

Databases

Aayush Bajaj

July 10, 2026

Contents

1	the relational model	1
2	relational algebra	2
3	SQL, actually run	2
4	normalisation	4
4.1	a worked decomposition	4
5	indexing	5
5.1	why B+-trees and not binary trees	5
5.2	the other index shapes	6
6	query processing	6
7	transactions and ACID	6
7.1	isolation levels and their anomalies	7
7.2	2PL vs MVCC	7
8	distribution: CAP and the NoSQL taxonomy	7
9	see also	8

a database is a data structure that survives a power cut, shared by programs that don't trust each other, queried in a language older than most of its users. the relational model has been declared dead roughly once a decade since 1970 and has outlived every announced successor. this page covers the model, the algebra underneath SQL, normalisation, the storage structures that make queries fast, and the machinery that keeps concurrent transactions honest.

this page's original stub was a shopping list — postgres, mysql, cassandra, mongodb, and something called QRKDB i can no longer place. the taxonomy section sorts them out

1 the relational model

codd's 1970 move: store everything as *relations* — sets of tuples over named, typed attributes — and let the system, not the programmer, decide how to navigate them.

- a **relation** is a set of tuples; a **tuple** is one row; an **attribute** is a named column with a domain. “table” is the implementation-flavoured word for the same thing.
- a **superkey** is any attribute set that uniquely identifies tuples; a **candidate key** is a minimal superkey; the **primary key** is the candidate key you commit to; a **foreign key** is an attribute set referencing another relation's key.

- **integrity constraints** are the guarantees the DBMS enforces so your application doesn't have to:
 - **domain**: values come from the attribute's type (mark `INTEGER`, `CHECK (year BETWEEN 1 AND 6)`),
 - **entity**: no part of a primary key may be null — you cannot half-identify a row,
 - **referential**: every foreign key value must exist in the referenced relation — no enrolments for students who don't exist (Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S., 2019).

the deep idea is *data independence*: queries are written against the logical schema, and the physical layout (indexes, partitions, storage order) can change underneath without breaking a single query.

2 relational algebra

SQL is syntax; relational algebra is the semantics the optimiser actually manipulates. six primitive operators generate the whole language (Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S., 2019):

operator	notation	meaning
selection	$\sigma_{\theta}(R)$	keep tuples satisfying predicate θ
projection	$\pi_{A_1, \dots, A_k}(R)$	keep only the listed attributes (deduplicating)
union	$R \cup S$	tuples in either (schemas must match)
set difference	$R \setminus S$	tuples in R but not S
cartesian product	$R \times S$	every pairing of tuples
rename	$\rho_{S(B_1, \dots)}(R)$	relabel a relation/attributes (enables self-joins)

everything else is sugar: the natural join is $R \bowtie S = \pi_{\dots}(\sigma_{R.A=S.A}(R \times S))$, and intersection is $R \cap S = R \setminus (R \setminus S)$. the optimiser exploits algebraic identities — pushing a selection below a join is legal because $\sigma_{\theta}(R \bowtie S) = \sigma_{\theta}(R) \bowtie S$ whenever θ touches only R 's attributes, and it is usually the single biggest win in a query plan.

3 SQL, actually run

schema, inserts, a join, an aggregate, and a window function — the five things that cover most daily SQL. python's built-in `sqlite3` (engine version 3.50) executes all of it:

```

1 import sqlite3
2
3 con = sqlite3.connect(":memory:")
4 con.execute("PRAGMA foreign_keys = ON")  # off by default; REFERENCES is
5                                         ↳ decorative without it
6 cur = con.cursor()
7 cur.executescript("""
8 CREATE TABLE students (
9     sid    INTEGER PRIMARY KEY,
10     name   TEXT NOT NULL,
11     year   INTEGER CHECK (year BETWEEN 1 AND 6)
12 );
13 CREATE TABLE courses (
14     cid    TEXT PRIMARY KEY,
15     title  TEXT NOT NULL,

