Introduction NoSQL

Impedance mismatch: décalage modèle relationnel vs objets. Solutions: ORM ou Document (JSON). **Problème**: multiples jointures. **Solution**: tout en JSON. Relations 1:N faciles, M:N complexes.

Scalabilité: Verticale (↑ CPU/RAM, limites physiques, coûteux) vs Horizontale (↑ serveurs, économique, complexe avec relationnel).

Caractéristiques NoSQL: pas relationnel, scalabilité horizontale, schéma flexible, open-source, réplication, API simples, **eventually consistent**.

Catégories: Documents (MongoDB, Couchbase), Colonnes (Cassandra, HBase), Graphes (Neo4j), Clé-valeur (Redis, Riak).

BD Documents & Couchbase

Stockage: JSON natif, tout dans un document. **Schéma flexible**: dynamique, champs variables. **Couchbase**: documents + clé-valeur, langage **N1QL** (SQL-like).

Modélisation

Imbrication:
- Avantages: vitesse (1 seule requête, pas jointures), tolérance pannes (données sur même nœud/machine)
- Inconvénients: incohérence (duplication données), docs volumineux, requêtes complexes sur parties imbriquées

Séparation:
- Avantages: cohérence (pas duplication), cache efficace (docs canoniques fréquemment accédés), requêtes simples
- Inconvénients: jointures multiples, recherches multiples

Règles décision: 1:1/1:N → imbriqué. M:N → séparés. Lectures parent+enfant fréquentes → imbriqué. Cohérence prioritaire → séparés.

Physique: Bucket (DB), Collection (table), Scope (regroupement), Item (clé + JSON).

N1QL Mots-clés

Base: `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `LIMIT`, `RAW`, `DISTINCT`.

Découverte: `INFER`. **Accès**: `.`, `[]`, `LIKE`, `META().id`, `USE KEYS`.

Collections: `IN/NOT IN`, `WITHIN/NOT WITHIN`, `ANY...SATISFIES...END`, `EVERY...SATISFIES...END`.

Tableaux: `ARRAY...FOR...WHEN...END`, `FIRST...FOR...WHEN...END`.

Agrégation: `GROUP BY`, `HAVING`, `COUNT`, `ARRAY_AGG`, `ARRAY_COUNT`, `ARRAY_MAX/MIN`.

NULL: `IS VALUED`, `IS NULL`, `IS MISSING`.

Index: `CREATE PRIMARY INDEX`, `CREATE INDEX`, composite, couvrant. `DISTINCT ARRAY...FOR...END` (tableaux).

Jointures: `INNER JOIN`, `LEFT/RIGHT OUTER JOIN`, `ON`. `NEST` (imbrique résultat), `UNNEST` (aplatit tableau).

Avancé: Sous-requêtes, `LET`, `LETTING`, `EXPLAIN`.

BD Graphes

Property Graph: Nœuds (propriétés, labels) + Relations (dirigées, nommées, propriétés). Relations stockées (**index-free adjacency**) → temps constant vs JOIN.

Perf: constante quelle que soit taille. RDBMS dégrade exponentiellement. Ex: 1M personnes, prof. 5: RDBMS 1543s, Neo4j 2.1s.

Neo4j: BD native graphe, langage **Cypher**. Convention: labels `CamelCase`, relations `MAJUSCULES_TIRETS`.

Cas d'usage: réseaux sociaux, recommandations, détection fraude, gestion dépendances, chemins/routage.

Cypher

Nœuds: `()`, `(p:Person)`, `(p:Person {name: 'Alice'})`.

Relations: `(a)-->(b)`, `(a)-[r:TYPE]->(b)`, `(a)-[r:TYPE {p: v}]->(b)`, `(a)<-[:TYPE]-(b)`.

CRUD: `CREATE`, `MATCH...RETURN`, `WHERE`, `WHERE NOT`, `IS NOT NULL`, `SET`, `DELETE`, `DETACH DELETE` (avec relations), `REMOVE`, `MERGE`, `ON CREATE/MATCH`.

Index: `CREATE INDEX FOR (a:Actor) ON (a.name)`, composite. `SHOW indexes`.

Contraintes: `CREATE CONSTRAINT...REQUIRE...IS UNIQUE`. `SHOW constraints`.

Agrégations: `count()`, `collect()`, `avg()`, `max()`, `min()`, `sum()`, `count(DISTINCT x)`.

Avancé: `WITH` (pipeline résultats), `UNWIND` (transforme liste en lignes), `ORDER BY`, `DISTINCT`, `LIMIT`.

Chemins: `[:TYPE*]` (1+), `[:TYPE*3]` (exactement 3), `[:TYPE*3..5]` (3 à 5), `[:TYPE*3..]` (3+), `[:TYPE*..5]` (max 5), `shortestPath()`.

OPTIONAL MATCH: comme LEFT JOIN, retourne NULL si pas match.

