Formulaire - MAC

Leonard Cseres | November 12, 2025

Impedance Mismatch Objet-Relationnel Disconnect between data layer and application layer

- Solutions: *ORM, NoSQL*

NoSQL

- Horizontal scaling
- No schema, fields can be added later
- Easy replication
- Simple API
- Not ACID

Column oriented

- Écriture à grande échelle
- Accès aux données co-localisé (pour la lecture et l'écriture)

Key-value

- Caches
- Domaine simple avec accès en lecture rapide
- Systèmes massivement concurrents
- Opaque value

Document oriented

- Suited for agile dev.
- When data modeling follows the structures of natural documents
- No need for migration schema
- No need for ORM layer
- Replicated, each document is independent
- Separation approach: normalized data
  + Data consistency through a single canonical source
  + Simpler queries (closer to relational databases)
  + Better cache efficiency and hardware utilization
  – Requires multiple lookups and joins
  – Forced consistency may be undesirable in some contexts
- Imbrication approach: unnormalized data
  + Faster access (no joins, single document retrieval)
  + Fewer failure points in distributed systems
  + Simpler application logic
  – Risk of data inconsistency due to redundancy
  – More complex queries on nested data
  – Larger, heavier documents

Models

- Represent n-to-m relations
  - document model: *difficult*
  - relation model: *easy*
- Optional fields
  - document model: *possible*
  - relations model: *not possible*

Couchbase

Clause	Scope of alias
`WITH`	Anywhere
`FROM`	Anywhere
`LET`	Anywhere
`LETTING`	`HAVING`, `SELECT` and `ORDER BY` clauses
`SELECT`	`SELECT` and `ORDER BY` clauses
`FOR`	The local collection expression

Indexes

- Primary (on the document key): `CREATE PRIMARY INDEX .. ON ..;`
- Secondary (on any key-value): `CREATE INDEX .. ON ..;`
- Secondary composite (on any key-value): `CREATE INDEX .. ON ..();`
- Covering (the query only selects filed in the index, no need to go to the document)

Graph

- Interconnected data
- When the domain can be represented by nodes and relations
- Social media, recommendation engines
- Relational DBs compute the relations during the query, graph DBs store them.
- Native: custom underlying storage
- Non-Native: underlying relational DB
- **Index-free adjacency**: the relations are stored instead of being indexed

---

Why distribute data? Scalability (Evolutivité), fault tolerance/high availability, reduced latency

Shared memory architecture

- Vertical scaling (scale up)
- Processors share unique memory (SMP or symmetric multiprocessing)

– higher costs
– limited data growth
– limited fault tolerance

Shared disks architecture

- Independent CPU/RAM, but shared disks via fast network
- Used in data warehouses

– limited scaling due to locking/conflict management

Shared nothing architecture

- Horizontal scaling (scale out)
- Each node manages its own CPU/RAM
- Coordination via software/network

Distribution Goals

- **Évolutivité** (Scalability) for high volume/load
- **Tolérance aux pannes/haute disponibilité** for continuous function
- **Latence réduite** serving users from nearby centers

Preferred Architecture Architectures sans partage (Scale Out). Each node manages its own CPU, RAM, and disks. Coordination via network/software. Excellent price/performance.

Distribution Mechanisms

- **Partitionnement (Sharding)** divides large DB into smaller partitions.
- **Réplication** maintains copies for latency, availability, and read throughput.

Single-Leader Replication One **leader** handles all writes and sends changes via logs to **followers**.

Replication Timing Trade-offs **Synchrone** guarantees up-to-date followers but adds latency/risk of leader waiting. **Asynchrone** reduces latency but risks data loss if leader fails before replication.

Leader Failure Requires **Failover**. A follower is promoted. **Consensus algorithm** (Raft, Paxos) selects the new leader, usually the one with the most recent data.

Replication Lag/Consistency Replication delay causes **cohérence éventuelle (eventual consistency)**. Followers eventually catch up if writes stop.

Read Guarantees

- **Reading Your Own Writes:** Requires ensuring the user reads from the leader, or tracking the user's latest update timestamp to guarantee the replica reflects that change.
- **Lectures Monotones (Monotonic Reads):** A user never sees an older value after a newer one. Solution: Hash the user ID to ensure reads stick to the **same replica**.

Leaderless Replication Any replica accepts writes. Consistency handled via **Quorums**.

Quorum Requirement Need $W + R > N$ (Writes + Reads > Replicas). This ensures at least one node queried for a read has the latest written data.

Partitioning by Key Range Assigns continuous key ranges. Good for **range queries**. Risk of **hot spots** if sequential keys (like timestamps) are frequently written.

Partitioning by Key Hash Uses hashing for even key distribution. Eliminates hot spots. Sacrifices efficient range queries.

Rebalancing Process of moving load/data between nodes.

- **Avoid** `hash mod N` because changing N requires moving most keys.
- **Preferred Strategy:** Use a **fixed number of partitions** (more than nodes, N). Only entire partitions are moved during rebalancing.

Concurrency Conflicts (Centralized DB)

- **Dirty Write (Écriture Sale):** One transaction overwrites another's uncommitted value. Avoided by acquiring **write locks**.
- **Dirty Read (Lecture Sale):** Reading data written by an uncommitted transaction.

CAP Theorem System cannot simultaneously guarantee all three: Consistency, Availability, Partition Tolerance.

CAP Trade-off Partition Tolerance (P) is unavoidable. When network partitioning occurs, designers must choose between **Consistency** (sacrificing A by rejecting requests) or **Availability** (responding, even if data is stale).