```

```

15     uoc    INTEGER DEFAULT 6
16 );
17 CREATE TABLE enrolments (
18     sid    INTEGER REFERENCES students,
19     cid    TEXT    REFERENCES courses,
20     mark   INTEGER,
21     PRIMARY KEY (sid, cid)          -- composite key
22 );
23
24 INSERT INTO students VALUES
25 ↪ (1,'ada',3),(2,'grace',2),(3,'edsger',4),(4,'barbara',3);
26 INSERT INTO courses  VALUES
27 ↪ ('COMP3311','databases',6),('COMP3331','networks',6),
28 ↪ ('COMP3231','operating systems',6);
29 INSERT INTO enrolments VALUES
30 ↪ (1,'COMP3311',91),(1,'COMP3331',84),(2,'COMP3311',78),
31 ↪ (2,'COMP3231',88),(3,'COMP3331',95),(3,'COMP3231',72),
32 ↪ (4,'COMP3311',85);
33
34 print("-- join: who is enrolled in what")
35 for row in cur.execute("""
36     SELECT s.name, c.title, e.mark
37     FROM enrolments e
38     JOIN students s ON s.sid = e.sid
39     JOIN courses  c ON c.cid = e.cid
40     ORDER BY s.name, c.cid"""):
41     print(row)
42
43 print("-- aggregate: course averages, hardest first")
44 for row in cur.execute("""
45     SELECT cid, COUNT(*) AS n, ROUND(AVG(mark),1) AS avg_mark
46     FROM enrolments GROUP BY cid
47     HAVING COUNT(*) >= 2
48     ORDER BY avg_mark"""):
49     print(row)
50
51 print("-- window: rank within each course without collapsing rows")
52 for row in cur.execute("""
53     SELECT cid, sid, mark,
54            RANK() OVER (PARTITION BY cid ORDER BY mark DESC) AS rk,
55            ROUND(AVG(mark) OVER (PARTITION BY cid), 1)      AS course_avg
56     FROM enrolments ORDER BY cid, rk"""):
57     print(row)

```

```

-- join: who is enrolled in what
('ada', 'databases', 91)
('ada', 'networks', 84)
('barbara', 'databases', 85)
('edsger', 'operating systems', 72)
('edsger', 'networks', 95)
('grace', 'operating systems', 88)
('grace', 'databases', 78)
-- aggregate: course averages, hardest first
('COMP3231', 2, 80.0)
('COMP3311', 3, 84.7)
('COMP3331', 2, 89.5)
-- window: rank within each course without collapsing rows
('COMP3231', 2, 88, 1, 80.0)
('COMP3231', 3, 72, 2, 80.0)

```

```

('COMP3311', 1, 91, 1, 84.7)
('COMP3311', 4, 85, 2, 84.7)
('COMP3311', 2, 78, 3, 84.7)
('COMP3331', 3, 95, 1, 89.5)
('COMP3331', 1, 84, 2, 89.5)

```

the window function is the one worth internalising: **GROUP BY** collapses rows to one per group, while **OVER (PARTITION BY ...)** computes the same aggregate but keeps every row — rank and group average side by side with the raw mark.

sqlite has had window functions since 3.25 (2018); if your mental model of sqlite is “toy”, update it

4 normalisation

redundancy is the disease; update, insertion, and deletion anomalies are the symptoms. functional dependencies are the diagnostic tool.

a **functional dependency** $X \rightarrow Y$ holds when tuples agreeing on attribute set X must agree on Y . armstrong’s axioms (reflexivity, augmentation, transitivity) generate all implied dependencies.

