

A PRACTICAL GUIDE

NoSQL Databases Explained

Document, Key-Value, Wide-Column, Graph, Search, Time-Series, and Vector Databases

MongoDB · Redis · DynamoDB · Cassandra · Neo4j
Elasticsearch · OpenSearch · Vector Search

MODEL · PARTITION · REPLICATE · SCALE

THENEXBOOKS

NoSQL Databases Explained

A Practical Guide to Document, Key-Value, Wide-Column,
Graph, Search, Time-Series, and Vector Databases

First Edition

THENEXBOOKS

NoSQL Databases Explained
A Practical Guide to Document, Key-Value, Wide-Column, Graph, Search, Time-Series, and Vector Databases

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages — including data loss — resulting from the use of the information contained herein. Database products, their features, and their licenses change over time; always consult the official documentation and license terms for the versions you use.

This book is an independent publication and is not affiliated with, sponsored by, or endorsed by any database vendor. MongoDB, Redis, Valkey, Amazon DynamoDB, Apache Cassandra, ScyllaDB, Neo4j, Elasticsearch, OpenSearch, InfluxDB, TimescaleDB, PostgreSQL, Docker, and other product names are trademarks of their respective owners; their use here is for identification only. Pulsefit is a fictional company.

Preface

For decades, "database" meant "relational database". Tables, rows, joins, and SQL solved an enormous range of problems — and still do. But as applications grew to serve millions of users, stream data from devices, rank players in real time, search free text, recommend friends, and power AI features, teams discovered workloads where a different data model fits better. The result is a diverse family of systems grouped under a loose label: **NoSQL**.

That diversity is both the strength and the challenge. A document database, a key-value store, a wide-column database, a graph database, a search engine, a time-series database, and a vector database make very different trade-offs. Choosing well — and modeling data correctly once you have chosen — requires understanding how each one stores data, distributes it, keeps it consistent, and answers queries.

This book explains those systems clearly and practically. We start with the ideas that all of them share: data models, partitioning, replication, consistency, and query-first modeling. Then we work hands-on with the leading database in each family — MongoDB, Redis, DynamoDB, Cassandra, Neo4j, Elasticsearch and OpenSearch, time-series databases, and vector search — before turning to the application-level concerns of choosing, integrating, migrating, securing, and operating them. Throughout, we stay vendor-neutral and honest about trade-offs, including when a relational database is still the right answer.

Who This Book Is For

- **Backend and full-stack developers** who need to choose and use the right database for each part of an application.
- **Developers who know SQL** and want to understand how NoSQL systems think differently about data.
- **Data engineers and architects** designing polyglot systems, pipelines, and change data capture.
- **Students and interview candidates** who want a solid grasp of distributed data concepts such as CAP, replication, and sharding.

What You Will Learn

- The NoSQL families, their data models, and the distributed-systems concepts behind them: partitioning, replication, quorums, CAP, and PACELC.
- How to model data around access patterns, and when to denormalize.
- MongoDB in depth: CRUD, schema design patterns, aggregation, indexes, transactions, replication, and sharding.
- Redis data structures and patterns, persistence and clustering, and DynamoDB single-table design.

- Cassandra, Neo4j, Elasticsearch and OpenSearch, time-series databases, and vector search for AI applications.
- How to choose a database, integrate it into application code, evolve schemas, capture changes, secure data, and monitor performance.
- Four end-to-end projects that build a polyglot fitness platform.

How This Book Is Organized

Part I — NoSQL Foundations (Chapters 1–4) introduces the families, data models, distributed-systems essentials, and access-pattern modeling. **Part II — Document Databases with MongoDB** (Chapters 5–9) goes deep on the most widely used document database. **Part III — Key-Value and In-Memory Databases** (Chapters 10–13) covers Redis and DynamoDB. **Part IV — Wide-Column, Graph, Search, Time-Series, and Vector Databases** (Chapters 14–19) explores the specialized families. **Part V — Building Applications with NoSQL** (Chapters 20–24) covers choosing, integrating, migrating, securing, and operating databases. **Part VI — Real-World Projects** (Chapters 25–28) brings everything together. The appendices provide a command cheat sheet, modeling templates, a production checklist, and a glossary.

Every chapter ends with a **Try It Yourself** section — designed to run locally with Docker — and a **Key Takeaways** box.

