BE SOFTWARE ARCHITECTURE WISE · CHAPTER 7

Data Architecture

Data outlives code, and in distributed systems it is the hardest part to get right. This chapter covers who owns each piece of data and why a shared database couples services, how to choose between relational and other databases, what transactions and isolation levels really guarantee, how replication and partitioning work and fail, CQRS and event sourcing, caching, evolving schemas without downtime, and analytics, streaming and data mesh. It is for developers and architects who already know the distributed styles of Chapter 6 and need to make sound, defensible decisions about data.

bewiseapps.com/software-architecture-wise/data-architecture/ · Version 2026-10-26. Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Explain why a shared database couples services, and assign ownership of data, including cases where several services write the same data.
- Choose a database for each data set by data shape, access patterns, consistency needs, scale, operations and team skills, and justify the cost of each additional store.
- Distinguish the isolation levels and the anomalies they allow, including lost updates and write skew, and protect a read-modify-write path with locking, version checks or serializable transactions.
- Explain leader-follower, multi-leader and leaderless replication, the anomalies caused by replication lag, and the risks of failover and split brain.
- Choose a partitioning scheme and shard key from measured volumes and queries, and explain consistent hashing, hot keys and secondary indexes.
- Decide when CQRS, event sourcing and caching are worth their costs, and avoid their common failure modes.
- Plan schema changes with compatibility rules and the expand-and-contract pattern so that they need no downtime.
- Distinguish operational and analytical data architectures: warehouses, lakes, lakehouses, stream processing and data mesh.

Before you start

Chapter 6, especially the sections on microservices, event-driven architecture, consistency, CAP and PACELC, and sagas. Chapter 0's section on storing data covers the database basics assumed here.

Time

About 3½ hours to read; 45 minutes for the test.

Key terms

system of record · database per service · polyglot persistence · isolation level · snapshot isolation · write skew · replication lag · consistent hashing · CQRS · event sourcing · cache-aside · expand and contract · lakehouse · data mesh

Why data is the hard part

Code can be rewritten in months; data lives for decades. Customer accounts, ledgers, orders and medical histories outlast every framework and every service that touches them, and moving them is slow, risky and expensive. That is why the decisions in this chapter are among the most significant an architect makes, in the sense of Chapter 1: they are hard to reverse, wide in scope and decisive for quality attributes such as availability, performance and modifiability.

Chapter 6 split systems into services that communicate over a network. Splitting code is the easy half. As soon as there are several services, someone has to decide who may change each piece of data, how others see it, which guarantees hold when many users act at once, and what happens when copies of the data disagree. The chapter follows two running examples throughout: a food-delivery platform, with its evening rush, courier locations and menus, and an online bank, where losing or double-counting a single payment is unacceptable.

Owning, storing and protecting data

The first four sections deal with the operational data that systems run on. Data ownership establishes the most important rule: every important fact should have one system of record and one writer. It explains why a shared integration database couples services however cleanly their code is separated, how to resolve cases where several services want to write the same data, how to read data you do not own, and how data contracts make shared data dependable.

Choosing a database surveys relational, document, key-value, wide-column, graph, time-series, search, vector and distributed SQL stores, and argues that the relational database remains the right default for systems of record, with specialised stores earning their place by meeting a requirement the default cannot. Transactions and isolation goes beneath the familiar ACID acronym to the anomalies that concurrency causes, such as lost updates and write skew, what isolation levels really guarantee in PostgreSQL, MySQL, Oracle and SQL Server, and when to use locks, version checks or serializable transactions. Replication and partitioning explains how databases copy and split data across machines, what replication lag and failover do to correctness, and how to choose a shard key from the numbers.

Deriving, caching and evolving data

The second half deals with data that is derived from the systems of record, and with change over time. CQRS and event sourcing separates the model that accepts changes from the models that answer queries, and shows how storing a history of events instead of current state enables audit and temporal queries, at a real cost in complexity. Caching covers the patterns, levels and failure modes of keeping copies close to where they are needed, from the browser to the database, including the stampedes that can take a system down when a popular entry expires.

Schema evolution addresses the fact that every schema, whether database tables, API payloads or events, will change while the system is running, and shows how compatibility rules and the expand-and-contract pattern allow change without downtime. Finally, analytics and streaming moves from running the business to understanding it: warehouses, lakes and lakehouses, stream processing, and data mesh as a way of organising analytical data around domains.

How the sections fit together

The sections form a chain. Ownership decides where each fact is written. The choice of database and its transaction guarantees decide which rules can be enforced cheaply inside that owner. Replication and partitioning decide how the owner's data survives failures and grows beyond one machine. Everything else in the chapter is a deliberate copy: read models, caches, search indexes, warehouses and streams. Each copy is faster or more convenient for some purpose, and each is stale by some amount. Schema evolution keeps the owner and all its copies able to change without breaking each other.

The chapter assumes the distributed-systems vocabulary of Chapter 6, especially consistency models, event-driven architecture and sagas and the outbox pattern. It hands on to Chapter 8, which protects this data, and to Chapter 9, which keeps the systems that hold it fast and available.

The one idea to take away

For every important piece of data, you should be able to answer three questions: who is its only writer, where are its rules enforced, and how stale may each copy of it be? Most data problems in real systems, such as orders that change state backwards, reports that disagree with the ledger, double bookings and customers who cannot see their own changes, are failures to answer one of those questions deliberately. The techniques in this chapter are ways of answering them well, and every one of them is a trade-off between consistency, availability, performance, cost and the effort of running the system.

SECTION 7.1 OF 7.8 · INTERMEDIATE · 24 MIN READ

Data ownership and database per service

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 6.3 Microservices
- 6.4 Event-driven

Terms it uses from earlier: Saga, Eventual consistency, Schema, Transaction

THE IDEA IN BRIEF

Think of a shared kitchen notice board where anyone can rub out and rewrite the day's orders. It works with two cooks. With ten, orders get overwritten, nobody knows who changed what, and redesigning the board means asking everyone's permission. A better kitchen gives each station its own board and its own responsibility: the grill writes grill orders, and other stations read them or ask the grill to change them. Data ownership means deciding, for each piece of information, which part of a system is the one allowed to change it.

WHAT YOU NEED TO KNOW

- A system of record is the authoritative source for a piece of data; data ownership means one component writes it, enforces its rules, controls its schema and publishes it to others.
- A shared (integration) database couples services through schema, deployment, semantics and runtime load; Fowler (2004) and most architects advise against it between independently developed applications.
- Database per service keeps each service's data private (private tables, schema or server) at the cost of cross-service joins, transactions and referential integrity, which must be replaced by APIs, events, sagas and reporting pipelines.
- Joint ownership can be resolved by table split, a small shared data domain, delegation to one owner or service consolidation, each with a distinct trade-off (The Hard Parts, 2021).
- Data you do not own can be read through synchronous calls, column schema replication, replicated caches or events; every copy is read-only and possibly stale.
- Data contracts state schema, meaning, quality, service levels and ownership for data shared between teams, and are most useful when checked automatically.

- For each important fact, a sound design can name its only writer, where its rules are enforced and how stale each copy may be.

Why ownership is the first data question

Chapter 6 split systems into services that are deployed, scaled and changed independently. Code splits relatively easily; data does not. An order refers to a customer, a menu item, a restaurant and a courier, and every team wants to read, and often change, some part of it. The most important question in a distributed data architecture is therefore not which database to buy but **who owns each piece of data**: which component is allowed to change it, which one enforces its rules, and how everyone else gets to see it.

Ownership matters because data outlives code. Services are rewritten every few years; customer accounts, ledgers and medical records are kept for decades. Getting ownership wrong produces the expensive kind of architectural debt: two services that cannot be released separately because they share tables, a status field that three teams overwrite in turn, or a monthly report that nobody can reproduce because the numbers come from five copies of the same data that disagree.

This section uses two running examples that recur throughout the chapter: a food-delivery platform (customers, restaurants, couriers, orders) and an online bank (customers, accounts, payments, a ledger).

System of record and data ownership

A system of record is the authoritative source for a given piece of data: the place whose value wins when copies disagree. In the bank, the core ledger is the system of record for account balances; the balance shown in the mobile app, the one in the fraud engine's cache and the one in last night's data warehouse load are all copies. In the food-delivery platform, the restaurant's own menu service is the system of record for prices, even though the search index, the mobile app's offline cache and the order lines all hold copies of them.

Data ownership turns this into a responsibility. The owning component (and the team behind it) is the only one that writes the data, enforces its *invariants* (rules that must always hold, such as “a ledger entry is never edited, only reversed”), decides its schema, and publishes it to others through an interface it controls. Ownership is a design decision, not a property of the database: two services can sit on the same database server and still have clear ownership, and two services with separate databases can still fight over the same facts if the boundary between them is drawn badly.

A useful rule from *Software Architecture: The Hard Parts* (Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani, 2021) is that the service that writes to a table owns it. Reading is cheap to share; writing is not, because whoever writes must enforce the rules.

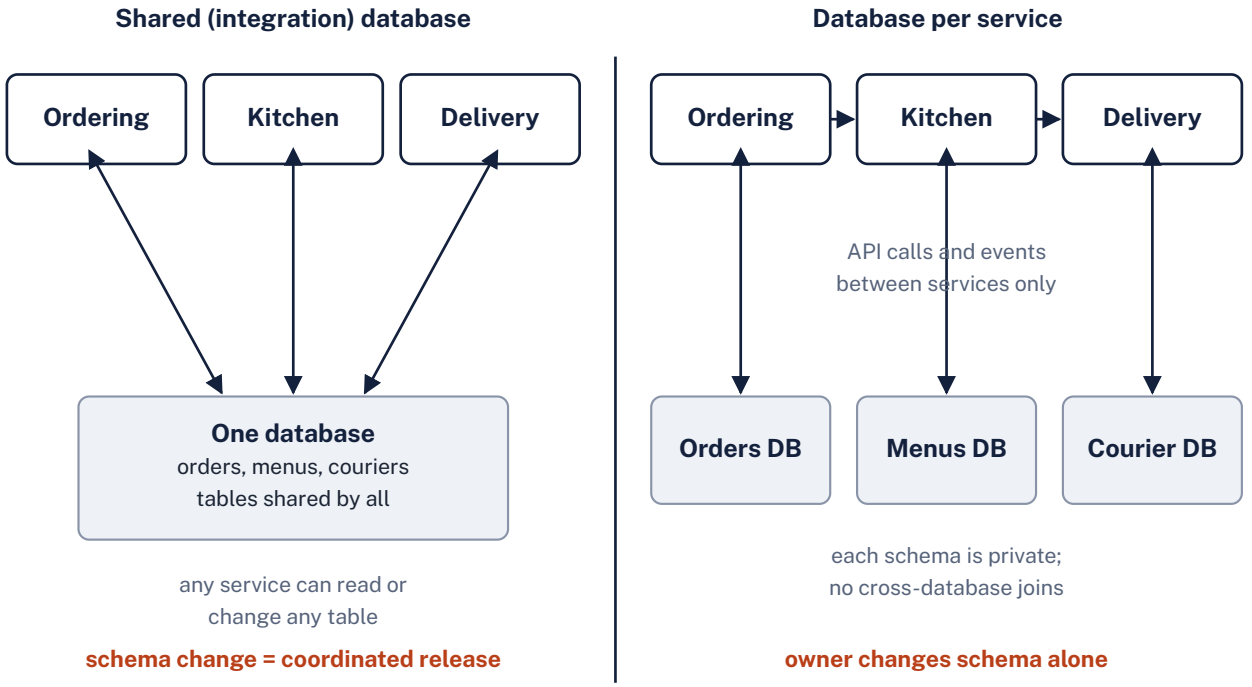
The shared database and why it couples services

The simplest way to let several applications work with the same data is to give them all access to one database. Martin Fowler called this an integration database (2004): a database that is the integration point between applications, as opposed to an *application database* that serves one application only. It looks attractive: no APIs to build, joins work across everything, and one transaction can update orders and payments together. Fowler's verdict, shared by most architects, is that integration databases should be avoided, because the coupling they create is deep and mostly invisible.

Consider the forms of coupling it introduces. **Schema coupling**: every table and column is effectively a public API. Renaming `orders.state` to `orders.status` breaks any service that reads it, and nobody can be sure which ones do, because SQL queries are scattered through many codebases, reports and spreadsheets. **Deployment coupling**: a schema change has to be coordinated across every team that touches the table, so services that were meant to be released independently are released together. **Semantic coupling**: the schema must serve every client, so it grows more general and more confusing (Fowler notes that it ends up trying to unify concepts that belong in separate bounded contexts), and the business rules for a table end up duplicated, inconsistently, in every service that writes it. **Runtime coupling**: one service's slow report query or lock-heavy batch job degrades every other service, and the database becomes a shared single point of failure and a ceiling on connections.

None of this is fatal for a single application or a modular monolith whose modules respect each other's tables, or for the coarse-grained service-based architecture in which a handful of domain services often share a database deliberately. It becomes a problem when teams expect independent deployability but keep the shared tables, which is a classic route to a *distributed monolith*.

Shared database versus database per service



With a shared database every table is a public interface; with a database per service each schema is private and data crosses boundaries only through APIs and events.

Database per service

The alternative, associated with microservices, is the database-per-service pattern: each service's persistent data is private to it and other services can reach it only through the service's API or the events it publishes. Chris Richardson's pattern catalogue (microservices.io) notes that this does not have to mean a separate server per service. There are three levels of separation, and the important line is the first one.

Three ways to keep a service's data private

Level	What is separate	Strength of the boundary	Typical use
Private tables per service	Each service has its own set of tables on a shared database, with separate credentials that can only reach those tables.	Weakest: relies on permissions and discipline; still shares capacity and failure.	Early in a migration; small systems; modular monoliths.
Schema per service	Each service has its own schema (namespace) on a shared server.	Clear ownership and access control; still shares the server's performance and availability.	Most teams starting with several services; cheaper to run.
Database server per service	Each service has its own database instance, possibly of a different type.	Strongest: separate scaling, failure and technology choice.	Services with very different load, data shape or availability needs.

The benefits mirror the costs of the shared database: owners change their schemas freely, scale and tune their stores independently, choose the right kind of database, and contain failures. The costs are real and should be stated in any decision record. Joins across services are gone and must be replaced by API composition or replicated data. A single ACID transaction can no longer span orders and payments; cross-service changes become sagas with compensating steps. Referential integrity across services (every order points to a real customer) is no longer enforced by foreign keys. Reporting needs a separate pipeline. And there are more databases to back up, patch and monitor.

The Hard Parts frames the decision as a balance of forces. **Disintegrators** push towards splitting data: change control (how many services break when a table changes), connection limits, scalability, fault tolerance, the wish for independently deployable units and the need for different database types. **Integrators** pull data together: relationships that need referential integrity, and operations that genuinely need one atomic transaction. If two tables are always changed together in one transaction and have no independent life, that is strong evidence they belong to the same owner.

When several services write the same data

Ownership questions become hard when more than one service legitimately changes the same entity. *The Hard Parts* distinguishes three situations. With **single ownership**, only one service writes the table, and it simply owns it. With **common ownership**, most services write the same table (an audit log or a set of configuration settings, for example); the usual answer is to create

a dedicated service that owns the data and give everyone else an API, often an asynchronous one so writers are not slowed down. With joint ownership, a small number of services, typically two, write the same table, usually within the same domain. The book offers four techniques for joint ownership.

Resolving joint ownership

Technique	How it works	Main trade-off
Table split	Split the table so that each service owns the columns it writes, in its own table. The food-delivery `orders` row becomes an order owned by Ordering and a kitchen ticket owned by Kitchen.	Clean ownership, but creating or deleting the parent entity now has to be synchronised between services, and any view of the whole needs both.
Data domain	Keep the table shared by the few services involved, inside a small shared schema that they jointly control.	Simple and fast, with transactions still possible, but it is a deliberately limited shared database: the services are coupled on schema change and the boundary between them weakens.
Delegate	Pick one owner; the other service sends it requests (synchronous calls or asynchronous commands) instead of writing. The owner is chosen by primary domain (whose concept is it?) or by operational characteristics (which service needs the fastest, most available writes?).	Clear ownership and rules in one place, at the cost of an extra network hop, a new failure mode and possibly eventual consistency for the delegating service.
Service consolidation	Merge the services that share the data into one service.	Removes the problem and restores local transactions, but creates a larger service with coarser deployment, scaling and fault isolation.

No technique is free, and the right one depends on how often the data changes, how strict its rules are and how independent the teams need to be. In practice, delegation is the most common answer for business entities with a lifecycle (orders, payments, bookings), table split works well when the jointly written columns are really different facts that happened to be stored together, and consolidation is the honest answer when the two services were split too finely in the first place.

Reading data you do not own

Once ownership is settled, every other service still needs to *read* the data. The kitchen display needs the customer's delivery notes; the fraud engine needs the account's recent payments. There are four broad options, and each trades freshness against coupling. Note that event-driven designs, covered in Chapter 6, provide the third and fourth.

Ways to read another service's data

Option	How it works	Freshness	Coupling and cost
Synchronous call	Ask the owner's API at the moment the data is needed.	Current, as of the call.	Runtime coupling: if the owner is slow or down, so is the caller; latencies add up along a chain.
Column schema replication	Copy the few columns needed (for example, restaurant name and address) into the reader's own database, kept up to date by events or change data capture.	Eventually consistent: stale by the replication delay.	Fast local reads and joins; the reader must handle late or missed updates and must never write the copy.
Replicated cache	An in-memory data grid (such as Hazelcast or Apache Ignite) keeps a copy of the owner's data in each consuming service, synchronised by the grid.	Near real time.	Very fast reads; memory cost, start-up loading and the operational complexity of the grid.
Events (event-carried state transfer)	The owner publishes events that carry the changed data; readers keep whatever they need.	Eventually consistent.	Loose temporal coupling; readers must cope with duplicates, ordering and schema evolution.

Fowler calls the fourth option event-carried state transfer: events such as `MenuItemPriceChanged` carry the new values, so consumers do not need to call back. A related mechanism, change data capture (CDC), reads the owner's database log and turns each committed change into an event, which avoids the *dual-write problem* (writing to the database and publishing an event separately, so that one can succeed while the other fails). The transactional outbox, which solves the same problem from inside the application, is covered with sagas.

WATCH OUT

A copy is a cache, not a second master

Replicated data is safe only if it is treated as read-only and as possibly stale. The moment a consumer starts correcting its copy (fixing a restaurant's address locally because the owner's update was late), there are two systems of record and they will drift apart. Decide for each copy how stale it may be, what happens when it is wrong, and how it is rebuilt from the owner when it is lost.

IF YOU ARE NEW TO THIS

Step by step: ask the owner, or keep a copy

This section's hardest idea is how a service should use data that it does not own. It can ask the owner every time, or keep its own copy, and each choice trades freshness against dependence on the owner.

- To send couriers to the right door, Delivery must know where every restaurant is and what it is called. That data belongs to the Restaurant service, which is its system of record.
- Option one: Delivery calls Restaurant's API every time it assigns a courier. The answer is always current, but if Restaurant is slow or down, courier assignments slow down or stall too, though nothing is wrong with Delivery itself.
- Option two: Delivery copies just those two columns into its own database, which the table above calls column schema replication. Whenever a restaurant's details change, Restaurant publishes an event carrying the new values and Delivery updates its copy: Chapter 6's event-carried state transfer. Reads are now local and fast, and they keep working while Restaurant is down.
- The price is staleness. When a restaurant corrects its address, Delivery holds the old one until the event arrives. If the event is lost, the copy stays wrong until it is refreshed from the owner.
- So the copy is read-only. If Delivery spots a wrong address, it tells Restaurant, which corrects it and publishes the fix; editing the copy would create a second master, as the warning above explains.
- **The lesson:** asking the owner gives fresh data but ties you to it while you run; a copy frees you but is read-only and possibly stale. For every copy, decide how stale it may be and how it is rebuilt from the owner.

Reporting across services

Database per service makes one old task harder: the cross-cutting report. “Revenue by restaurant by week, net of refunds, split by courier type” needs orders, payments, restaurants and deliveries. Running it by calling four APIs page by page is slow and puts reporting load on operational services. Querying their databases directly reintroduces the shared-database coupling through the back door.

The usual answer is to separate reporting from operations. Fowler described a *reporting database*: a separate store, shaped for queries and fed from the operational systems, which reporting users may read freely. Today this is usually a data warehouse or lakehouse fed by CDC or events, discussed in analytics and streaming. For small, near real-time views inside the product (a restaurant's dashboard of today's orders), a service can build its own read model from events, the idea behind CQRS. Either way, the pipelines that feed reporting depend on the owners' data structures, which is why they need contracts.

Data contracts

A data contract is an explicit, versioned agreement between the owner of a data set and its consumers. It states the schema (fields, types, which may be null), the meaning of each field (is `amount` in pence or pounds, including VAT or not?), quality expectations (no duplicate order identifiers, at most 0.1% of rows with a missing restaurant), service levels (how fresh the data is, how long it is kept), who owns it and how changes are announced. It applies the idea of an API contract to data that flows between teams through events, CDC streams and warehouse tables.

Contracts are most useful when they are machine-readable and checked automatically: a pipeline that rejects a schema change which would break registered consumers, or a test that fails when a quality rule is violated. The Open Data Contract Standard (ODCS), maintained by the Bitol project under the Linux Foundation AI & Data, defines a YAML format covering schema, quality rules, service levels, roles and support channels. Data contracts are not free: someone must write and maintain them, and overly strict contracts slow down change. *The Hard Parts* contrasts strict contracts (precise, verifiable, tightly coupling) with loose ones (easier to evolve, needing more testing), and suggests consumer-driven contracts, in which consumers state what they actually use, as a middle path. How to change a schema without breaking consumers is the subject of schema evolution.

WORKED EXAMPLE

Worked example: who owns order status in the food-delivery platform?

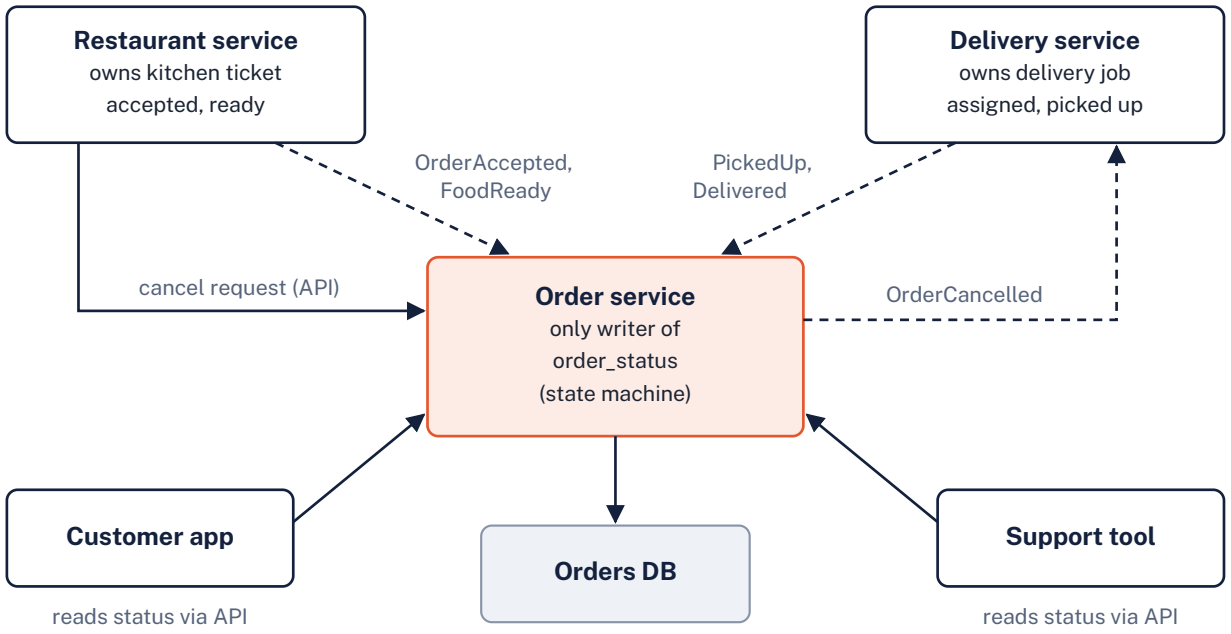
The platform has three services that care about an order's progress: Ordering (takes and pays for orders), Restaurant (the kitchen tablet: accepts orders, marks food ready) and Delivery (assigns couriers, tracks pick-up and drop-off). Today all three write the `status` column of a shared `orders` table. Customers have seen orders flip from “Picked up” back to “Preparing”, and nobody can say where the cancellation rules live.

Diagnose	Joint ownership of one column by three writers. Each writes what it knows, last write wins, and late messages move the status backwards. The rule “an order can be cancelled only before the kitchen starts” is checked in two services, differently.
Separate the facts	There are really three facts: the kitchen ticket's state (owned by Restaurant), the delivery job's state (owned by Delivery) and the customer-facing order status, which combines them and carries the business rules (cancellation, refunds, compensation).
Option: table split	Give each service its own status column and let readers combine them. Rejected: every reader (app, support tool, notifications) would reimplement the combination, and the cancellation rule would still have no home.
Option: service consolidation	Merge all three into one service. Rejected: three teams, very different load (courier location updates arrive every few seconds) and different release cycles.
Decision: delegate	Ordering is the only writer of `order_status`. Restaurant and Delivery own their own facts and publish events (`OrderAccepted` , `FoodReady` , `PickedUp` , `Delivered`). Ordering consumes them and applies a state machine. A cancellation from the restaurant tablet is a request to Ordering's API, which accepts or refuses it.
State machine	PLACED, ACCEPTED, PREPARING, READY, PICKED_UP, DELIVERED, with CANCELLED allowed only from PLACED or ACCEPTED. Transitions only move forward, so a late `FoodReady` arriving after `PickedUp` is recorded but does not change the status; duplicates are ignored.
Consequences	One place for the rules; no backwards jumps; each team releases independently. Costs: the status lags the kitchen by the event delay (typically well under a second), Ordering must be available to accept cancellations, and the events become a contract that must be versioned.

The key move was not choosing a technique from the table but noticing that one column was hiding three different facts with three natural owners.