4.1 a worked decomposition

start with one wide relation recording marks, one row per enrolment:

enrol(sid, cid, sname, ctitle, dept, head, mark)

with dependencies

$\text{sid} \rightarrow \text{sname}$
 $\text{cid} \rightarrow \text{ctitle}, \text{dept}$
 $\text{dept} \rightarrow \text{head}$
 $\text{sid}, \text{cid} \rightarrow \text{mark}$

- **1NF** demands atomic values and no repeating groups — storing a student’s courses as a comma-separated string violates it; one row per enrolment fixes it. the relation above is already 1NF.
- **2NF** bans *partial* dependencies: non-key attributes depending on part of the composite key. **sname** depends on **sid** alone, **ctitle** and **dept** on **cid** alone. redundancy is visible: ada’s name is repeated once per enrolment, and you cannot record a course before anyone enrolls (insertion anomaly). decompose:

student(sid, sname), course(cid, ctitle, dept, head), enrol(sid, cid, mark)

- **3NF** bans *transitive* dependencies: **cid** determines **dept**, **dept** determines **head**, so **head** is stored once per course instead of once per department — change a head of department and you update many rows or corrupt the data. split again:

course(cid, ctitle, dept), department(dept, head)

- **BCNF** tightens 3NF: for *every* nontrivial $X \rightarrow Y$, X must be a superkey. the classic residual case: tute(sid, cid, tutor) where each tutor teaches exactly one

course ($\text{tutor} \rightarrow \text{cid}$) and each student has one tutor per course ($\text{sid}, \text{cid} \rightarrow \text{tutor}$). **tutor** is not a superkey, so BCNF says split into $\text{tutorof}(\text{tutor}, \text{cid})$ and $\text{assigned}(\text{sid}, \text{tutor})$ — but now the dependency $\text{sid}, \text{cid} \rightarrow \text{tutor}$ spans two tables and can no longer be checked without a join. BCNF is not always dependency-preserving; 3NF always is. every decomposition above is lossless (the shared attributes are keys of one side), which is the non-negotiable property (Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S., 2019).

normalise until it hurts, denormalise until it works — but know which dependencies you are knowingly breaking.

5 indexing

5.1 why B+-trees and not binary trees

disks and SSDs are read in pages (4–16 KiB), and each random page fetch is the expensive unit. a balanced binary tree on 10^8 keys is $\log_2 10^8 \approx 27$ levels deep — 27 page fetches if each node lands on a different page, which they do, because a binary node is a few dozen bytes rattling around alone. the fix: make each node exactly one page wide and branch as hard as the page allows (Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S., 2019).

a **B+-tree** keeps all records (or record pointers) in the leaves, threaded into a sorted linked list; internal nodes hold only separator keys and child pointers, and every leaf sits at the same depth.

fanout arithmetic. take a 4 KiB page, 8-byte keys, 8-byte pointers. an internal node holding n child pointers needs $n - 1$ keys: $8n + 8(n - 1) \leq 4096$ gives $n \approx 256$. for $N = 10^8$ keys with 256-entry leaves:

- leaves: $\lceil 10^8 / 256 \rceil = 390,625$
- level above: $\lceil 390,625 / 256 \rceil = 1,526$
- above that: $\lceil 1,526 / 256 \rceil = 6$, then a root of 6 pointers.

four levels for a hundred million keys, and since the root and second level (about 7 pages) live permanently in the buffer pool, a cold lookup costs roughly *two* page reads — against 27 for the binary tree. the same structure answers range queries by walking the leaf chain, which a hash index cannot do.

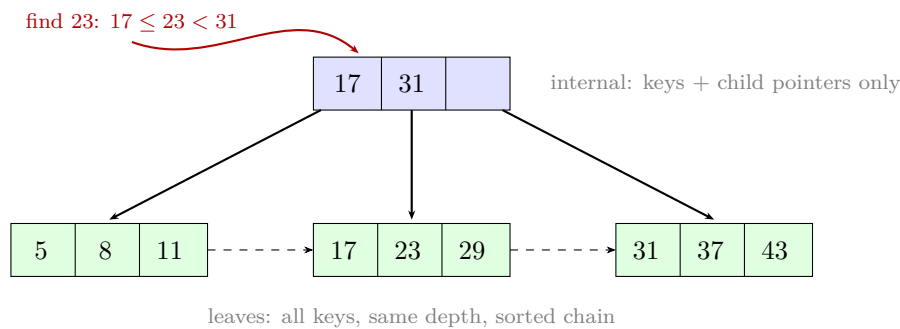


Figure 1: a B+-tree of order 4: separator keys route the search, all data sits in the linked leaves. dashed arrows are the range-scan chain.