The Running Example: Pulsefit

Throughout the book we build the data layer of **Pulsefit**, a fictional fitness and wellness platform. Members wear fitness bands, log workouts, join clubs and challenges, follow friends, and watch an activity feed and leaderboards. Coaches publish training plans, and devices stream heart-rate and step data. Pulsefit deliberately uses several databases, each for the job it does best — which lets us compare the NoSQL families on the same, familiar domain.

Versions and Licensing

Examples target recent versions of each database. Where a feature arrived in a specific release, we say so; otherwise we write "recent versions" and point you to the official documentation. Several NoSQL products have changed their licenses in recent years, and some have community forks; we describe that history neutrally where it matters, but this book does not give legal advice — review the license terms for your own use case.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause data loss, outages, or security problems.

Code appears in `monospace`. When a code block shows a file, its path appears above it. Connection strings and credentials use placeholders and are never real.

Contents

Preface

PART I — NOSQL FOUNDATIONS

1 What Is NoSQL? Beyond the Relational Model

2 Data Models: Key-Value, Document, Wide-Column, Graph, Search, Time-Series, and Vector

3 Distributed Systems Essentials: Partitioning, Replication, and Consistency

4 Modeling for Access Patterns: Thinking in Queries

PART II — DOCUMENT DATABASES WITH MONGODB

5 MongoDB Fundamentals: Documents, Collections, and CRUD

6 Schema Design in MongoDB: Embedding, Referencing, and Patterns

7 Querying and Aggregation in MongoDB

8 Indexes, Performance, and Transactions in MongoDB

9 Replication, Sharding, and Operating MongoDB

PART III — KEY-VALUE AND IN-MEMORY DATABASES

10 Redis Fundamentals: Data Structures and Commands

11 Redis Patterns: Caching, Sessions, Rate Limiting, Leaderboards, and Streams

12 Redis Persistence, Replication, and Clustering

13 Amazon DynamoDB: Keys, Indexes, and Single-Table Design

PART IV — WIDE-COLUMN, GRAPH, SEARCH, TIME-SERIES, AND VECTOR DATABASES

14 Apache Cassandra: Architecture and Data Modeling

15 CQL, Consistency Levels, and Operating Cassandra

16 Graph Databases with Neo4j and Cypher

17 Search Engines as Databases: Elasticsearch and OpenSearch

18 Time-Series Databases

19 Vector Databases and Semantic Search

PART V — BUILDING APPLICATIONS WITH NOSQL

20 Choosing the Right Database: A Decision Framework

21 NoSQL in Application Code: Drivers, ODMs, and Connection Management

22 Data Migration, Schema Evolution, and Change Data Capture

23 Security, Backup, and Disaster Recovery

24 Performance, Monitoring, and Troubleshooting

PART VI — REAL-WORLD PROJECTS

25 Project: Pulsefit Profiles, Workouts, and Activity Feed with MongoDB

26 Project: Pulsefit Real-Time Leaderboards, Sessions, and Rate Limiting with Redis

27 Project: Pulsefit at Scale — DynamoDB Challenges and Cassandra Telemetry

28 Project: Polyglot Pulsefit — Graph Recommendations, Search, Vectors, and Next Steps

APPENDICES

A NoSQL Command Cheat Sheet

B Data Modeling Templates and Reference

C Production Checklist

D Glossary

PART I

NoSQL Foundations

What NoSQL means, the major data models, the distributed-systems ideas every NoSQL database relies on, and how to model data around the queries you need to answer.

1 · What Is NoSQL?

2 · Data Models

3 · Partitioning, Replication, and Consistency

4 · Modeling for Access Patterns

CHAPTER 1 · PART I — NOSQL FOUNDATIONS

What Is NoSQL? Beyond the Relational Model

NoSQL is not a single technology but a family of database designs that grew out of real problems with scale, availability, and changing data. In this chapter we trace where it came from, what the relational model still does brilliantly, and how the major NoSQL families differ, and we introduce Pulsefit, the fitness platform we will build on throughout the book.

Why This Question Matters

If you have written backend software in the last twenty years, you have almost certainly used a relational database. Tables, rows, primary keys, foreign keys, joins, and SQL are the shared vocabulary of our industry. They are taught in universities, supported by every framework, and understood by every data analyst. So when a team proposes storing member profiles in MongoDB, caching sessions in Redis, or pushing sensor data into Cassandra, a reasonable engineer asks: *why?* What does this buy us that PostgreSQL or MySQL does not, and what does it cost?

