Modern Database Architectures: From SQL to NoSQL and Distributed SQL

Andrei Manakov

About me

- 10+ years of experience in distributed systems development
- Currently building ML infrastructure at indian tik-tok like app (300M MAU)
- I write posts about distributed systems in my personal blog

Prehistoric databases (1950–1960s)

📚 Types of Databases:

1. **Hierarchical DBs** (e.g., IBM IMS)
2. **Network DBs** (e.g., CODASYL, IDMS)
3. **File Systems**

❌ Key issues:

- 📍 **Complex data access** that required navigating step-by-step through the structure.
- 🔧 **Applications tightly depended on data structure.**
- 🔄 **Even small changes required rewriting code.**
- 🧩 **No universal query language:** developers had to write low-level code.
- ♻️ **Data redundancy and inconsistency:** the same data was stored in multiple places.
- 🧱 **No built-in integrity constraints** (e.g., uniqueness or relationships).
- 🚫 **Lack of logical and physical data independence**.

Edgar F. Codd’s breakthrough (1970)

🧠 What did Codd do?

- In 1970, Edgar F. Codd introduced the relational data model, revolutionizing the way data was stored and processed. .
- For the first time, he applied mathematical principles to databases:
    a. Set theory
    b. First-order predicate logic
- He also formulated the famous 12 rules, solving all previous database issues:
    a. 📍 **Complex data access** — ✅
    b. 🔧 **Tight application dependency on data structure** — ✅
    c. 🔄 **Modification challenges** — ✅
    d. 🧩 **No universal query language** — ✅
    e. ♻️ **Data redundancy and inconsistency** — ✅
    f. 🧱 **No built-in integrity constraints** — ✅
    g. 🚫 **Lack of logical and physical data independence** — ✅

The era of relational DBs (1970–1980s)

✅ **Proven and tested storage structure**

✅ **Clear formal foundation (set theory + logic)**

✅ **Standardized query language (SQL)**

✅ **Transaction support (ACID):**

- **A** — Atomicity
- **C** — Consistency
- **I** — Isolation
- **D** — Durability

✅ **Data reliability and consistency**

✅ **Data normalization**

Examples: Oracle and FoxPro

Challenges of the 1990s

🌍 **The Internet Boom of the 1990s**

- Mass Internet access **transformed** how businesses operated.
- Companies rapidly developed **web applications**, e-commerce platforms, and online services.
- The number of users and data requests skyrocketed — growing **by tens or even hundreds of times.**

💾 **Impact on databases:**

- 📈 **Explosive data growth:**
  - User profiles, logs, actions, transactions.
- 🔁 **Demand for real-time processing:**
  - Instant responses became crucial.
- 🧩 **More complex data structures:**
  - Databases had to store everything from products and orders to images and user behavior.

⚠️ **Consequences:**

➡️ Relational databases struggled with **scaling issues.**

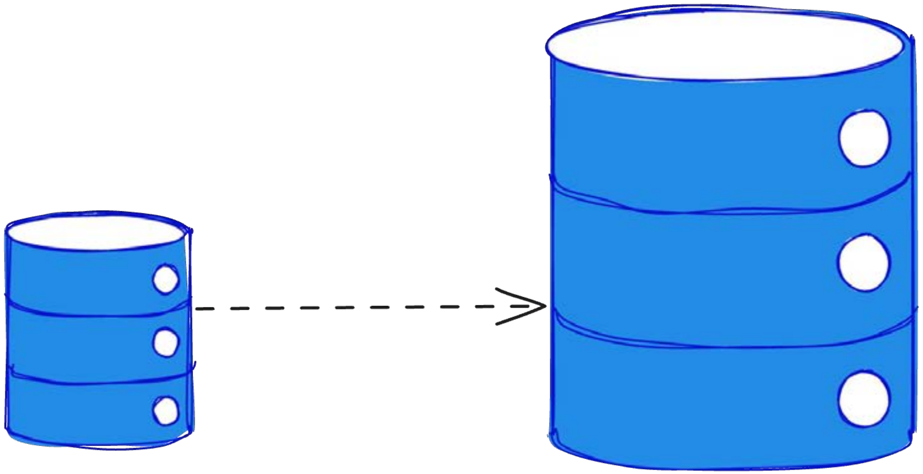
➡️ **New database architectures** were needed to meet rising demands.