Write the decision down as an Architecture Decision Record, including the rejected options; the next team to ask “why can't we just update the status from the courier app?” will need it.

Order status after the decision



Each fact has exactly one writer; everyone else reads or asks

Each fact has one writer: Restaurant and Delivery publish what they own, and the Order service alone turns those events into the customer-facing status.

Common mistakes

Splitting services but keeping the shared tables. Teams get the operational cost of distribution without the independence it was meant to buy. If services cannot be deployed without coordinating a schema change, they are not independent. **Entity services.** A `CustomerService` that is nothing but create, read, update and delete over a table, called by every other service, is a shared database with extra latency; ownership should follow business capabilities and their rules, not nouns. **Copies that quietly become masters.** See the warning above. **Ignoring reporting until the end.** The first finance report after the split often reveals that nobody planned how data would be brought back together. **Treating ownership as purely technical.** Ownership is also organisational: someone must answer for the data's quality, and per Conway's law data boundaries that cut across team boundaries tend not to last.

A good test of any data design is to ask, for each important fact, three questions: who is the only writer, where are its rules enforced, and how stale may each copy be? If any answer is “it depends” or “several services”, the design is not finished.

CHECK YOUR UNDERSTANDING

▼ Two services each update the `status` column of the same table. What kind of ownership is this, and name two ways to resolve it.

Joint ownership. It can be resolved by splitting the table, delegating writes to one owner, keeping a small shared data domain or consolidating the services.

▼ Why does a shared database undermine independent deployment?

Every table is effectively a public interface, so a schema change must be coordinated with every service that reads or writes it, forcing joint releases.

▼ What is the difference between column schema replication and a synchronous call when reading another service's data?

Replication keeps a local, eventually consistent copy that is fast and available even if the owner is down; a synchronous call returns current data but couples the caller's availability and latency to the owner.

▼ What does a data contract add beyond a schema?

Meaning of fields, quality expectations, freshness and retention levels, ownership and how changes are announced and versioned.

SOURCES

- Martin Fowler: IntegrationDatabase (2004): https://martinfowler.com/bliki/IntegrationDatabase.html
- Martin Fowler: ReportingDatabase: https://martinfowler.com/bliki/ReportingDatabase.html
- Chris Richardson, microservices.io: Pattern: Database per service: https://microservices.io/patterns/data/database-per-service.html
- Chris Richardson, microservices.io: Pattern: Shared database: https://microservices.io/patterns/data/shared-database.html
- Bitol (Linux Foundation AI & Data): Open Data Contract Standard: https://bitol-io.github.io/open-data-contract-standard/latest/

SECTION 7.2 OF 7.8 · INTERMEDIATE · 25 MIN READ

Choosing a database: relational, document, key-value, column, graph and time-series

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 6.6 CAP and consistency
- 7.1 Data ownership

Terms it uses from earlier: System of record, Transaction, SQL (Structured Query Language), Change data capture (CDC), Architecture Decision Record (ADR)

THE IDEA IN BRIEF

Choosing a database is like choosing storage for a home. A filing cabinet with labelled folders (a relational database) suits paperwork you need to cross-reference. A wardrobe of complete outfits (a document store) suits things you always take out together. A row of numbered lockers (a key-value store) is quickest when you know the number. A corkboard with string between pins (a graph) shows who knows whom. Most homes need a filing cabinet plus one or two specialised pieces, not one of everything, because every extra piece needs space, cleaning and someone who knows how it works.

WHAT YOU NEED TO KNOW

- Relational databases remain the default for business data because they provide transactions, constraints, flexible queries, mature tooling and widely available skills, and now also handle JSON, text, geospatial and vector data at moderate scale.
- Document, key-value, wide-column, graph, time-series, search and vector stores each make particular access patterns cheap and others expensive; they excel as specialised or derived stores.
- Distributed SQL (Spanner, CockroachDB, YugabyteDB, TiDB) keeps SQL and transactions while scaling across nodes and regions, at the cost of write latency, money and operational complexity.
- Choose per data set using evidence: data shape, access patterns, consistency needs, measured scale, operational burden and team skills; write the key queries and volumes down first.

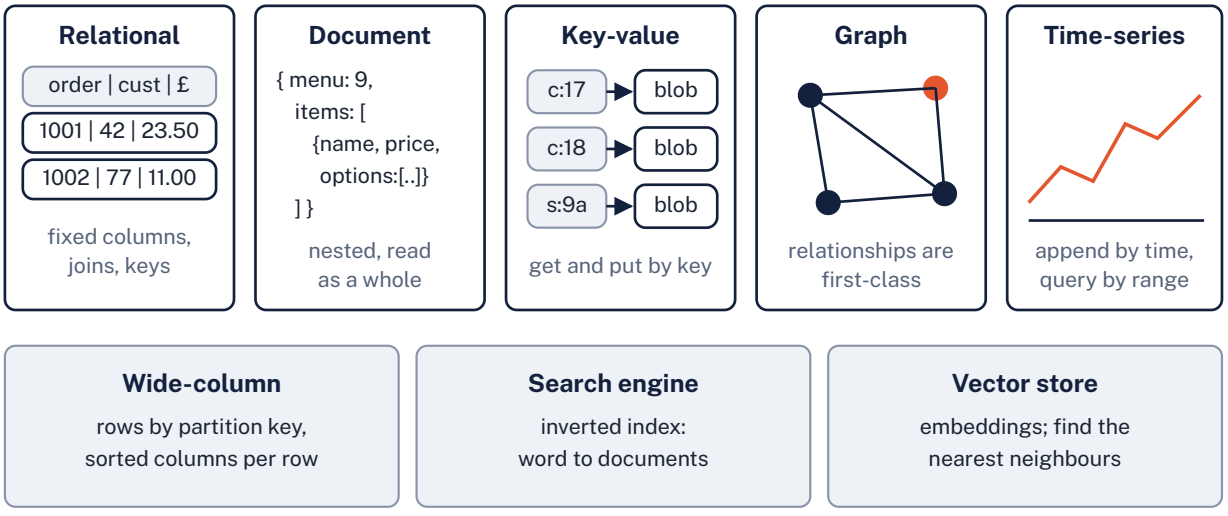
- Separate systems of record, which need the most conservative technology, from derived stores such as search indexes and caches, which can be specialised because they can be rebuilt.
- Polyglot persistence lets each data set use a fitting store, but every extra technology adds lasting operational and cognitive cost; McKinley's 'Choose Boring Technology' (2015) argues for spending innovation sparingly.

Why the choice deserves care

A database is one of the hardest components to replace. Applications are rewritten around their data, not the other way round; migrating terabytes of live data while the business keeps running can take a year. The choice of store is therefore a one-way door that deserves an explicit decision, while many teams make it by habit or by fashion.

Chapter 0 introduced relational databases and the four NoSQL families. This section goes further: what each family is really good at, the newer categories (time-series, search, vector and distributed SQL), how to choose by evidence rather than enthusiasm, and what it costs to run several kinds at once. The answer is rarely one database for everything, and almost never a different database for every service.

Data models at a glance



Each family organises data differently, and that shape decides which queries are cheap and which are impossible.

Relational: still the default, for good reasons

A relational database stores data in tables with declared types, keys and constraints, and answers declarative SQL queries, an idea from Edgar Codd's 1970 paper. Fifty years on it remains the sensible default for business data, for four reasons. First, **transactions and constraints**: the database itself guarantees that a payment and its ledger entries commit together and that no order points at a missing customer, so these rules do not have to be reimplemented, imperfectly, in every service. Second, **query flexibility**: you do not need to know every future question when you design the schema; a new report is a new query, not a new table. Third, **maturity**: decades of tooling for backup, replication, monitoring, migration and performance diagnosis, and engineers who already know it. Fourth, **range**: modern relational systems store and index JSON (PostgreSQL's `jsonb`), support full-text search, geospatial data (PostGIS), time partitioning and vector similarity (the pgvector extension), so one well-run database can cover many needs at moderate scale.

The real limits are specific. Writes on a single primary are bounded by one machine; very large tables need careful partitioning; schema changes on huge tables need technique (schema evolution); and some access patterns, such as deep graph traversals or relevance-ranked text search, are served much better by specialised engines. A single modern server with fast solid-state drives can serve thousands of transactions per second and terabytes of data, which covers the needs of most organisations.

The non-relational families

A document database stores each record as a self-contained document, usually JSON or a binary form of it. It fits *aggregates* in the domain-driven design sense: things read and written as a whole, such as a restaurant's menu with its sections, items and option groups. Reading one document is one lookup with no joins. The trade-off appears when the same data must be queried in another shape (all items containing peanuts across all menus) or changed in many documents at once; joins and multi-document transactions exist in modern products but are costlier and less natural than in SQL.

A key-value store is a distributed dictionary: put a value under a key, get it back. Because the store never looks inside values, it can be extremely fast and easy to partition. Redis and its open-source fork Valkey keep data in memory with optional persistence and add data structures (sorted sets, streams, geospatial indexes) and expiry; Amazon DynamoDB is a managed, partitioned key-value and document store. They suit sessions, carts, rate-limit counters, feature flags, caches and the latest position of each courier.

A wide-column store (Bigtable, Apache Cassandra, ScyllaDB, HBase) groups rows by a partition key and stores them sorted by a clustering key within each partition. It is designed for massive write volumes spread across many nodes and data centres. The discipline it demands is *query-first modelling*: you design one table per query, duplicating data freely, because queries that do not start from the partition key are expensive or impossible. Cassandra descends in part from Amazon's 2007 Dynamo paper, whose replication ideas appear in the next section but one.

A graph database stores nodes and relationships as first-class records and answers traversal queries such as “customers within three hops of this fraud ring” without the cascading joins that make such queries slow in SQL. Neo4j's Cypher is the best-known query language, and ISO published a standard graph query language, GQL (ISO/IEC 39075), in 2024. Graphs are excellent for fraud detection, recommendations and network analysis, and usually a poor primary store for transactional business data.

Time-series, search and vector stores

A time-series database is optimised for timestamped measurements that arrive in order and are rarely updated: metrics, sensor readings, prices, location history. It compresses heavily, partitions by time, downsamples old data (keep every point for a week, one-minute averages for a year) and deletes expired data cheaply by dropping whole time partitions. Examples include InfluxDB, Prometheus (for monitoring) and TimescaleDB, which is a PostgreSQL extension and shows how often a relational database can stretch to fit.

A search engine such as Elasticsearch, OpenSearch or Apache Solr (all built on the Apache Lucene library) maintains an *inverted index* from words to the documents containing them, with text analysis (stemming, synonyms, typo tolerance) and relevance ranking. It answers “vegan ramen near me” in milliseconds. It should almost never be a system of record: indexes are refreshed in near real time rather than transactionally, and the usual design feeds the search engine from the owning database and rebuilds the index from it when needed.

A vector database stores *embeddings*, lists of numbers that represent the meaning of text, images or products, and finds the items nearest to a query vector. Exact search compares the query with every vector, so these stores use approximate nearest-neighbour indexes such as HNSW graphs, trading a little recall for speed. They power semantic search, recommendations and retrieval for large language models, covered in retrieval-augmented generation. Dedicated products exist (Milvus, Qdrant, Pinecone, Weaviate), but for millions rather than billions of vectors, an extension to an existing database such as pgvector is often enough.

Distributed SQL

Distributed SQL databases, sometimes called *NewSQL* (a term coined by the analyst Matthew Aslett in 2011), keep the relational model, SQL and ACID transactions but partition and replicate data automatically across many nodes, often across regions. Google Spanner, described in a 2012 OSDI paper, uses tightly synchronised clocks (its TrueTime API, backed by GPS and atomic clocks) to give globally consistent transactions. CockroachDB, YugabyteDB and TiDB offer similar models on ordinary hardware; CockroachDB uses serializable isolation by default.

The price is physics and operations. A transaction that must be agreed by replicas in several regions waits for a consensus round trip, so single-row writes take milliseconds to tens or hundreds of milliseconds rather than well under a millisecond. Query planning across partitions is subtler, costs are higher, and fewer engineers know the failure modes. Distributed SQL is the right tool when you genuinely need relational transactions beyond one server's capacity or across regions (a global payments ledger, a multi-region bank), and an expensive one when a single PostgreSQL primary with replicas would do.

Comparison of database families

Family	Data model and query	Strengths	Weaknesses	Examples
Relational	Tables, keys, SQL with joins and aggregates	Transactions and constraints, flexible ad hoc queries, mature tooling, widely known	Vertical scaling limit for writes on one node; rigid schema needs migrations	PostgreSQL, MySQL, SQL Server, Oracle
Document	JSON-like documents, queries on fields and nested values	Natural fit for aggregates read as a whole; flexible structure	Joins and cross-document transactions weaker or costlier; schema enforced in code	MongoDB, Couchbase, Firestore
Key-value	Opaque value per key; get, put, delete	Very fast, simple, easy to partition	No queries across values; modelling burden on the application	Redis, Valkey, DynamoDB, etcd
Wide-column	Rows grouped by partition key, sorted by clustering key	Huge write throughput, linear horizontal scale, multi-data-centre	Tables built per query; limited joins and ad hoc queries	Apache Cassandra, ScyllaDB, HBase, Bigtable
Graph	Nodes and edges with properties; traversal queries	Multi-hop relationship queries	Harder to partition; smaller ecosystem	Neo4j, Amazon Neptune
Time-series	Timestamped points, tags; range and rollup queries	Compression, retention, downsampling	Poor at general updates and relationships	InfluxDB, TimescaleDB, Prometheus
Search engine	Inverted index over documents; relevance-ranked text queries	Full-text search, fuzzy matching, facets	Not a system of record; near-real-time refresh	Elasticsearch, OpenSearch, Apache Solr

Family	Data model and query	Strengths	Weaknesses	Examples
Vector	Embeddings; approximate nearest-neighbour search	Semantic similarity for retrieval and recommendations	Approximate results; index build cost; young tooling	pgvector, Milvus, Qdrant, Pinecone
Distributed SQL	Relational model, SQL, partitioned and replicated	Transactions and SQL with horizontal scale and multi-region survival	Higher write latency, cost and operational complexity	Google Spanner, CockroachDB, YugabyteDB, TiDB

Choosing: criteria that matter

Choose per data set, not per system. For each important data set, collect evidence on six criteria and write the decision in an ADR.

Data shape. Is it naturally tabular with many relationships, a self-contained aggregate, a network, a time series, free text or vectors? **Access patterns.** Which queries run, how often, with what latency targets, and how many are ad hoc? Read-heavy or write-heavy? Always by key, or by many attributes? **Consistency needs.** Does the data need multi-item transactions and constraints (balances, stock, bookings), or is a few seconds of staleness harmless (likes, recommendations)? See consistency models. **Scale.** Current and projected data size, writes per second and growth, measured, not guessed; a store that scales to petabytes is no advantage for 50 gigabytes. **Operations.** Who will run it: backups, upgrades, failover, monitoring, security patches? Is there a managed service in your cloud, and what does it cost at your volumes? **Team skills.** What does the team already know well enough to debug at 3 a.m.? Unfamiliar technology carries unknown failure modes.

Two habits improve the choice. Write down the five or ten most important queries with their target latencies *before* choosing, and test the candidate with realistic data volumes; many disappointments come from a store that was benchmarked on a laptop with a thousand rows. And separate the *system of record* from *derived stores*: the system of record needs the strongest guarantees and the most conservative technology, while derived stores (search indexes, caches, read models) can be specialised precisely because they can be rebuilt.

DECISION GUIDE

Decision guide: which kind of store for this data?

Ask these questions for each data set (not once for the whole system), in this order. Stop at the first *yes* that fits; if you reach the end, the relational default stands.

Q1 **Is this data a system of record that needs multi-row transactions, constraints or ad hoc queries (orders, payments, accounts, bookings)?**

- **Yes** Use a relational database. Only look further if you can show it cannot meet a measured scale or latency requirement.
- **No** Continue: the data may be derived, specialised or extremely high volume.

Q2 **Is it a copy derived from a system of record to serve one special query (full-text search, recommendations, similarity search)?**

- **Yes** Use the specialised store for that query (search engine, graph or vector index), fed from the system of record and rebuildable from it.
- **No** Continue.

Q3 **Is it read and written as a single item by key, with no queries across items (sessions, carts, latest positions, feature flags)?**

- **Yes** A key-value store fits, often in memory with expiry. Decide what happens if it is lost.
- **No** Continue.

Q4 **Is it an append-mostly stream of timestamped measurements queried by time range (metrics, sensor readings, location history)?**

- **Yes** A time-series database, or time-partitioned tables in a relational database at moderate volumes.
- **No** Continue.

Q5 **Will write volume or data size exceed what one well-tuned server can handle, with access always by a known key?**

- **Yes** A wide-column or key-value store designed for partitioning (Cassandra, DynamoDB), with tables designed around the queries.
- **No** Continue.

Q6 **Do you need relational transactions and SQL across regions or beyond one server's capacity?**

Yes Consider distributed SQL (Spanner, CockroachDB, YugabyteDB), accepting higher write latency and operating cost.

No Stay with the relational default, and revisit when a measured requirement changes.

IF YOU ARE NEW TO THIS

Step by step: the only copy, or a copy

This section's hardest idea is the difference between a system of record and a derived store, and why it decides how adventurous a database choice may be. Follow one menu edit through the food-delivery platform's stores.

- A restaurant changes the description of a dish, and the edit is stored in the menus PostgreSQL database. That database is the system of record: whenever another copy of a menu disagrees with it, its version wins.
- Change data capture picks up the committed change and passes it to the search engine, which updates its index a few seconds later. For those seconds a search can still show the old description, a lag the team has accepted.
- Now suppose the search cluster fails and its index is lost. Customers cannot search for a while, but no information has gone: the team rebuilds the index by reading every menu again from PostgreSQL.
- Imagine instead that restaurants' edits were saved only in the search engine. Its index is refreshed in near real time rather than inside transactions, and it may guard each write less carefully than the database does, so the same failure could lose edits with nothing to rebuild them from.
- That is why the platform can run a specialised store for search, chosen for that one job, while orders and menus stay on PostgreSQL, the database the team already runs well.
- **The lesson:** before choosing a store, ask whether it will hold the only copy of anything. If it will, choose the most conservative technology you can; if it holds a copy that can be rebuilt, you are free to specialise.

Polyglot persistence and its cost

Polyglot persistence means using different storage technologies for different data within one system, chosen by how the data is used. Martin Fowler popularised the term in 2011, crediting Scott Leberknight with first using it. It follows naturally from database per service: each owner

can pick the right tool.

The cost is easy to underestimate. Each additional database type needs its own expertise, backup and restore procedures, monitoring, capacity planning, security hardening, upgrade cycle and on-call knowledge. Data now moves between stores, so you need pipelines (CDC, events) and must reason about the lag and failure of each. Consistency between stores is eventual at best. For a small team, five databases can consume more effort than the product. A reasonable rule is to justify each new store by a requirement the existing ones demonstrably cannot meet, and to prefer managed services for the specialised ones.

WHERE THIS CAME FROM

Choose Boring Technology

In a 2015 essay, Dan McKinley, drawing on his years as an engineer at Etsy, argued that every company gets only a few *innovation tokens*, about three, to spend on unfamiliar technology. Mature, “boring” tools have known failure modes; new ones bring unknown unknowns. Each new technology also adds a permanent operational and cognitive cost to the whole organisation. His advice: ask first how you would solve the problem with what you already run, and treat adding a technology as a decision for the organisation, not one team. The essay is widely cited in database choices, where the cost of running something unfamiliar lasts for years.

WORKED EXAMPLE

Worked example: stores for the food-delivery platform

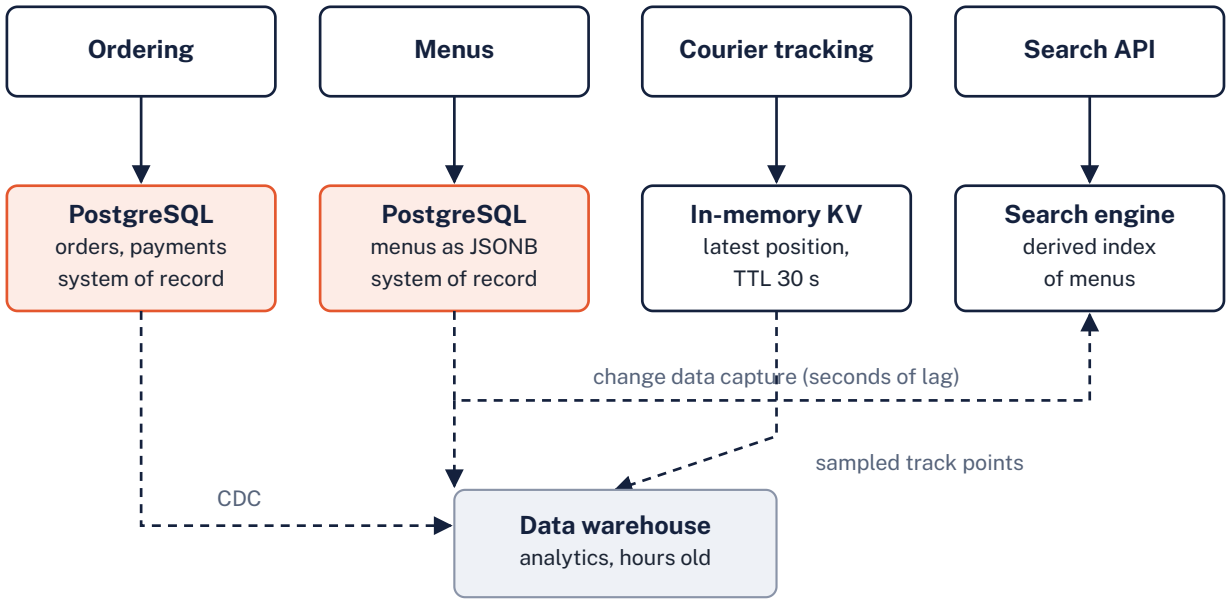
The platform operates in 40 cities. At the evening peak it takes about 300 orders a minute, has 20,000 couriers online, serves 50,000 restaurant menus and is searched about 2,000 times a second. The team of 35 engineers already runs PostgreSQL well.

Orders and payments	System of record; multi-row transactions (order, lines, payment authorisation); strict rules; ad hoc queries by support. About 5 new orders a second at peak (a few dozen row writes). **PostgreSQL**, with a replica for failover. No case for anything else.
Menus	Read-heavy, hierarchical (sections, items, option groups), edited by restaurants a few thousand times a day. Prices must be exactly what the order records. **PostgreSQL** with a relational core (restaurant, item, price) and `jsonb` for variable options. A document store would fit the shape, but would add a second technology without meeting any requirement PostgreSQL misses.
Courier locations	20,000 couriers each sending a position every 4 seconds is 5,000 writes a second, of which only the latest per courier matters for dispatch, and only for about 30 seconds. **In-memory key-value store** (Redis or Valkey) with geospatial queries and a 30-second expiry. Losing it on restart costs one update cycle. A sample of points (say one every 30 seconds) goes to the warehouse for analytics.
Restaurant and dish search	Full-text, typo-tolerant, filtered by location, open hours and dietary tags, ranked by relevance. **Search engine** (OpenSearch), fed from the menus database by change data capture, rebuildable from it. Accepts a few seconds of lag after a menu edit.
Analytics	Weekly revenue, delivery times by city, courier utilisation: large scans and aggregates over months of history. **Cloud data warehouse** fed by CDC and events, hours old by design. See analytics and streaming.
Rejected	A graph database for recommendations (not yet needed; revisit with evidence), a separate time-series database for locations (the warehouse sample answers the questions asked), and distributed SQL (the whole platform's orders, about 5 a second at peak, fit comfortably on one primary; there is no multi-region transaction requirement).

The result is two systems of record on one well-known technology plus three specialised, derived stores. Each extra technology is justified by a requirement the default cannot meet: 5,000 short-lived writes a second, and relevance-ranked text search.

Writing the queries and numbers down first made the decisions almost obvious, and gave the team a record of the assumptions to revisit when volumes change.

The resulting store landscape



Two systems of record; everything else is derived and can be rebuilt

Orders and menus have a single relational system of record each; the search index, the warehouse and the location store are derived or disposable.

Common mistakes

Résumé-driven choices: picking a store because it is new or because a large company uses it, without the requirement that justified it there. **Choosing for imagined scale**: designing for a billion users with a product that has a thousand, paying the complexity tax from day one. **Using a derived store as a system of record**: a search index or cache that becomes the only copy of data, with weaker durability than anyone realised. **Ignoring the query side**: picking a key-value or wide-column store and then discovering the business needs ad hoc reports and filters it cannot serve. **Underrating operations**: forgetting that someone must restore it from backup, upgrade it without downtime and diagnose it under load. Finally, **treating the choice as permanent truth**: record the assumptions (volumes, queries, team skills) so the decision can be revisited when they change.

CHECK YOUR UNDERSTANDING

▼ Why should a search engine usually not be the system of record?

Its index is refreshed in near real time rather than transactionally and its durability guarantees are weaker; it should be fed from the owning database and rebuilt from it when needed.

▼ Give two signs that a wide-column store may be a good fit.

Very high write volume beyond one server, and access that always starts from a known partition key so tables can be designed per query.

▼ What does an 'innovation token' represent in McKinley's essay?

One of a small number of chances a company has to adopt unfamiliar technology, each of which brings unknown failure modes and lasting operational cost.

▼ When is distributed SQL worth its cost?

When you need relational transactions and SQL beyond one server's capacity or across regions, such as a global ledger; not when a single primary with replicas meets the requirements.

SOURCES

- Dan McKinley: Choose Boring Technology (2015): https://mcfunley.com/choose-boring-technology
- Martin Fowler: PolyglotPersistence (2011): https://martinfowler.com/bliki/PolyglotPersistence.html
- Google Research: Spanner: Google's Globally-Distributed Database (OSDI 2012): https://research.google/pubs/spanner-googles-globally-distributed-database-2/
- Google Cloud: Spanner TrueTime and external consistency: https://cloud.google.com/spanner/docs/true-time-external-consistency
- DeCandia et al.: Dynamo: Amazon's Highly Available Key-value Store (SOSP 2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

SECTION 7.3 OF 7.8 · ADVANCED · 28 MIN READ

Transactions, ACID and isolation levels

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data

Terms it uses from earlier: SQL (Structured Query Language), Foreign key, Index, Linearisability

THE IDEA IN BRIEF

Imagine two cashiers sharing one paper ledger for a customer's account. Each looks up the balance, works out the new one on a notepad and writes it in. If they do this at the same moment, one cashier's change is lost. A database transaction is a rule about how cashiers take turns: whether they may peek at each other's half-written entries, whether the page can change while they are reading it, and whether one must wait for the other. Stricter rules mean fewer mistakes but more waiting, and the names for these rules mean slightly different things in different banks.