That question is the heart of this book. NoSQL databases are not better or worse than relational databases in the abstract. They make different trade-offs, and those trade-offs line up with some workloads beautifully and with others terribly. A database that shines at storing a billion small sensor readings per day may be painful for ad-hoc financial reporting. Choosing well requires understanding the designs, not just the brand names.

A Short History of Storing Data

Databases did not begin with tables. Early commercial systems in the 1960s used **hierarchical** and **network** models. In a hierarchical database, records were arranged in a tree: a customer record owned order records, which owned line items. In a network database, records could be connected by explicit pointers in more flexible ways. Both approaches were fast for the access paths they were designed around, but they forced application programs to navigate the data physically, following pointers from record to record. If the business wanted a new kind of question answered, programmers often had to rewrite navigation code.

In 1970, Edgar F. Codd published his paper describing the **relational model**. The idea was radical in its simplicity: represent all data as relations (tables) of tuples (rows), and let users describe *what* data they want rather than *how* to navigate to it. The database would figure out the access path. Over the

following decade, research systems proved that this could be efficient, SQL emerged as the dominant query language, and commercial relational database management systems (RDBMSs) appeared. By the 1990s, the relational model had become the default choice for nearly every business application.

The Web Changes the Workload

The 2000s brought a different kind of application. Large consumer web services had to serve very large numbers of users around the world, around the clock, with traffic that could grow unpredictably. Their data was often semi-structured (user-generated content, logs, clickstreams), their read and write volumes were enormous, and downtime was highly visible. The traditional answer to growth, buying a bigger server (**scaling up** or vertical scaling), eventually runs out of road: there is always a largest machine, and a single machine is a single point of failure.

Engineers at these companies started **scaling out** (horizontal scaling): spreading data and load across many commodity machines. Doing that with a traditional RDBMS was possible but painful. Teams sharded MySQL by hand, wrote routing logic in the application, gave up cross-shard joins and transactions, and built caching layers in front of everything. At that point they were already living without many relational features, so some asked whether a database designed from the ground up for distribution would be simpler.

Two influential research papers shaped what came next. Google's **Bigtable** paper (2006) described a distributed storage system for structured data organized by row key, column family, and timestamp. Amazon's **Dynamo** paper (2007) described a highly available key-value store that used consistent hashing, replication across nodes, and eventual consistency to keep a shopping cart writable even during failures. Many open-source projects borrowed ideas from one or both. Apache Cassandra, for example, combined a Dynamo-style distribution design with a Bigtable-style data model.

Where the Name Came From

The word "NoSQL" itself has a slightly accidental history. It was used in the late 1990s as the name of a small relational database that did not expose a SQL interface. The term was revived in 2009 as a short, catchy label for a meetup about open-source, distributed, non-relational databases. It stuck, even though it describes these systems by what they are not rather than what they are. Over time the community reinterpreted it as **"Not only SQL"**, which is a more useful framing: NoSQL databases complement relational ones rather than replacing them.

> **NOTE**
>
> "NoSQL" is a marketing-era umbrella term, not a technical specification. There is no standard that a database must meet to be called NoSQL, and several NoSQL databases now offer SQL-like query languages. When you read "NoSQL" in this book, read it as "a non-relational data model, usually designed with distribution in mind."

What the Relational Model Does Well

Before we celebrate alternatives, we should be precise about what we would be giving up. Relational databases are excellent at a long list of things, and many teams that adopt NoSQL later discover they miss some of them.

Declarative Queries and Ad-Hoc Questions

SQL lets you ask questions nobody anticipated when the schema was designed. A product manager can ask "which coaches have more than fifty members on plans longer than eight weeks?" and an analyst can answer it with a join and a `GROUP BY`, without changing the storage layout. The query optimizer chooses indexes and join algorithms. This flexibility is enormously valuable, and most NoSQL systems deliberately trade some of it away in exchange for predictable performance on known access patterns.

Normalization and a Single Source of Truth

In a normalized schema, each fact lives in exactly one place. A coach's display name is stored once in the `coaches` table, and every training plan references it by ID. Rename the coach, and every query sees the new name immediately. As we will see in Chapter 4, NoSQL modeling often duplicates data on purpose, and then you own the job of keeping the copies in sync.