Distribution & Cohérence

Pourquoi distribuer: scalabilité (↑ charge), haute disponibilité (tolérance pannes), latence réduite (centres proches utilisateurs).

Architectures:
- Mémoire partagée (SMP): coûteux, limites scalabilité
- Disques partagés: conflits accès
- **Sans partage/Scale Out**: nœuds indépendants, matériel standard, coordination logicielle (modèle privilégié)

Modèles: Partitionnement (sharding: diviser données) + Réplication (copies multiples). Souvent combinés.

Réplication

3 types principaux:

1. Leader unique (PostgreSQL, MySQL):
- Leader écrit, followers répliquent
- Lecture: leader ou followers
- **Sync**: lent, données garanties. **Async**: rapide, risque perte
- **Semi-sync**: 1 follower sync, autres async (compromis courant)
- **Failover** (panne leader): timeout (30s) → élection nouveau leader (consensus: Raft, Paxos) → reconfiguration clients

Logs réplication:
- Statement-based (SQL): problème fonctions non-déterministes (`NOW()`, `RAND()`)
- WAL shipping (octets disque): couplage moteur, incompatibilité versions
- Logical/row-based (lignes): découplage, rétrocompatible
- Trigger-based: flexible, coûteux

Replication lag: délai leader→follower. **Cohérence éventuelle**: incohérence temporaire.

Problèmes lag:
- **Read-your-writes** (ne pas voir ses modifs): lire du leader pour données modifiables, suivre timestamp
- **Lectures monotones** (régression temps): même réplique par utilisateur (hash ID)

2. Multi-leader:
- Plusieurs acceptent écritures
- Cas d'usage: offline, multi-datacenter, collaborative editing
- **Problème**: conflits d'écriture
- **Solutions**: last write wins, vecteurs de version, résolution applicative

3. Sans leader (Dynamo: Cassandra, Riak):
- Toute réplique accepte écritures
- **Quorums**: n répliques, w écritures confirmées, r lectures interrogées
- **w + r > n** garantit données à jour
- Ex: n=5, w=3, r=3 → tolérance 2 nœuds
- Ajustement possible: w=n, r=1 (lectures rapides, écritures bloquées si panne)

Partitionnement

Objectif: diviser données volumineuses/débit élevé.

Terminologies: shard (MongoDB), region (HBase), tablet (Bigtable), vnode (Cassandra), vBucket (Couchbase).

Partitionnement équitable évite hotspots (charge disproportionnée sur partition). Chaque partition: leader + followers.

Stratégies:
- **Intervalle de clé** (range): plages continues, données triées, range queries OK. **Risque**: hotspots (ex: timestamp → tout sur partition aujourd'hui)
- **Hachage de clé**: répartition équitable, **pas** de range queries (interroge toutes partitions)

Rééquilibrage:
- NE PAS `hash(key) mod N` (changement N → tout bouge)
- **Partitions fixes**: nombre >> nœuds, déplacement partitions entières, nombre constant
- Ex: Riak, Elasticsearch, Couchbase
- Automatique vs manuel (spectre: Couchbase suggère, admin approuve)

Service discovery: comment clients trouvent bonne partition?
1. Client contacte n'importe quel nœud (round-robin), transmet si besoin
2. Couche routage (load balancer conscient partitions)
3. Client conscient (connexion directe partition)

ZooKeeper: coordination, mappage partitions↔nœuds, nœuds s'enregistrent, routage s'abonne changements.

Cohérence

Cohérence: absence contradiction. SGBDR centralisé: forte cohérence. NoSQL distribué: assouplissement → cohérence à terme.

Problèmes SGBD centralisé:
- **Dirty Write**: 2 transactions MAJ simultanées → incohérent. Solutions: verrous (pessimiste) ou détection conflits (optimiste)
- **Dirty Read**: lire écritures non-validées (ex: message inséré, compteur pas encore incrémenté)

Niveaux isolation: read uncommitted (permet dirty reads), read committed, repeatable reads, serializable (complet).

Transactions NoSQL:
- Centralisé: journalisation (log modifications, rollback si échec)
- Distribué: plus complexe (logs séparés, coordination nécessaire)

Cohérence réplication: même valeur sur répliques différentes.
- Cohérence à terme: propagation prend temps, données périmées (stale) temporairement
- **Read-your-writes**: sticky session (affinité nœud) ou lire leader

Théorème CAP: système distribué ne peut garantir simultanément les 3:
- **Consistency**: tous nœuds voient mêmes données au même moment
- **Availability**: chaque requête → réponse (succès/échec)
- **Partition tolerance**: système fonctionne malgré isolation réseau (coupure)

Face au partitionnement réseau (inévitable): choisir **cohérence** (rejeter requêtes jusqu'à résolution) OU **disponibilité** (accepter données périmées).

Compromis cohérence ↔ temps de réponse selon besoins opérations.