Solution: vertical scaling

✅ Simple

✅ No issues with consistency, transactions, or joins

❌ Hard scalability limits

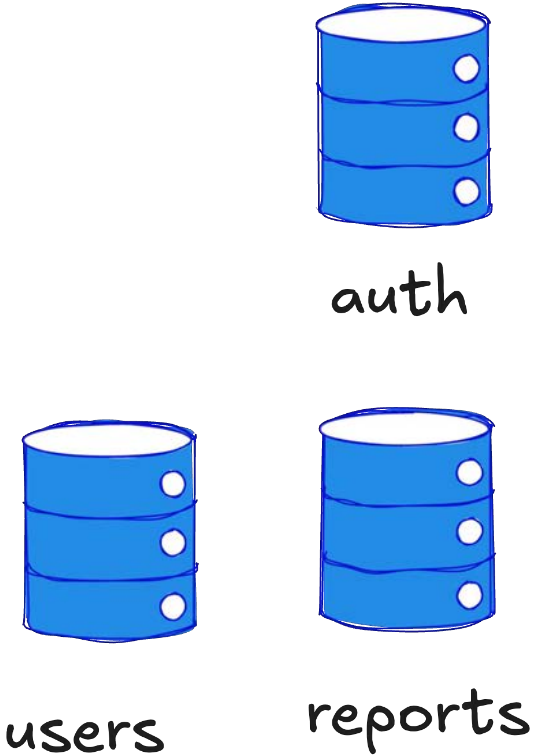


Solution: logical database partitioning

✅ Still simple

✅ Increased overall storage capacity

❌ Still has a limit



Solution: sacrificing Codd’s model

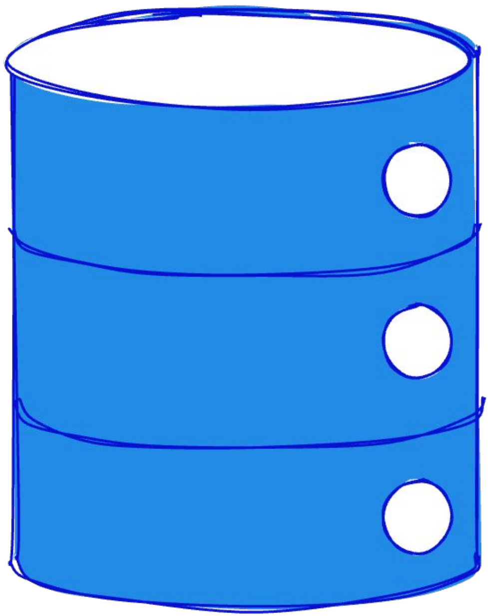
📌 Denormalization

📌 No transactions

✅ Increased overall system capacity

❌ Still a ceiling

❌ Requires careful planning

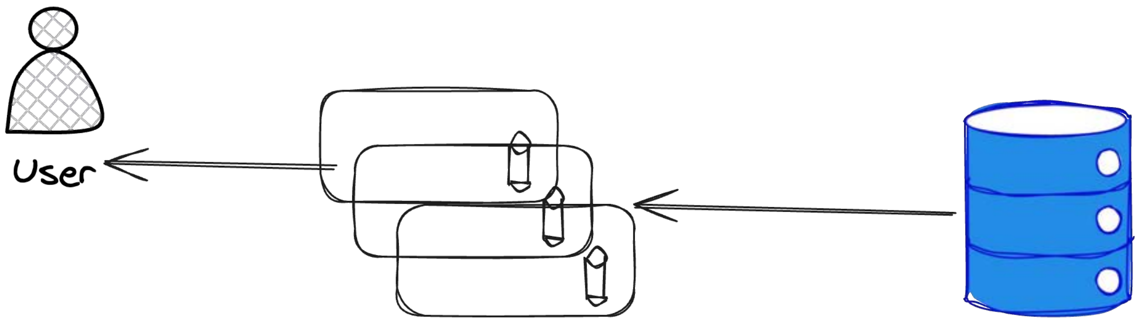


Solution: caching

✅ Lower database load

❌ Sometimes users end up reading outdated data from the cache

❌ Still a ceiling



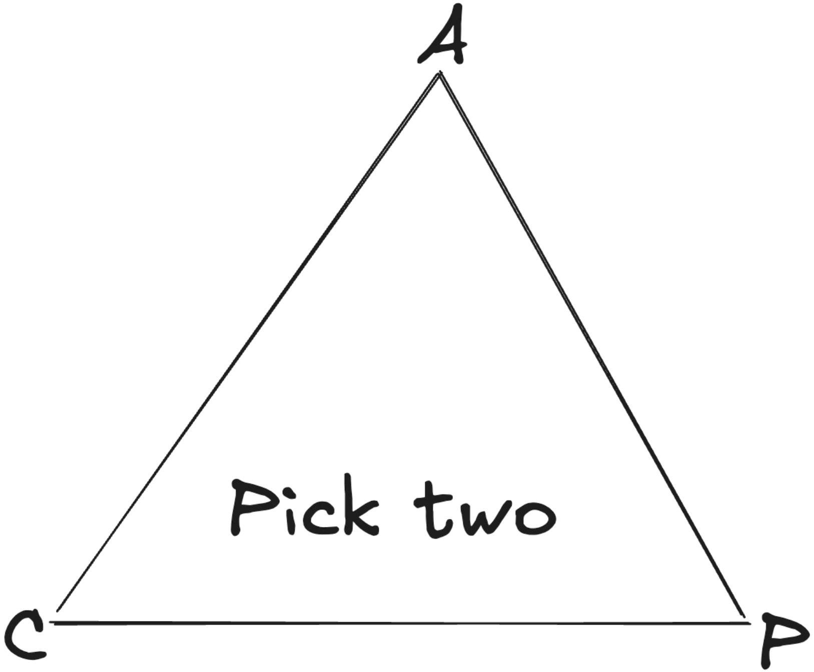
CAP theorem

P (Partition Tolerance) — the system continues to function even **if some nodes become unavailable.**

C (Consistency) — all nodes see the same data at the same time.

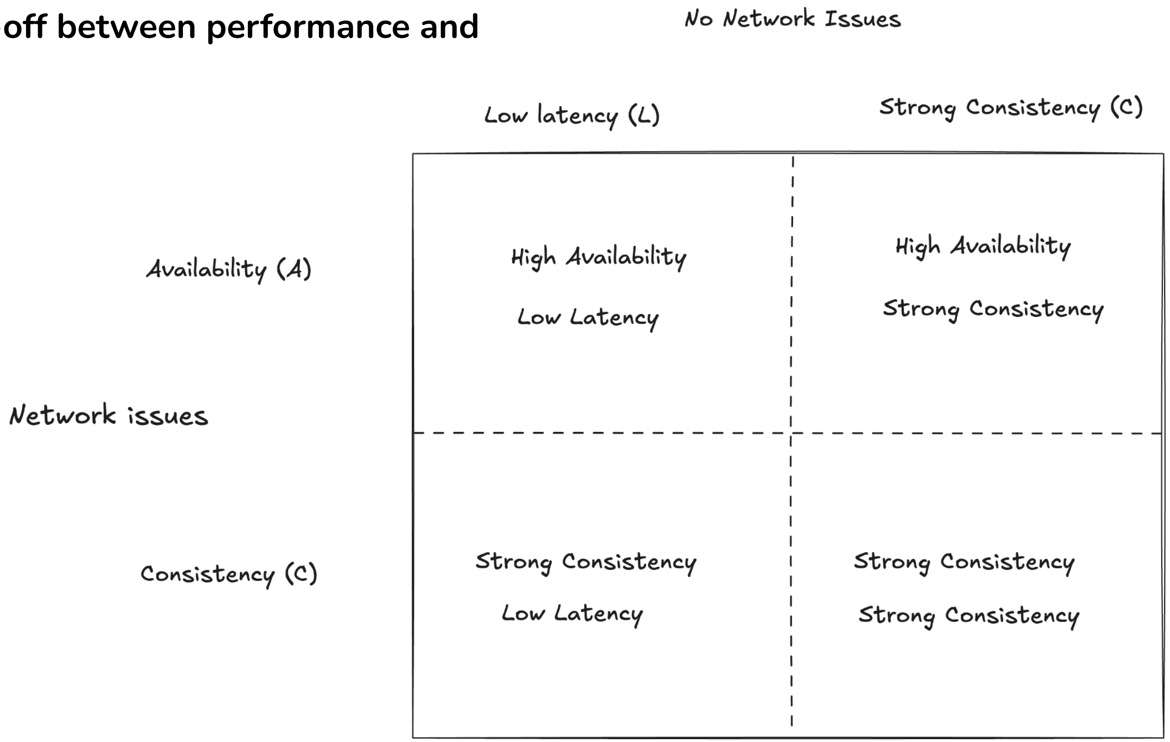
A (Availability) — every request receives a response.

