

Databases

Reading time: ~10 min · Prereqs: Lessons 01–06 · *Snapshot 2026 — concepts are stable; the vector-DB wing especially is fluid.*

Why this matters to you

Databases are the most boring-sounding, most consequential decision in software. Get the database right and the rest of the app stays tractable for a decade — new features slot in, queries stay fast, the on-call rotation stays quiet. Get it wrong and every feature request comes with a multi-week migration tax, the cloud bill creeps up, and "we can't change that because of the database" becomes a recurring meeting note. This lesson is the survey: what kinds of databases exist, what each is good at, and why a meaningful chunk of senior engineers *as of 2026* will roll their eyes and say "just use Postgres" before letting you pick anything else. After this lesson you should be able to read a database name in an architecture diagram and instantly know what kind of workload it's solving for.

1. Why a database, instead of files?

A folder of `.csv` files *is* a database — for exactly one user, on exactly one machine, with no failures. It collapses immediately past that under five forces. **Many users at once**: two concurrent writes to the same CSV corrupt the file. **Durability**: one crash mid-write and your file is half-old, half-new and unreadable. **Query speed**: with no index, every "find all orders over \$100" scans every row, every time — fine at a thousand rows, agony at a billion. **Structure**: nothing stops you from putting `"not-an-email"` in the email column. **Reasoning across data**: joining two CSVs by a foreign key is a multi-hour ad-hoc script. Databases solve all five of these as their day job. The price you pay is a separate piece of infrastructure to install, manage, back up, and (eventually) scale — which is why this whole lesson exists.

2. SQL — the 50-year incumbent

SQL (Structured Query Language) was invented at IBM in the 1970s and is still the most-deployed shape of database in the world. SQL databases — **PostgreSQL**, **MySQL** / **MariaDB**, **Microsoft SQL Server**, **Oracle**, **SQLite** — store data in **tables** with fixed columns. Each row in a table has the same shape; if you want to add a new field, you alter the table. You query with a declarative language: the famous `SELECT name FROM users WHERE country='UA'` — you describe *what* you want, not *how* to get it, and the engine figures out the fastest plan.

SQL databases have decades of work behind them on three things that matter enormously and are extremely hard to build. First, **ACID transactions** — writes are all-or-nothing, even across crashes, even when multiple users are writing at once. Second, **referential integrity** — foreign keys make it impossible to have an order pointing to a customer that doesn't exist. Third, **query planning** — the engine inspects your query and your indexes and picks the fastest way to answer, often beating what a human would write by hand.

The simple rule: for **anything involving money, accounts, identity, or other "I'd rather lose my hair than lose this record" data**, you want a SQL database.

PostgreSQL is the default choice for new projects *as of 2026*; it's free, broadly loved, and increasingly does everything (see "just use Postgres" below). Think of a SQL database as a *well-organized filing cabinet*: every drawer (table) has a fixed set of labeled folders (columns), every document (row) goes in exactly one place, and you have a librarian (the query planner) who can pull any cross-reference in milliseconds.

3. The NoSQL family

In the 2000s, web companies (Google, Amazon, Facebook) hit limits with single-machine SQL: too much write traffic for one box, too many JSON-shaped objects that didn't want to be tables, too rigid a schema for products that pivoted every quarter.

NoSQL is the catch-all for the alternatives that emerged. The name is a misnomer (it's not "anti-SQL", more "not-only-SQL") but it stuck. Four useful shapes show up in 2026:

- **Document stores** — store JSON-like documents instead of rows; schema can vary per document. Brand names *as of 2026*: **MongoDB** is the icon; **DynamoDB** (AWS), **Firestore** (Google), **Cosmos DB** (Azure), **Couchbase**. Good fit for user profiles, product catalogs, and content where every item is shaped slightly differently. Analogy: a shoebox of receipts instead of a spreadsheet.