WHAT YOU NEED TO KNOW

- ACID: atomicity means all-or-nothing on failure; consistency means the application's invariants hold; isolation limits interference between concurrent transactions; durability means committed data survives crashes, usually through a write-ahead log and replicas.
- The SQL standard defines read uncommitted, read committed, repeatable read and serializable by the dirty reads, non-repeatable reads and phantoms they forbid; Berenson et al. (1995) showed these definitions are ambiguous and incomplete.
- Lost updates and write skew arise when transactions read data, decide in the application and write back; snapshot isolation prevents lost updates but allows write skew.
- MVCC keeps multiple row versions so that readers and writers do not block each other; long-running transactions bloat storage.
- Defaults and names differ: PostgreSQL and Oracle default to read committed, MySQL InnoDB to repeatable read, CockroachDB to serializable; Oracle's serializable is snapshot isolation.
- Protect read-modify-write paths with constraints, atomic conditional updates, SELECT ... FOR UPDATE (pessimistic) or version columns (optimistic), or use serializable isolation with whole-transaction retries.

What a transaction promises

A transaction groups several reads and writes into one unit that either takes effect completely (commits) or not at all (aborts and rolls back). Chapter 0 introduced the idea with a bank transfer. This section is about the part most developers misunderstand: what happens when many transactions run at the same time, which guarantees databases really give, and how to choose. The examples use SQL, but the reasoning applies to any store that offers transactions.

Transactions matter architecturally because they decide where business rules can be enforced cheaply. Inside one database, a transaction can keep a ledger balanced or stop a table being double-booked with a few lines of code. Across services there is no such tool, which is why data ownership and sagas are so important. Knowing exactly what a single database guarantees tells you which rules belong inside one owner's boundary.

ACID in detail

ACID was coined by Theo Härder and Andreas Reuter in 1983, building on Jim Gray's earlier work. Each letter is narrower than it sounds. **Atomicity** is about failure, not concurrency: if anything goes wrong part-way (a crash, a constraint violation, a deadlock, the application deciding to abort), every change made so far is undone, so the application can safely retry. *Abortability* would be a more accurate name.

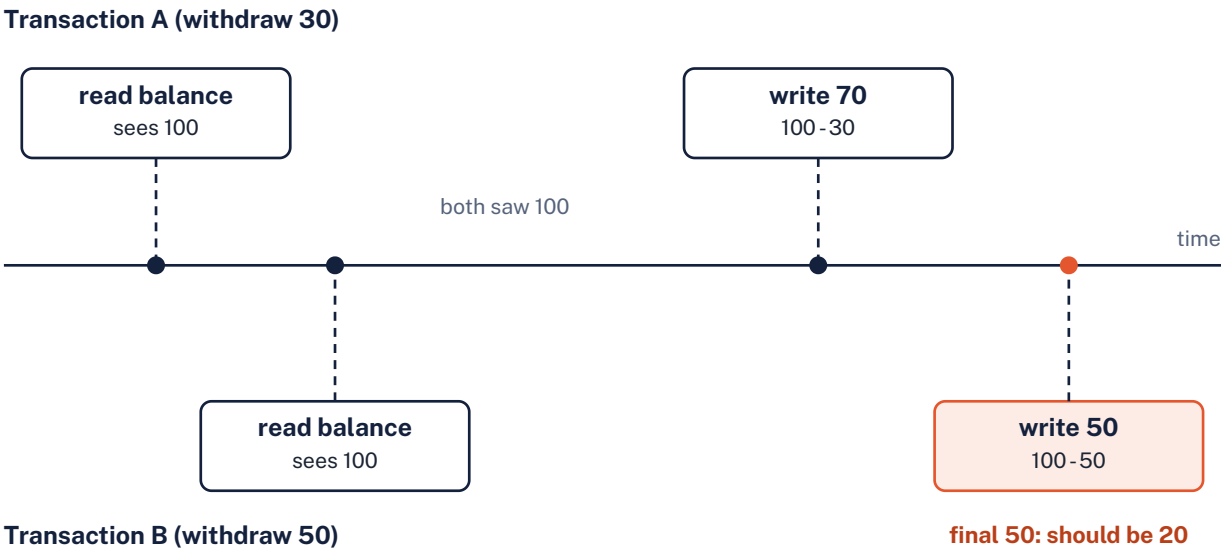
Consistency means that a transaction takes the database from one valid state to another, where "valid" means your invariants hold: debits equal credits, stock is never negative, every order has a customer. The database can enforce some invariants itself (keys, foreign keys, `CHECK` constraints, unique indexes), but most are business rules that only the application's transactions keep. Consistency is therefore a property of the application's use of atomicity and isolation, not something the database provides on its own. It is also unrelated to the "C" in CAP, which means linearisability (Chapter 6).

Isolation means that concurrently running transactions do not interfere with each other. In its strictest form, serializability, the result of running transactions concurrently is the same as if they had run one at a time in some order. Weaker levels allow some interference in exchange for performance, and the rest of this section is about exactly which interference. **Durability** means that once a commit is acknowledged, the data survives crashes. Single-node databases achieve it with a write-ahead log flushed to disk before the commit returns; replicated databases add copies on other machines. Durability is never absolute: a disk can lie about a flush, and asynchronous replication can lose the last few acknowledged writes when the leader fails (see replication).

What goes wrong without isolation

Concurrency bugs are hard to find because they depend on timing: they pass every test on a quiet developer machine and appear at the dinner-time peak. The clearest way to understand isolation levels is through the specific anomalies (also called phenomena) they allow or prevent. The lost update below is the most common in business code.

A lost update, step by step



Each transaction is correct on its own; interleaved, the second write silently erases the first withdrawal.

The anomalies, from mildest to subtlest

Anomaly	What happens	Food-delivery or banking example
Dirty write	A transaction overwrites data another transaction has written but not yet committed.	Two transactions assigning a courier interleave so the order row names one courier and the courier row a different order. Every real isolation level prevents this.
Dirty read	A transaction reads data written by another transaction that has not committed (and may roll back).	A report counts a payment of 500 that is then rolled back.
Non-repeatable (fuzzy) read	A transaction reads the same row twice and gets different values because another transaction committed a change in between.	Checkout reads a dish price of 9.50, a restaurant edit commits 11.00, and checkout's second read sees 11.00.
Read skew	A transaction sees parts of the database at different points in time, so related values do not add up.	A customer with 500 split across two accounts sees 450 in total because a transfer of 50 committed between reading the two balances.
Phantom read	A transaction re-runs a query with a condition and finds rows added or removed by another committed transaction.	Counting a courier's open deliveries twice returns 2 and then 3.
Lost update	Two transactions read a value, each computes a new one and writes it, and one write silently overwrites the other.	Two withdrawals from the same account (diagram above).
Write skew	Two transactions read an overlapping set of data, make decisions based on it, and update different rows, so together they break a rule each one checked.	Two on-call doctors each see that someone else is still on call and both go off call (worked example below).

The SQL standard isolation levels

The SQL standard (from SQL-92 onwards) defines four isolation levels by which of three phenomena they forbid: dirty reads, non-repeatable reads and phantoms. The standard makes serializable the default, but almost no mainstream product follows that; most default to read committed or repeatable read for performance.

The SQL standard levels and the phenomena they rule out

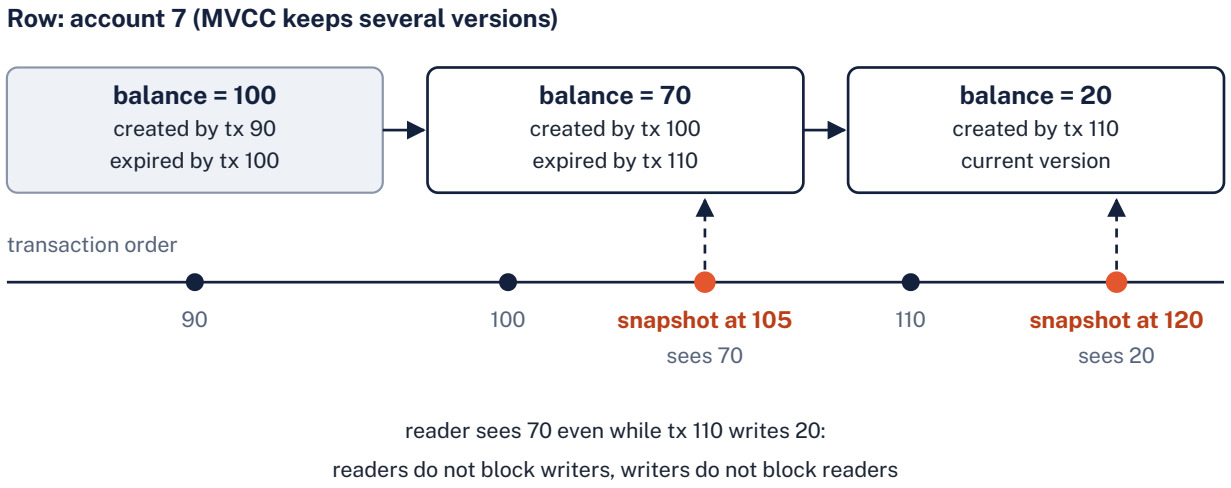
Isolation level	Dirty read	Non-repeatable read	Phantom read
Read uncommitted	possible	possible	possible
Read committed	prevented	possible	possible
Repeatable read	prevented	prevented	possible
Serializable	prevented	prevented	prevented, and the result must equal some serial order

In 1995, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil published *A Critique of ANSI SQL Isolation Levels*. They showed that the standard's English definitions were ambiguous, that forbidding the three phenomena does not by itself give serializability, and that the standard's levels failed to describe isolation levels that real products used. They added phenomena the standard missed, notably the dirty write, the lost update, read skew and write skew, and defined snapshot isolation: each transaction reads from a consistent snapshot of the committed data as of its start, and a transaction cannot commit a write to a row that a concurrent transaction has already changed and committed ("first committer wins"). Snapshot isolation prevents dirty reads, non-repeatable reads, read skew and lost updates, and the phantoms that affect reads, but it allows write skew, so it is not serializable. It fits awkwardly between the standard's levels, which is why products label it differently.

MVCC: how snapshots work

Most modern databases implement snapshot reads with multi-version concurrency control (MVCC). Instead of overwriting a row in place, a write creates a new version tagged with the writing transaction's identifier, and the old version is kept until no running transaction can still need it. Each reader is given a snapshot, essentially a list of which transactions had committed when it started, and sees only the versions those transactions created. The great benefit is that readers never wait for writers and writers never wait for readers; only two writers to the same row conflict.

Row versions under MVCC



Each update creates a new row version; a transaction sees the newest version that was committed before its snapshot.

The cost is housekeeping. Old versions must be cleaned up: PostgreSQL keeps them in the table and removes them with *vacuum*, while Oracle and MySQL InnoDB keep before-images in undo logs. A transaction left open for hours pins old versions, so tables and undo logs bloat and performance degrades. Long-running transactions are therefore an operational smell in any MVCC database, and a reason to keep analytical queries off the operational database. In **read committed** under MVCC, each *statement* gets a fresh snapshot; in **snapshot isolation**, the whole *transaction* uses one.

Real defaults, and why the names mislead

Isolation level names are not portable. The same word can mean different guarantees in different products, and the defaults differ. Martin Kleppmann's Hermitage project tested the actual behaviour of many databases against the anomalies above; vendor documentation confirms the defaults. Treat this table as a starting point and check the documentation for the version you run.

Same names, different behaviour: what real databases do (summarised from vendor documentation and the Hermitage tests)

Database	Default level	What the levels really are	Notes
PostgreSQL	Read committed	Read uncommitted behaves as read committed. Repeatable read is snapshot isolation. Serializable is serializable snapshot isolation (SSI).	Repeatable read aborts a transaction that tries to update a row changed since its snapshot (*could not serialize access due to concurrent update*), so lost updates are prevented; write skew is not.
MySQL (InnoDB)	Repeatable read	Plain SELECTs read a snapshot taken at the first read; locking reads, UPDATE and DELETE act on the latest committed rows with record and gap locks. Serializable turns plain SELECTs into locking reads when autocommit is off.	At repeatable read, a read-modify-write written as SELECT then UPDATE can still lose an update or produce write skew unless the SELECT uses FOR UPDATE.
Oracle Database	Read committed	Offers read committed and serializable; its serializable is snapshot isolation.	Prevents lost updates at serializable but allows write skew.
Microsoft SQL Server	Read committed (locking) on SQL Server; Azure SQL Database turns on read committed snapshot by default	Locking read committed, read committed snapshot, repeatable read (locking), snapshot (snapshot isolation) and serializable (locking).	On SQL Server, snapshot isolation must be enabled per database (ALLOW_SNAPSHOT_ISOLATION) before use.
CockroachDB	Serializable	Serializable by default; read committed is also available.	Clients must retry transactions that fail with a serialization error.

WATCH OUT

Two traps hidden in the names

First, “repeatable read” in PostgreSQL and in MySQL are different things: PostgreSQL's is true snapshot isolation that aborts conflicting updates, while MySQL's mixes snapshot reads with locking writes against the latest data and can lose updates written as read-then-write. Second, “serializable” in Oracle is snapshot isolation and allows write skew. Code that is correct on one database can be wrong on another with an identically named setting.

Achieving serializability

Databases that offer true serializability use one of three techniques. **Actual serial execution** runs transactions one at a time on a single thread, which is fast if every transaction is short and the data is in memory (VoltDB, and Redis scripts and transactions, work this way). **Two-phase locking (2PL)**, used by SQL Server's and MySQL's serializable levels, makes readers take shared locks and writers exclusive locks, including locks on ranges to stop phantoms, and holds them to the end of the transaction; it is correct but readers and writers block each other, throughput drops under contention and deadlocks are common. **Serializable snapshot isolation (SSI)**, described by Michael Cahill, Uwe Röhm and Alan Fekete in 2008 and used in PostgreSQL since version 9.1 (CockroachDB uses a related optimistic technique), runs transactions on snapshots without blocking, tracks read-write dependencies between them, and aborts one transaction when a dangerous pattern appears.

Whichever is used, serializable transactions can fail with a serialization error (SQLSTATE `40001` in PostgreSQL and CockroachDB) or a deadlock, and the application must retry the whole transaction. This is the hidden cost of the strongest level: every transactional code path needs a retry loop that is safe to repeat, which means no side effects such as sending emails or calling payment providers inside the transaction.

Pessimistic and optimistic concurrency control

In practice most applications run at read committed or repeatable read and protect specific read-modify-write paths explicitly. There are three tools. The best, when possible, is an **atomic update** that lets the database do the arithmetic and the check in one statement, such as `UPDATE accounts SET balance = balance - 30 WHERE id = 7 AND balance ⩾ 30`, then checking that exactly one row changed. Under PostgreSQL's read committed, a concurrent update makes the second statement wait and then re-evaluate its condition against the new row, so no update is lost.

When the decision needs logic the database cannot express, use pessimistic locking: lock the rows you read with `SELECT ... FOR UPDATE`, so concurrent transactions that want them wait until you commit. It is simple and suits short, highly contended operations (assigning one courier to one order). The costs are waiting, the risk of deadlocks (lock rows in a consistent order, such as by ascending id) and reduced throughput. Variants include `NOWAIT`, which fails immediately instead of waiting, and `SKIP LOCKED`, which skips rows already locked and is the standard way to build a reliable job queue on a relational table.

```
BEGIN;
-- Lock the row: other FOR UPDATE readers and writers must wait
SELECT balance FROM accounts WHERE id = 7 FOR UPDATE;
-- application checks: balance (100) ⩾ 30, so allow it
UPDATE accounts SET balance = 70 WHERE id = 7;
INSERT INTO ledger (account_id, amount, kind)
  VALUES (7, -30, 'withdrawal');
COMMIT;  -- lock released; the waiting transaction now sees 70
```

Pessimistic locking: the second withdrawal waits at the SELECT until the first commits, then reads the new balance.

Optimistic concurrency control takes no locks while the user or the code is thinking. Each row carries a version number (or a timestamp or hash). The writer updates only if the version is still the one it read, and increments it. If no row changes, someone else got there first, and the application reloads and retries or tells the user. This suits low-contention data and long “think time”: a restaurant manager editing a menu in a browser for five minutes must not hold a database lock all that time. HTTP has the same idea built in, with ETags and the `If-Match` header.

```
-- Read (no lock); remember version = 12
SELECT id, name, price, version FROM menu_items WHERE id = 501;

-- Later, write only if the version is unchanged
UPDATE menu_items
   SET price = 11.00, version = version + 1
 WHERE id = 501 AND version = 12;

-- Rows affected = 1: success.
-- Rows affected = 0: a concurrent edit won; reload and retry
-- or show the user the other change.
```

Optimistic concurrency with a version column: the update succeeds only if nobody changed the menu item since it was read.

WORKED EXAMPLE

Worked example: write skew with two on-call doctors

A hospital rota system enforces one rule: at least one doctor must be on call for each shift. Alice and Bob are both on call tonight. Both feel unwell and press “Go off call” at almost the same moment. Each request runs this logic in a transaction: count the doctors on call for the shift; if there are at least two, set my own row to off call. The database runs at snapshot isolation (PostgreSQL repeatable read or Oracle serializable).

Start	Table `doctors`: Alice on_call = true, Bob on_call = true. Rule: count(on_call) >= 1.
Step 1: T1 (Alice) begins	T1 takes a snapshot.
Step 2: T2 (Bob) begins	T2 takes a snapshot. Both snapshots show two doctors on call.
Step 3: T1 reads	`SELECT count(*) FROM doctors WHERE shift = 1234 AND on_call` returns 2. Check passes.
Step 4: T2 reads	The same query in T2's snapshot also returns 2. Check passes.
Step 5: T1 writes	`UPDATE doctors SET on_call = false WHERE name = 'Alice' AND shift = 1234`
Step 6: T2 writes	`UPDATE doctors SET on_call = false WHERE name = 'Bob' AND shift = 1234`. A different row, so no write-write conflict is detected.
Step 7: both commit	Snapshot isolation sees no conflict: each transaction wrote a row the other did not. Result: nobody is on call. Each transaction was correct in isolation; together they broke the rule.
Under serializable (SSI)	The database notices that each transaction read data the other then changed (a dangerous cycle of read-write dependencies). One commit fails with SQLSTATE 40001; the retry recounts, finds one doctor and refuses.
Fix at a weaker level	Lock the rows the decision depends on: `SELECT name FROM doctors WHERE shift = 1234 AND on_call FOR UPDATE`, then count the rows returned. Now T2 waits for T1's locks. At read committed, T2 then re-reads, finds only Bob on call and refuses; at PostgreSQL repeatable read it fails with a serialization error instead, and the retry refuses.

Write skew is a generalisation of the lost update: the transactions read the same data but write *different* rows, so snapshot isolation's first-committer-wins rule never fires.

The same pattern appears in booking systems (two people booking the last seat or overlapping meeting slots), usernames (two sign-ups checking that a name is free), and banking (two withdrawals from different accounts that together breach a joint overdraft

limit).

When the rows that would prove a conflict do not yet exist, as with two overlapping bookings, `FOR UPDATE` has nothing to lock. Options are serializable isolation, a unique or exclusion constraint (PostgreSQL can forbid overlapping time ranges declaratively), or *materialising the conflict*: create rows such as one per room per time slot, purely so they can be locked.

IF YOU ARE NEW TO THIS

Step by step: making a read-then-write safe

This section's hardest idea is that putting a read and a write inside one transaction does not stop another transaction slipping in between them. Two orders for one dish show why, and how to fix it.

- A restaurant's daily special has 8 portions left, stored in one row. Orders X (2 portions) and Y (3) arrive together, both at read committed, the default in PostgreSQL and Oracle.
- Each reads 8, subtracts its portions and saves: X saves 6, then Y saves 5. X's 2 portions have vanished from the count, a lost update, so 2 portions that do not exist could still be sold. Atomicity did not help: it is about failures, not taking turns.
- Fix one, an atomic update: the database does the arithmetic and the check in one statement, such as 'take 3 if at least 3 remain'. In PostgreSQL, Y's statement waits for X to commit, re-checks against 6 and leaves 3.
- Fix two, a pessimistic lock: each transaction locks the row as it reads it. X gets there first, so Y's read waits until X commits, and then sees 6, not 8.
- Fix three, optimistic concurrency: no lock, but each write states the version number it read, and succeeds only if the row still carries that number. X's write goes through and bumps the version; Y's matches no row, so Y reads again and retries with 6.
- Serializable isolation also prevents it, but may make one transaction fail, to be retried.
- **The lesson:** the danger lies in every read-decide-write, whatever the isolation level is called. Protect each one with an atomic statement, a lock, a version check or serializable isolation with retries.

Choosing an isolation strategy

There is no single right level; there is a right level for each piece of logic. A practical approach starts from the invariants, not from the settings.

Checklist: making a transactional code path safe

- [ ] **Name the invariant**
  Write the rule the code protects (balance never below zero, one courier per order, at least one doctor on call). If there is none, the default level is probably fine.
- [ ] **Prefer constraints and atomic statements**
  Unique indexes, foreign keys, CHECK and exclusion constraints, and single-statement conditional updates are enforced by the database whatever the isolation level.
- [ ] **Spot read-modify-write**
  Any code that reads a value, decides in the application and writes back is a lost-update or write-skew risk. Protect it with FOR UPDATE, a version column or serializable isolation.
- [ ] **Know your database's real semantics**
  Check what your product's level names actually mean (PostgreSQL and MySQL repeatable read differ; Oracle serializable is snapshot isolation).
- [ ] **Retry correctly**
  Serializable and repeatable read can abort with serialization errors and deadlocks. Retry the whole transaction with back-off, and keep external side effects outside it.
- [ ] **Keep transactions short**
  No user think time, network calls or large reports inside a transaction; long transactions hold locks and bloat MVCC storage.
- [ ] **Test under concurrency**
  Run the critical paths concurrently in an integration test against the real database engine, not an in-memory substitute with different semantics.

As a rule of thumb, read committed with explicit locking or version checks is a sound, fast default for most business applications, provided the team is disciplined about finding read-modify-write paths. Serializable is the safer default when the rules are complex, the team is large or the cost of a violated invariant is high (money, medicine, safety), and many teams now use it for the critical paths, accepting retries. Practitioners genuinely disagree here: some argue that serializable by default prevents a whole class of bugs that code review will never catch; others argue that its retries and contention cost too much and that explicit locking makes intent visible. Both positions are defensible if the choice is deliberate and tested.

CHECK YOUR UNDERSTANDING

▼ Why is snapshot isolation not serializable?

It allows write skew: two transactions can read overlapping data and write different rows, together breaking a rule each checked, without any write-write conflict being detected.

▼ What does the C in ACID depend on?

Mostly on the application: the database enforces declared constraints, but business invariants hold only if the application's transactions preserve them.

▼ When would you choose optimistic over pessimistic concurrency control?

When conflicts are rare or there is long think time between read and write, such as a user editing a form, so holding locks would be wasteful or impossible.

▼ Why must code using serializable isolation be safe to retry?

Serializable transactions can be aborted with serialization failures or deadlocks, and the correct response is to rerun the whole transaction, so it must have no external side effects.

SOURCES

- Berenson, Bernstein, Gray, Melton, O'Neil and O'Neil: A Critique of ANSI SQL Isolation Levels (1995): https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- PostgreSQL documentation: Transaction Isolation: https://www.postgresql.org/docs/current/transaction-iso.html
- MySQL 8.4 Reference Manual: Transaction Isolation Levels: https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html
- Microsoft Learn: SET TRANSACTION ISOLATION LEVEL (Transact-SQL): https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql
- Jepsen: Consistency Models: https://jepsen.io/consistency

SECTION 7.4 OF 7.8 · ADVANCED · 30 MIN READ

Replication, partitioning and sharding

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 2.5 Availability and reliability
- 6.6 CAP and consistency

Terms it uses from earlier: Availability, Latency, Index, Write-ahead log (WAL), Linearisability

THE IDEA IN BRIEF

Imagine a popular library. To keep serving readers if one building floods, it keeps identical copies of its catalogue in several branches: that is replication. The copies are updated by courier, so a branch may be a little behind, and a reader who has just returned a book at one branch may find another branch still lists it as on loan. When the collection becomes too big for any one building, the library splits it, A to M here and N to Z there: that is partitioning. Splitting well means no branch is overwhelmed, and readers rarely need to visit every branch for one answer.

WHAT YOU NEED TO KNOW

- Replication keeps copies of the same data on several nodes for availability, read scaling and locality; partitioning (sharding) splits data across nodes for write throughput and storage; real systems usually combine them.
- In leader-follower replication all writes go to one leader; synchronous followers protect acknowledged writes at the cost of latency, while asynchronous followers are fast but can lose recent writes on failover.
- Replication lag breaks read-your-writes, monotonic reads and consistent prefix reads; fixes include reading recent own writes from the leader, waiting for a log position and pinning users to one replica.
- Failover must detect failure, elect the most up-to-date node and fence the old leader; split brain, where two nodes accept writes, can cause long, painful recoveries, as GitHub's 2018 incident showed.
- Multi-leader replication must resolve write conflicts; leaderless (Dynamo-style) replication uses quorums with R + W > N, read repair and anti-entropy, but quorums alone do not guarantee linearisability.

- Range partitioning supports range scans but creates hot spots for time-ordered keys; hash partitioning spreads load but scatters ranges; consistent hashing or fixed logical partitions keep rebalancing cheap.
- Hot keys, skew and secondary indexes (local means scatter-gather reads; global means remote, usually asynchronous index writes) must be designed for from the most important queries.