Transactions and Integrity

Relational databases provide **ACID** transactions: atomicity (all or nothing), consistency (constraints hold before and after), isolation (concurrent transactions do not see each other's partial work, to a degree set by the isolation level), and durability (committed data survives a crash). Foreign keys, unique constraints, and check constraints let the database itself reject invalid data. For money, inventory, bookings, and anything where two users must not both "win", these guarantees are hard to replace.

Maturity and Ecosystem

Decades of engineering have gone into relational query planners, backup tools, replication, monitoring, and reporting integrations. Almost every BI tool, ORM, and data pipeline speaks SQL. That ecosystem is a real advantage, not a nostalgic one.

Here is how a small slice of Pulsefit would look in a classic relational schema. Notice how cleanly it expresses relationships and constraints.

sql/pulsefit_relational.sql

```
CREATE TABLE members (
  member_id    TEXT PRIMARY KEY,          -- e.g. 'm_1042'
  display_name TEXT NOT NULL,
  email        TEXT NOT NULL UNIQUE,
  joined_at    TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

```
CREATE TABLE workouts (
  workout_id   BIGSERIAL PRIMARY KEY,
  member_id    TEXT NOT NULL REFERENCES members(member_id),
  kind         TEXT NOT NULL CHECK (kind IN ('run', 'ride', 'strength', 'yoga')),
  started_at   TIMESTAMPTZ NOT NULL,
  duration_sec INTEGER NOT NULL CHECK (duration_sec > 0)
);

CREATE TABLE workout_sets (
  workout_id   BIGINT NOT NULL REFERENCES workouts(workout_id) ON DELETE CASCADE,
  set_no       INTEGER NOT NULL,
  exercise     TEXT NOT NULL,
  reps         INTEGER,
  weight_kg    NUMERIC(6,2),
  PRIMARY KEY (workout_id, set_no)
);

