Introduction NoSQL

Impedance mismatch: décalage entre modèle relationnel (tables) et modèles objets/flexibles. Solutions: ORM (mappage objets-tables) ou approche Document (JSON).

Problème relationnel: CV LinkedIn nécessite multiples jointures (users, positions, education, etc.). **Solution Document**: tout dans un seul JSON, structure arborescente naturelle. Relations 1:N faciles, M:N plus complexes.

Scalabilité: Verticale (↑ CPU/RAM d'un serveur, limites prix/matériel) vs Horizontale (↑ nombre serveurs, économique, résilient, complexe avec relationnel).

Caractéristiques NoSQL

- Pas de modèle relationnel ni SQL
- Scalabilité horizontale
- Schéma flexible (ajout/suppression champs dynamique)
- Open-source, réplication aisée
- API simples, **eventually consistent** (pas ACID)

Catégories NoSQL

Documents (MongoDB, Couchbase): JSON/BSON, index sur champs, examinables. Cas: CMS, apps web/mobile, ML, réseaux sociaux.

Colonnes (Cassandra, HBase): map 2 niveaux, efficace pour analytics. Cas: logs, comptage, reporting.

Graphes (Neo4j): nœuds + arêtes, exploration relations. Cas: réseaux sociaux, SIG, recommandations.

Clé-valeur (Redis, Riak): hash distribué, lecture/écriture rapides. Cas: sessions, cache, préférences, recommandations.

BD Documents & Couchbase

Stockage: format natif JSON, tout dans un document (pas de jointures multiples). **Schéma flexible**: auto-descriptif, dynamique, champs variables, pas de migration. Distribution facile (unité indépendante).

Couchbase: orientée documents + clé-valeur. Langage **N1QL** (syntaxe SQL-like pour JSON). Architecture distribuée.

Modélisation

2 approches: Séparation (normalisé, docs séparés avec références, type comme clé étrangère) vs Imbrication (dénormalisé, tout dans un doc parent).

Imbrication: + vitesse (1 recherche), tolérance pannes. - incohérence (redondance), requêtes complexes, docs volumineux.

Séparation: + cohérence (copie canonique), requêtes simples, cache efficace. - recherches multiples, jointures nécessaires.

Règles: 1:1 ou 1:N → imbriqué. M:N → séparés. Lectures parents+enfants → imbriqué. Cohérence prioritaire → séparés.

Concepts physiques: Bucket (database), Collection (table), Scope (regroupement collections), Item (clé unique + valeur JSON/binaire).

N1QL Mots-clés

Base: `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `LIMIT`. `RAW` (sans wrapper), `DISTINCT` (éliminer doublons).

Découverte: `INFER` (métadonnées schéma: nb docs, % docs, types).

Champs: `.` (enfants), `[]` (tableaux), `LIKE` (pattern). `META().id` (clé document). `USE KEYS` (recherche directe par clé).

Collections: `IN/NOT IN` (contenu direct), `WITHIN/NOT WITHIN` (direct/indirect), `ANY...SATISFIES...END` (au moins un), `EVERY...SATISFIES...END` (tous).

Tableaux: `ARRAY...FOR...WHEN...END` (construire), `FIRST...FOR...WHEN...END` (premier élément).

Agrégation: `GROUP BY`, `HAVING` (filtre après agrégation), `COUNT`, `ARRAY_AGG` (valeurs→tableau), `ARRAY_COUNT` (compte non-NULL), `ARRAY_MAX/MIN`.

NULL/MISSING: `IS VALUED` (a valeur), `IS NULL` (explicitement null), `IS MISSING` (absent), `IS NOT MISSING`.

Index: `CREATE PRIMARY INDEX` (sur clé), `CREATE INDEX` (secondaire sur champs), index composite (plusieurs champs), index couvrant (tous champs requête). Index sur tableaux: `DISTINCT ARRAY...FOR...END`.

Jointures: `INNER JOIN`, `LEFT/RIGHT OUTER JOIN` (ANSI recommandé), `ON` (condition). `NEST` (imbrique résultats en tableau, pas ligne par match). `UNNEST` (aplatit tableau→lignes).

Sous-requêtes: dans WHERE, FROM, SELECT. `LET` (variables locales), `LETTING` (après GROUP BY).

Optimisation: `EXPLAIN` (plan exécution), index couvrants (évite récupération docs).

BD Graphes

Structure: Graphe = sommets (nœuds) + arêtes (relations). Omniprésent: réseaux sociaux, recommandations, détection fraudes.

Property Graph:
- **Nœuds**: entités avec propriétés (clé-valeur) et labels (0 à N catégories)
- **Relations**: dirigées, nommées, avec propriétés. Toujours entre 2 nœuds (source→destination). Citoyens 1ère classe (stockées, pas calculées)
- **Index-free adjacency**: relations stockées avec nœuds → traversée temps constant (vs JOIN coûteux)

Graphes vs Relationnel: Performance BD graphe constante quelle que soit taille données. RDBMS dégrade avec données connectées (JOINs). Ex: amis profondeur 5 sur 1M personnes: RDBMS 1543s (prof. 4), Neo4j 2.1s (prof. 5).