Two different reasons to spread data

A single database server eventually runs out of something: availability (it is one machine that can fail), read capacity, write capacity, storage, or proximity to users on other continents. There are two complementary ways to spread data over several machines, and it is important not to confuse them. Replication keeps a *copy of the same data* on several nodes, for availability, read scaling and lower latency near users. Partitioning, called *sharding* in many products, *splits the data* so that each node holds a subset, for write throughput and storage beyond one machine. Real systems usually do both: each partition is replicated.

Both have serious consequences for correctness. Replication raises the question of what a reader sees while copies disagree, which is the subject of consistency models, CAP and PACELC. Partitioning raises the question of what happens to queries, indexes and transactions that span partitions. This section explains the mechanisms in enough depth to judge a database's claims and to design a shard key.

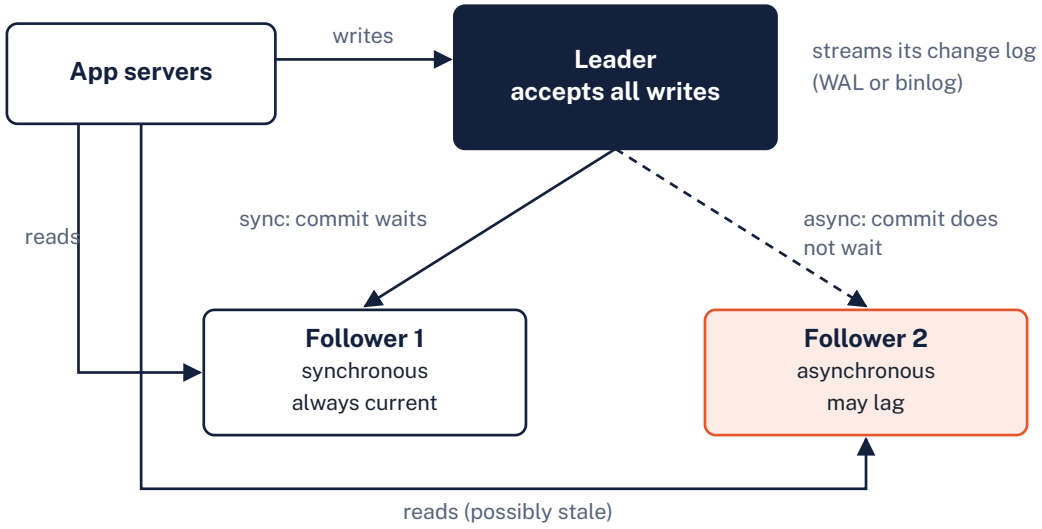
Leader-follower replication

The most common scheme is leader-follower replication (also called primary-replica; older documentation says master-slave). One node, the leader, accepts all writes. It records every change in its log (PostgreSQL's write-ahead log, MySQL's binary log) and streams that log to followers, which apply the changes in the same order. Reads can go to the leader or to followers. PostgreSQL streaming replication, MySQL replication, SQL Server availability groups, MongoDB replica sets and most managed cloud databases work this way.

The key design choice is when the leader reports a commit as successful. With **synchronous** replication it waits until a follower confirms it has the change, so a committed write survives the loss of the leader; the price is that every commit includes a network round trip, and if the synchronous follower is down, writes stall. With **asynchronous** replication the leader confirms immediately and ships the change afterwards; commits are fast and followers can be far away, but if the leader dies, the last changes that had not reached any follower are lost even though the client was told they were committed. A common compromise, called semi-synchronous in MySQL, waits for one follower and lets the rest be asynchronous. In the bank, losing an

acknowledged payment is unacceptable, so at least one synchronous copy (or a consensus-based database) is normal; for the food-delivery menu database, a few seconds of potential loss may be an acceptable price for speed. This is the RPO trade-off from availability and reliability in concrete form.

Leader with a synchronous and an asynchronous follower

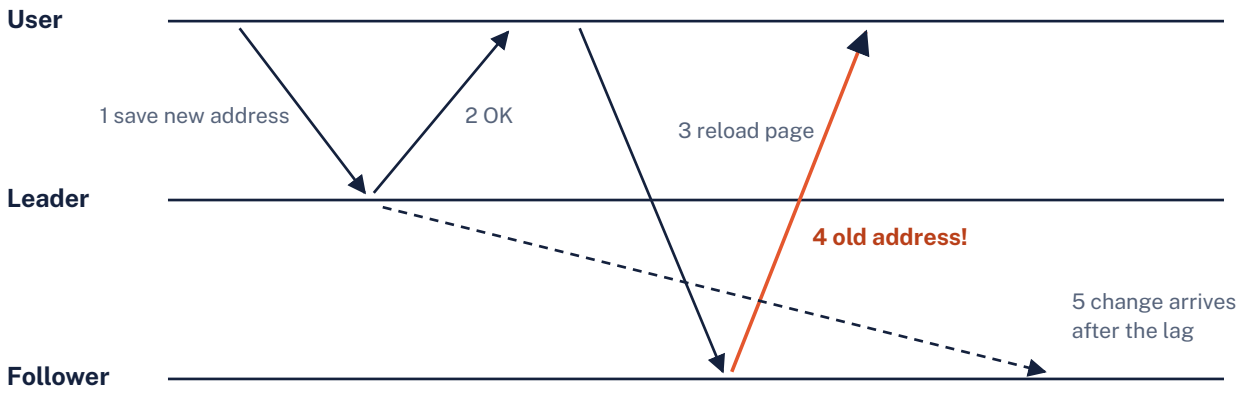


All writes go to the leader; its change log flows to followers, and the commit waits only for the synchronous one.

Replication lag and the anomalies it causes

Asynchronous followers are usually milliseconds behind, but under load, during a large batch update or across a slow network, the replication lag can grow to seconds or minutes. Reading from followers then produces anomalies that users notice. Douglas Terry and colleagues defined the first two guarantees below, read-your-writes and monotonic reads, as *session guarantees* in 1994; the third matters most when data is also partitioned.

Read-your-writes violated



Fix: read the user's own recent changes from the leader, or wait for the follower to catch up

The user's own change is acknowledged by the leader but not yet applied on the follower that serves her next read.

Lag anomalies and their fixes

Guarantee	What goes wrong without it	Common fixes
Read-your-writes (read-after-write)	A customer changes her delivery address, reloads, and sees the old one: it looks as if the save failed, so she tries again.	Read data the user may have just changed from the leader (for example, for one minute after their last write); or record the leader's log position at write time and read from a follower only once it has applied that position.
Monotonic reads	A courier's app refreshes twice; the first read hits an up-to-date follower and shows an order as assigned, the second hits a lagging one and the order seems unassigned again. Time appears to go backwards.	Route each user's reads to the same follower (for example, by hashing the user id), failing over only when it is unavailable.
Consistent prefix reads	Readers see an answer before the question, or an order's “delivered” event before “picked up”, because the two were written to different partitions that lag differently.	Write causally related data to the same partition, or track causal dependencies explicitly.

None of these anomalies exists if every read goes to the leader, so the simplest fix is often to send reads that must be fresh to the leader and use followers only for reads that tolerate staleness: product listings, reports, search feeds. Design this deliberately per query instead of routing “all reads to replicas” by default.

Failover and split brain

Failover replaces a failed leader with a follower. It has three steps: detect that the leader has failed (usually by a timeout), choose the new leader (ideally the most up-to-date follower, often by a consensus protocol such as Raft among the nodes or an external coordinator such as etcd or ZooKeeper), and redirect writes to it. Each step can go wrong. A timeout that is too short fails over during a brief network blip or load spike, which makes things worse; one that is too long extends the outage. With asynchronous replication, writes the old leader had not yet shipped are lost, and if the old leader comes back and still holds them, they conflict with new writes.

The most dangerous failure is split brain: two nodes both believe they are the leader and both accept writes, so the data diverges in ways that may be impossible to merge automatically. It typically follows a network partition in which the old leader is cut off but alive. Protection needs a quorum (a node may lead only while a majority agrees), *fencing* (the new leader's authority is enforced by an ever-increasing token that storage checks, or the old leader is forcibly powered off, a technique nicknamed STONITH: shoot the other node in the head), and clients that follow the current leader reliably.

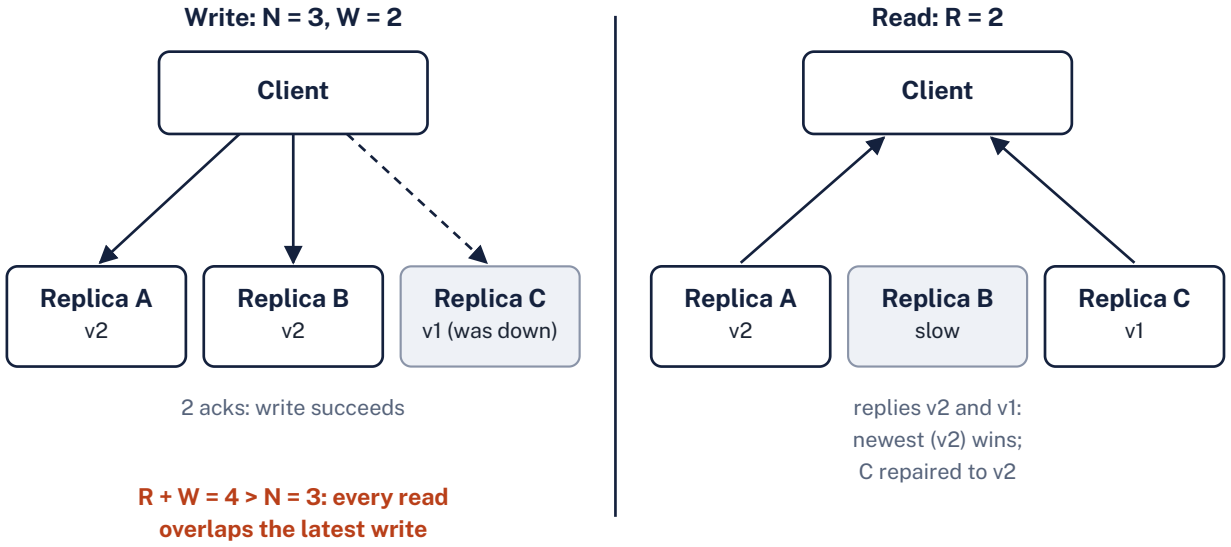
GitHub's incident of 21 October 2018 shows how hard this is in practice. A network fault cut the connection between its US East Coast hub and its main East Coast data centre for 43 seconds. Its failover tooling, working as designed, promoted West Coast database primaries. When connectivity returned, both coasts held writes the other lacked, so GitHub could not simply fail back; to protect data integrity it chose to restore and reconcile, and service was degraded for 24 hours and 11 minutes. Automatic failover across regions had turned a 43-second blip into a day-long recovery.

Multi-leader and leaderless replication

In multi-leader replication several nodes accept writes and replicate to each other. It is used for multiple data centres (each region writes locally, so users avoid cross-ocean latency), for clients that work offline (a calendar or notes app is effectively a leader on each device) and for collaborative editing. The unavoidable problem is **write conflicts**: two leaders change the same record concurrently. Resolutions include last write wins (simple, but silently discards data and depends on clocks), merging values, keeping both and asking a person, or using data types designed to merge automatically (CRDTs, conflict-free replicated data types). The most reliable approach is to avoid conflicts: give each record a home region that alone accepts its writes, such as a customer's home country.

Leaderless replication, popularised by Amazon's Dynamo paper (DeCandia and colleagues, 2007) and used in Apache Cassandra, ScyllaDB and Riak, has no leader at all. The client, or a coordinator node acting for it, sends each write to all N replicas of a key and treats it as successful when W of them acknowledge; it sends each read to replicas and waits for R answers, taking the newest version. If R + W > N, every read set overlaps every successful write set in at least one replica, so a read normally sees the latest acknowledged write. Replicas that missed writes are repaired by *read repair* (the coordinator writes back the newest version when it sees a stale one) and background *anti-entropy* that compares replicas. Cassandra exposes these choices per request as consistency levels such as ONE, QUORUM and ALL.

A quorum write and read with N = 3



With W = 2 and R = 2, at least one replica in every read has seen the latest acknowledged write.

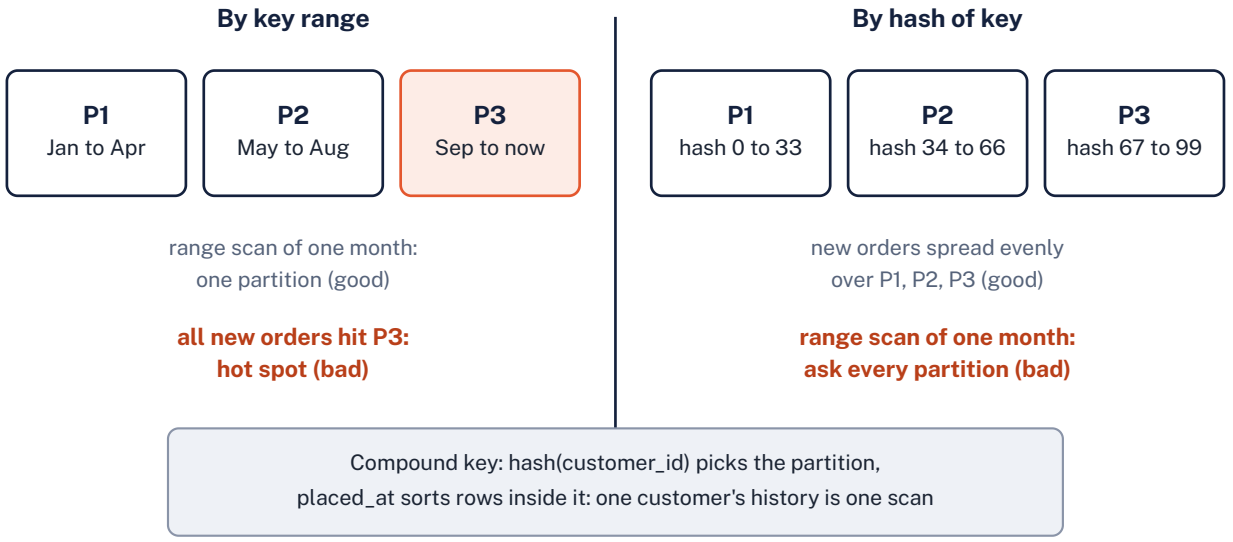
Quorums are often over-trusted. R + W > N does not by itself make the system linearisable: concurrent writes, writes that failed on some replicas but succeeded on others, *sloppy quorums* (writing to substitute nodes during failures) and last-write-wins with skewed clocks all allow stale or lost data. Chapter 6 works through quorum arithmetic and these edge cases in CAP and PACELC. Leaderless stores are excellent for high write availability across failures and data centres; they are a poor fit for invariants such as “never sell the same seat twice” unless they offer a lightweight transaction or consensus feature for those operations.

Partitioning: by key range or by hash

Partitioning assigns each record to one partition by its **partition key** (or shard key), and the aim is to spread data and load evenly. A partition that receives disproportionate load is a hot spot, and an uneven spread is called *skew*. There are two basic schemes.

Key-range partitioning gives each partition a contiguous range of keys, like volumes of an encyclopaedia. Range scans are efficient because adjacent keys are stored together (all orders for October in one or two partitions), and partitions can be split when they grow, as in Bigtable, HBase, Spanner and MongoDB's range sharding. The danger is a key that increases with time: if the key is a timestamp or a sequential order number, every new write lands on the last partition, which becomes a hot spot while the others sit idle. **Hash partitioning** applies a hash function to the key and assigns ranges of hash values to partitions, spreading even sequential keys uniformly. The price is that range queries on the key now have to ask every partition. Many stores combine the two with a *compound key*: Cassandra hashes the partition key to choose the node and sorts rows within the partition by clustering columns, so `(customer_id, placed_at)` puts one customer's orders together, in time order, on one node.

Range versus hash partitioning for orders



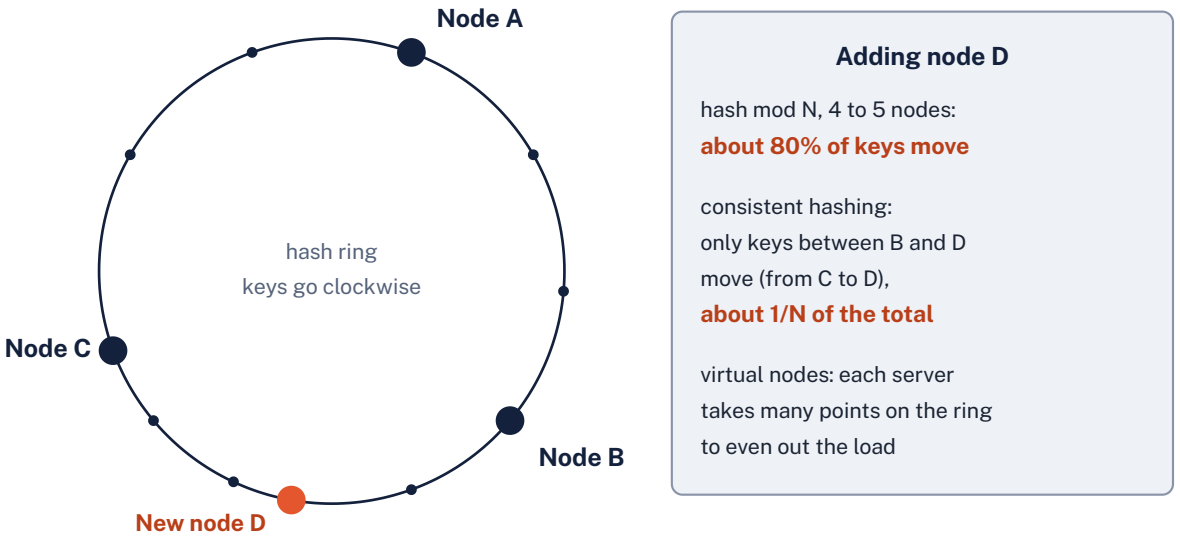
Range partitioning keeps ranges together but concentrates time-ordered writes; hashing spreads writes but scatters ranges.

Consistent hashing and rebalancing

The obvious way to map hashes to nodes, `hash(key) mod N`, has a fatal flaw: when N changes, almost every key moves. Going from 4 to 5 nodes relocates about 80% of the keys (any key whose remainder differs for 4 and 5), and from 10 to 11 about 91%, which means copying almost the whole data set just to add one machine. Consistent hashing, introduced by David Karger and colleagues in 1997 for web caches, places both nodes and keys on a circular hash space; each key belongs to the next node clockwise. Adding a node takes over only the keys between it and its predecessor, about 1/N of the total, and removing one hands its keys to its neighbour.

With few nodes the arcs are uneven, so each physical node is given many *virtual nodes* (tokens) at different points on the ring, which evens out the load and lets a new node take a little from every existing node. Dynamo and Cassandra use this scheme.

Consistent hashing on a ring



Keys belong to the next node clockwise, so a new node takes over one arc instead of reshuffling everything.

Two other rebalancing strategies are common. With a **fixed number of partitions**, you create many more partitions than nodes at the start (say 1,024 for 10 nodes) and assign whole partitions to nodes; adding a node moves a few whole partitions and never re-hashes keys. Elasticsearch shards, Couchbase vBuckets and many hand-built sharding schemes use this, and the main decision is choosing a partition count large enough for future growth. With **dynamic partitioning**, as in HBase and MongoDB, a partition splits when it exceeds a size threshold and the halves can move to different nodes. Whatever the scheme, rebalancing moves large amounts of data over the network while the system is serving traffic, so many teams prefer that a human approves each move, and they throttle it to protect latency.

Requests also need routing to the right partition: either any node accepts a request and forwards it (Cassandra nodes learn the ring by gossip), or a routing tier knows the partition map (MongoDB's `mongos`), or the client library holds the map. The map itself is usually kept in a coordination service such as etcd or ZooKeeper or in the database's own consensus group.

IF YOU ARE NEW TO THIS

Step by step: adding a node without moving everything

This section's hardest idea is why the obvious way of sharing keys among nodes falls apart when a node is added, and how consistent hashing avoids it. Follow the food-delivery platform's orders as a cluster grows.

- A hash function turns each order's key into a number. Suppose, to keep it small, the numbers run from 0 to 11, and an order goes to node 'hash mod 3', the remainder after dividing by 3. Order 7 lives on node 1.
- A fourth node is added and the rule becomes 'hash mod 4'. Order 7 now belongs on node 3.
- Of the twelve numbers, only 0, 1 and 2 give the same remainder for 3 and 4. The other nine orders, 75% of them, must be copied to another node just to add one machine.
- Consistent hashing instead places nodes and keys on a circle of hash values, and each key belongs to the next node clockwise. In the circle drawn above, new node D sits between B and C.
- Only keys on the arc between B and D change owner, moving from C to D. Every other key stays put, so about one node's share of the data moves, not most of it.
- With few points the arcs are uneven and D relieves only C, so each server gets many points round the circle, called virtual nodes, and a new server takes a small slice from every existing one.
- **The lesson:** judge a placement rule by what happens when the cluster changes, not only by how evenly it spreads keys today. Consistent hashing, or many partitions created up front and moved whole, keeps the move small when a node joins.

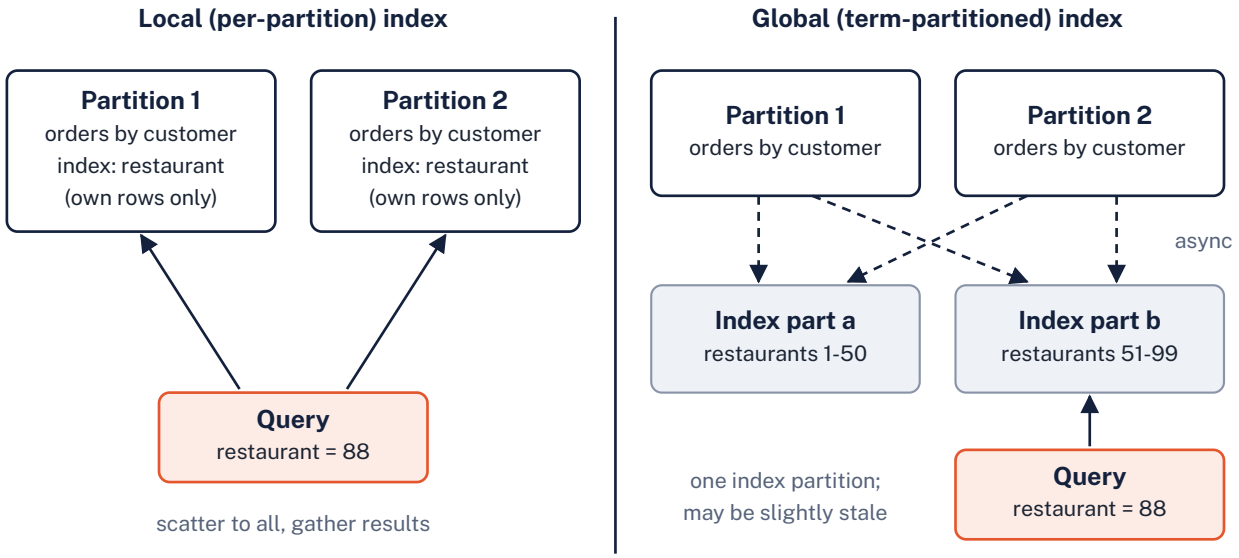
Hot keys and skew

Hashing spreads *keys* evenly, not *load*. If one key is far more active than the rest, its partition is hot whatever the scheme: a celebrity's account in a social network, a national chain's flagship restaurant during a promotion, or a bank's own settlement account that touches every payment. Managed stores make the limits explicit; Amazon DynamoDB, for example, documents a ceiling per partition of 3,000 read units and 1,000 write units a second. Mitigations include caching hot reads, splitting a hot key by appending a small random suffix (writes to `restaurant:88#0` to `restaurant:88#9`, at the cost of reading all ten and combining), moving a very large tenant to dedicated partitions, and aggregating counters in memory before writing them. All of them add complexity, so first measure which keys are actually hot.

Secondary indexes across partitions

Partitioning works smoothly for lookups by the partition key. Queries by anything else need a secondary index, and there are two ways to partition one. A **local index** (document-partitioned) lives with each partition and covers only that partition's rows. Writes update a single partition, but a query by the indexed attribute must be sent to every partition and the results merged (*scatter-gather*), so its latency is set by the slowest partition and its cost grows with the number of partitions. A **global index** (term-partitioned) is itself partitioned by the indexed value, so a query reads one index partition; but a single write may now need to update index partitions on other nodes, which is usually done asynchronously, so the index can briefly lag the data. DynamoDB shows both: a local secondary index shares the table's partition key, while a global secondary index has its own and is eventually consistent.

Local and global secondary indexes



A local index makes writes cheap and queries scatter; a global index makes queries targeted and index updates remote and usually asynchronous.

WORKED EXAMPLE

Worked example: sizing and choosing a shard key for orders

The food-delivery platform has grown to 30 countries and 10 million orders a day, with the evening peak at three times the average rate. Each order generates about 10 writes over its life (creation, lines, status changes, payment references) and occupies about 4 KB including indexes. Customers can view their full order history; regulation requires orders to be retrievable for six years, but support staff need online access only to the last two.

Write rate	10,000,000 / 86,400 s = about 116 orders a second on average; peak about 350 a second; at 10 writes each, about 3,500 writes a second at peak. A well-tuned single primary can handle this; throughput alone does not force sharding.
Storage growth	10,000,000 x 4 KB = 40 GB a day, about 14.6 TB a year, about 29 TB for two years online.
Do we need to shard?	If only 90 days had to be online (3.6 TB), one primary plus replicas, with time-based table partitioning and archiving to the warehouse, would be the boring and correct answer. With two years online at 29 TB, restore and replica rebuild times on one node become unacceptable, so the team shards.
Size the shards	Target about 2 TB per shard so a replica can be rebuilt in hours: 29 TB / 2 TB = about 15, so 16 shards, each a leader with two followers. Create 1,024 logical partitions mapped onto the 16 shards so growth means moving whole partitions, not re-hashing. Peak writes per shard: 3,500 / 16 = about 220 a second.
Candidate: order_id (hash)	Perfectly even writes, but a customer's history and a restaurant's live queue both become scatter-gather across 16 shards. Rejected as the primary key for placement.
Candidate: placed_at (range)	Great for time-range reports, but every new order writes to the newest partition: a guaranteed hot spot. Rejected.
Candidate: restaurant_id	A restaurant's queue is local, but large chains and promotions create hot keys, and customer history scatters. Rejected.
Candidate: country or city	Supports data residency and local queries, but one large city may carry 15% of all orders: skew. Kept only as a coarse first level for residency (EU and non-EU clusters), not as the shard key.
Decision	Shard by `hash(customer_id)` with rows sorted by `placed_at` inside each partition. The order id embeds the customer's partition number so a lookup by order id goes straight to one shard.

