word.

Read in context: 8.3.6

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, 50.1.6

Service-level objective

the target for an indicator — a specified level over a stated population and window.

Read in context: 50.1.6

Session window

a group linked by activity gaps — a window that may merge when an arriving event connects earlier groups.

Read in context: 41.3.6

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

Shared guard record

one coordination point for a shared limit — a record whose guarded mutation serializes changes to the represented resource.

Read in context: 22.5.6

Side effect

an action beyond calculating the returned value — an external change that must not accidentally be repeated during reconstruction.

Read in context: 6.4.6

Significant figures

the digits deliberately retained in reporting; technically, a numerical presentation convention that does not alone specify a full uncertainty model.

Read in context: 4.4.6

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: 14.5.6, 35.3.6

Skipped check

a check not executed under current conditions — neither a pass nor proof of failure of the underlying feature.

Read in context: 52.5.6

Sliding window

overlapping intervals — windows whose width exceeds their step, allowing one event to contribute to multiple results.

Read in context: 41.3.6

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

Slot entry

a small reference locating an item within a page — metadata mapping an item identifier to its stored position and length.

Read in context: 27.1.6

Slowly changing dimension

a policy for evolving descriptions — a modelling approach that may overwrite attributes or preserve dated versions.

Read in context: 42.2.6

Snapshot

a saved position at a known boundary — a serialised state with enough metadata to resume a compatible interpretation.

Read in context: 6.4.6

Snapshot establishment

the event that chooses the view — the implementation-defined point at which a transaction's snapshot is acquired.

Read in context: 23.2.6

Snapshot-to-stream handover

joining the initial copy to later changes — a protocol preventing gaps and managing overlap at a capture boundary.

Read in context: 40.3.6

Snippet

a displayed excerpt from a result — contextual text that requires source attribution and safe rendering.

Read in context: 44.5.6

Sort key

the values deciding order — the ordered sequence of expressions and comparison rules used to arrange records.

Read in context: 14.3.6

Sorted run

one already ordered batch — an intermediate sequence used by an external merge process.

a sequence already ordered by its search key — a file or logical collection that can be searched and merged using key order.

Read in context: 14.4.6, 29.1.6

Space amplification

physical footprint exceeds live logical data — a ratio whose scope includes the selected files, versions and temporary maintenance state.

Read in context: 29.6.6

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

Spill

working data moved out of the operation's memory allowance — temporary external storage used to continue an operation such as sorting or hashing.

Read in context: 14.4.6

Split

revising an association into distinct entities; technically, a correction of mapping relationships whose downstream effects may require separate repair.

Read in context: 5.6.6

Split propagation

a full parent also needs room — upward restructuring triggered when adding a child exceeds an internal node's capacity.

Read in context: 16.4.6

SQL

a language for working with data; technically, Structured Query Language and its standardised and product-specific features.

Read in context: 1.3.6

SQL expression

a rule for calculating an output value — a combination of column references, literals, operators or functions evaluated under engine semantics.

Read in context: 12.4.6

Stale client copy

an earlier result still held outside the database — data whose original read validity does not guarantee current edit validity.

Read in context: 25.2.6

Stale read

an observation that no longer describes the relevant current state — a previously read version that may be unsafe as an unconditional write basis.

an answer lacks required newer information — a read that does not meet the application's freshness or consistency contract.

Read in context: 22.2.6, 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

Stale request

a request based on an older position — an operation whose assumed version no longer matches the accepted state.

Read in context: 6.2.6

Standard uncertainty

uncertainty expressed like a standard deviation; technically, measurement uncertainty expressed as a standard deviation under the stated model.

Read in context: 4.6.6

Standardization

applying a common comparison population — reweighting subgroup results under explicitly chosen weights.

Read in context: 46.2.6

Startup cost

estimated work before the first result — a planner quantity relevant to operations that must prepare or consume input before emitting rows.

Read in context: 18.4.6

State

the position the model currently holds — a representation at a defined logical boundary.

Read in context: 6.1.6

State checkpoint

a recoverable state boundary — a saved representation coordinated with the progress and effects it claims.

Read in context: 41.6.6

State transition

an allowed move between positions — a specified change from one model state to another.

one defined move from an old state to a new state — an operation with explicit preconditions, writes and outcomes.

Read in context: 6.2.6, 26.1.6

Statement rollback

undoing effects of one failed statement — a failure response distinct from abandoning all earlier work in the active transaction.

Read in context: 11.6.6

Statement snapshot

a view chosen for one statement — the visibility boundary used by a query under statement-level snapshot rules.

Read in context: 23.2.6

Statistical unit

the entity providing an observation — the request, person, item or time unit whose independence and weighting need definition.