- **Key-value stores** — a giant hash map. `SET user:42:session "abc..."` / `GET user:42:session`. Microseconds-fast because they usually live in memory. **Redis** is king as of 2026; **Valkey** is its open-source fork; **Memcached** is the old guard. Used as caches, session stores, leaderboards, rate limiters, and lightweight pub/sub buses.
- **Wide-column stores** — **Cassandra**, **ScyllaDB**, **HBase**, **BigTable**. Built for huge, write-heavy, time-series-ish data spread across many machines. Less common in everyday application work; common in telemetry, IoT, and ad-tech pipelines.
- **Graph databases** — **Neo4j**, **Memgraph**, **Amazon Neptune**. For relationship-heavy data: social networks, fraud detection, knowledge graphs. If your central question is "how is X connected to Y, two hops away?", a graph DB answers it natively where a SQL JOIN would crawl.

4. Search engines

When you need full-text search — typos, synonyms, relevance ranking, faceted filtering — SQL's `LIKE '%foo%'` is the wrong tool (it's slow and dumb about meaning). Search engines build **inverted indexes** over text (for every word, the list of documents it appears in) and rank results by relevance. Brand names as of 2026: **Elasticsearch** (and its AWS-led fork **OpenSearch**) is the heavyweight; **Algolia** is the SaaS-friendly choice with the fastest "drop it in and it works" story; **MeiliSearch** and **Typesense** are lightweight, developer-friendly upstarts. If your app has a search bar that needs to feel like Google's — tolerant of typos, ranked by relevance, fast on millions of records — you want one of these alongside your primary database, not instead of it.