Covering the other queries	The restaurant's live queue is served by a separate read model keyed by restaurant and built from order events (a deliberate, eventually consistent global index). Analytics run in the warehouse. Cross-customer transactions are not needed: each order touches one customer.

The numbers changed the question. Throughput never required sharding; data retained online did. Shrinking the online window would have removed the need entirely, which is worth raising with the business before building a sharded system.

The shard key was chosen from the most important queries and their consistency needs, not from the data model alone. Every other query was given an explicit, separately designed path.

Choosing a shard key is one of the hardest decisions to reverse: changing it later means moving every row. Record it in an ADR with the volumes and queries it assumed.

Common mistakes

Treating replicas as backups. A replica faithfully copies a mistaken `DELETE` within milliseconds; backups and point-in-time recovery are still needed (Chapter 0). **Reading from replicas by default** without deciding which queries tolerate lag. **Trusting automatic failover that has never been rehearsed**, or that can promote a node while the old leader still accepts writes. **Sharding too early**: a sharded database makes joins, transactions, schema changes and operations harder, and modern single nodes are very large; exhaust indexing, caching, read replicas, archiving and vertical scaling first. **Sharding by a monotonically increasing key**, which concentrates every insert on one partition. **Forgetting secondary access paths**, discovering after the split that an important query must now ask every shard. As the rest of this chapter shows, the good alternatives often involve derived data: read models, caches and analytical stores, each built for one kind of query.

CHECK YOUR UNDERSTANDING

▼ **Why does asynchronous replication risk losing writes that clients were told had committed?**

The leader acknowledges before any follower has the change; if it fails before shipping it, the promoted follower never received those writes.

▼ **A user saves a change and immediately sees the old value. Which guarantee is broken, and what is one fix?**

Read-your-writes. Serve that user's reads of recently changed data from the leader, or wait until the follower has applied the write's log position.

▼ **Why is hash mod N a poor way to assign keys to nodes?**

Changing N moves most keys (about 80% when going from 4 to 5 nodes), whereas consistent hashing or fixed logical partitions move only about 1/N.

▼ **What is the trade-off between local and global secondary indexes?**

Local indexes keep writes on one partition but force queries to ask every partition; global indexes make queries target one index partition but require cross-partition, usually asynchronous, index updates.

SOURCES

- DeCandia et al.: Dynamo: Amazon's Highly Available Key-value Store (SOSP 2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- GitHub: October 21 post-incident analysis (2018): https://github.blog/2018-10-30-oct21-post-incident-analysis/
- PostgreSQL documentation: High Availability, Load Balancing, and Replication (log-shipping standby servers): https://www.postgresql.org/docs/current/warm-standby.html
- Amazon DynamoDB Developer Guide: Best practices for designing and using partition keys effectively: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html
- Microsoft Azure Architecture Center: Sharding pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding

SECTION 7.5 OF 7.8 · ADVANCED · 26 MIN READ

CQRS and event sourcing

BEFORE YOU READ THIS

This section builds on:

- 5.7 Domain-driven design
- 6.4 Event-driven
- 7.3 Transactions

Terms it uses from earlier: Aggregate, Bounded context, Eventual consistency, Optimistic concurrency control

THE IDEA IN BRIEF

A bank keeps a ledger in which every deposit and withdrawal is written down, one line after another, and nothing is ever rubbed out; mistakes are fixed by adding a correcting line. Your balance is simply the total of all the lines. Your monthly statement is a tidy summary prepared from that ledger for easy reading. Event sourcing stores data the way the ledger does: as a list of things that happened. CQRS keeps the ledger (for making changes) separate from the statements (for looking things up). Both are powerful, and both add real complexity.

WHAT YOU NEED TO KNOW

- Command-query separation (Bertrand Meyer, 1988) says a method should either change state or return an answer, not both; CQRS (Greg Young, around 2010) applies the same split to whole models.
- In CQRS a write model enforces business rules on commands, while projections keep one or more denormalised read models up to date for queries; read models are derived data that can be rebuilt.
- Event sourcing stores every change as an immutable, past-tense event in an append-only stream per aggregate; current state is a deterministic left fold of the events.
- Event sourcing gives a complete audit trail, temporal queries, rebuildable and retroactive read models, and replay for debugging; snapshots shorten long folds but are only a cache.
- The costs are real: complexity, eventual consistency of read models, event versioning by upcasting or new event types, and erasing personal data from an immutable log, usually by crypto-shredding.

> - Martin Fowler warns that CQRS adds risky complexity for most systems; apply either pattern to specific bounded contexts with genuinely different read and write needs, never to a whole system by default.

From command-query separation to CQRS

The idea starts small. In *Object-Oriented Software Construction* (first edition 1988), Bertrand Meyer proposed command-query separation: every method should either be a command that changes state and returns nothing, or a query that returns an answer and changes nothing. Asking a question should never change the answer. Martin Fowler points out that the rule has sensible exceptions (popping an item off a stack both changes the stack and returns the item), but as a default it makes code easier to reason about, because queries can be called freely, in any order, without side effects.

Around 2010 Greg Young lifted the principle from methods to whole models and called the result Command Query Responsibility Segregation (CQRS). Instead of one object model, and usually one set of tables, serving both updates and reads, the system has a **write model** that handles commands and enforces business rules, and one or more **read models** shaped for the questions users ask. The insight is that writing and reading are different jobs. Writing needs strict rules, small transactions and a normalised structure so that a fact is stored once. Reading needs speed, denormalised shapes that match screens and reports, and often far more traffic. A single model that tries to do both well tends to do both badly once the domain becomes rich.

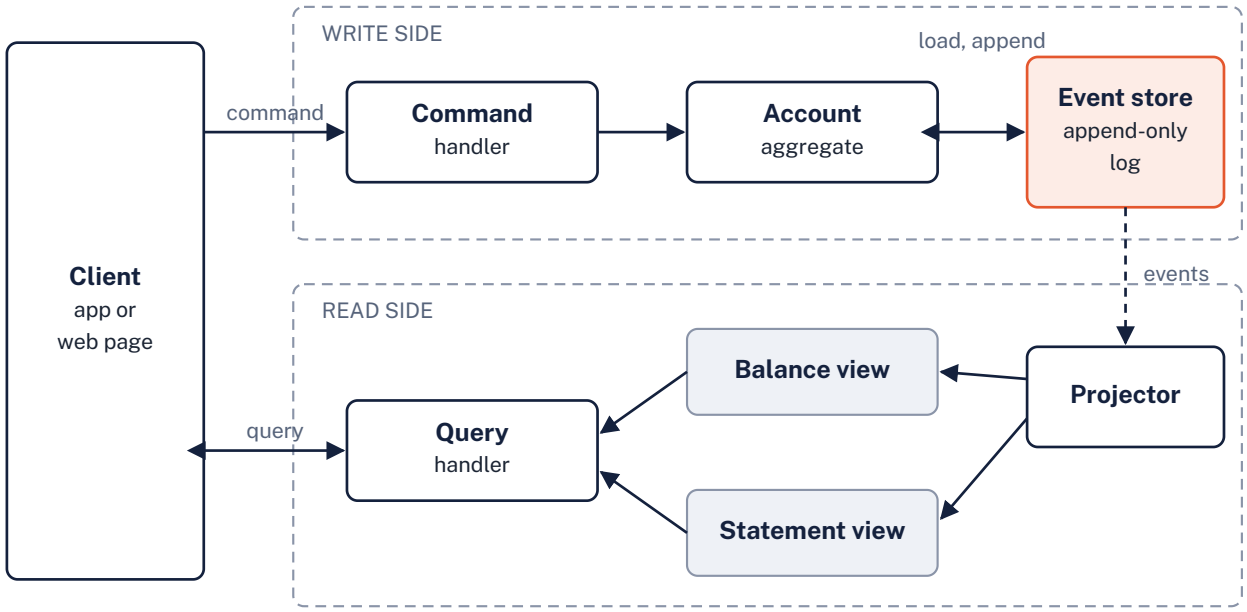
What CQRS actually changes

On the write side, a command such as `WithdrawMoney(account 42, £50)` arrives at a command handler. The handler loads the relevant aggregate (the cluster of data that must stay consistent inside one transaction, a term from domain-driven design covered in Section 5.7), asks it to apply the business rules, and saves the result. Commands are requests and can be refused: the account may be frozen or the funds insufficient. On the read side, a query such as "show me the last 30 transactions with running balances" is answered from a read model built for exactly that screen, with no joins, no business logic and no locks on the write tables.

Something must keep the read models up to date. That job belongs to a projection: code that listens to changes on the write side and updates one read model. A projection may maintain a table of balances, a search index of transactions, a monthly statement document and a fraud-scoring feature store, each from the same stream of changes. Read models are therefore **derived data**: they can always be thrown away and rebuilt from the write side. That property is what makes it safe to add a new read model next year, or to change the shape of an old one.

CQRS comes in degrees, and the lightest degree is often all a system needs. At the lightest, the write and read models are different classes over the same database, with reads served by views or dedicated query code. In the middle, the read models live in separate tables updated in the same transaction as the write. At the heaviest, the read models live in different stores (a search engine, a document database, a cache) and are updated asynchronously from events, which brings scale and flexibility but also eventual consistency. CQRS does not require event sourcing, a message broker or microservices; those are separate decisions that are often, but not necessarily, made together.

CQRS with event sourcing



Commands and queries take different paths; the event store is the source of truth and every read model is derived from it by a projector.

IF YOU ARE NEW TO THIS

If you are new to this

Think of a bank's ledger and a bank statement. The ledger is where every deposit and withdrawal is written, one line at a time, under strict rules. The statement is a summary prepared from the ledger in a format that is easy to read. You never write on the statement to move money, and you never read the ledger line by line to find your balance. CQRS gives software the same split: one model for changing things, other models for looking things up. Event sourcing goes one step further and makes the ledger itself the only thing that is permanently stored.

Event sourcing: state as a sequence of facts

Most systems store current state. When Asha withdraws £120, the `accounts` row changes from £500 to £380 and the old value is gone unless someone also writes an audit row. Event sourcing, described by Martin Fowler in 2005, turns this round: the system stores every change as an immutable event in an append-only log, and current state is whatever you get by replaying the events in order. The events are the source of truth; the balance is a calculation.

Events are named in the past tense because they record facts that have already happened and been accepted: `AccountOpened`, `MoneyDeposited`, `MoneyWithdrawn`. A command (`WithdrawMoney`) is a request that may be refused; an event (`MoneyWithdrawn`) can no longer be refused, only compensated by a later event. Each aggregate has its own event stream, numbered from version 1, and the store guarantees that events in a stream are kept in order.

Rebuilding state is a **left fold**: start with nothing, apply the first event, apply the second to the result, and so on. The apply function must be pure and deterministic: it may not call a clock, a random-number generator or another service, because replaying the same events must always give the same state. Any value that came from outside (an exchange rate, a timestamp, a fraud score) is recorded inside the event when it is first decided.

Writing follows a fixed cycle. The command handler loads the stream, folds it into current state, runs the business rule, and appends the new events with the version it expects to write. If another command appended to the same stream in the meantime, the append fails and the handler retries with fresh state. This is the optimistic concurrency of Section 7.3, applied per stream, and it is why two simultaneous withdrawals cannot both spend the same £100.

WORKED EXAMPLE

Worked example: a bank account as events

Asha's current account is stream `account-42`. Amounts are stored in pence, as Section 0.2 recommends for money. Five events have been appended; then she tries to withdraw £200.

v1 AccountOpened (overdraft limit £0)	State: balance £0.00, version 1
v2 MoneyDeposited £500.00	£0.00 + £500.00 = £500.00, version 2
v3 MoneyWithdrawn £120.00	£500.00 − £120.00 = £380.00, version 3
v4 MoneyDeposited £75.50	£380.00 + £75.50 = £455.50, version 4
v5 MoneyWithdrawn £300.00	£455.50 − £300.00 = £155.50, version 5
Command: WithdrawMoney £200.00	Rule: balance minus amount must not go below minus the limit. £155.50 − £200.00 = −£44.50, below £0. Command refused; no event is appended.
Command: WithdrawMoney £20.00	£155.50 − £20.00 = £135.50 is allowed. Handler appends v6 MoneyWithdrawn £20.00, expecting version 6 to be free.
Temporal query: balance after v3	Fold v1 to v3 only: £380.00. No history table was needed.
Audit question: why is the balance £135.50?	The six events are the complete answer, in order, with who, when and why recorded in each.

The refused command left no trace in the stream. If the bank must record refusals (for fraud analysis, say), it appends a separate `WithdrawalRefused` event, which does not change the balance.

A double-entry ledger is the oldest event-sourced system there is: accountants never erase a line, they post a correcting entry. Banking domains therefore fit event sourcing naturally, which is why it is often taught with an account.

```
# A bank account as a stream of events, folded into current state.
# Amounts are in pence so that no floating-point rounding creeps in.
from functools import reduce

events = [
    {"v": 1, "type": "AccountOpened",  "overdraft_limit": 0},
    {"v": 2, "type": "MoneyDeposited", "amount": 50000},
    {"v": 3, "type": "MoneyWithdrawn", "amount": 12000},
    {"v": 4, "type": "MoneyDeposited", "amount": 7550},
    {"v": 5, "type": "MoneyWithdrawn", "amount": 30000},
]

def apply(state, e):                      # pure: old state + event → new state
    if e["type"] == "AccountOpened":
        return {"balance": 0, "limit": e["overdraft_limit"], "version": e["v"]}
    sign = 1 if e["type"] == "MoneyDeposited" else -1
    return {**state, "balance": state["balance"] + sign * e["amount"], "version": e["v"]}

def decide_withdraw(state, amount):       # command side: check the rule, emit events
    if state["balance"] - amount < -state["limit"]:
        raise ValueError("insufficient funds")
    return [{"v": state["version"] + 1, "type": "MoneyWithdrawn", "amount": amount}]

now = reduce(apply, events, None)         # {'balance': 15550, 'limit': 0, 'version': 5}
as_at_v3 = reduce(apply, events[:3], None) # temporal query: balance 38000
```

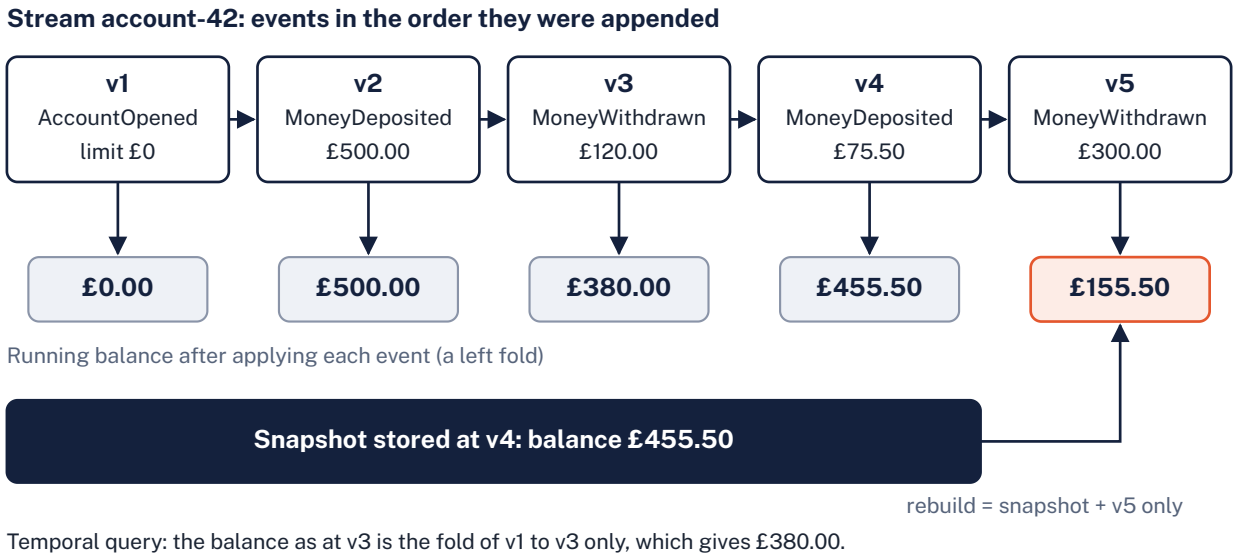
Folding the stream into current state, deciding a command against that state, and answering a temporal query by folding only part of the stream.

```
-- An event store can be an ordinary table. The primary key enforces
-- optimistic concurrency: only one writer can append version 6.
CREATE TABLE account_events (
  account_id  uuid        NOT NULL,
  version     integer     NOT NULL,
  event_type  text        NOT NULL,
  data        jsonb       NOT NULL,
  recorded_at timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (account_id, version)
);

-- The command handler read the stream up to version 5, decided, and now appends:
INSERT INTO account_events (account_id, version, event_type, data)
VALUES ('2f9c0d6e-4a51-4c7e-9a43-6d1f2b7e8a10', 6, 'MoneyWithdrawn', '{"amount": 2000}');
-- If a concurrent command already appended version 6, this fails with a
-- unique violation; the handler reloads the stream and decides again.
```

A minimal event store in a relational database: the composite primary key turns a concurrent append into a detectable conflict.

Folding, snapshots and temporal queries



Temporal query: the balance as at v3 is the fold of v1 to v3 only, which gives £380.00.

Current state is the fold of every event; a snapshot shortens the fold, and folding a prefix of the stream answers questions about the past.

What event sourcing gives you

A complete audit trail by construction. In a state-based system the audit log is a second write that someone can forget, and it can disagree with the data. In an event-sourced system the log is the data, so it cannot drift. Regulators in banking, insurance and healthcare value this highly.

Temporal queries. “What was the balance at 23:59 on 31 March?” or “Which orders were in status PREPARING when the kitchen display failed?” become folds over a prefix of each stream rather than forensic work on backups.

Rebuilds and new read models. Because read models are derived, a team can fix a bug in a projection, delete its output and replay every event to produce a corrected view. It can also build a read model nobody thought of when the events were first recorded, such as a report of how often customers withdraw cash within an hour of being paid, over the full history.

Debugging and what-if analysis. Copying a stream into a test environment and replaying it reproduces the exact state that caused a bug. Replaying the same events through changed rules shows what the new rules would have done.

A natural source of integration events. The events can feed other services, although the events stored in a stream are best treated as a private detail of the owning service. Publishing them directly couples every consumer to the internal model; many teams translate them into a smaller set of public integration events instead, designed as the long-lived published contracts described in Section 6.4.

Snapshots and long streams

Folding five events is instant; folding 400,000 events for a busy merchant account on every command is not. A snapshot stores the folded state of a stream at a known version, so loading becomes "read the latest snapshot, then apply only the events after it". Snapshots are a cache, not a source of truth: they can be deleted and recreated at any time, in the background, without stopping the system. Take them every N events or on a schedule, and store the version of the apply code alongside them, because a snapshot produced by old code is wrong once the fold logic changes.

Before reaching for snapshots, check the stream design. Very long streams often mean the aggregate is too large. A bank account that lives for 40 years can be modelled as a series of shorter streams, one per statement period, each opened with a `BalanceCarriedForward` event that records the closing balance of the previous period. This mirrors how accountants close books, and it keeps every stream short enough to load quickly.

The costs, honestly stated

Complexity and unfamiliarity. Event sourcing changes how every developer on the team thinks about persistence. Simple questions such as "which customers live in Leeds?" cannot be asked of the event store at all; they need a projection. Tooling for inspecting, migrating and repairing event stores is less mature than for relational tables, and fewer engineers have run such a system in production.

Eventual consistency of read models. When projections run asynchronously, a user can make a withdrawal and then see a balance that does not yet include it. The usual mitigations are to return the new stream version from the command and let the query side wait until its projection has processed that version (a read-your-writes guarantee, as in Section 7.4); to update the user interface optimistically from the command's result; or to update critical read models synchronously in the same transaction as the append. Projections must also be idempotent and record the position they have processed, because events may be delivered more than once after a crash.

Event versioning. Events are stored forever, so their schema can never simply be changed. When `MoneyDeposited` gains a currency field, years of old events still lack it. The standard techniques, catalogued by Greg Young in *Versioning in an Event Sourced System*, include tolerant readers that supply defaults for missing fields, upcasting (a function that converts an old event version into the current shape as it is read, so the domain code sees only one version), introducing a new event type alongside the old one, and, as a last resort, copying a whole stream into a new one with transformed events. Section 7.7 covers the general compatibility rules.

Deleting personal data. An immutable log collides with the right to erasure under data protection law. The common technique is crypto-shredding: personal fields in events are encrypted with a key held per customer in a separate key store, and erasure means destroying that key so the ciphertext can no longer be read. The alternative is to keep personal data out of events entirely and store only a reference to a record that can be deleted. Whether destroying a key counts as erasure is a legal judgement rather than a technical one, so agree the approach with the data protection officer; Section 8.7 covers the wider obligations.

Operational load. Rebuilding a projection over a billion events can take hours, so teams need a way to run the new projection beside the old one and switch over when it catches up. Event stores grow without limit, and storage, backup and replay times grow with them.

WATCH OUT

Fowler's warning

Martin Fowler's 2011 article on CQRS is unusually blunt: he writes that “for most systems CQRS adds risky complexity”, and he recommends using it only for specific parts of a system, typically one bounded context with genuinely different read and write needs, never as the architecture of the whole system. The same applies with greater force to event sourcing. Teams that applied both everywhere because they were fashionable have reported years of accidental complexity, especially in simple create-read-update-delete areas such as reference data or user profiles.

DECISION GUIDE

Decision guide: should this part of the system use CQRS or event sourcing?

Ask these questions about one bounded context, not the whole system.

Q1 **Is this area mostly simple create, read, update and delete with few business rules?**

- **Yes** Use a conventional model with one database. Neither pattern will pay for itself here.
- **No** Continue: the domain has real behaviour worth modelling.

Q2 **Do reads and writes differ sharply in shape or volume (for example, many denormalised screens, search, or reads outnumbering writes a thousand to one)?**

- **Yes** Consider CQRS. Start with the lightest form: separate read code or views over the same database.
- **No** Keep a single model; add a read replica or cache if read load alone is the problem.

Q3 **Is the history itself valuable: regulatory audit, temporal queries, replaying with corrected logic, or many future read models from the same facts?**

- **Yes** Consider event sourcing for this context, ideally a ledger-like domain such as accounts, payments or order lifecycles.
- **No** Store current state; publish events with an outbox if other services need to know about changes.

Q4 **Does the team have the experience, tooling and time to handle event versioning, projection rebuilds and eventual consistency?**

- **Yes** Proceed, keep the event-sourced area small, and record the decision and its reasons in an ADR.
- **No** Store current state plus an append-only audit table, which gives most of the audit benefit at a fraction of the cost.

When not to use either, and common mistakes

Neither pattern suits a domain that is mostly forms over data, a small team building its first version, or a system whose only problem is read load (a read replica or a cache is far cheaper). Event sourcing is also a poor fit where data must be truly deleted on demand at high volume, or where most queries cut across many aggregates, since every such query needs its own projection.

The common mistakes are predictable. Teams apply the patterns system-wide instead of to one context. They expose stored events as the public contract, freezing their internal model. They put derived values or external calls in the apply function, so replay gives different answers. They design events as generic `AccountUpdated` blobs, which record that something changed but not what happened or why, and lose most of the value. And they forget that a read model is a cache: they start writing to it directly, after which it can no longer be rebuilt.

Used selectively, the combination is powerful. A food-delivery platform might event-source the order lifecycle (placed, accepted, preparing, picked up, delivered, refunded), because disputes and refunds need the exact history, while keeping menus, restaurant profiles and user settings as ordinary tables. A bank might event-source the ledger and payments while storing customer contact details conventionally, which also keeps most personal data out of the immutable log.

CHECK YOUR UNDERSTANDING

▼ Why must the function that applies events to state never call the system clock?

Because replaying the same events must always produce the same state. A clock (or random number, or external call) would make replay give different answers; any such value must be captured inside the event when it is first decided.

▼ What is the difference between the command WithdrawMoney and the event MoneyWithdrawn?

The command is a request that can be refused by the business rules; the event is a recorded fact that has already been accepted and can only be compensated by a later event, never removed.

▼ A projection has a bug that has produced wrong statements for three months. How does event sourcing help?

Because the read model is derived, the team fixes the projection, builds a new read model by replaying all events (usually beside the old one) and switches over, with no need to repair data by hand.

▼ Is a snapshot part of the source of truth?

No. It is a cached fold of a stream at a given version, which can be deleted and recreated from the events at any time.

SOURCES

- Martin Fowler: CQRS (2011): https://martinfowler.com/bliki/CQRS.html

- Martin Fowler: Event Sourcing (2005): https://martinfowler.com/eaaDev/EventSourcing.html
- Martin Fowler: Command Query Separation: https://martinfowler.com/bliki/CommandQuerySeparation.html
- Microsoft Azure Architecture Center: Event Sourcing pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
- Greg Young: Versioning in an Event Sourced System (Leanpub): https://leanpub.com/esversioning

SECTION 7.6 OF 7.8 · INTERMEDIATE · 24 MIN READ

Caching strategies and their pitfalls

BEFORE YOU READ THIS

This section builds on:

- 0.2 Data and algorithms
- 0.4 How computers talk
- 2.4 Performance and scalability

Terms it uses from earlier: Latency, Throughput, HTTP, System of record, Read-your-writes

THE IDEA IN BRIEF

Imagine a busy café. Instead of grinding beans for every order, the barista keeps a small jar of freshly ground coffee on the counter. Most orders are served from the jar in seconds; only when it runs out does anyone go to the store room. The jar is a cache. It makes service fast and spares the store room, but it raises questions: how long can ground coffee sit before it tastes stale, what happens when the jar empties during the morning rush, and who remembers to throw it out when the beans change? Software caches raise exactly the same questions.