5.2 the other index shapes

- **hash indexes** map key to bucket in expected $O(1)$ — faster than a B+-tree for pure equality lookups, useless for ranges, **ORDER BY**, or prefix matches, because the hash deliberately destroys ordering.
- **covering indexes**: if an index on **(cid, mark)** contains every column a query touches, the engine answers from the index alone and never visits the table — an “index-only scan”. the fastest table read is the one that doesn’t happen.
- indexes are not free: every **INSERT**, **UPDATE**, and **DELETE** must maintain all of them, so write-heavy tables want few, carefully chosen indexes.

rule of thumb: index what you filter and join on, cover what you read hottest, and measure before adding a fourth

6 query processing

a join $R \bowtie S$ (with $|R| = n$, $|S| = m$ tuples; B_R , B_S pages) can be executed three classic ways (Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S., 2019):

algorithm	in-memory cost	i/o cost (pages)	when it wins
nested loop	$O(nm)$	$B_R + n \cdot B_S$	tiny inner relation, or an index on it
block nested loop	$O(nm)$	$B_R + B_R \cdot B_S$ (worst)	small memory, no order, no index
hash join	$O(n + m)$	$\approx 3(B_R + B_S)$ (grace)	equality joins, enough memory to partition
sort-merge join	$O(n \log n + m \log m)$	sort cost + $B_R + B_S$	inputs already sorted, or output must be

- **nested loop** is the naive double **for**; with an index on the inner join key it becomes *index nested loop* and is excellent for small outer relations.
- **hash join** builds a hash table on the smaller relation, probes with the larger; when neither fits in memory, both are partitioned to disk by hash first (grace hash join).
- **sort-merge** sorts both on the join key and zips them; it is the reason **ORDER BY** on the join key is sometimes free afterwards.

the optimiser picks between them using table statistics — cardinalities, histograms, index availability — which is why a stale **ANALYZE** can turn a millisecond plan into a minute-long one.

7 transactions and ACID

a **transaction** is a group of statements that must behave as one indivisible action:

- **atomicity** — all or nothing; crash mid-transfer and the money is in exactly one account. implemented with a write-ahead log and rollback.
- **consistency** — every commit moves the database between constraint-satisfying states.
- **isolation** — concurrent transactions cannot observe each other’s intermediate garbage.
- **durability** — once committed, survives power loss; the WAL is fsynced before “committed” is reported.

7.1 isolation levels and their anomalies

full serialisability costs concurrency, so SQL defines discounted tiers. what each level permits:

level	dirty read	non-repeatable read	phantom
read uncommitted	possible	possible	possible
read committed	prevented	possible	possible
repeatable read	prevented	prevented	possible
serialisable	prevented	prevented	prevented

- **dirty read**: seeing another transaction’s uncommitted write.
- **non-repeatable read**: re-reading a row and finding it changed under you.
- **phantom**: re-running a predicate query and finding *new rows* that now match.

the table is the standard’s contract, not a description of any real engine: postgres’s “repeatable read” is snapshot isolation, which also prevents phantoms but admits *write skew* — two transactions each reading the condition the other is about to falsify (two doctors both going off-call because “at least one other doctor is on call”). snapshot isolation passes the table above and still is not serialisable — a genuinely non-obvious fact, explained beautifully in kleppmann’s *designing data-intensive applications* (<https://dataintensive.net/>).