Classical database – CA



PACELC

📌 **Also considers the trade-off between performance and consistency**



When to choose relational databases?

1. **Always, if there is not a high load (around 1K RPS)**

    ✅ Scaling is not required

    ✅ All the benefits of Codd’s model remain

    ✅ No trade-offs

    📌 **Must account for potential failures**

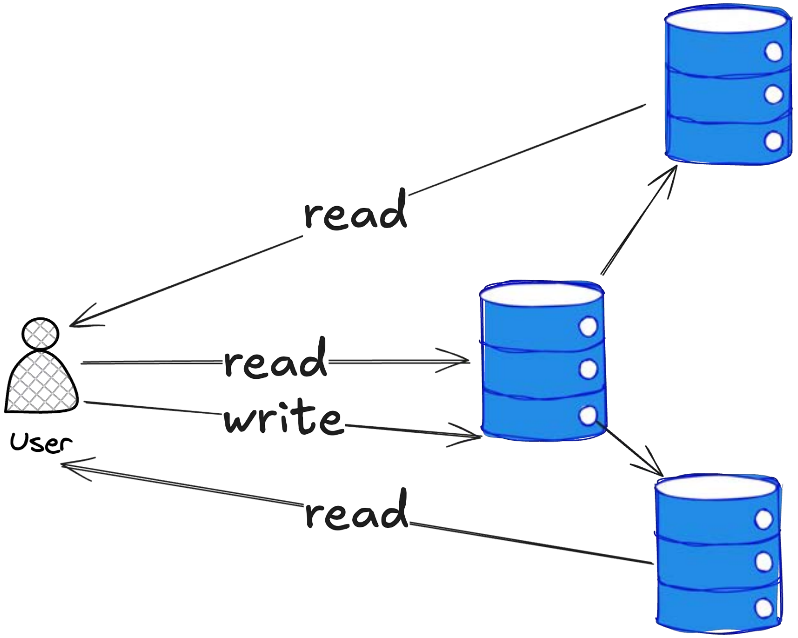
2. **If you’re ready to invest significant effort in database configuration**

Replication

📌 AP — sacrificing C*

✅ No limits on read scalability!

❌ There's a ceiling on write scalability



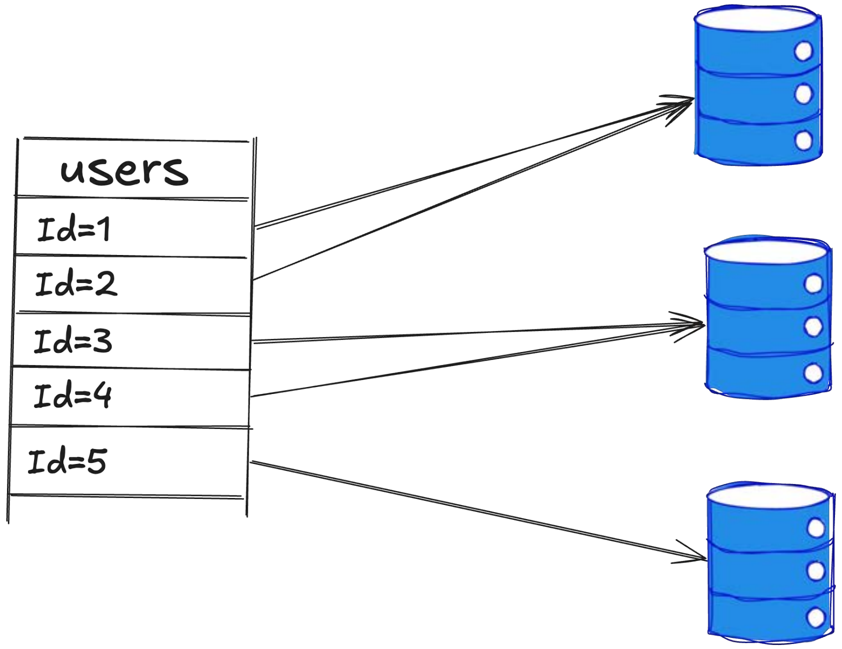
Sharding

📌 CP — sacrificing A*

✅ Can scale both reads and writes

❌ No transactions

❌ Everything becomes much more complex!