Read in context: 46.6.6

Steady state

workload and maintenance remain sustainable together — an operating regime without unbounded growth of deferred work under the stated conditions.

Read in context: 29.6.6

Stock adjustment

an explicit change to the expected stock record — a domain operation distinct from a goods movement or a count observation.

Read in context: 6.1.6

Storage observation

a measured fact about one physical configuration — metadata or instrumentation interpreted with version, scope and timing.

Read in context: 27.6.6

Storage type

the engine's representation category; technically, a supported type or storage class with documented operations and limits.

the kind of value an engine stores in a field — an implementation-level representation with defined conversion and operation rules.

Read in context: 3.1.6, 10.4.6

Store inventory

the declared destinations to address — an explicit set whose completeness must be assessed, not assumed.

Read in context: 48.3.6

Stream

an ongoing supply of records — input processed as a continuing sequence rather than only as a completed finite batch.

Read in context: 41.1.6

Stream sequence

a record's position in one accepted history — a scoped ordering value used for interpretation or concurrency checks.

Read in context: 6.3.6

Structural integrity check

inspect engine representation rules — a diagnostic procedure for covered page, tree or table/index invariants.

Read in context: 32.3.6

Structural validation

check that the representation obeys its rules — inspection of ordering, ranges, links, occupancy and depth invariants.

Read in context: 16.6.6

Success boundary

the event a positive response promises has occurred — the precise acceptance or persistence condition represented to the caller.

Read in context: 30.1.6

Successor link

the route to the next ordered region — a reference or traversal mechanism connecting neighbouring entries or leaf nodes.

Read in context: 16.5.6

Surrogate key

an extra identifier created for the record; technically, a key whose principal role is row identity rather than existing business meaning.

Read in context: 5.2.6

Symptom

what was observed — a result needing explanation, not an already-proven cause.

Read in context: 52.3.6

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

Syntactic validity

being written in the right shape — conformance to a format's grammar, distinct from business rules.

Read in context: 7.2.6

Syntax

the shape of a representation; technically, the grammar used to recognise valid constructions.

Read in context: 2.1.6

System metadata

information the engine stores about its own objects — structural descriptions exposed through catalogues or metadata interfaces.

Read in context: 10.5.6

T

Table

a collection of records with a declared shape — a managed SQL structure whose columns and constraints give rows a defined interpretation.

Read in context: 9.1.6

Table snapshot

a coherent table version — metadata selecting the files and interpretation visible to a reader.

Read in context: 42.4.6

Tail latency

the unusually slow end of the distribution — high-percentile delays whose meaning depends on the measured population and sample size.

Read in context: 20.2.6

Tare mass

the packaging amount taken away; technically, the measured or assigned container mass used in a net-mass calculation.

Read in context: 4.1.6

Target cardinality

how many records an operation is intended to affect — a result-scope requirement distinct from whether each resulting row is structurally valid.

Read in context: 12.5.6

Target population

everything the question is meant to describe; technically, the defined set of units about which an inference or descriptive claim is intended.

Read in context: 4.5.6

Temporal interpretation

which point or period a value describes — the time-related meaning that distinguishes a current attribute from a historical observation or agreement.

Read in context: 10.2.6

Tenant

one separately scoped customer organisation — an isolation domain within a service that may share infrastructure.

Read in context: 47.4.6

Tenant scope

the customer boundary in a shared service; technically, the domain boundary used to associate records and permitted operations with a tenant, requiring access control beyond identification.

Read in context: 5.3.6

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

Test fixture

the controlled input state — synthetic or appropriately authorised data used for a specified check.

Read in context: 52.5.6

Text literal

text written as a value in a statement — a quoted string such as '0-1042', distinct from a column identifier.

Read in context: 12.1.6

Theoretical availability

required requests eventually complete — a liveness property under a stated distributed-system model.

Read in context: 36.2.6

Three-valued logic

reasoning with true, false and unknown — SQL expression semantics needed when operands can be NULL.

Read in context: 12.2.6

Throughput

completed work per unit time — a rate whose counted operation and success condition must be defined.

Read in context: 20.3.6

Tie-break

a rule ordering equal-scored results — a deterministic convention that does not itself establish relevance.

Read in context: 44.4.6

Tie-breaker

an additional rule for equal primary sort values — a subsequent sort expression used to distinguish otherwise tied rows.

an extra rule for equal first choices — additional ordering attributes that distinguish otherwise tied results.

an additional ordering key — a rule making otherwise tied results deterministic where required.

Read in context: 12.3.6, 14.3.6, 52.2.6

Time bucket

the period a record contributes to — a reporting interval defined by clock semantics, timezone and boundary rules.

Read in context: 19.4.6

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

Tokenisation

splitting text into searchable units — applying a specified segmentation rule to content and queries.