WHAT YOU NEED TO KNOW

- A cache trades a second, possibly stale copy of data for lower latency, higher throughput and protection of the source; first decide what may be stale, by how much and for whom.
- Cache-aside is the safe default; read-through moves the loading into the cache layer; write-through keeps the cache current at the cost of slower writes; write-behind risks data loss and must never hold money; refresh-ahead suits a few hot keys.
- Combine invalidation on change with a TTL as a safety net; on writes, update the database and then delete the cache entry rather than overwriting it.
- Prevent stampedes with request coalescing, jittered expiry, stale-while-revalidate, probabilistic early refresh and cache warming.
- Hit ratio drives database load far more than average latency: each extra nine of hits divides database queries by about ten, and a cold cache can overwhelm a database sized for cached traffic.
- Caches exist in browsers, CDNs, reverse proxies, applications and databases; HTTP controls them with Cache-Control, ETag and conditional requests (RFC 9111, RFC 9110).

- LRU and LFU are the common eviction policies; keys must include everything that changes the answer, and personal or financial data must stay out of shared caches.

Why cache, and what it costs

A cache keeps a copy of data somewhere faster or closer than its original home, so that repeated requests do not pay the full price each time. Section 0.2 introduced the idea as a trade of space for time. At architecture scale the trade is more interesting, because a cache changes three qualities at once. It lowers latency (a Redis lookup in the same data centre typically takes well under a millisecond; a database query that joins five tables may take tens of milliseconds). It raises throughput and protects the database, because every hit is a query the database never sees. And it can improve availability, because a cache that can serve slightly old data keeps working while the source is slow or down.

The price is paid in correctness and complexity. A cache is a second copy of the data, and two copies can disagree. Every cache therefore embodies a decision about how stale an answer is allowed to be, and for whom. A restaurant menu that is five minutes old is acceptable; a bank balance that is five minutes old, shown to a customer deciding whether they can afford a payment, is not. Caches also add a component that can fail, consume memory that costs money, and create new failure modes such as stampedes and cold starts that do not exist without them. The first question is therefore not “which cache?” but “what may be stale, by how much, and what happens if it is?”

The effectiveness of a cache is measured by its hit ratio: hits divided by all lookups. Hit ratio depends on how skewed the access pattern is. Read traffic in most consumer systems is highly skewed (a small fraction of restaurants, products or accounts receives most of the reads), so a cache holding a few per cent of the data can often answer most of the reads. Uniformly random access, by contrast, caches badly whatever you do.

The five classic patterns

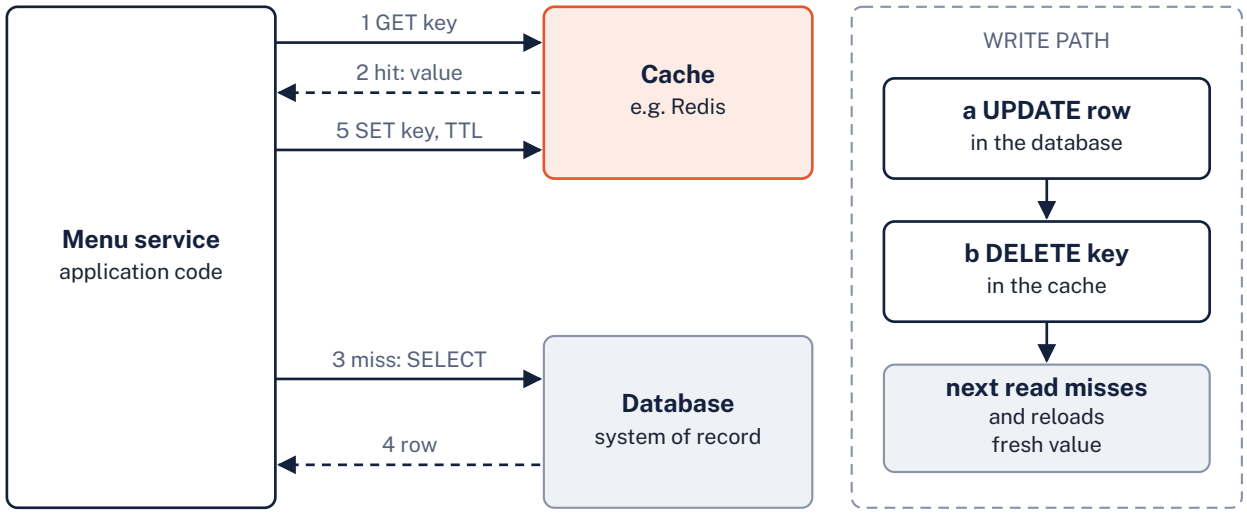
Caching patterns differ in who loads the cache, when writes reach the source, and therefore how stale and how durable the data can be. The table compares them; the paragraphs after it explain where each fits.

Caching patterns compared

Pattern	How reads work	How writes work	Strengths	Risks
Cache-aside (lazy loading)	Application checks the cache; on a miss it reads the database and stores the result	Application writes the database, then deletes (or updates) the cache entry	Simple, only caches what is used, cache failure degrades to slower reads	First read after expiry is slow; races can leave stale entries; stampedes
Read-through	Application asks the cache; the cache library itself loads from the database on a miss	Usually combined with write-through or cache-aside writes	Loading logic lives in one place	Same staleness and stampede issues; the cache layer must know the data source
Write-through	As read-through or cache-aside	Write goes to the cache and the database synchronously before acknowledging	Cache is never older than the database for data written through it	Every write pays both latencies; caches data that may never be read
Write-behind (write-back)	Served from the cache	Write is acknowledged once in the cache; flushed to the database later, often batched	Very fast writes; absorbs bursts; batches database work	Data loss if the cache fails before flushing; database lags; ordering and conflicts
Refresh-ahead	Served from the cache	Not about writes: hot entries are reloaded shortly before they expire	Popular items never expire in users' faces	Wasted reloads of items that would not have been read; needs access statistics

Cache-aside is the default for good reason: it is easy to reason about and fails safe, because if the cache disappears the application simply reads the database (more slowly). Read-through moves the same logic into a cache library or product, which is tidy but no different in behaviour. Write-through suits data that is read soon after being written, such as a user's session or shopping basket. Write-behind is the risky one: it turns the cache into the temporary system of record, so it is appropriate only where losing the last few seconds of writes is acceptable (view counters, analytics events, game scores) or where the cache itself is replicated and durable. Never use write-behind for money. Refresh-ahead is an optimisation for a small set of very hot keys, such as the home-page restaurant list in a busy city.

Cache-aside in practice



Reads fill the cache lazily on a miss; writes change the database first and then delete the cached copy rather than overwrite it.

```
import json, random

TTL = 300  # seconds

def get_menu(restaurant_id):
    key = f"menu:{restaurant_id}:v3"           # version the key with the data shape
    cached = cache.get(key)
    if cached is not None:
        return json.loads(cached)               # hit
    with cache.lock(f"lock:{key}", timeout=5):  # coalesce: one loader per key
        cached = cache.get(key)                 # someone may have filled it meanwhile
        if cached is not None:
            return json.loads(cached)
        menu = db.load_menu(restaurant_id)      # miss: go to the system of record
        jitter = random.randint(0, TTL // 10)   # spread expiry times out
        cache.set(key, json.dumps(menu), ex=TTL + jitter)
        return menu

def update_price(restaurant_id, item_id, price):
    db.update_price(item_id, price)             # 1. change the system of record
    cache.delete(f"menu:{restaurant_id}:v3")    # 2. invalidate, do not overwrite
```

Cache-aside with a versioned key, a per-key lock to coalesce concurrent misses, jittered expiry, and invalidation by deletion after the database write.

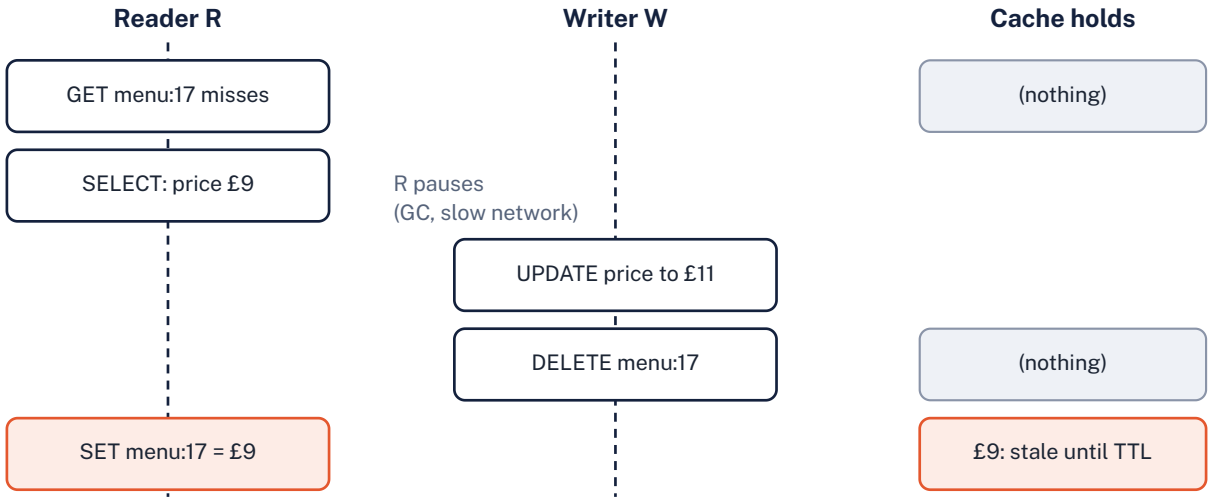
Time to live and invalidation

There are two ways to stop a cache serving data that has changed. The first is a time to live (TTL): every entry expires after a fixed period, which bounds staleness without any coordination. A five-minute TTL on menus means a price change is visible everywhere within five minutes. The second is explicit invalidation: when the source changes, something deletes or updates the cached copy. Invalidation gives fresher data but requires every code path that changes the data to know every cache that holds it, which is exactly what goes wrong as systems grow. In distributed systems invalidation is often driven by events: the menu service publishes `MenuChanged`, and each service holding a copy deletes its entry. Most professional designs combine both: invalidate on change for freshness, and keep a TTL as a safety net for the invalidation that was missed.

The difficulty is captured in a quip attributed to Phil Karlton of Netscape: “There are only two hard things in Computer Science: cache invalidation and naming things.” It is hard because invalidation is a distributed-systems problem in disguise. The database write and the cache delete are two operations on two systems with no shared transaction, so one can succeed while the other fails, and operations from different requests can interleave.

Two rules avoid most problems. First, on a write, update the database and then **delete** the cache entry; do not write the new value into the cache. Two concurrent writers that each set the cache can finish in the opposite order to their database commits and leave the older value cached indefinitely, whereas a delete can only cause an extra miss. Second, accept that even delete-on-write leaves a narrow race, shown below, and bound it with a TTL. Facebook's 2013 paper on scaling memcache describes **leases** for this case: the cache gives a token to the client that missed, and refuses that client's later write if the key was deleted in the meantime.

The stale-set race



Deleting on write narrows the window for stale data but does not close it; a TTL bounds how long the stale value survives.

IF YOU ARE NEW TO THIS

Step by step: why delete instead of update

This section's hardest idea is cache invalidation: changing the database and changing the cache are separate steps on two systems, so overlapping requests can leave an old value behind. Two quick address changes show why to delete, not overwrite.

- The platform caches each customer's profile, delivery address included, using cache-aside: a request that misses loads the profile from the database and stores it. The cache says Flat 1. Within a second the customer saves Flat 2, then corrects it to Flat 3: requests A and B.
- Suppose each save commits to the database, then copies its address over the cached one. A commits Flat 2, B commits Flat 3: the database is right.
- But A stalls between its two steps. B puts Flat 3 in the cache; then A, running late, puts Flat 2. The cache now contradicts the database, and every reader sees the wrong flat.
- Nothing repairs it: no further save is coming, so the wrong address stays until the entry expires, or indefinitely if it never does.
- Now let each save remove the cached profile instead. In whatever order the removals land, the cache ends up empty; the next request misses and stores Flat 3 from the database. A late removal costs one extra miss at most.
- Removing narrows the gap without closing it: a slow reader can still store an old value just after a removal, as the race above shows. A time to live on every entry limits how long that survives.
- **The lesson:** on a change, commit to the database first, then remove the cached copy instead of replacing it, and give every entry an expiry time to catch what invalidation misses.

Stampedes and the thundering herd

A cache stampede happens when a popular entry expires, or a cache node restarts empty, and hundreds of concurrent requests all miss at the same instant. Each one goes to the database to compute the same value, and the database, sized for the cache's normal miss rate, is overwhelmed. Slower responses mean more requests pile up, and the overload can cascade. The related **thundering herd** describes many waiting clients woken at once to compete for one resource. Facebook reported that, for a set of keys particularly prone to this, leases cut the peak database query rate from 17,000 to 1,300 per second.

The defences combine well. Request coalescing (also called single flight) lets only one request per key fetch from the source while the others wait for its result; within one process this is a map of in-flight futures, across processes a short lock in the cache, as in the code above. **Jittered expiry** adds a random amount to each TTL so that entries written together (after a deployment or a cache flush) do not expire together. **Stale-while-revalidate** serves the expired value to everyone while one background request fetches a fresh one, which removes the latency spike entirely when slight staleness is acceptable. **Probabilistic early expiration** (Vattani, Chierichetti and Lowenstein, VLDB 2015) has each reader decide at random, with a probability that rises as expiry approaches, to refresh early, so that one request usually refreshes before the entry actually expires. Finally, **warming** a new cache from a list of hot keys before it takes traffic avoids the cold-start stampede after a restart.

WORKED EXAMPLE

Worked example: what hit ratio does to latency and database load

The food-delivery platform's menu service receives 20,000 menu reads per second at the evening peak. A cache lookup takes 0.5 ms; a database query takes 8 ms. On a miss the service pays both (8.5 ms) and stores the result asynchronously. The database can sustain about 3,000 such queries per second before latency climbs steeply.

Formula	Average latency = 0.5 + (1 − h) × 8 ms. Database load = (1 − h) × 20,000 queries per second.
No cache at all	8 ms average; 20,000 queries per second, nearly seven times what the database can sustain.
Hit ratio 80%	0.5 + 0.2 × 8 = 2.1 ms; 4,000 queries per second: still above capacity, so the database queues and latency rises.
Hit ratio 95%	0.5 + 0.05 × 8 = 0.9 ms; 1,000 queries per second: comfortable.
Hit ratio 99%	0.5 + 0.01 × 8 = 0.58 ms; 200 queries per second.
From 95% to 99%	Average latency improves by only 0.32 ms, but database load falls five-fold, from 1,000 to 200 queries per second.
Cache cluster restarts empty	Hit ratio drops to near 0% for a while: 20,000 queries per second hit a database sized for 3,000. Without coalescing, warming or rate limiting, the database falls over.
Percentiles	At 95% hits, 5% of requests take 8.5 ms, so p95 sits on the edge and p99 is 8.5 ms, not 0.9 ms.

Hit ratio matters far more for the database than for average latency: each extra nine of hits divides database load by about ten.

A cache that the system cannot run without is no longer an optimisation; it is a critical dependency, and its failure must be planned for as Section 2.5 plans for any other.

Averages flatter caches; as Section 2.4 showed, the misses define the tail.

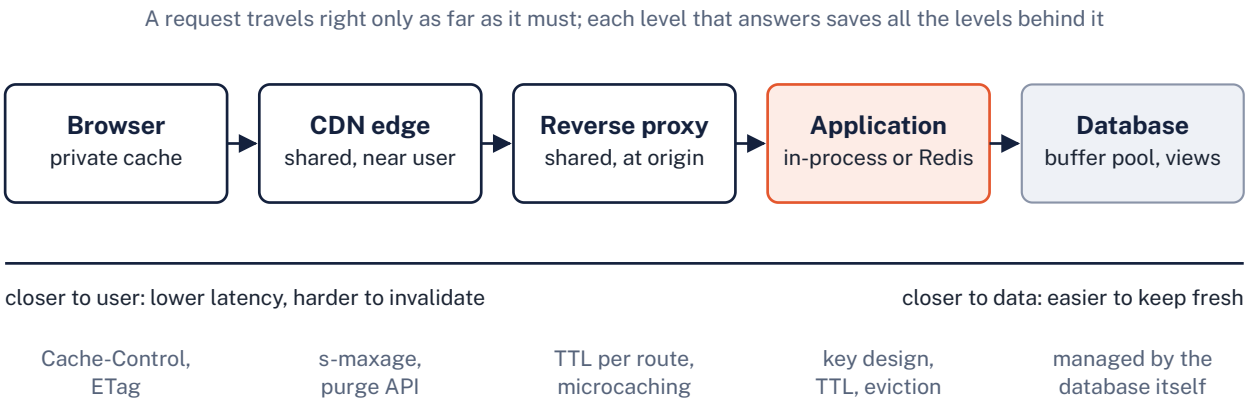
Levels of caching

Caches exist at every level between the user and the disk, and a request is answered by the first level that has a usable copy. The **browser** keeps a private cache controlled by HTTP headers. A content delivery network (CDN) keeps shared copies at edge locations near users, ideal for images, scripts and public API responses such as menus. A **reverse proxy** at the origin

(Nginx, Varnish) can cache whole responses for a few seconds, which is called microcaching and can absorb a sudden spike. The **application** caches computed objects, either in-process (fastest, but each instance holds its own copy and they disagree) or in a shared cache such as Redis or Memcached. The **database** caches pages in memory (PostgreSQL's shared buffers, the InnoDB buffer pool) and can precompute expensive results in materialised views.

The further from the data, the larger the saving per hit and the harder the invalidation. You can delete a key from Redis in a millisecond; you can purge a CDN, more slowly; you cannot reach into a customer's browser at all, so whatever `max-age` you send is a promise you must live with. Long browser lifetimes are therefore used for files whose URL changes when their content changes (`app.3f9a1c.js`), and short ones, or revalidation, for anything else.

Where caches live



Each level that answers a request spares every level behind it; levels nearer the user save more but are harder to invalidate.

HTTP caching headers

HTTP has a complete caching model, defined in RFC 9111 (June 2022), and using it well can remove load before it reaches your servers. `Cache-Control: max-age=60` lets any cache reuse a response for 60 seconds; `s-maxage` overrides that for shared caches such as CDNs; `private` forbids shared caches from storing it (use it for anything personal); `no-cache` confusingly means “store it, but check with the server before every reuse”; and `no-store` means “never keep a copy”, the right setting for bank balances and statements. The `stale-while-revalidate` and `stale-if-error` extensions (RFC 5861, 2010) apply the stampede and availability ideas above at the HTTP level.

When a cached copy is stale, the client need not download it again. A validator lets it ask whether anything has changed: the server sends an `ETag` (an opaque version identifier) or a `Last-Modified` date with the response, and the client later sends `If-None-Match` or `If-Modified-Since`. If nothing changed, the server replies `304 Not Modified` with no body,

saving bandwidth and often most of the server's work. These conditional requests are defined in RFC 9110. The `Vary` header tells caches which request headers (such as `Accept-Language`) produce different responses, so that a French menu is never served to an English request.

```
GET /restaurants/17/menu HTTP/1.1
Host: api.example-food.test

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=30
ETag: "menu-17-v42"

... 18 KB of menu JSON ...

# 100 seconds later the copy is past max-age plus the stale-while-revalidate
# window, so the browser must revalidate before using it:
GET /restaurants/17/menu HTTP/1.1
If-None-Match: "menu-17-v42"

HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=30
ETag: "menu-17-v42"

# A bank balance, by contrast, must never sit in any cache:
Cache-Control: no-store
```

A public menu response that browsers may reuse for 60 seconds and CDNs for 300, revalidated cheaply with an ETag; personal financial data uses no-store.

Eviction, sizing and consistency risks

A cache is smaller than the data behind it, so when it is full it must evict something. Least recently used (LRU) evicts the entry untouched for longest and works well when recent use predicts future use. Least frequently used (LFU) evicts the entry used least often and resists one-off scans (a nightly report that touches every restaurant once would flush an LRU cache of its genuinely hot entries). Production caches approximate both: Redis, for example, samples a few keys and evicts the best candidate among them, and offers policies such as `allkeys-lru`, `allkeys-lfu` and `volatile-ttl`. Its default policy, `noeviction`, rejects writes when memory is full, which suits Redis used as a store but not as a cache, so set the policy deliberately.

Beyond staleness, caches introduce subtler consistency risks. A user who changes their delivery address and then sees the old one, because the read came from a cache another instance had not yet invalidated, experiences a broken read-your-writes guarantee (Section 7.4); a common fix is to bypass the cache for a user's own recent writes. In-process caches on

many instances disagree with each other, so consecutive requests routed to different instances can show a value going backwards. Caching an error or an empty result without a short TTL turns a brief outage into a long one. Caching data that mixes users (a page with the customer's name, cached under the URL alone) leaks personal data to the next visitor, a well-known class of security incident. And cache keys must include everything that changes the answer: tenant, language, currency, permissions and the version of the data format.

WATCH OUT

Common mistakes

- Adding a cache before measuring: often the real fix is an index or a better query, which removes the problem without a second copy of the data.
- No TTL at all, relying entirely on invalidation that one forgotten code path will miss.
- Updating the cache on write instead of deleting it, creating ordering races between writers.
- Sizing the database for the cached load, so that a cache restart takes the whole platform down.
- Caching personal or financial data in shared caches, or forgetting `private` and `Vary` on HTTP responses.
- Ignoring hit ratio, eviction rate and memory in monitoring, so that a cache quietly stops helping.

CHECK YOUR UNDERSTANDING

▼ **Why should a write delete the cache entry rather than put the new value into the cache?**

Because two concurrent writers can finish their cache updates in the opposite order to their database commits, leaving the older value cached; a delete can at worst cause one extra miss that reloads the current value.

▼ **A cache lookup costs 1 ms and a database read 10 ms. Roughly what is the average latency at a 90% hit ratio with cache-aside?**

1 + 0.1 × 10 = 2 ms (every request pays the lookup; one in ten also pays the database read).

▼ **What is the difference between Cache-Control: no-cache and no-store?**

no-cache allows a cache to store the response but requires it to revalidate with the server before each reuse; no-store forbids keeping a copy at all.

▼ **Why can raising the hit ratio from 99% to 99.9% matter even though average latency barely changes?**

Because database load is proportional to the miss rate, which falls from 1% to 0.1%, a ten-fold reduction in queries reaching the database.

SOURCES

- RFC 9111: HTTP Caching (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9111.html
- RFC 5861: HTTP Cache-Control Extensions for Stale Content: https://www.rfc-editor.org/rfc/rfc5861.html
- MDN Web Docs: HTTP caching: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching
- Nishtala et al.: Scaling Memcache at Facebook (NSDI 2013): https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala
- Microsoft Azure Architecture Center: Cache-Aside pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

SECTION 7.7 OF 7.8 · INTERMEDIATE · 25 MIN READ

Schema evolution, migrations and compatibility

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 0.8 How software gets built
- 6.5 Communication and APIs

Terms it uses from earlier: Breaking change, Protocol Buffers, Continuous delivery, Replication lag, Idempotent

THE IDEA IN BRIEF

Imagine a hospital changing the paper form used for every admission while patients keep arriving. If it simply swaps the forms overnight, nurses on the night shift will be holding the old form while the day shift expects the new one, and filed records still use the old layout. A careful hospital prints a form with both the old and new boxes, uses it until everyone has switched and the old files are copied over, and only then removes the old boxes. Software teams change the structure of their data in exactly this careful, step-by-step way.

WHAT YOU NEED TO KNOW

- Schemas must change while old and new code run side by side and old data still exists, so every deployed schema state must work with at least two versions of the code.
- Backward compatibility means new readers can read old data; forward compatibility means old readers can read new data; full compatibility allows readers and writers to upgrade in any order.
- Expand and contract (parallel change) splits a breaking change into compatible steps: add the new structure, dual-write and backfill, switch readers, stop writing the old structure, then remove it.
- Migrations are versioned scripts kept with the code and applied automatically by tools such as Flyway, Liquibase and Alembic; never edit an applied migration, and rehearse on production-sized data.
- Large backfills run as separate, batched, throttled, idempotent jobs; online techniques such as concurrent index builds and NOT VALID constraints avoid long locks.
- Schema registries enforce compatibility rules for event schemas; Avro relies on defaults and writer-reader schema resolution, Protocol Buffers on never reusing field numbers.

Why schemas must evolve without stopping the world

Every piece of stored or transmitted data has a schema: the tables and column types of a database, the fields of a JSON API response, the record definition of an event on a stream. Schemas change constantly, because the business does: the bank adds a second account holder, the food-delivery platform starts recording tips, a regulator demands a new field. The architectural question is how to change a schema while the system keeps running, when the old shape is still in use by code, by stored data and by other teams.

Three facts make this hard. First, in any system deployed without downtime, old and new versions of the code run side by side: a rolling deployment replaces instances one at a time, a canary sends a small share of traffic to the new version, and a rollback brings the old version back. Every schema state must therefore work with at least two versions of the code. Second, data outlives code. Rows written five years ago, events in a log retained forever, messages waiting in a queue and backups all keep the old shape. Third, schemas are shared: a column is read by reports, extract jobs and sometimes other services, and an event schema is read by consumers the producer may not even know about. Section 7.1 argued for owning data behind an interface; schema evolution is where that discipline pays off, because a private schema can be changed with one team's agreement.