Neo4j: BD graphe native NoSQL (Java/Scala). Stockage natif graphes dès conception. Langage **Cypher** déclaratif (projet openCypher). Éditions communautaire + entreprise.

Convention nommage: Labels nœuds `CamelCase` (`:VehicleOwner`), Relations `MAJUSCULES_TIRETS` (`:OWNS_VEHICLE`), propriétés/variables `camelCase` (`businessAddress`).

Syntaxe Cypher

Nœuds: `()` (anonyme), `(p:Person)` (label+var), `(:Tech)` (label seul), `(p:Person {name: 'Alice'})` (avec propriété).

Relations: `(a)-->(b)` (dirigée), `(a)-[r:TYPE]->(b)` (type+var), `(a)-[r {prop: val}]->(b)` (propriété). Direction obligatoire création, optionnelle recherche.

CRUD:
- `CREATE (m:Movie {title: 'Matrix'})`
- `MATCH (m:Movie) RETURN m.title`
- `WHERE m.released >= 1990 AND m.released < 2000`
- `WHERE NOT p.name = 'X'`, `WHERE p.birth IS NOT NULL`
- `SET p.birthdate = date('1980-01-01')`
- `DELETE r` (relation), `DETACH DELETE m` (nœud+relations)
- `REMOVE n.birthdate` ou `SET n.birthdate = null`
- `MERGE (m:Person {name: 'Mark'})` (évite doublons)
- `ON CREATE SET x.created = timestamp()`
- `ON MATCH SET x.lastSeen = timestamp()`

Index: `CREATE INDEX FOR (a:Actor) ON (a.name)` (simple), `ON (a.name, a.born)` (composite). Accélère recherche points départ parcours. `SHOW indexes`.

Contraintes: `CREATE CONSTRAINT FOR (m:Movie) REQUIRE m.title IS UNIQUE`. Crée implicitement index. `SHOW constraints`.

Requêtes avancées

Agrégations: `count()`, `collect()`, `avg()`, `max()`, `min()`, `sum()`. `count(DISTINCT x)` (dédoublonnage). `RETURN collect(p2.name)` (tableau).

WITH: calculs intermédiaires, filtrage sur agrégation. `WITH other, count(*) AS conn WHERE conn > 1 RETURN other.name`.

UNWIND: liste→lignes. `UNWIND ['A','B'] AS x MATCH (p)-[:LIKES]-(t:Tech {type: x})`. `WITH DISTINCT x` puis `collect(x)` (liste unique).

ORDER BY: tri résultats. `ORDER BY p.exp DESC`.

DISTINCT: résultats uniques. `RETURN DISTINCT user.name`.

Chemins variables:
- `[:KNOWS*]` (1+ sauts)
- `[:KNOWS*3]` (exactement 3)
- `[:KNOWS*3..5]` (entre 3 et 5)
- `[:KNOWS*3..]` (3+), `[:KNOWS*..5]` (≤5)
- `p = (a)-[:KNOWS*3..5]->(b)` (nommer chemin)

Plus courts chemins: `shortestPath((a)-[*]-(b))` (Bacon number).

Distribution & Cohérence

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

Architectures: Mémoire partagée (SMP, coûteux), Disques partagés (conflits), Sans partage/Scale Out (nœuds indépendants, matériel standard, coordination logicielle).

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

Réplication

3 types:
- **Leader unique** (PostgreSQL, MySQL): leader écrit, followers répliquent. Lecture: leader ou followers
- **Multi-leader**: plusieurs acceptent écritures. Cas: offline, multi-DC, collaborative editing. Problème: conflits, vecteurs de version
- **Sans leader** (Dynamo: Cassandra, Riak): toute réplique accepte écritures

Sync vs Async: Synchrone (lent, données garanties) vs Asynchrone (rapide, risque perte). **Semi-synchrone**: 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:
- Read-your-owns-writes (ne pas voir ses modifs): lire du leader pour données modifiables, suivre timestamp
- Lectures monotones (régression): même réplique par utilisateur (hash ID)

Sans leader - 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: 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). Chaque partition: leader + followers.

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

Rééquilibrage: Ne PAS utiliser `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, ex: Couchbase suggère, admin approuve).

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

Défi: apprendre changements affectations. **ZooKeeper**: coordination, mappage partitions↔nœuds, nœuds s'enregistrent, routage s'abonne.

Cohérence

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

Problèmes SGBD centralisé:
- **Dirty Write**: 2 transactions MAJ simultanées (ex: Alice facture, Bob listing → incohérent). Solutions: verrous (pessimiste) ou détection (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). Distribué: plus délicat (logs séparés).

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

Théorème CAP: système distribué ne peut avoir les 3:
- **Consistency**: tous voient mêmes données
- **Availability**: chaque requête → réponse
- **Partition tolerance**: fonctionne malgré isolation

Face au partitionnement réseau: choisir cohérence (rejeter requêtes) OU disponibilité (données périmées). Compromis cohérence ↔ temps de réponse selon opérations.