NoSQL (2000s)

✅ Designed for **scalability** and **big data**

- 📌 Built-in support for **sharding** and **replication**

✅ Prefer flexible schemas (or no schema at all)

❌ Do not necessarily use tables and SQL

❌ Do not support transactions*

📦 Examples:

- 🔹 **Key-Value:** Redis and DynamoDB
- 📄 **Document-based:** MongoDB
- 🧠 **Graph-based:** Neo4j
- 🔢 **Columnar:** Cassandra и ScyllaDB

Distributed transactions

Distributed transactions require **multiple nodes** (or databases) to perform a series of operations in a coordinated manner — *either all together or none at all*. This is achieved through **coordination protocols** that ensure:

- 📦 Atomicity
- 🔁 Consistency
- 🧷 Fault tolerance
- 🔐 Isolation

⚠️ Key issues:

- 🕒 **Poor performance**
  — Waiting for responses from all nodes.
  — Slow consensus, even with one slow participant.
  — Resource locks until Phase 2 is complete.
- 🔌 **Vulnerability during failures**
  — If the coordinator fails at a critical moment, a freeze may occur.
- 🧩 **Implementation complexity**
  — Manual handling of timeouts, failures, and retries is required.

Distributed SQL (2010s)

📌**CP, often sacrificing A**

✅ **Maintain ACID** (transactions, consistency)
✅ Use **SQL — a familiar and tested language**

✅ **Scale horizontally**, like NoSQL databases

❌ **Sharding needs to be considered during design**

❌ **Joins and transactions can be very slow across different shards**

📌 **Examples of distributed SQL databases:**

- Open source: **CockroachDB, YugabyteDB**
- Cloud-based**: Google Spanner, AWS Aurora**