Read in context: 44.2.6

Tombstone

a newer marker hiding an older value — a deletion record retained until older visible copies and snapshot requirements permit removal.

Read in context: 29.5.6

Top-N

keep the requested best few — an optimisation that retains a bounded candidate set under an ordering, without necessarily avoiding a full input scan.

Read in context: 14.3.6

Torn page

only part of a page update survives — a mixed old/new physical page caused by an interrupted multi-part write.

Read in context: 30.4.6

Trace

connected steps of an operation — spans and context describing work across a request path.

Read in context: 50.2.6

Traceability matrix

a record showing how a requirement is addressed — a mapping among assumptions, representation, enforcement, tests and remaining gaps.

Read in context: 10.6.6

Transaction

a controlled unit of database work; technically, operations managed together under an engine's atomicity and related semantics.

a defined group of database work — a unit with a controlled commit or rollback boundary.

Read in context: 1.5.6, 8.4.6

Transaction boundary

the work accepted or rejected together — the scope of a database transaction's reads, writes and completion decision.

Read in context: 21.1.6

Transaction owner

the code responsible for completion — the layer that controls commit and rollback for a defined unit of work.

Read in context: 21.1.6

Transaction ownership

responsibility for starting and ending a transaction — a contract defining which component may commit, roll back or leave the connection active.

responsibility for completion — the component entitled to commit or roll back the particular unit of work.

Read in context: 11.5.6, 51.2.6

Transaction rollback

abandoning a transaction's pending changes — an explicit or engine-triggered reversal within that transaction's defined scope.

Read in context: 11.6.6

Transaction snapshot

a retained view for related statements — a visibility boundary reused through a transaction under the chosen implementation.

Read in context: 23.2.6

Transformation precondition

what must be true before changing a row — the provenance, version or state justifying the operation.

Read in context: 49.3.6

Transformation version

identify the calculation rules used — a versioned definition of parsing, filtering, joining and aggregation semantics.

which processing rule was used — an identifier binding output to a particular implementation and its parameters.

Read in context: 19.6.6, 40.1.6

Tree height

how many levels a lookup crosses — the root-to-leaf depth under a stated counting convention.

Read in context: 16.2.6

Trigger

the rule deciding when to publish — a condition for emitting a window result or pane.

Read in context: 41.5.6

Truncation

an input ending before its promised content is complete — insufficient bytes to satisfy a declared structure or length.

Read in context: 7.2.6

Trusted expected digest

an integrity reference with justified provenance — a digest bound to an intended artifact through a trusted source or process.

Read in context: 32.2.6

Tuple

one same-shaped collection of attribute values — a row-like element of a relation in the relational model.

Read in context: 9.1.6

Two-phase commit

prepare followed by a final decision — an atomic-commit protocol across participating transactional resources.

Read in context: 38.2.6

Type

the kind of value; technically, a classification that governs representation and permitted or defined operations.

Read in context: 3.1.6

U

Unbounded input

no known final record — a collection whose completion is not generally available during normal operation.

Read in context: 41.1.6

Uncertainty budget

the explanation behind the uncertainty figure; technically, a statement of contributing uncertainty components, their evaluation and their combination in the measurement model.

Read in context: 4.6.6

Undo information

evidence for reversing unaccepted changes — recovery data used by designs that require explicit restoration of earlier state.

Read in context: 28.5.6

Uniform model

assume every output is equally likely — a mathematical distribution assumption whose applicability must be justified separately.

Read in context: 17.5.6

Unique constraint

a rule against repeated key values; technically, database enforcement that the relevant value combination cannot appear in conflicting rows under its null and comparison rules.

a prohibition on repeated constrained values — an invariant over one column or a composite value under the engine's comparison and NULL rules.

Read in context: 5.2.6, 11.2.6

Unit

the agreed size of a numerical step; technically, a reference quantity used to express values of quantities of the same kind.

Read in context: 4.2.6

Unknown field

a member the contract does not recognise — data requiring an explicit reject, ignore, preserve or extension policy.

Read in context: 7.4.6

Unknown outcome

the caller lacks reliable completion evidence — a state in which a request may have committed even though its response was not received.

the caller lacks enough evidence to classify completion — a state distinct from confirmed success or confirmed rollback.

Read in context: 22.6.6, 34.5.6

Unknown result

a comparison that cannot be classified as true or false from the supplied values — the third truth outcome commonly produced by SQL expressions involving NULL.

Read in context: 11.1.6

UPDATE

changing values on selected records — a SQL write operation applying assignments to rows matching its predicate.

Read in context: 12.5.6

Update anomaly

one correction leaves conflicting copies — a state inconsistency caused by storing a dependent fact in multiple independently editable rows.