-- An ad-hoc question nobody planned for:
SELECT m.display_name, count(*) AS strength_sessions
FROM members m
JOIN workouts w ON w.member_id = m.member_id
WHERE w.kind = 'strength'
  AND w.started_at >= now() - interval '30 days'
GROUP BY m.display_name
ORDER BY strength_sessions DESC
LIMIT 10;
```

This is good design. If Pulsefit were a small product with modest traffic and a team fluent in SQL, a single PostgreSQL instance could run the whole thing for a long time. We will return to that point in Chapter 20, because "stay relational" is frequently the right answer.

Why NoSQL Emerged

So why did so many teams look elsewhere? The motivations fall into a handful of recurring themes. Few projects face all of them, but most NoSQL adoptions can be traced to at least one.

1. Horizontal Scale

Many NoSQL databases were designed so that data is automatically partitioned across nodes and you add capacity by adding machines. The data model is shaped to make that work: if every query includes a partition key, the database can route it to one small group of nodes without coordinating across the whole cluster. Relational databases can also be distributed (and several modern distributed SQL systems do so impressively), but general-purpose joins and multi-row transactions across partitions are inherently harder to scale than single-partition reads and writes.

2. Availability Across Failures

When you run on many machines, something is always failing: a disk, a node, a network link, an entire data center. Systems like Cassandra and DynamoDB were designed so that data is replicated automatically and requests can continue to succeed while some replicas are unreachable. That goal forces decisions about consistency that we will study carefully in Chapter 3.

3. The Impedance Mismatch

Application code works with objects and nested structures: a workout has a list of sets, each with an exercise and a list of heart-rate zones. Relational storage flattens that into several tables, and an ORM or hand-written code reassembles it. This translation is called the **object-relational impedance mismatch**. Document databases reduce it by storing the nested structure directly, so what you read is close to what your code uses.

4. Schema Flexibility

Fast-moving products change their data shape constantly. A new device model reports blood oxygen; a new workout type needs a "laps" field. In a relational database, each change is a migration. Many NoSQL systems let records in the same collection have different fields, which lets teams ship faster, with the important caveat that the schema has not disappeared, it has moved into application code. We will come back to that caveat under myths.

5. Specialized Workloads

Some workloads simply do not look like rows and joins. Traversing a social graph six hops deep, ranking documents by text relevance, storing millions of timestamped measurements per minute, or finding the nearest neighbors of a 768-dimensional vector are all possible in SQL, but specialized engines are built around those access patterns from the storage layer up. The NoSQL movement made room for these purpose-built engines to be taken seriously as primary data stores.

The NoSQL Families at a Glance

"NoSQL" covers several quite different designs. This book organizes them into seven families. Some products span more than one family (these are often called **multi-model** databases), but the families are a useful map.

Family	Core idea	Typical strengths	Examples
Key-value	Opaque value looked up by a unique key	Very fast point reads and writes; caching; sessions	Redis, Valkey, DynamoDB
Document	Self-describing nested records (JSON-like)	Natural fit for objects; flexible fields; rich queries	MongoDB, Couchbase
Wide-column	Rows grouped by partition key, sorted by clustering columns	Massive write throughput; time-ordered data; multi-region	Apache Cassandra, ScyllaDB, HBase

Graph	Nodes and relationships with properties	Multi-hop traversals; recommendations; fraud rings	Neo4j, Amazon Neptune
Search	Inverted index over analyzed text	Full-text relevance; faceting; log analytics	Elasticsearch, OpenSearch
Time-series	Measurements indexed by time and tags	High ingest; downsampling; retention policies	InfluxDB, TimescaleDB
Vector	High-dimensional embeddings with similarity search	Semantic search; recommendations; retrieval for AI	pgvector, Atlas Vector Search

A few observations about this table are worth making now. First, the boundaries are blurry. DynamoDB is usually called a key-value store, but its items are document-like and it supports sorted ranges within a partition, which makes it resemble a wide-column store in modeling practice. Second, some entries are not "NoSQL" in the strict sense: TimescaleDB and pgvector are PostgreSQL extensions, fully SQL. We include them because the *data model* they serve (time-series, vectors) is the point, not the query language. Third, the table says nothing about scale or speed in absolute terms, deliberately. Performance depends on your data, your access patterns, your hardware, and your configuration far more than on the family label.

Chapter 2 takes the same Pulsefit data and models it in each family side by side, so you can see the differences concretely rather than from a table.

Meet Pulsefit

Throughout this book we build pieces of **Pulsefit**, a fictional fitness and wellness platform. Pulsefit is invented, but its requirements are typical of modern consumer products, and it is deliberately varied enough to justify several databases.

The Product

- **Members** wear a fitness band (a **Device**, with IDs like `dev_7Q2K`) and use a mobile app. A member has an ID like `m_1042`.
- Members log **Workouts** made up of **Exercises** (sets, reps, weights, distances, heart-rate zones).
- **Coaches** publish **Training Plans**: multi-week programs with daily sessions.
- Members join **Clubs** and enter **Challenges** ("run 100 km in March"); each attempt is a **ChallengeEntry**.
- Members **Follow** each other and see an activity feed of **FeedItems** and leaderboards.
- Devices stream **Readings**: heart rate every few seconds during workouts and step counts throughout the day.

The Architecture

Pulsefit uses **polyglot persistence**: each workload lives in the database whose model fits it best. This is not a recommendation that every product should run eight databases. Every additional database is another system to secure, monitor, back up, upgrade, and understand. Pulsefit is a teaching device that lets us compare the families on one coherent domain. In Chapter 20 we discuss how a real team would decide how many of these to adopt.



Here is how responsibilities are split, and the one-line reason for each choice:

- **MongoDB** stores member profiles, training plans, workouts, and the activity feed, because these are naturally nested documents read as a whole.
- **Redis** holds sessions, caches, leaderboards (sorted sets), rate-limit counters, and lightweight event streams, because these need very low latency and fit in memory.
- **DynamoDB** keeps workout history and challenge entries at scale using a single-table design, because the access patterns are well known and a managed service removes operational burden.
- **Cassandra** ingests high-volume device telemetry partitioned by device and day, because the workload is write-heavy and append-only.
- **Neo4j** models the social graph (follows, club membership) and powers friend and plan recommendations, because those queries are multi-hop traversals.
- **Elasticsearch or OpenSearch** provides full-text search across plans, clubs, and exercises.
- A **time-series database** holds per-member daily metrics and their downsampled history.
- **Vector search** powers semantic search of training plans and an "AI coach" feature. Model names in this book are placeholders like `"<your-embedding-model>"`.

One Workout, Two Shapes

To make the impedance mismatch concrete, here is the same workout that required three relational tables, stored as a single MongoDB document. We insert it from `mongosh`, the MongoDB shell.

mongosh: insert a workout document