CAP theorem with Redis example

📌 **In-memory key-value** database

📌 Highly customisable

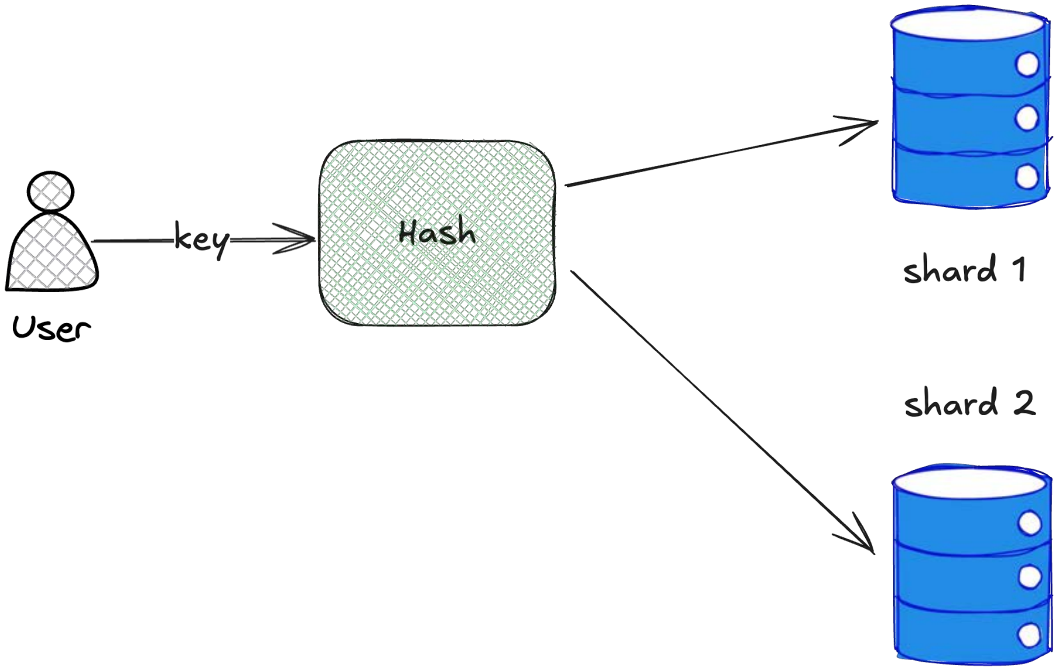
✅ **Consistency**

❌ **Availability**

✅ **Partition tolerance**

✅ **Scalability**

Example — distributed locking



CAP theorem with Redis example

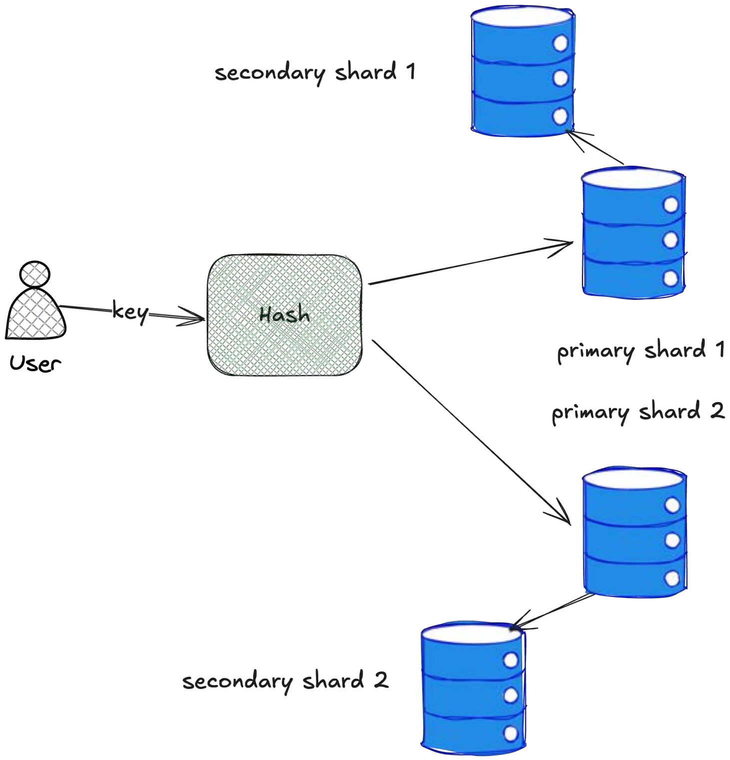
❌ **Consistency**

✅ **Availability**

✅ **Partition tolerance**

✅ **Scalability**

Example — distributed caching



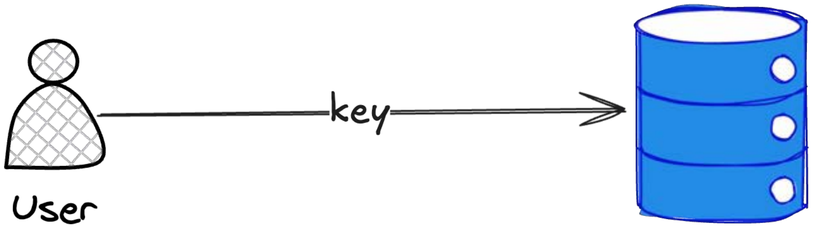
CAP theorem with Redis example

✅ Consistency

✅ Availability

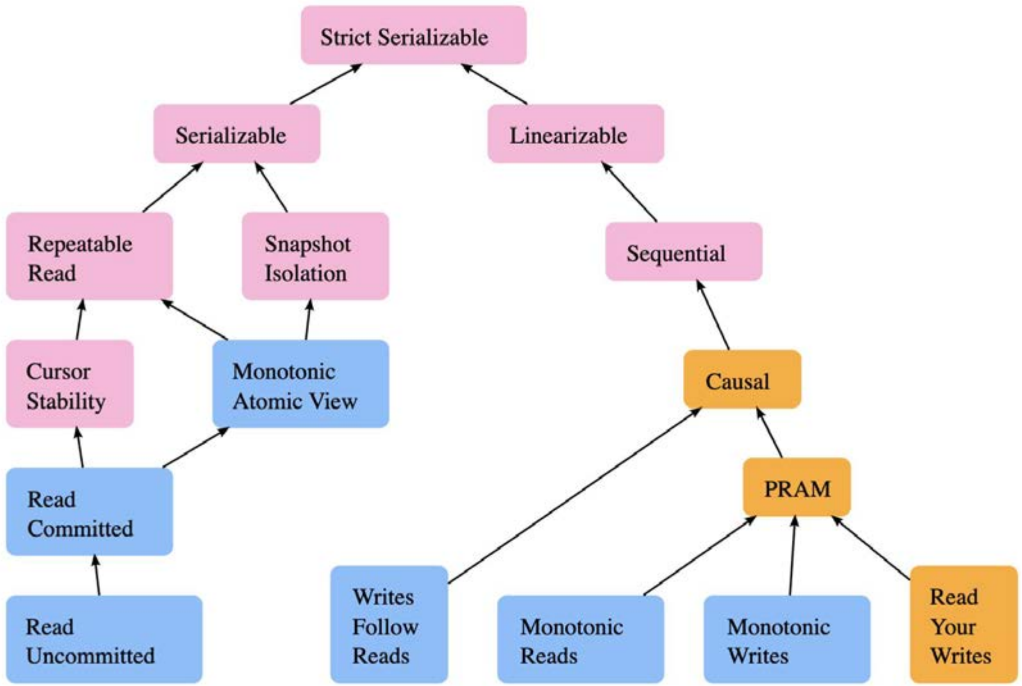
❌ Partition tolerance

❌ Scalability



More on consistency in distributed systems

📚 Academically, many consistency models are distinguished, each with different guarantees



Consistency in practice

✅ **Strong Consistency**

- All clients immediately see **the same data** after a write operation.

🔁 **Eventual Consistency**

- After a write, all nodes will **eventually** contain the same data, but it might not be immediately consistent.

🛠️**Tunable Consistency**

- The level of consistency can be configured at the query level.

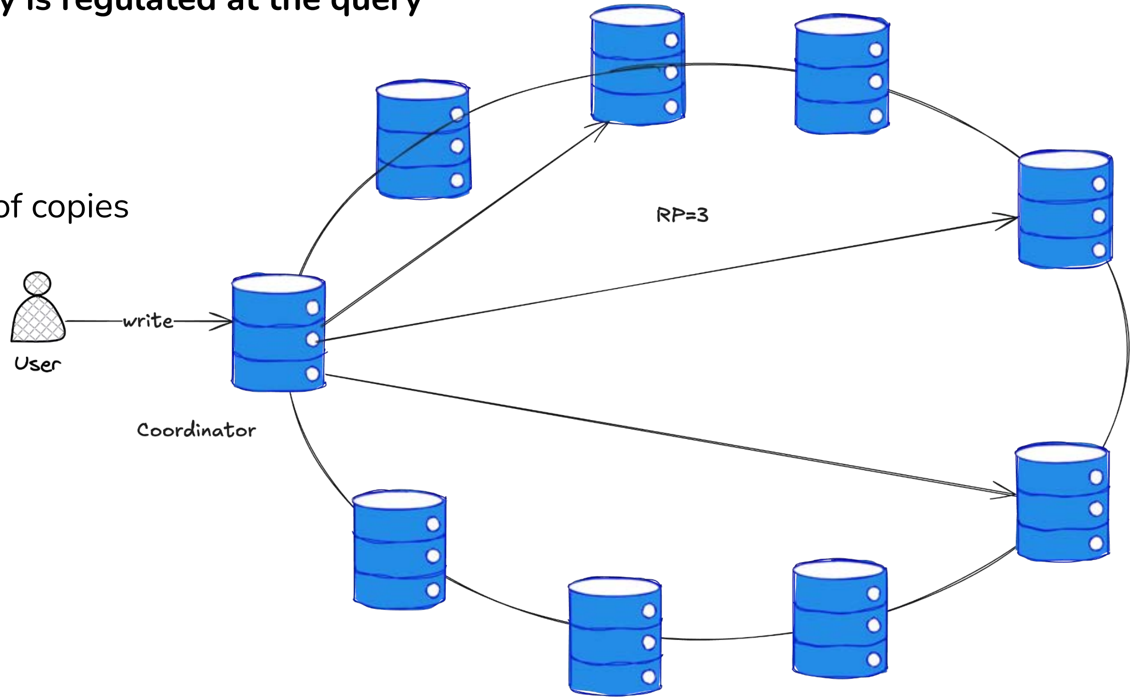
ScyllaDB

✅ **Tunable consistency — consistency is regulated at the query level**

✅ No single point of failure.