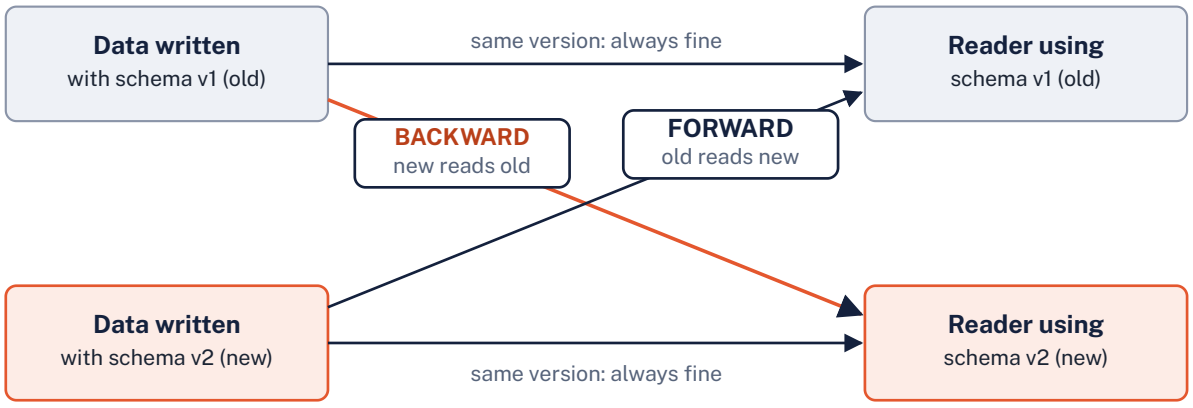
Backward, forward and full compatibility

Compatibility is defined in terms of readers and writers. A change is backward compatible when code using the **new** schema can read data written with the **old** one. Adding an optional field with a default is backward compatible: the new reader supplies the default when the old data lacks the field. A change is forward compatible when code using the **old** schema can read data written with the **new** one. That requires old readers to ignore fields they do not know, which is the tolerant reader discipline. A change is **fully compatible** when it is both, which means readers and writers can be upgraded in any order.

The direction you need follows from who upgrades first. If consumers upgrade before producers (common when a stream has a few consumers that you control), you need backward compatibility. If producers upgrade first, which is the usual case for a database (new code starts writing before every reader has been redeployed) and for public APIs, you need forward compatibility. When you cannot control the order, or need to roll back either side, aim for full compatibility. Confluent Schema Registry, for example, defaults to backward compatibility checked against the previous version, and offers transitive variants that check against every earlier version, which matters when consumers may replay old data from the beginning of a log.

A word of warning on vocabulary. In API design, “backward compatible” is often used loosely to mean “existing clients keep working”, which in reader and writer terms is forward compatibility for the clients that read your responses and backward compatibility for the requests you read from them. When a discussion turns on the term, ask which side changes and which side must keep working.

Which reader can read which data



FULL = both diagonals work, so readers and writers can upgrade in any order

Backward compatibility protects new readers from old data; forward compatibility protects old readers from new data.

Common changes and their compatibility

Change	Backward (new reads old)	Forward (old reads new)	Safe approach
Add an optional field or nullable column with a default	Yes	Yes, if readers ignore unknown fields	The standard safe change
Add a required field with no default	No: old data lacks it	Yes	Add as optional, backfill, then enforce
Remove a field	Yes	Only if old readers have a default for it	Stop reading it everywhere first, then remove
Rename a field or column	No	No	Expand and contract: add new, copy, switch, remove old
Change a type (text to integer, int to long)	Depends on the format: widening is often safe, narrowing is not	Depends	Add a new field of the new type and migrate
Change meaning without changing shape (pence to pounds)	Silently wrong	Silently wrong	Never: create a new field with a new name

IF YOU ARE NEW TO THIS

Step by step: old readers, new readers

This section's hardest idea is the difference between backward and forward compatibility, and why the one you need depends on which side upgrades first. Follow one new field through the food-delivery platform.

- The platform starts recording tips, so a new version of the OrderPlaced event adds a field, tip_pence. Events already written and stored have no such field.
- Suppose a consumer is upgraded first. Its new code reads events written before tips existed, including old ones replayed from the start of the log, so it must fill the missing field with a default, here null, meaning no tip recorded. This case, where the new version copes with old data, is called backward compatibility.
- Suppose instead the producer is upgraded first. Consumers still running old code now receive events with a field they have never heard of, and must simply ignore it, as a tolerant reader does. This case, where the old version copes with new data, is called forward compatibility.
- With tolerant readers, an optional field with a default works in both directions, so the change is fully compatible: either side can upgrade first, and either can be rolled back.
- Contrast renaming total_pence to amount_pence, for readers that find fields by name. New readers look for amount_pence and cannot find it in old events; old readers look for total_pence and cannot find it in new ones. The change breaks in both directions at once.
- **The lesson:** before changing a schema, ask who will read new data with old code and who will read old data with new code. Additive changes with defaults are safe both ways; anything else needs the step-by-step technique that comes next.

Expand and contract: the core technique

Most breaking changes can be split into a sequence of compatible ones. The pattern is called expand and contract, or parallel change (Danilo Sato's 2014 description on martinfowler.com names the phases expand, migrate and contract). First **expand**: add the new structure alongside the old, so that both old and new code work. Then **migrate**: change writers to maintain both, copy existing data across, and move readers to the new structure one by one. Finally **contract**: once nothing uses the old structure, remove it. Each step is deployed separately, each is compatible with the version before it, and each can be rolled back without losing data.

The cost is time and temporary mess: the change spans several releases, and for days or weeks the schema carries both shapes. The benefit is that no release ever needs downtime, no deployment needs to be coordinated across teams, and every step can be paused. Scott

Ambler and Pramod Sadalage's *Refactoring Databases* (2006) catalogues dozens of database refactorings built on the same idea, with a **transition period** during which old and new schemas coexist, kept in step by the application or by triggers.

WORKED EXAMPLE

Worked example: renaming a column without downtime

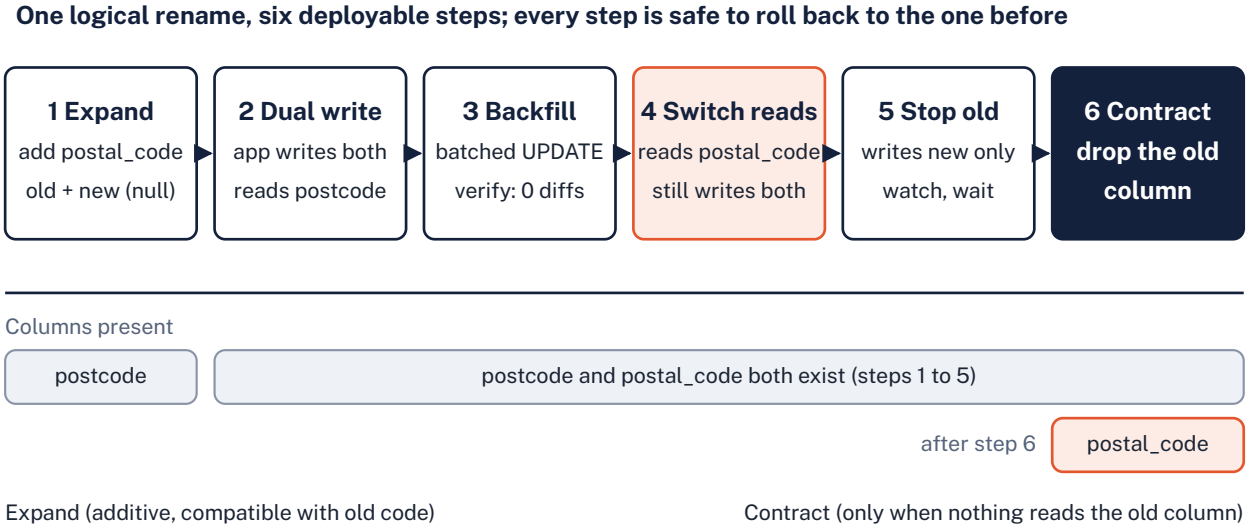
The restaurants table of the food-delivery platform has a column `postcode`. The platform is expanding to countries that use other postal formats, and the team wants it called `postal_code`. The table has 4 million rows, the restaurant service is deployed several times a day with rolling updates, and a nightly extract job also reads the column. A single `ALTER TABLE ... RENAME COLUMN` would break every running instance of the old code the moment it committed.

Release 1: expand	Migration adds `postal_code text NULL`. In PostgreSQL 11 and later, adding a column with no default or a constant default is a metadata-only change. Old code ignores the new column; nothing breaks.
Release 2: dual write	Application writes the same value to both columns on every insert and update, and still reads `postcode`. (A database trigger that copies one column to the other is an alternative when several applications write the table.)
Step 3: backfill	A background job copies `postcode` into `postal_code` for existing rows in batches of 5,000 by primary-key range, pausing between batches and watching replication lag. Then a verification query counts rows where the two differ: it must return 0.
Release 4: switch reads	Application reads `postal_code` but still writes both columns, so rolling back to release 2 or 3 remains safe. The extract job is changed to read the new column too.
Release 5: stop writing the old column	Application writes only `postal_code`. The old column must not be `NOT NULL` by now, or inserts fail. Wait long enough to be confident no rollback to release 4 will be needed.
Release 6: contract	Migration enforces `NOT NULL` on `postal_code` (using a `NOT VALID` check constraint, validated separately, which PostgreSQL 12 and later can use to set `NOT NULL` without a long blocking scan) and finally drops `postcode`.

Six deployments for one rename sounds excessive, but steps can be combined when the team controls every reader and accepts a short rollback window; the principle that matters is that each deployed state works with the code before and after it.

The contract step is the one teams forget. Track it explicitly (a ticket, an ADR status, a lint rule that flags the old column) or the schema fills up with half-migrated columns.

The rename as six deployable steps



During steps 1 to 5 both columns exist, so any version of the code from the previous step can still run.

```
-- Migration V41 (expand): metadata-only change, no table rewrite in PostgreSQL 11+
SET lock_timeout = '3s';            -- fail fast rather than queue behind long queries
ALTER TABLE restaurants ADD COLUMN postal_code text;

-- Backfill job (run repeatedly by a script, not inside one big migration)
UPDATE restaurants
SET    postal_code = postcode
WHERE  id ⩾ :from_id AND id < :from_id + 5000   -- one small batch per transaction
AND    postal_code IS DISTINCT FROM postcode;

-- Verify before switching reads
SELECT count(*) FROM restaurants WHERE postal_code IS DISTINCT FROM postcode;  -- expect
0

-- Migration V44: enforce NOT NULL without a long blocking scan
ALTER TABLE restaurants ADD CONSTRAINT postal_code_nn CHECK (postal_code IS NOT NULL) NOT
VALID;
ALTER TABLE restaurants VALIDATE CONSTRAINT postal_code_nn;  -- scans, but allows writes
ALTER TABLE restaurants ALTER COLUMN postal_code SET NOT NULL;  -- PostgreSQL 12+: uses
the check, skips scan
ALTER TABLE restaurants DROP CONSTRAINT postal_code_nn;

-- Migration V45 (contract), weeks later, once nothing reads postcode
ALTER TABLE restaurants DROP COLUMN postcode;
```

PostgreSQL statements for the expand, backfill, verify and contract steps, written to avoid long locks.

Migrations as code in the delivery pipeline

Database changes belong in version control next to the application code, as an ordered series of migrations, each a small script that moves the schema from one version to the next. Pramod Sadalage and Martin Fowler's *Evolutionary Database Design* (first published 2003, rewritten 2016) set out the practice, and the tools are mature. **Flyway** runs versioned SQL files such as `V41__add_postal_code.sql` in order and records each in a `flyway_schema_history` table. **Liquibase** describes changes as changesets in SQL, XML, YAML or JSON, tracked in a `DATABASECHANGELOG` table. **Alembic**, for Python and SQLAlchemy, generates revision scripts with `upgrade()` and `downgrade()` functions and records the current revision in `alembic_version`. Flyway and Liquibase also store a checksum of every applied migration and stop with an error if one has since been edited, which protects environments from silently diverging.

In a continuous delivery pipeline, the same migrations run against a fresh database in tests, then against staging, then production, so production never receives a change that has not been rehearsed. Good practice adds a few rules. Never edit a migration that has been applied anywhere; write a new one. Keep schema changes and the code that depends on them in separate deployments, with the schema change first for expansions and last for contractions. Test migrations against a copy of production-sized data, because a statement that takes 50 ms on a test database can lock a large table for an hour. Prefer rolling forward with a new migration over down-migrations that drop columns and with them data. And let the database tell you about locks: in PostgreSQL an `ALTER TABLE` waiting for its lock blocks every query that arrives after it, so set a short `lock_timeout` and retry rather than queueing behind a long-running report.

Large backfills and online schema changes

A backfill fills a new column or table for existing rows. Done as one `UPDATE` over 400 million transactions, it holds locks for hours, generates a vast amount of write-ahead log that replicas must replay (causing replication lag), and in PostgreSQL leaves dead row versions that vacuum must clean. The professional approach is a separate, restartable job that processes small batches by primary-key range, commits each batch, records its progress, throttles itself when replication lag or database load rises, and is idempotent so that re-running a batch does no harm. Run it outside the migration tool, so that a deployment never waits on it.

Some changes cannot be made in place without locking. PostgreSQL offers `CREATE INDEX CONCURRENTLY` (which cannot run inside a transaction block, so migration tools need to be told) and the `NOT VALID` then `VALIDATE CONSTRAINT` sequence shown above. For MySQL, tools such as gh-ost and pt-online-schema-change build a shadow copy of the table with the new

schema, keep it in step with ongoing writes and swap it in with a brief rename. Changing a column's type is usually best done as an expand and contract onto a new column rather than an in-place `ALTER`, which in most databases rewrites the whole table.

Schema registries, Avro and Protocol Buffers

For events and messages, the equivalent of a migration tool is a schema registry. Producers register each schema version; the registry checks it against the configured compatibility rule and rejects a breaking change at build or deploy time rather than in production. Each message carries a short schema identifier, so consumers can fetch the exact schema it was written with.

Apache Avro makes the writer's schema explicit: decoding needs both the writer's schema and the reader's schema, and Avro's resolution rules match fields by name, ignore writer fields the reader does not know, and fill reader fields missing from the data with their declared defaults. Hence the practical rule: give every field you might add or remove a default. In the example below, `currency` and `tip_pence` were added in later versions with defaults, so old events still decode with new consumers.

Protocol Buffers identify fields by number, not name, on the wire. Adding fields is safe, and old readers keep unknown fields when re-serialising. Renaming a field does not affect the binary encoding (though it does change the JSON form). The cardinal rules are never to change or reuse a field number, and to mark the numbers and names of deleted fields as `reserved` so that nobody reuses them by accident. Confluent recommends backward-transitive compatibility for Protobuf, because adding a new message type is not forward compatible. JSON Schema can be registered and checked too, but JSON's openness (is an unknown property allowed or not?) makes compatibility rules subtler.

```
{
  "type": "record",
  "name": "OrderPlaced",
  "namespace": "food.orders.v1",
  "fields": [
    {"name": "order_id",       "type": "string"},
    {"name": "restaurant_id",  "type": "string"},
    {"name": "total_pence",    "type": "long"},
    {"name": "currency",       "type": "string", "default": "GBP"},
    {"name": "tip_pence",      "type": ["null", "long"], "default": null}
  ]
}
```

An Avro schema for an OrderPlaced event; fields added after version 1 carry defaults, so new consumers can read old events.

Evolving API and event schemas

APIs and events follow the same principles with different tools. For APIs, prefer additive change, require clients to be tolerant readers, and keep breaking changes for a new version run alongside the old one, with usage monitoring to see when the old version can be retired (Section 6.5 covers versioning approaches). Consumer-driven contract tests let each consumer publish the fields it actually relies on, so the provider knows which changes are safe.

Events are harder than APIs because they are stored: a consumer may replay a topic from the beginning, and an event-sourced system (Section 7.5) rebuilds state from events written years ago. Treat a published event schema as a public contract, version it in the registry, and when a change is genuinely breaking, publish a new event type or topic (`OrderPlaced.v2`) for a period alongside the old one rather than mutating the existing one. Section 6.4 discusses event design more generally.

Checklist: before shipping a schema change

- [ ] **Is the change additive?**
  If not, split it into expand, migrate and contract steps, each deployable on its own.
- [ ] **Does every deployed state work with the previous and next code versions?**
  Rolling deployments and rollbacks mean two versions always run together.
- [ ] **Who else reads this data?**
  Search for reports, extract jobs, other services and event consumers; check registry subscribers and data catalogue lineage.
- [ ] **Has the migration been run on production-sized data?**
  Measure lock duration, table rewrites and log volume; use concurrent index builds and NOT VALID constraints where available.
- [ ] **Is a lock timeout set?**
  Fail fast and retry rather than letting a waiting ALTER block all traffic.
- [ ] **Is the backfill separate, batched, throttled, idempotent and resumable?**
  It should never block a deployment or flood replicas.
- [ ] **Do new fields have defaults, and are no field numbers or names reused?**
  Essential for Avro and Protocol Buffers; enforced by the registry's compatibility check.
- [ ] **Has any field's meaning changed?**
  Units, currency or semantics changes need a new field name, never a reused one.
- [ ] **Is the contract step tracked?**
  Record a ticket or ADR so the old structure is actually removed once unused.

CHECK YOUR UNDERSTANDING

▼ **A team adds a required field with no default to an Avro schema. Which compatibility does this break, and why?**

Backward compatibility: a consumer using the new schema reading old data finds the field missing and has no default to fill in.

▼ **Why does the dual-write step continue after reads switch to the new column?**

So that rolling back to the previous release, which still reads the old column, finds that column up to date.

▼ **Why should a large backfill not be written as one UPDATE statement inside a migration?**

One huge transaction holds locks for a long time, generates a burst of log that causes replication lag, leaves much cleanup work, cannot be resumed and blocks the deployment; small committed batches avoid all of these.

▼ **What does marking a Protocol Buffers field number as reserved prevent?**

It stops anyone reusing the number of a deleted field for a new field, which would make old data be decoded as the wrong field.

SOURCES

- Danilo Sato: ParallelChange (martinfowler.com, 2014): https://martinfowler.com/bliki/ParallelChange.html
- Pramod Sadalage and Martin Fowler: Evolutionary Database Design: https://martinfowler.com/articles/evodb.html
- Confluent documentation: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Protocol Buffers: Language Guide (proto3), updating a message type: https://protobuf.dev/programming-guides/proto3/
- PostgreSQL documentation: ALTER TABLE: https://www.postgresql.org/docs/current/sql-altertable.html

SECTION 7.8 OF 7.8 · ADVANCED · 27 MIN READ

Analytics, streaming, warehouses, lakes and data mesh

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 6.4 Event-driven
- 7.1 Data ownership

Terms it uses from earlier: Data contract, Schema, Foreign key, Join

THE IDEA IN BRIEF

A supermarket's tills are built to serve each customer quickly, one basket at a time. Nobody would ask the till staff to stop and work out which products sold best in every store over the last two years. Instead, each night the day's sales are sent to head office, where analysts keep years of history arranged for that kind of question. Some figures cannot wait until tomorrow, such as a sudden run on bread, so they are also watched live. Analytics architecture is about building the head office and the live watch without slowing down the tills.

WHAT YOU NEED TO KNOW

- OLTP serves many small transactions by key; OLAP scans and aggregates large volumes; separating them protects the operational systems and lets analytics use columnar, denormalised designs.
- Inmon builds an integrated, normalised enterprise warehouse first; Kimball builds dimensional star schemas per business process linked by conformed dimensions; every fact table needs a declared grain.
- ELT loads raw data and transforms it inside the warehouse; change data capture streams changes from database logs, but published data contracts are more robust than reading another service's tables.
- A lakehouse adds transactions, schema evolution, deletes and time travel to open files on object storage through open table formats such as Delta Lake, Apache Iceberg and Apache Hudi.
- Stream processing must compute by event time, cut streams into tumbling, sliding or session windows, and use watermarks plus an explicit policy for late data.
- Lambda (Marz) combines batch and speed layers at the cost of duplicate logic; Kappa (Kreps, 2014) uses one stream processor and reprocesses by replaying a retained log.

- Data mesh (Dehghani, 2019) rests on domain ownership, data as a product, a self-serve platform and federated computational governance; it suits large organisations with mature domain teams.

Two different workloads: OLTP and OLAP

Everything in this chapter so far has been about online transaction processing (OLTP): many small, fast reads and writes of individual records, such as placing an order, posting a payment or updating a courier's position. Analytics asks a different kind of question. “What was revenue by cuisine and region, week by week, for the last two years?” touches millions of rows, reads only a few columns of each, and changes nothing. This is online analytical processing (OLAP), and it is so different from OLTP that running both on the same database usually serves neither well: one large analytical query can saturate the disks and buffer cache that the checkout depends on, and the normalised schema that keeps transactions correct forces analysts to join a dozen tables.

The standard architectural answer is to separate the workloads. Operational systems keep their own databases, owned by their services as Section 7.1 described. Data is copied, continuously or periodically, into an analytical platform designed for scans: columnar storage that reads only the needed columns, heavy compression, massively parallel query engines and schemas shaped for analysis. The rest of this section is about how that platform is designed, how data reaches it, how streaming changes the picture and who owns the result.

OLTP and OLAP compared

Aspect	OLTP (operational)	OLAP (analytical)
Typical question	Fetch order 81723; mark it delivered	Average delivery time by city by hour, last 12 months
Rows touched per query	One to a few hundred, by key	Millions to billions, by scan
Writes	Constant small inserts and updates	Bulk loads or appends; few updates
Schema	Normalised for integrity (Section 0.5)	Denormalised, dimensional or wide tables
Storage layout	Row-oriented, indexed by key	Column-oriented, compressed, partitioned by date
Users	Customers and applications	Analysts, data scientists, dashboards, ML pipelines
Key qualities	Low latency, consistency, availability	Throughput, query flexibility, history
Examples	PostgreSQL, MySQL, DynamoDB	Snowflake, BigQuery, Redshift, ClickHouse, Databricks SQL

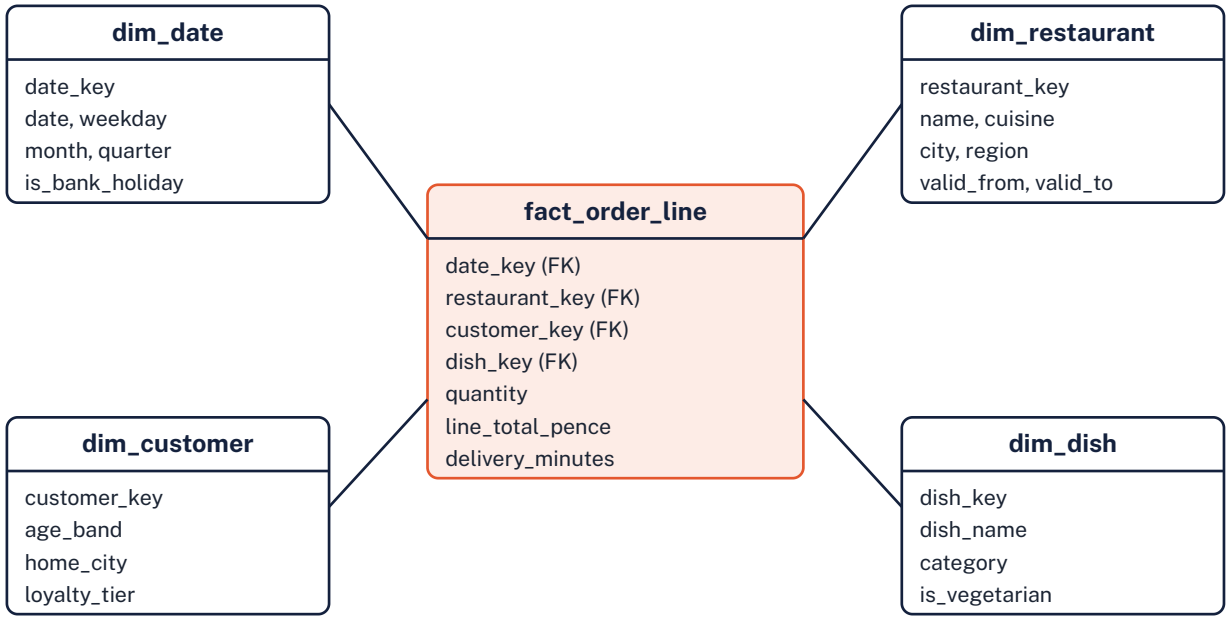
Data warehouses and dimensional modelling

A data warehouse is a database built for analysis that integrates cleaned data from many operational systems and keeps its history. Two schools shaped how warehouses are designed, and practitioners still argue about them. **Bill Inmon**, often called the father of the data warehouse (*Building the Data Warehouse*, 1992), advocates a top-down design: an enterprise-wide, normalised, integrated warehouse is built first as the single source of truth, and departmental data marts are then derived from it. **Ralph Kimball** (*The Data Warehouse Toolkit*, first published 1996, third edition with Margy Ross in 2013) advocates a bottom-up design: build dimensional models one business process at a time, and integrate them through shared, **conformed dimensions** (one agreed definition of customer, date or restaurant used by every model). Inmon's approach gives stronger integration at a higher up-front cost; Kimball's delivers usable results sooner but needs discipline to keep the dimensions conformed. Many modern platforms blend the two: a cleaned, integrated layer feeding dimensional models for consumption.

Kimball's star schema remains the most widely taught analytical model. A central fact table records business events at a declared **grain** (one row per order line, per payment, per courier ping), with numeric measures such as quantity, amount and minutes, and foreign keys to

dimension tables. Each dimension holds descriptive attributes used to filter and group: dates (with weekday, month and bank-holiday flags precomputed), restaurants (name, cuisine, city), customers and dishes. Queries become one fact table joined to a few small dimensions, which columnar engines execute very efficiently. Attributes that change over time, such as a restaurant moving city, are handled by **slowly changing dimension** techniques; the most common, type 2, adds a new dimension row with validity dates, so historical facts still point to the attributes that were true at the time.

A star schema for food-delivery orders



Grain: one row per dish on a delivered order. Facts are numeric and additive; dimensions describe them.

One fact table at a declared grain, surrounded by the dimensions that describe each fact.

```
-- Revenue and average delivery time by cuisine and weekday for last quarter.
-- One join per dimension; the fact table is scanned, never updated.
SELECT r.cuisine,
       d.weekday,
       SUM(f.line_total_pence) / 100.0 AS revenue_gbp,
       AVG(f.delivery_minutes)          AS avg_delivery_min
FROM   fact_order_line f
JOIN   dim_date       d ON d.date_key       = f.date_key
JOIN   dim_restaurant r ON r.restaurant_key = f.restaurant_key
WHERE  d.quarter = '2026-Q2'
  AND  r.region  = 'North West'
GROUP  BY r.cuisine, d.weekday
ORDER  BY revenue_gbp DESC;
-- Caveat: delivery_minutes is per order, so averaging it over order LINES
-- weights big orders more. Grain decisions matter (see the worked example).
```

A typical star-schema query: filter and group by dimension attributes, aggregate the fact measures.