```
use pulsefit

db.workouts.insertOne({
  memberId: "m_1042",
  kind: "strength",
  startedAt: ISODate("2026-03-14T07:05:00Z"),
  durationSec: 2700,
  deviceId: "dev_7Q2K",
  sets: [
    { setNo: 1, exercise: "back squat", reps: 8, weightKg: 70 },
    { setNo: 2, exercise: "back squat", reps: 8, weightKg: 75 },
    { setNo: 3, exercise: "romanian deadlift", reps: 10, weightKg: 60 }
  ],
  heartRate: { avg: 128, max: 164 },
  tags: ["legs", "morning"]
})

// Read the whole workout back in one round trip, no joins:
db.workouts.findOne({ memberId: "m_1042" }, { sets: 1, heartRate: 1 })
```

The document mirrors the way the app displays a workout. There is no join, and the entire workout is read or written atomically because it is a single document. On the other hand, that "ad-hoc" question from earlier (top members by strength sessions in 30 days) now requires an aggregation pipeline rather than a simple join, and if exercise names needed to be renamed globally, we would have to update them inside every document. Neither model is free.

And here is a leaderboard, the kind of thing Redis makes almost trivially easy, using `redis-cli`:

redis-cli: weekly distance leaderboard

```
ZINCRBY lb:distance:2026-W11 5.2 m_1042
ZINCRBY lb:distance:2026-W11 8.0 m_2210
ZINCRBY lb:distance:2026-W11 3.4 m_1042
ZREVRANGE lb:distance:2026-W11 0 9 WITHSCORES
ZREVRANK lb:distance:2026-W11 m_1042
EXPIRE lb:distance:2026-W11 1209600
```

A sorted set keeps members ordered by score as you update it, so "top ten" and "what is my rank?" are single commands rather than a sort over every row. Chapter 11 builds this out properly.

Common Myths About NoSQL

Because "NoSQL" started as a buzzword, a lot of folklore travels with it. Clearing up these myths early will save you from design mistakes later.

Myth 1: "NoSQL means schemaless"

Your data always has a schema, because your code expects certain fields with certain types. The question is *where* that schema is enforced. A relational database enforces it on write (**schema-on-write**). Many NoSQL databases accept any shape and leave enforcement to readers (**schema-on-read**) or to optional validation rules. MongoDB, for example, supports JSON Schema validation with `$jsonSchema`. "Flexible schema" is the accurate phrase. Teams that treat it as "no schema" tend to accumulate documents in five historical shapes and a tangle of `if (doc.fieldX === undefined)` checks. Chapter 22 covers schema evolution in depth.

Myth 2: "NoSQL databases do not support transactions"

This was closer to true in the early years and is misleading today. Single-record (single-document, single-item) operations are atomic in essentially all mainstream NoSQL databases, and good modeling often keeps related data in one record precisely so that updates are atomic. Beyond that, MongoDB has supported multi-document ACID transactions since version 4.0 (replica sets) and 4.2 (sharded clusters); DynamoDB offers transactional APIs across multiple items; Redis offers `MULTI` / `EXEC` and Lua scripts with their own specific semantics; Cassandra offers lightweight transactions for compare-and-set on a single partition. The guarantees, limits, and costs differ from system to system, and you must read the details, but "no transactions" is not accurate.

Myth 3: "NoSQL is always faster than SQL"

No database is fast in general. A key-value lookup by primary key is fast in PostgreSQL and fast in Redis; Redis avoids some overhead by keeping data in memory and doing less work per request, but PostgreSQL with a warm cache is also quick. A NoSQL database used against its design (for example, scanning an entire Cassandra table to answer an unplanned question) can be dramatically slower than a well-indexed relational query. Speed comes from matching the data model and access patterns to the engine. Be suspicious of any claim of "N times faster" that does not come with the exact workload, hardware, and configuration used to measure it, and measure your own workload before deciding.

Myth 4: "Relational databases cannot scale"

Relational databases scale vertically a long way, scale reads with replicas, and can be sharded (by the application, by middleware, or by distributed SQL systems designed for it). What is genuinely harder is scaling arbitrary cross-partition joins and transactions. If your workload avoids those, a sharded relational setup can work well; if it depends on them, no database makes them free.

Myth 5: "You have to pick one"

Most real systems mix. A relational database as the system of record plus Redis as a cache plus a search engine for text queries is an extremely common architecture. PostgreSQL itself offers JSONB columns for document-style data, full-text search, and extensions for time-series and vectors. The skill is choosing the *smallest* set of databases that meets your needs, not the largest.

Myth 6: "Eventual consistency means the data might be wrong forever"

Eventual consistency means that if writes stop, all replicas will converge to the same value after some period.Many systems let you request stronger consistency for individual operations when you need it. It does mean your application must tolerate reading a stale value in some situations, and you must design for that deliberately. Chapter 3 explains exactly what the guarantees are.