RP (Replication level) — the number of copies

in which data is stored.



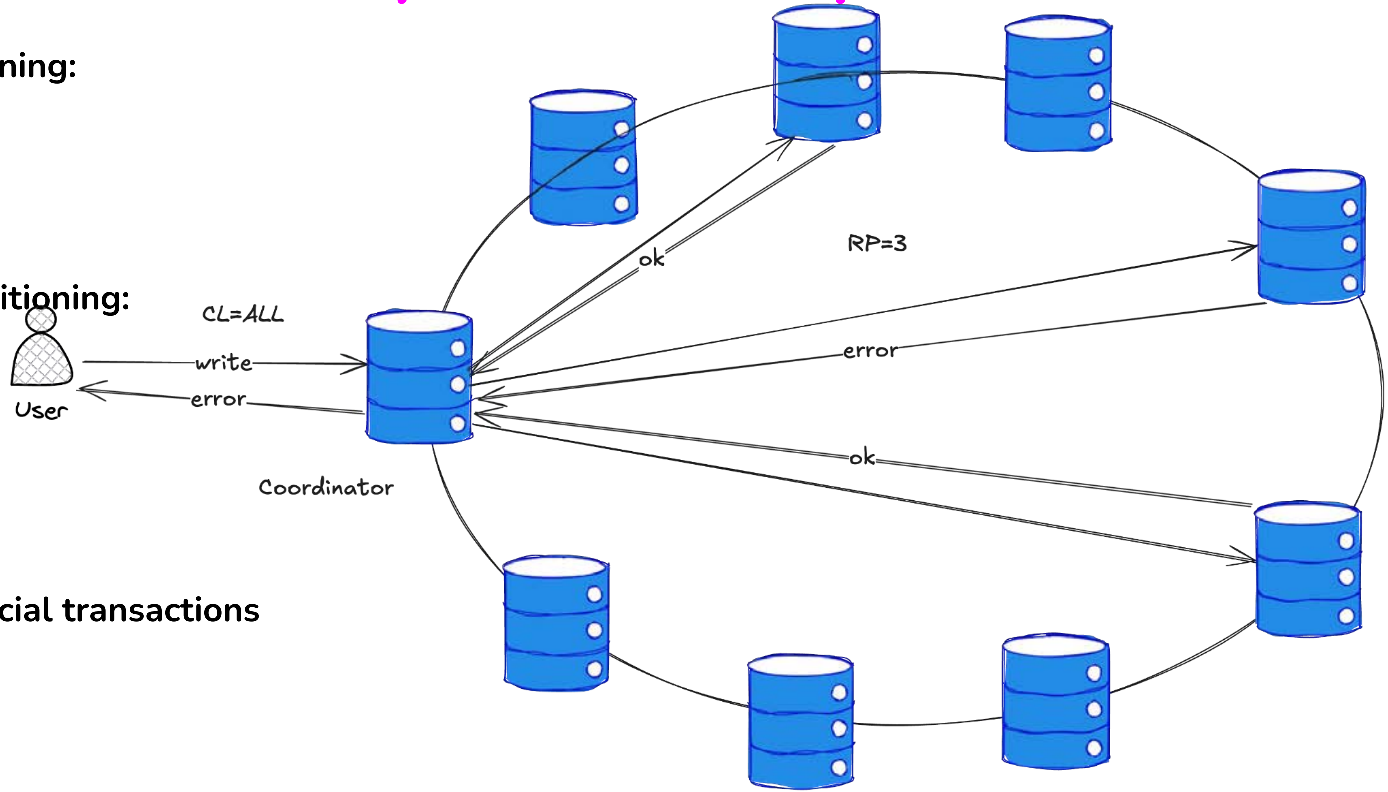
CAP theorem with ScyllaDB example

With network partitioning:

- ✅ Consistency
- ❌ Availability

Without network partitioning:

- ✅ Consistency
- ❌ Performance



Example: critical financial transactions

CAP theorem with ScyllaDB example

With network partitioning:

❌ Consistency

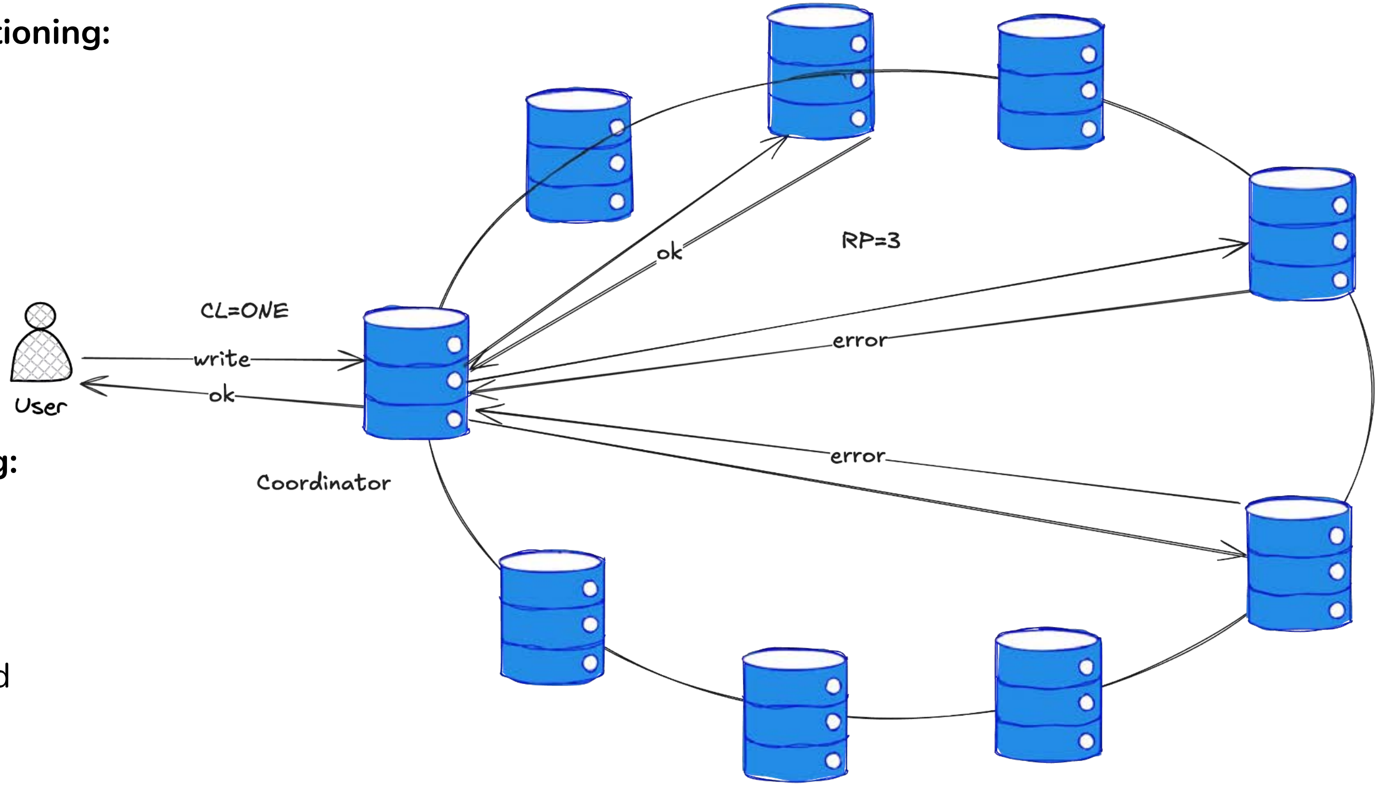
✅ Availability

Without network partitioning:

❌ Consistency

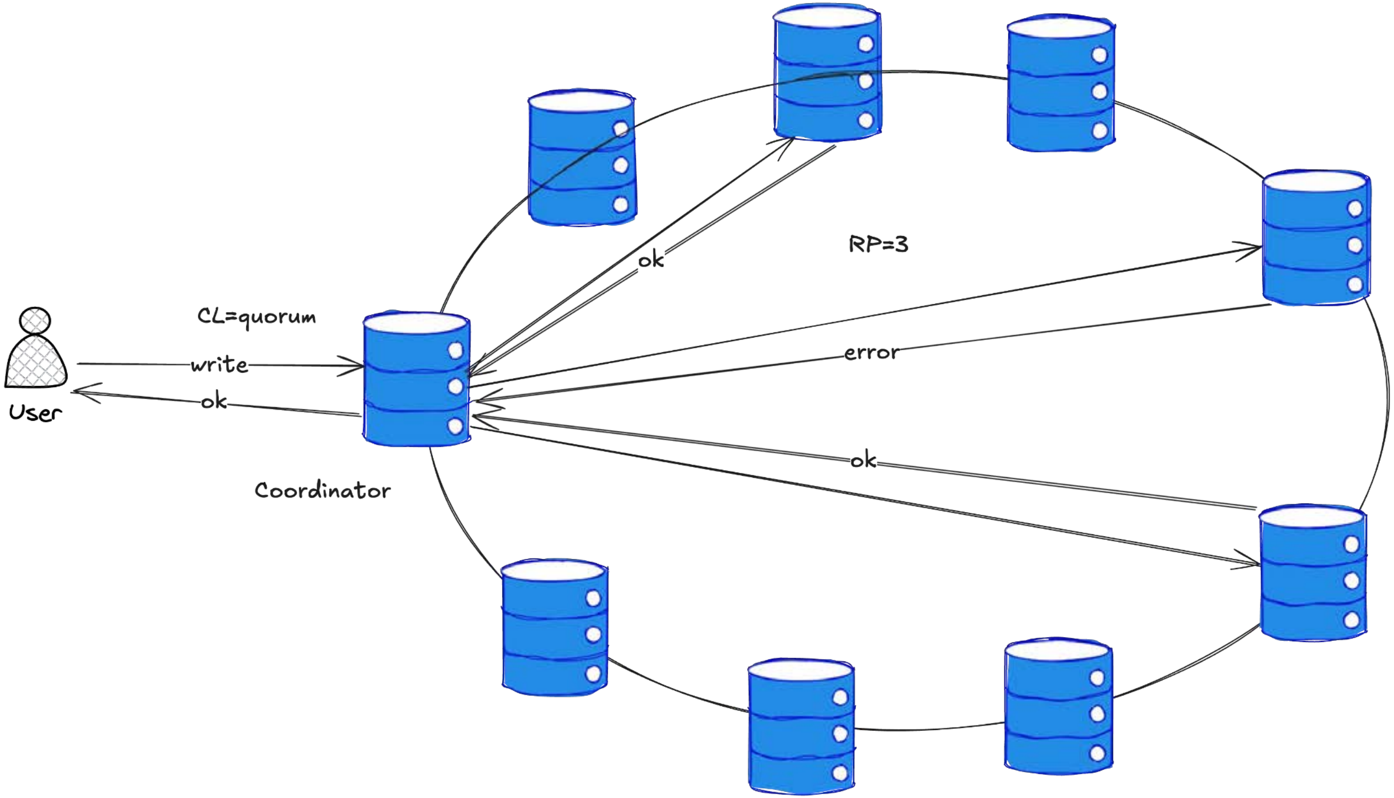
✅ Performance

Examples: news feed



CAP theorem with ScyllaDB example

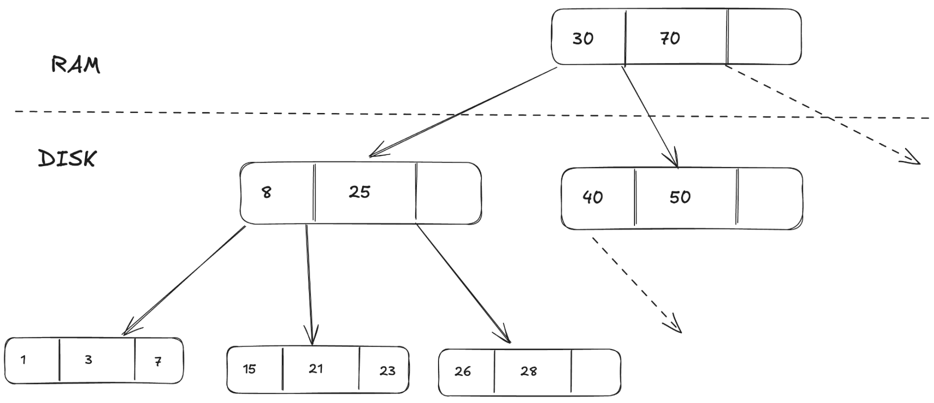
- 🛠️ Consistency
- 🛠️ Availability
- 🛠️ Performance



Database engine: LST tree and B-tree

🌳 B-tree

- Widely used in both **relational database management systems** (RDBMS) and NoSQL databases.



📌 Key properties:

- Optimized for reading -> minimizes the number of disk reads.
- Not optimized for heavy write operations.

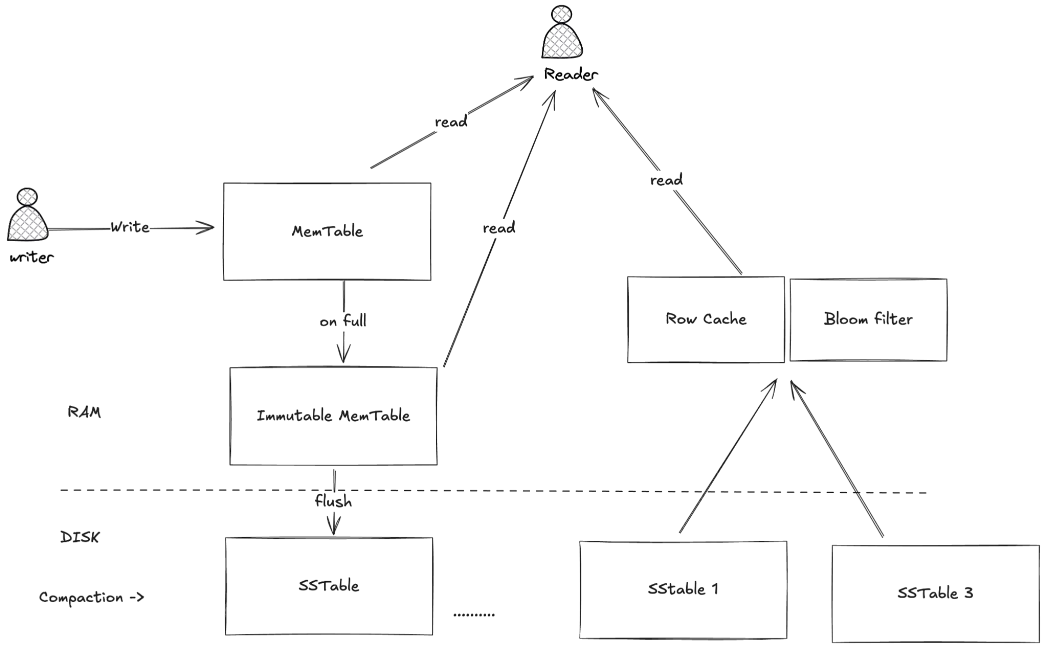
Database engine: LST tree and B-tree

🧱 LSM-Tree (Log-Structured Merge-Tree)

- An LSM-tree is a structure optimized for high write input.
- All changes are first recorded in **memory (MemTable)**.
- Data is periodically written to disk.

Used in:

- **NoSQL DBs**: Cassandra, RocksDB, LevelDB, ScyllaDB



Modern trends

🚀 Companies are rewriting systems while keeping the API intact

🟠 ScyllaDB

- A fully rewritten **alternative to Cassandra** in C++.
- 5–10 times faster than Cassandra with the same API, thanks to the one thread per core architecture.

🟣 Redpanda

- A rewritten version of Kafka, but **without JVM, in C++.**
- 10 times faster thanks to the one thread per core architecture.

🟡 Dragonfly

- A high-performance alternative to Redis.
- **5–25 times faster than Redis** thanks to the one thread per core architecture.

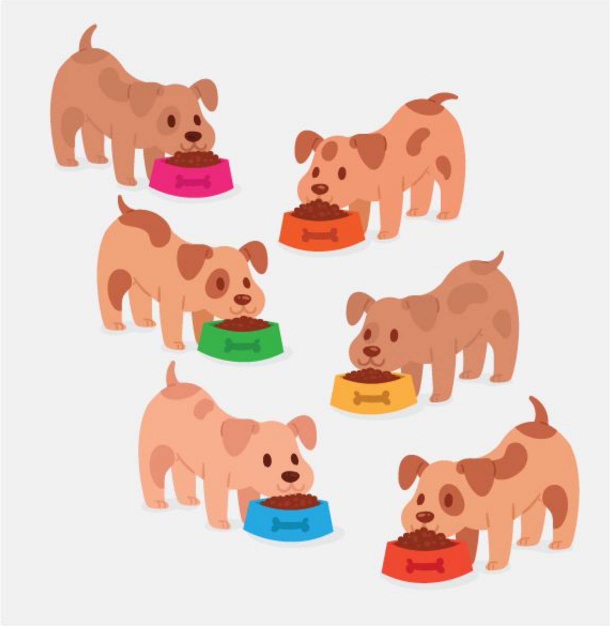
Why faster?

📌 One thread per core.

📌 Each core is assigned its own chunk of data.

✅ No thread contention — faster data processing.

✅ Better resource utilisation without performance degradation.



Contacts

Blog - https://andection.substack.com/

Twitter - @AndreyManakov

BlueSky - https://andection.bsky.social/

Email - andection@gmail.com