7.2 2PL vs MVCC

- **two-phase locking**: acquire shared/exclusive locks as you go (growing phase), release only at the end (shrinking phase — in practice, at commit: strict 2PL). guarantees serialisability, but readers block writers and vice versa, and deadlocks must be detected and a victim aborted.
- **multi-version concurrency control**: writers create new versions instead of overwriting; each transaction reads the snapshot that was current when it began. readers never block writers, which is why postgres, mysql/innodb, and oracle all live here. the honest price: garbage from dead versions (postgres’s vacuum), and plain snapshot isolation admits the anomalies above — postgres’s **SERIALIZABLE** level layers serialisation-conflict detection (SSI) on top and aborts one of the offenders, trading blocking for retries.

8 distribution: CAP and the NoSQL taxonomy

once data spans machines, the network becomes part of the database. the CAP theorem: under a network **partition**, a distributed store must sacrifice either **consistency** (every read sees the latest write) or **availability** (every request gets a non-error answer). the folk version — “pick two of three” — is misleading, because partition tolerance is not optional: partitions happen. the real content is the C-vs-A choice *during* the partition, and kleppmann argues even that framing hides more than it reveals ([please stop calling databases CP or AP](#)).

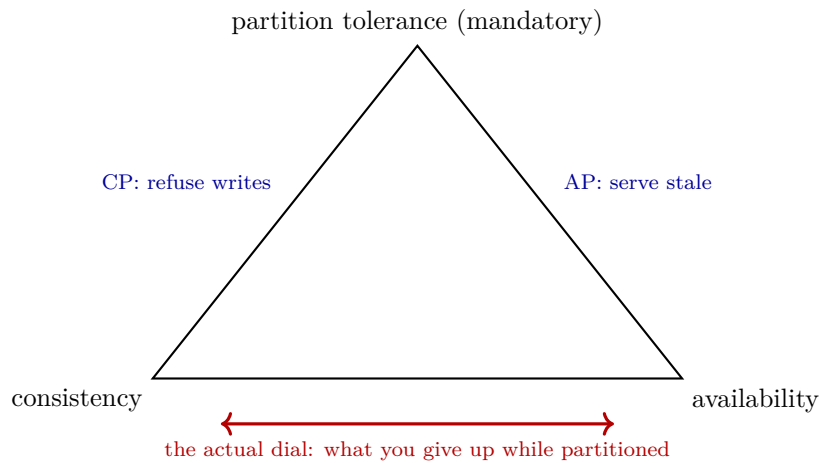


Figure 2: CAP: a partition forces the choice on the bottom edge. the top corner is the part you don't get to negotiate away.

the “NoSQL” label bundles four genuinely different data models. sorting the stub’s shopping list into it:

family	data model	exemplars	reach for it when
key-value	opaque blob per key	redis, dynamodb	caches, sessions, counters — access is only ever by key
document	nested JSON-ish trees	mongodb, couchdb	aggregates read/written whole, schema still settling
column-family	wide rows, sorted within partitions	cassandra, hbase	write-heavy append streams, time-series at scale
graph	nodes + edges, first-class	neo4j	queries that are mostly multi-hop relationship traversal

postgres and mysql are the relational workhorses the rest get compared against — and postgres’s **JSONB** quietly covers a large slice of the document use-case with transactions included. as for **QRKDB**, the last name on the stub’s list: i can find no trace of it anywhere, including in my own notes, so it stays here as a monument to writing down context, not just names.

if future-me remembers what QRKDB was: edit this page and apologise to past-me

the deeper divide than SQL-vs-NoSQL is the consistency model: single-node transactions are a solved problem; cross-partition transactions are where the distributed-systems dragons live.

9 see also

- **internet networks** — the layer your database listens on
- **linux** — fsync, pages, and the kernel under the WAL
- **data structures & algorithms** — B-trees and hashing in their natural habitat
- **memory** — the cache hierarchy that makes page-sized thinking mandatory

references

Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S. (2019). *Database System Concepts*, McGraw-Hill Education.