> **WARNING**
>
> The most expensive NoSQL mistake is adopting a database for its reputation rather than its data model. If your core workload is relational (many entities, many relationships, frequent ad-hoc queries, strict cross-entity integrity) and your scale fits one server, a NoSQL primary store will usually make your life harder. Write down your access patterns first, then choose.

Trade-offs You Are Signing Up For

Every NoSQL family asks you to accept some combination of the following trade-offs. Being explicit about them helps you ask the right questions when you evaluate a database.

- **Queries are designed up front.** Most NoSQL systems are fast for the queries you designed the data model around and awkward or slow for others. Adding a new access pattern may mean a new index, a new table, or a data migration.
- **Integrity moves into the application.** With fewer (or no) foreign keys and cross-record constraints, your code and processes must prevent orphans and inconsistencies.
- **Duplication is normal.** Denormalizing data to avoid joins means one fact may live in several places, and you must decide how and when copies are updated.
- **Consistency is a spectrum.** Distributed databases often let you pick consistency per operation, and the default may be weaker than you expect.
- **Operations are different.** Compaction, rebalancing, hot partitions, and memory limits are new failure modes to learn.

In exchange, you can gain horizontal scalability with predictable latency, high availability during failures, a data model that mirrors your code, and engines specialized for text, graphs, time, and vectors. Whether that exchange is worth it depends entirely on the workload.

Licensing and the Open-Source Landscape

Licensing is not a technical topic, but it affects which distribution of a database you can use and how, so every practitioner should know the broad history. In the last several years a number of databases that were once under permissive open-source licenses changed their terms, largely in response to cloud providers offering the software as a managed service. We describe the history neutrally here; the individual chapters add detail.

- **MongoDB** moved its Community Server from the AGPL to the Server Side Public License (SSPL) in 2018. Recent Community releases are distributed under the SSPL. MongoDB Atlas is the company's managed service.
- **Elasticsearch** changed from the Apache 2.0 license to a dual SSPL and Elastic License model in 2021. An AWS-led fork named **OpenSearch** was created from the last Apache-licensed version and is now under the Linux Foundation. Elastic later added the AGPL as an additional licensing option.
- **Redis** moved from the BSD license to a dual RSALv2 and SSPLv1 model in 2024. The Linux Foundation fork **Valkey** followed, based on the last BSD-licensed release. Later Redis releases (Redis 8) added the AGPLv3 as an option.
- **Apache Cassandra** is an Apache Software Foundation project under the Apache 2.0 license. Compatible alternatives such as ScyllaDB have their own licensing histories; check the current license of any distribution you evaluate.

What these changes mean for you depends on how you use the software: running it internally, embedding it in a product, or offering it as a service to others can each be treated differently by different licenses. We do not give legal advice in this book. Review the current license text for the exact version you deploy, and involve your organization's legal counsel when the answer matters.

> **TIP**
> Record the database distribution, version, and license in your architecture decision records. Licenses change between versions, and a note like "Valkey 8.x, BSD-3-Clause, reviewed 2026-03" saves a painful archaeology exercise during the next audit or upgrade.

What Goes Wrong in Practice

It is worth closing the conceptual part of this chapter with the failure stories that recur across NoSQL adoptions, described generically. You will meet each of them again in later chapters with concrete fixes.

- **The relational schema, transplanted.** A team moves tables one-to-one into collections, then reimplements joins in application code with many round trips. Performance is worse than before, and the conclusion is "NoSQL is slow." The real problem is that the data model was never redesigned for the new engine.
- **The hot key.** A celebrity coach's plan, a single global counter, or a date-based partition key concentrates traffic on one node while the rest of the cluster idles.
- **The unbounded record.** A document that holds "all followers" or "all readings" grows until it hits size limits or becomes expensive to rewrite on every update.
- **Surprised by consistency.** A member logs a workout, refreshes the app, and it is missing because the read went to a replica that had not caught up yet. Nothing is broken; the application simply assumed a guarantee the configuration did not provide.