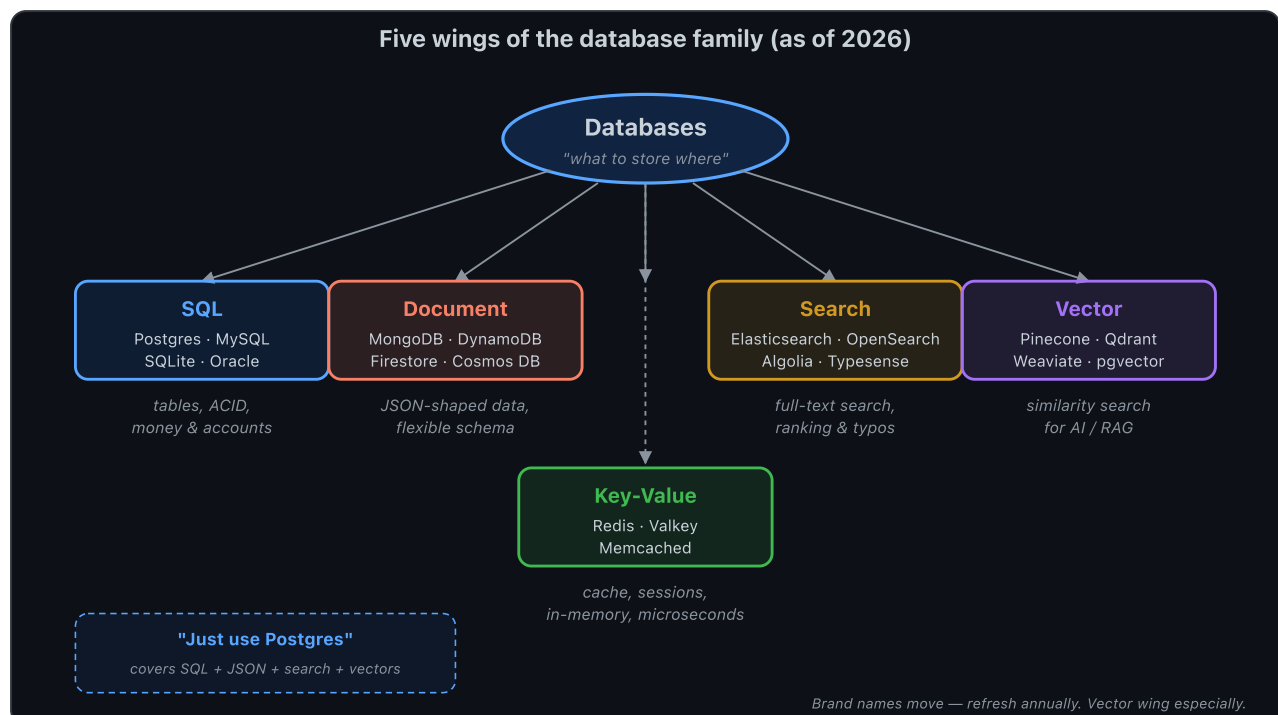
5. Vector databases — the new wing

New as of 2026, born of the LLM era. LLMs and embedding models turn text (or images, or audio) into high-dimensional vectors of numbers — ~1024 dimensions is typical. Searching by **semantic similarity** ("find me documents that mean roughly this") becomes a problem of finding the nearest neighbors of a query vector in that high-dimensional space. Specialized stores have optimized indexes for exactly that. Names as of 2026: **Pinecone** (the SaaS pioneer), **Weaviate**, **Qdrant**, **Chroma** (lightweight, developer-friendly), and **pgvector** (a Postgres extension — the "just use Postgres" path). Every RAG-based AI app (Lesson 13 returns to RAG in depth) uses

one. This whole section is on [refresh-checklist.md](#) as high-churn — vendors come and go quarterly, and the 2027 list will look different.

6. The database family, one diagram

Here is the same idea drawn as a taxonomy: a central *Databases* node branching into five wings, each with a couple of representative brand names *as of 2026* and a one-line use-case caption. When you see one of these names in an architecture diagram, you can place it on this tree.



7. The cost of getting it wrong

Database choice is a long-tail commitment. A handful of failure modes show up over and over in postmortems. **Silent data loss:** some NoSQL stores have, historically, lost writes during a primary failover — sometimes silently, which is the worst kind. **Wrong shape forever:** denormalized so hard, or schema-locked so rigidly, that adding a new query pattern takes weeks of migration. **Egress cost:** cross-region replication of a chatty database runs up a five-figure monthly cloud bill that nobody noticed in the design doc. **Schema rigidity:** changing a column type in a 1 TB SQL table takes hours of downtime if you didn't plan for it. **Search relevance:** "our users say they can't find anything" is sometimes a six-month engineering project to fix. This is why the next section exists.

8. "Just use Postgres"

A 2026 meme that's also serious advice. Modern PostgreSQL handles: relational data (its day job), JSON documents (with `JSONB`), full-text search (built-in `tsvector`), vectors (via `pgvector`), time-series (via **TimescaleDB**), and key-value (cheap `UNLOGGED` tables or `hstore`). For most teams, starting with one Postgres instance and adding specialized stores only when proven necessary keeps the operational surface small, the cost predictable, and the team's mental load manageable. At very large scale, or for genuinely extreme query patterns, specialized stores eventually earn their keep — but most products never reach that point, and the ones that do can migrate later with much better information than they had on day one.

What people argue about

- **SQL vs NoSQL.** The 2010s religious war has largely died. The honest take as of 2026: most apps need a SQL database first; some apps benefit from adding a NoSQL store later for specific shapes of data (a Redis cache, a Mongo store for user-generated documents). "Pick one and use it for everything" is rarely the right answer.
- **Vector DBs vs pgvector.** Specialized vendors say they scale better and offer faster nearest-neighbor indexes; the "just use Postgres" camp says for most teams that's premature optimization. Real workloads in 2026 increasingly favor pgvector at small-to-medium scale and specialized DBs at very large scale — with the crossover point moving each year.
- **NewSQL.** A real-but-niche third category: **CockroachDB**, **Spanner**, **YugabyteDB** — SQL semantics with horizontal scaling across many machines. Worth knowing exists; rarely the right first choice.

Reading on

- **Designing Data-Intensive Applications** by Martin Kleppmann — the canonical modern book on the trade-offs in this lesson. If you read one technical book this year, this one pays back the time many times over.

- **Use The Index, Luke!** — use-the-index-luke.com. Free; teaches the one thing that turns out to matter most about SQL performance (indexes, and when they're not used).
- **PostgreSQL documentation** — postgresql.org/docs. Surprisingly readable and well-organized; a good way to absorb what "modern Postgres" can actually do.

LAST-REVIEWED: 2026-05-25