WORKED EXAMPLE

Worked example: declaring the grain

The analytics team builds the fact table above at the grain of one row per dish on a delivered order. The order-level measure `delivery_minutes` was copied onto every line. Two orders are delivered: order A has 1 line and took 20 minutes; order B has 4 lines and took 40 minutes.

Correct average delivery time	Per order: (20 + 40) ÷ 2 = 30 minutes.
Average over the fact rows	Lines: 20, 40, 40, 40, 40. Sum 180 ÷ 5 rows = 36 minutes. Wrong: large orders count four times.
Sum of delivery minutes	180, which has no meaning: the measure is not additive at this grain.
Fix	Keep line-level measures (quantity, line total) in the order-line fact, and put order-level measures (delivery minutes, delivery fee, tip) in a separate fact_order table at the grain of one row per order, sharing the same conformed dimensions.
Check	In fact_order: (20 + 40) ÷ 2 = 30 minutes, and SUM(delivery_fee) now counts each fee once.

Kimball's first design step is to declare the grain precisely, before choosing dimensions or facts, because every measure must be true at that grain.

Mixing grains in one table is one of the most common causes of dashboards that disagree with each other.

Getting data in: ETL, ELT and change data capture

Data reaches the warehouse through pipelines. In classic ETL (extract, transform, load), a separate tool extracts data from source systems, cleans and reshapes it on its own servers, and loads only the result. Since cloud warehouses made storage cheap and compute elastic, **ELT** has become common: extract and load raw data first, then transform it inside the warehouse with SQL, often managed as version-controlled, tested models with a tool such as dbt. ELT keeps the raw data, so transformations can be fixed and rerun; its risk is that raw personal data spreads further than it needs to.

How data is extracted matters as much. Nightly full copies are simple but slow and stale. **Change data capture (CDC)** reads the database's own replication log (PostgreSQL's write-ahead log, MySQL's binlog) with a tool such as Debezium and streams every insert, update and

delete as an event, with low latency and little load on the source. Whatever the mechanism, reading another service's tables directly couples the analytics platform to that service's private schema, exactly the problem Section 7.1 described. The more robust pattern is for each service to publish data deliberately, through events or an agreed export, under a data contract.

Data lakes, lakehouses and open table formats

A data lake (the term is credited to James Dixon, 2010) stores raw data from many sources in its original form, usually as files on cheap object storage such as Amazon S3, often in columnar formats like Apache Parquet. It defers the schema until the data is read, which suits data scientists who want raw detail and suits data whose use is not yet known. Lakes earned a reputation for becoming “data swamps”: without ownership, documentation and quality checks, nobody trusts what is in them. Files on object storage also lack what a warehouse provides: transactions, updates and deletes, schema enforcement and consistent reads while data is being written.

The lakehouse (Armbrust, Ghodsi, Xin and Zaharia, CIDR 2021) adds those warehouse features on top of open files in the lake. The enabling technology is the open table format: **Delta Lake** (created at Databricks), **Apache Iceberg** (created at Netflix) and **Apache Hudi** (created at Uber). Each keeps a metadata layer (a transaction log or a tree of manifest files) that records which data files make up each version of a table. That gives atomic commits, concurrent writers with optimistic concurrency, schema evolution, row-level updates and deletes (needed for erasure requests), and **time travel**: querying a table as it was at an earlier version. Because the format is open, several engines (Spark, Trino, Flink, and increasingly the cloud warehouses themselves) can read and write the same tables, which reduces lock-in. Many lakehouses organise data in layers, popularised by Databricks as the medallion architecture: bronze for raw ingested data, silver for cleaned and conformed data, gold for business-level models such as star schemas.

Stream processing: event time, windows and watermarks

Batch pipelines answer questions about yesterday. Some questions cannot wait: is this card being used fraudulently right now, are couriers in Manchester running late this evening, which restaurants should be paused because their kitchens are overwhelmed? Stream processors such as Apache Flink, Kafka Streams and Spark Structured Streaming compute continuously over unbounded streams of events. The essential concepts come from Google's Dataflow model (Akidau and colleagues, VLDB 2015).

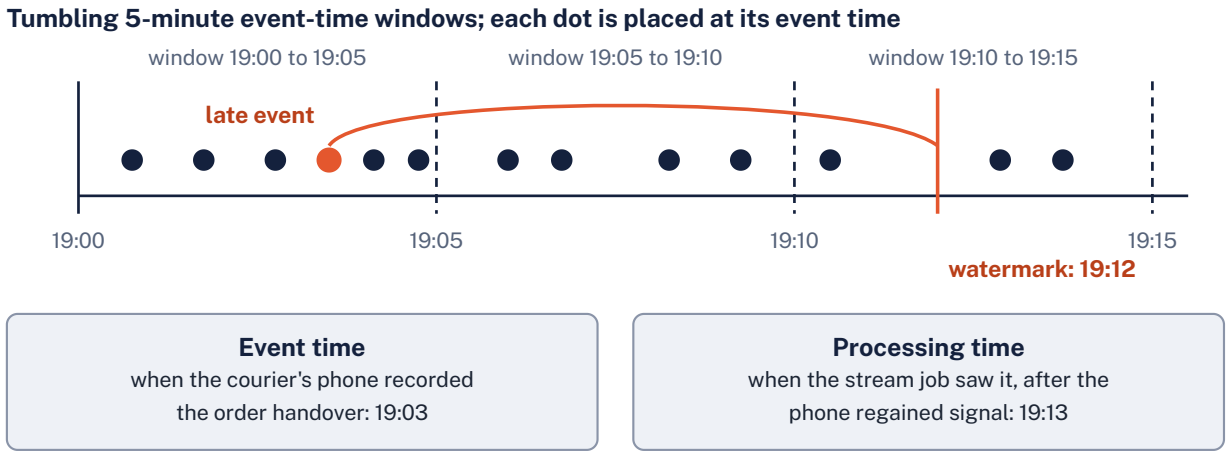
The first is the difference between event time, when something actually happened, and **processing time**, when the system observes it. They differ because of network delays, retries, batching and devices that go offline: a courier's phone in a lift records a handover at 19:03 but

delivers the event at 19:13. Any result that should reflect reality (orders per five minutes, average delivery time per hour) must be computed by event time, or it will shift whenever the pipeline is slow and be wrong when it is replayed.

The second is **windows**, which cut an unbounded stream into finite pieces to aggregate: **tumbling** windows are fixed and non-overlapping (every five minutes), **sliding** windows overlap (the last ten minutes, updated every minute) and **session** windows group bursts of activity per key separated by a gap of inactivity (one customer's browsing session).

The third is the watermark: the processor's moving estimate that all events up to a given event time have probably arrived. When the watermark passes the end of a window, the window's result can be emitted. Watermarks are heuristics, so some events still arrive after their window has closed. The pipeline needs an explicit policy for such **late data**: drop it, send it to a side output for later correction, or keep windows open for an allowed lateness and emit updated results. Waiting longer gives more complete answers; emitting sooner gives fresher ones. That latency-completeness trade-off is the stream-processing version of a quality attribute trade-off and should be decided with the business.

Event time, windows and a late event



The 19:00 window already closed when the watermark passed 19:05; the late event needs a policy: drop it, send it to a side output, or update the window if it is within the allowed lateness.

Windows are defined by event time; the watermark decides when they close, and late events need an explicit policy.

IF YOU ARE NEW TO THIS

Step by step: counting by when it happened

This section's hardest idea is that a stream job must group events by when they happened, not by when they arrived, and must then decide how long to wait for stragglers. Follow one delayed event through a simple count.

- A stream processing job counts the orders each restaurant's kitchen tablet accepts in five-minute windows, to spot kitchens that are overwhelmed. Between 20:40 and 20:45 one busy restaurant accepts 25 orders. Its tablet loses Wi-Fi just after accepting one at 20:43, and sends that acceptance only at 20:57.
- Counted by processing time, when the job sees each event, the 20:40 window shows 24 and the 20:55 window gains an order accepted long before. On a replay tomorrow, the whole evening is read within minutes, and the counts change again.
- Counted by event time, the time recorded in the event itself, the delayed acceptance goes into the 20:40 window where it belongs, and a replay gives the same answer every time.
- That raises a new question: when can the job publish the 20:40 count, if a straggler might still turn up? It uses a watermark, its moving estimate that all events up to a given time have probably arrived. When the watermark passes 20:45, the job publishes 24.
- At 20:57 the 20:43 acceptance arrives, after its window has closed. The pipeline needs a rule decided in advance: drop it, set it aside for a later correction, or allow some lateness and publish an updated count of 25.
- **The lesson:** compute by event time so that results reflect what really happened, and agree with the business how long to wait. Waiting longer gives more complete counts; publishing sooner gives fresher ones.

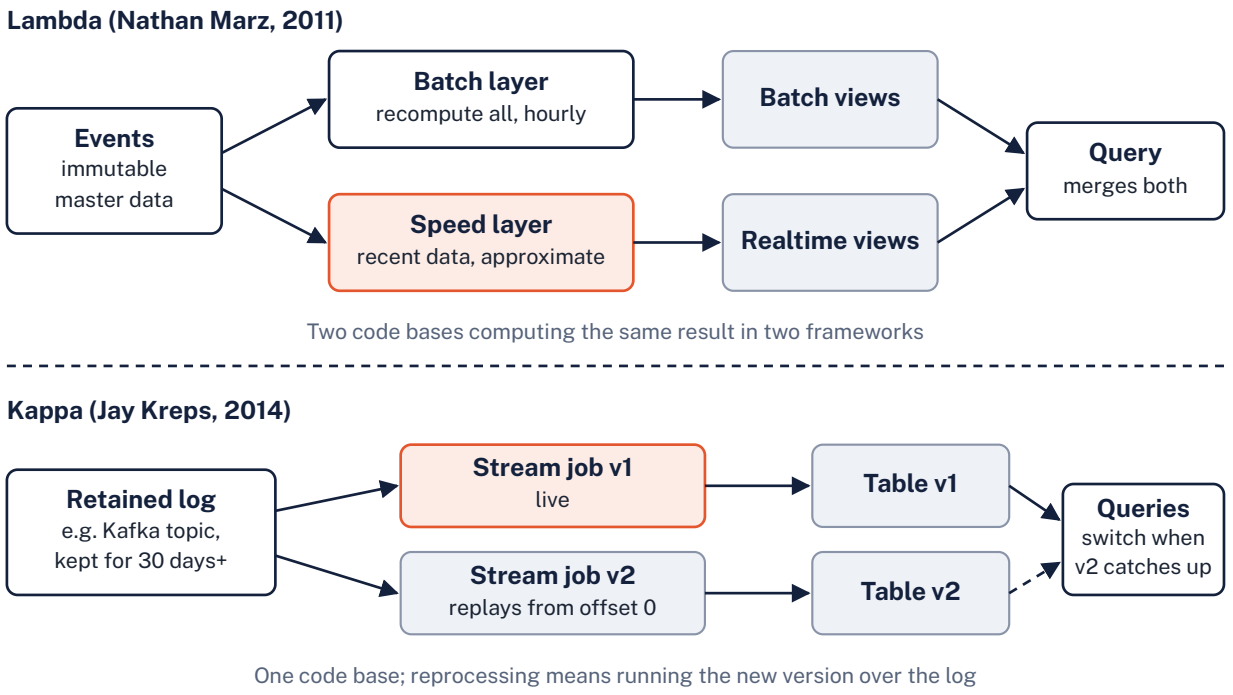
Lambda and Kappa architectures

How should batch and streaming results be combined? Nathan Marz's 2011 essay “How to beat the CAP theorem” described what became known as the **Lambda architecture**: an immutable master dataset feeds a **batch layer** that periodically recomputes accurate views from all data, and a **speed layer** that computes approximate views of recent data in real time; queries merge the two, and each batch run replaces the speed layer's approximations. Lambda gives accuracy and low latency, and the batch layer acts as a safety net for streaming bugs. Its cost is that the same logic must be written and kept identical in two different frameworks.

In 2014 Jay Kreps, one of the creators of Apache Kafka, questioned this design and proposed the **Kappa architecture**, a name he offered half in jest. Use only a stream processor, and retain the input in a log long enough to replay it. To change the logic, start a second instance of the

new job reading from the beginning of the log into a new output table, and switch readers to it once it has caught up. One code base, and reprocessing is just replaying. Kappa depends on long log retention and on stream processors good enough to handle event time and state correctly, which modern engines provide. Today most new designs lean towards Kappa-style streaming with a lakehouse for history, while Lambda-style designs persist where batch recomputation over very long histories is cheaper or where existing batch pipelines are trusted.

Lambda and Kappa side by side



Lambda keeps separate batch and speed paths and merges them at query time; Kappa has one streaming path and reprocesses by replay.

Data mesh: ownership of analytical data

As organisations grew, central data teams became bottlenecks: every new source and every new question queued behind one team that understood neither the source systems nor the business questions very well. In 2019 Zhamak Dehghani, then at Thoughtworks, proposed the data mesh (later expanded in her 2022 book *Data Mesh*). It applies to analytical data the same decentralisation that microservices applied to operational systems, and it rests on four principles.

Domain ownership: the teams closest to the data, such as orders, payments or couriers, own and serve their analytical data, just as they own their services. **Data as a product**: each team publishes data products that are discoverable, documented, trustworthy, addressable and secure, with an owner and quality objectives, treating other teams as customers. **Self-serve data platform**: a platform team provides the storage, pipelines, catalogue and access control

that make publishing a data product cheap, so each domain does not reinvent them. **Federated computational governance**: global rules (identifiers, privacy classifications, interoperability standards) are agreed jointly and enforced automatically by the platform rather than by a central review board.

Data mesh is an organisational design as much as an architecture, and opinions differ on when it pays. Its advocates point to scale: it removes the central bottleneck and puts data quality where the knowledge is. Its critics point out that it needs mature domain teams, a strong platform and real investment, and that smaller organisations are usually better served by a well-run central platform with clear ownership of each dataset. Conway's law (Section 3.7) applies directly: the data architecture will mirror whichever structure the organisation actually adopts.

Quality, lineage, catalogues and privacy

Analytical data is only useful if people trust it. **Data quality** checks run inside pipelines and test freshness (did today's load arrive?), volume (did the row count drop by half?), schema (did a column change type?), validity (are amounts positive and postcodes well formed?) and uniqueness. Failing checks should stop a pipeline or quarantine data rather than silently publish it. Data lineage records where each dataset came from and which transformations produced it, so that when a figure looks wrong, or a source schema changes, you can trace the impact upstream and downstream; the OpenLineage specification standardises how pipelines report it. A **data catalogue** makes datasets discoverable, with descriptions, owners, classifications and lineage in one place.

Analytics platforms concentrate data, which makes them attractive to attackers and risky under data protection law. Apply data minimisation (do not copy personal fields that no analysis needs), pseudonymise identifiers where possible, classify columns so that access policies follow the data, enforce retention periods with automated deletion, and make sure erasure requests reach the lake and warehouse as well as the operational systems; open table formats help because they support row-level deletes. Section 8.7 covers the obligations in detail.

WATCH OUT

Common mistakes

- Running heavy analytical queries against the production OLTP database, or against a replica that the application also depends on for failover.
- Building fact tables without declaring the grain, so that averages and sums are silently wrong.
- Reading other services' private tables directly into the warehouse, so every internal schema change breaks a pipeline.
- Computing streaming aggregates by processing time, so results change whenever the pipeline lags or is replayed.
- Treating a data lake as a dumping ground with no owners, quality checks or catalogue.
- Adopting data mesh vocabulary without funding the self-serve platform or giving domains real ownership.

CHECK YOUR UNDERSTANDING

▼ Why is it a bad idea to run a two-year revenue report against the checkout database?

The report scans millions of rows and competes for disk, memory and CPU with the small, latency-sensitive transactions the checkout needs, and the normalised schema makes the query complex and slow.

▼ What is the first decision Kimball's method asks for when designing a fact table?

Declaring the grain: exactly what one row represents, such as one dish on a delivered order.

▼ A stream job counts orders per minute by processing time. What happens when it falls ten minutes behind and then catches up?

Orders are counted in the minutes when they were processed rather than when they happened, so the counts are wrong, with a dip followed by a spike; computing by event time avoids this.

▼ How does the Kappa architecture handle a change to the processing logic?

It runs the new version of the stream job from the beginning of the retained log into a new output, then switches readers to it once it has caught up and retires the old job.

SOURCES

- Zhamak Dehghani: How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh (2019): https://martinfowler.com/articles/data-monolith-to-mesh.html
- Jay Kreps: Questioning the Lambda Architecture (O'Reilly Radar, 2014): https://www.oreilly.com/radar/questioning-the-lambda-architecture/
- Akidau et al.: The Dataflow Model (VLDB 2015): https://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf
- Armbrust et al.: Lakehouse (CIDR 2021): https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf
- Kimball Group: Dimensional modeling techniques: https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/

Further reading

- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. The standard modern reference on replication, partitioning, transactions, stream processing and derived data; most of this chapter's mechanisms are explained there in depth.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Data ownership, joint ownership techniques, distributed data access patterns and data disintegrators and integrators, with a running case study.
- *A Critique of ANSI SQL Isolation Levels*, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil (1995), Microsoft Research technical report MSR-TR-95-51 (also ACM SIGMOD 1995). The short paper that defined snapshot isolation and the anomalies missing from the SQL standard.
  https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- *Dynamo: Amazon's Highly Available Key-value Store*, Giuseppe DeCandia and colleagues (2007), ACM Symposium on Operating Systems Principles (SOSP). The paper behind leaderless replication, quorums and consistent hashing with virtual nodes in Cassandra and its relatives.
  https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- *Choose Boring Technology*, Dan McKinley (2015), mcfunley.com. A short, influential essay on innovation tokens and the long-term cost of adding technologies, especially databases.
  https://mcfunley.com/choose-boring-technology
- *The Data Warehouse Toolkit (3rd edition)*, Ralph Kimball and Margy Ross (2013), Wiley. The definitive guide to dimensional modelling, star schemas and fact and dimension tables for analytics.
- *Data Mesh: Delivering Data-Driven Value at Scale*, Zhamak Dehghani (2022), O'Reilly Media. The book-length account of domain-owned analytical data, data as a product, a self-serve platform and federated governance.

Glossary

Anomaly (phenomenon)
A wrong result caused by concurrent transactions interfering, such as a dirty read, lost update or write skew; isolation levels are defined by which anomalies they prevent.

Backfill
Filling a new column or table with values for existing rows, usually in small batches so that the live system is not disrupted.

Backward compatibility
A schema change is backward compatible when code using the new schema can still read data written with the old one.

Cache hit ratio
The fraction of lookups that the cache can answer from its own contents; the rest are misses that go to the slower source.

Cache invalidation
Removing or marking stale a cached copy when the original data changes, so that later reads fetch the new value.

Cache stampede
A surge of requests that all miss the cache at the same moment, usually when a popular item expires, and all hit the slower source at once; closely related to the thundering herd problem.

Cache-aside
A caching pattern in which the application checks the cache, loads from the database on a miss and puts the result into the cache itself; also called lazy loading.

Command-query separation (CQS)
Bertrand Meyer's principle that a method should either change state (a command) or return an answer (a query), but not both.

Consistent hashing
A way of assigning keys to nodes on a hash ring so that adding or removing a node moves only about 1/N of the keys (Karger and colleagues, 1997).

Crypto-shredding
Making data unreadable by destroying the encryption key that protects it, used to erase personal data from logs and backups that cannot be edited.

Data contract
An explicit, versioned agreement between a data owner and its consumers covering schema, meaning, quality, freshness, ownership and how changes are announced.

Data lake
A large, cheap store, usually object storage, that keeps raw data from many sources in its original form for later analysis.

Data lineage
A record of where a piece of data came from and every transformation it went through on the way.

Data mesh
Zhamak Dehghani's approach to analytical data in which domain teams own and publish their data as products on a self-serve platform under federated governance.

Data product
A dataset published by a domain team for others to use, with an owner, documentation, quality guarantees and a stable interface.

Data warehouse
A database built for analysis that integrates cleaned data from many operational systems and keeps its history.

Dimension table
A table of descriptive attributes, such as a restaurant's name, cuisine and city, used to filter and group facts.

Distributed SQL (NewSQL)
Relational databases such as Spanner and CockroachDB that keep SQL and ACID transactions while partitioning and replicating data across many nodes or regions.

ETag
An HTTP response header carrying an identifier for one version of a resource, which a client can send back in If-None-Match to ask whether it has changed.

Event stream
The ordered sequence of events for one aggregate, numbered by version, from which that aggregate's state is rebuilt.

Event time
The time at which an event actually happened, as opposed to processing time, when a system observes it.

Expand and contract
Making a breaking change in safe steps: add the new structure alongside the old, migrate readers, writers and data, then remove the old structure. Also called parallel change.

Fact table
The central table of a star schema, holding one row per business event at a stated grain, with numeric measures and keys to dimensions.

Forward compatibility
A schema change is forward compatible when code using the old schema can still read data written with the new one, usually by ignoring fields it does not know.

Integration database
A database shared by several applications, which integrate through it by reading and writing the same tables. Martin Fowler (2004) advises against it because it couples the applications deeply.

Isolation level
A setting that says how much concurrent transactions may affect each other: read uncommitted, read committed, repeatable read, snapshot or serializable. Names mean different things in different databases.

Joint ownership
A situation in which a few services all write the same data. It can be resolved by splitting the table, a small shared data domain, delegating writes to one owner or merging the services.

Lakehouse
An architecture that adds warehouse features such as transactions, schemas and fast queries to data held in open file formats on object storage.

Leader-follower replication
A replication scheme in which one node (the leader or primary) accepts all writes and streams its changes to followers (replicas), which can serve reads.

Leaderless replication
A Dynamo-style scheme in which clients write to and read from several replicas directly and use quorums instead of a single leader.

Least frequently used (LFU)
An eviction policy that removes the item that has been read least often when the cache is full.

Least recently used (LRU)
An eviction policy that removes the item that has gone longest without being read when the cache is full.

Lost update
An anomaly in which two transactions read a value, each computes a new one and writes it back, so one change silently overwrites the other.

Migration
A versioned script that changes a database schema or its data from one version to the next, kept in version control and applied automatically.

Multi-leader replication
A scheme in which several nodes accept writes and replicate to each other, requiring a way to detect and resolve conflicting writes.

Multi-version concurrency control (MVCC)
A technique in which writes create new row versions instead of overwriting, so readers see a consistent snapshot without blocking writers.

OLAP
Online analytical processing: workloads of large queries that scan and aggregate many records, such as revenue by region by month.

OLTP
Online transaction processing: workloads of many small reads and writes of individual records, such as placing orders or posting payments.

Open table format
A specification, such as Apache Iceberg, Delta Lake or Apache Hudi, that organises data files and metadata so many engines can treat them as tables with transactions and time travel.

Optimistic concurrency control
Detecting conflicts instead of preventing them: a write succeeds only if a version number or timestamp is unchanged since the data was read, otherwise the writer retries.

Partitioning (sharding)
Splitting a data set so that each machine holds only part of it, chosen by a partition key, to scale writes and storage beyond one machine.

Pessimistic locking
Preventing conflicts by locking rows when they are read (for example with SELECT … FOR UPDATE), so other transactions wait until the lock is released.

Polyglot persistence
Using different kinds of database for different data within one system, chosen by how each data set is used, at the cost of more technologies to run.

Projection
Code that listens to changes or events and keeps a read model up to date; also used for the read model it produces.

Replication
Keeping copies of the same data on several machines for availability, read capacity and lower latency near users.

Replication lag
The delay between a write on the leader and its appearance on a follower; it causes users to read stale data.

Request coalescing
Letting only one request fetch a missing item while other requests for the same item wait for and share its result; also called single flight.

Schema registry
A service that stores every version of each message schema and rejects new versions that break the agreed compatibility rule.

Search engine
A store built around an inverted index from words to documents, providing relevance-ranked, typo-tolerant full-text search; usually fed from a system of record.

Secondary index
An index on an attribute other than the primary or partition key. In a partitioned store it is either local to each partition or global and partitioned by the indexed value.

Serializability
The strongest isolation guarantee: concurrent transactions produce the same result as if they had run one at a time in some order.

Snapshot
A stored copy of an aggregate's state at a known event version, used to avoid replaying a long stream from the beginning; it is a cache, not a source of truth.

Snapshot isolation
An isolation level in which each transaction reads a consistent snapshot of committed data from its start and conflicting writes to the same row are rejected. It prevents lost updates but allows write skew.

Split brain
A failure in which two nodes both believe they are the leader and both accept writes, so their data diverges.

Star schema
A dimensional model with a central fact table of measurements surrounded by dimension tables that describe them, such as date, customer and product.

System of record
The authoritative source for a piece of data, whose value wins when copies disagree, such as a bank's core ledger for account balances.

Time to live (TTL)
How long a cached item may be used before it is considered stale and must be refreshed or revalidated.

Time-series database
A database optimised for timestamped measurements that arrive in order, with compression, downsampling and cheap deletion of old data.

Tolerant reader
A consumer that reads only the fields it needs and ignores anything it does not recognise, so that providers can add fields without breaking it.

Upcasting
Converting an event stored in an old schema version into the current version as it is read, so that application code only has to handle one shape.

Vector database
A store for embeddings (lists of numbers representing meaning) that finds the items most similar to a query, usually with approximate nearest-neighbour indexes.

Watermark
A stream processor's moving estimate that all events up to a certain event time have probably arrived, used to decide when a window's result can be emitted.

Write skew
An anomaly in which two transactions read overlapping data, each decides a change is safe, and they update different rows, together breaking a rule each one checked.

Write-ahead log (WAL)
An append-only log on disk in which a database records each change before applying it, so that committed work survives a crash; replicas are often fed from it.

Write-behind cache
A caching pattern in which writes are acknowledged once they reach the cache and are written to the underlying store later, in the background; also called write-back.

Write-through cache
A caching pattern in which every write goes to the cache and to the underlying store before it is acknowledged, so the cache is never older than the store.