Read in context: 13.1.6

User-space buffer

data waiting inside the program or library — memory used to batch output before passing it to the operating system.

Read in context: 30.2.6

UTC offset

the difference from UTC in a representation; technically, a signed hours-and-minutes offset, not a complete named-zone rule set.

Read in context: 3.5.6

UUID

a standardised large identifier; technically, a 128-bit identifier whose version and variant determine the interpretation and generation rules.

Read in context: 5.2.6

V

Vacuuming

maintenance that reclaims and manages old storage — PostgreSQL's process for handling obsolete tuples and other required maintenance tasks.

Read in context: 25.3.6

Validation

checking the stated rules; here, testing whether a representation meets declared structural or domain requirements.

Read in context: 1.2.6

Validation boundary

the exact state a check describes — its database, snapshot or time, schema and input scope.

Read in context: 49.4.6

Vectorized execution

processing batches of values together — an implementation approach reducing overhead and exploiting suitable data layout.

Read in context: 43.6.6

Verification scope

what a check could actually observe — the stores, identities, versions and authority included in the observation.

Read in context: 48.6.6

Version column

a field naming the accepted edit state — an application-managed revision value checked and advanced by relevant writes.

Read in context: 25.4.6

Version register

the declared implementation/document context — a record distinguishing documented versions from the runtime actually tested.

Read in context: 52.6.6

Visible sequence limit

the newest version a snapshot may use — a model or engine boundary excluding later updates from that reader.

Read in context: 29.3.6

Visible version

the row state allowed by a reader's snapshot — a version satisfying the engine's transaction-visibility rules.

Read in context: 25.2.6

Volatile write cache

acknowledged data still depends on power — intermediate storage that can lose pending writes when its power disappears.

Read in context: 30.4.6

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

Wait-for graph

a map of who waits for whom — a directed graph used to reason about resource dependencies.

Read in context: 24.3.6

Warm state

earlier work remains useful — cached data or prepared resources available to later operations under a specified scope.

Read in context: 20.4.6

Watermark

a statement of event-time progress — a source- and framework-dependent frontier used for window and timer decisions.

Read in context: 41.4.6

Weighted mean

larger contributions receive proportionate influence — the sum of value-times-weight divided by total valid weight.

Read in context: 19.2.6

WHERE predicate

the condition deciding which source rows qualify — a Boolean-like SQL expression for which only true rows are retained by the filter.

Read in context: 12.2.6

Wilson interval

a binomial-proportion interval — a score-based method whose limits remain within the valid probability range.

Read in context: 46.6.6

Window

a defined group of time-related records — a membership rule used to bound aggregation over a stream.

Read in context: 41.3.6

Window frame

the contributing region for one current row — a bounded subset of its partition under ROWS, RANGE or another supported frame mode.

Read in context: 19.3.6

Window function

calculate across related rows without collapsing each one — an SQL function evaluated over a partition and, where applicable, an ordered frame.

Read in context: 19.3.6

Workflow

the steps around completing a piece of work — a sequence of operations, decisions and failure paths with defined state transitions.

Read in context: 10.1.6

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

Workload

the work the system must actually handle — a defined mixture of reads, writes, sizes, timing and contention.

Read in context: 8.6.6

Workload envelope

the conditions a design was evaluated for — declared ranges of size, skew, query mix, concurrency and service requirements.

Read in context: 15.5.6

Workload mix

the combination of operations a system serves — their frequencies, parameters, sizes, concurrency and correctness requirements.

Read in context: 15.2.6

Workload specification

a description of the actual work — volumes, distributions, query patterns and operating requirements used to evaluate a design.

Read in context: 42.6.6

Write amplification

extra work caused by one logical change — a ratio of physical or internal writes to a clearly defined logical write baseline.

storage writes exceed logical input — a measured ratio of bytes written at a named layer to logical bytes accepted over a stated interval.

Read in context: 15.2.6, 29.1.6

Write path

a route through which information can change — an application, import, job or administrative mechanism capable of mutating stored state.

Read in context: 10.1.6

Write skew

separate writes jointly break a shared rule — an anomaly in which transactions act on overlapping read conditions while changing different records.

Read in context: 23.4.6

Write-ahead ordering

recovery evidence must precede dependent data persistence — the ordering constraint linking log durability and data-page writes.

Read in context: 28.2.6

Wrong-scope mutation

a valid change applied to the wrong set of records — an operation whose predicate does not express the intended target boundary.

Read in context: 12.5.6

Z**Zone map**

summary bounds for a data region — metadata such as minimum and maximum used for conservative elimination.

Read in context: 43.5.6

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://jcgmbipm.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://jcgmbipm.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://jcgmbipm.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://jcgmbipm.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/combo.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://jcgmbipm.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://jcgmbipm.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>