NoSQL, graphes et Neo4J

I.Mougenot

HAI914I - FDS

2022

UNIVERSITÉ DE MONTPELLIER

Plan général du cours

1. Introduction au NOSQL
   1. Relationnel / NOSQL
   2. Typologie et mécanismes clés
2. Principes généraux des systèmes orientés graphes
   - Positionnement contextuel (taille vs complexité/expressivité)
   - Accointances avec les systèmes à objets et navigationnels
   - Adossement à la théorie des graphes
   - modèle de données : graphe attribué et orienté
3. Un système en particulier : Neo4J
4. Neo4J face aux triplestores

Les motivations autour du NoSQL (Not Only SQL)

Recouvre différentes initiatives complémentaires aux modèles relationnels et relationnels-objets

1. évolution du Web, sources de données ouvertes (LOD), impulsion Google, Facebook, Amazon, Twitter, ...
2. ↗ volume des données ↗ interconnexion des données
3. limites des bases de données relationnelles face à de nouveaux besoins :
   - flexibilité : schémas très ouverts : nombreuses entités et associations entre ces entités
   - adaptabilité : évolutions très fréquentes des schémas
   - des milliers voire des millions d'usagers

Quand passer par un système NoSQL ?

Alternatives au relationnel dans des cas de figure ciblés

- recours fréquent à de l'évolution de schémas
  - entités munies de diverses caractéristiques souvent non renseignées
  - nombreuses associations avec des multiplicités 1..* aux extrémités
  - attributs composites
- un flux transactionnel (lecture, écriture) très élevé
- données distribuées dès l'origine (mondialisation)

NoSQL : se démarquer des SGBD relationnels

Réplication et partitionnement : deux techniques qui peuvent se combiner

- schémas normalisés vus comme des sophistications inutiles au détriment de l'efficacité
- modèle transactionnel et propriétés ACID : proposer une alternative moins exigeante
- passage à l'échelle par ajout de serveurs au niveau de l'architecture physique : diminuer le temps de réactivité lors de l'afflux de nouveaux usagers, de nouvelles transactions à servir
- systèmes distribués et mécanismes de tolérance aux pannes : fragmentation des schémas et réplication, médiateur, entrepôt de données ...

Passage à l'échelle ou scalabilité

Capacité de l'architecture à s'adapter à une montée en charge (nouveaux usagers, nouvelles transactions) sans besoin de refonte des applications

- scalabilité horizontale (scaling out) : ajouter des serveurs (noeuds) avec des mécanismes de répartition de charge $\Leftarrow$ NoSQL
- scalabilité verticale (scaling up) : rendre plus performant un serveur : ajout de processeurs (CPU), barrettes mémoire (RAM), disques secondaires, cartes réseaux ...

Scalabilité horizontale

Etablir une relation linéaire entre les ressources ajoutées et l'accroissement des performances

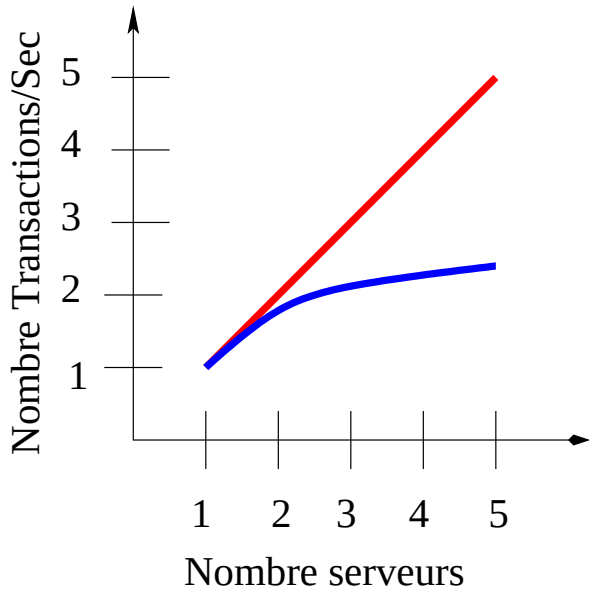


Figure: 1 serveur : 100 transactions/s ; 2 serveurs : 200 transactions/s

...

SGBDR et besoins applicatifs à large échelle

Limites face aux besoins des applications à large échelle sur le Web (à partir Web 2.0)

- partionnement : les schémas fragmentés (fragmentations horizontale, verticale, hybride) distribués sur l'ensemble des partitions doivent être des fragments d'un seul schéma de données initial
- réplication sur différents noeuds : les fondements OLTP (On Line transactional processing) imposent de maintenir une intégrité forte sur les données, dans une application faisant appel à de nombreux noeuds, la disponibilité des données va être pénalisée (surtout si les transactions impliquent de nombreuses écritures).

systèmes NoSQL : grands principes

- **Simplicité**
- **Flexibilité**
- **Efficacité**
- **Passage à l'échelle** : gros volumes de données distribués et interconnectés
  - partitionnement dynamique - sharding (partitionnement horizontal + plusieurs co-occurrences de schémas)
  - réplication à large échelle
  - architecture décentralisée

Complémentarité des systèmes NoSQL

One size doesn't fit all

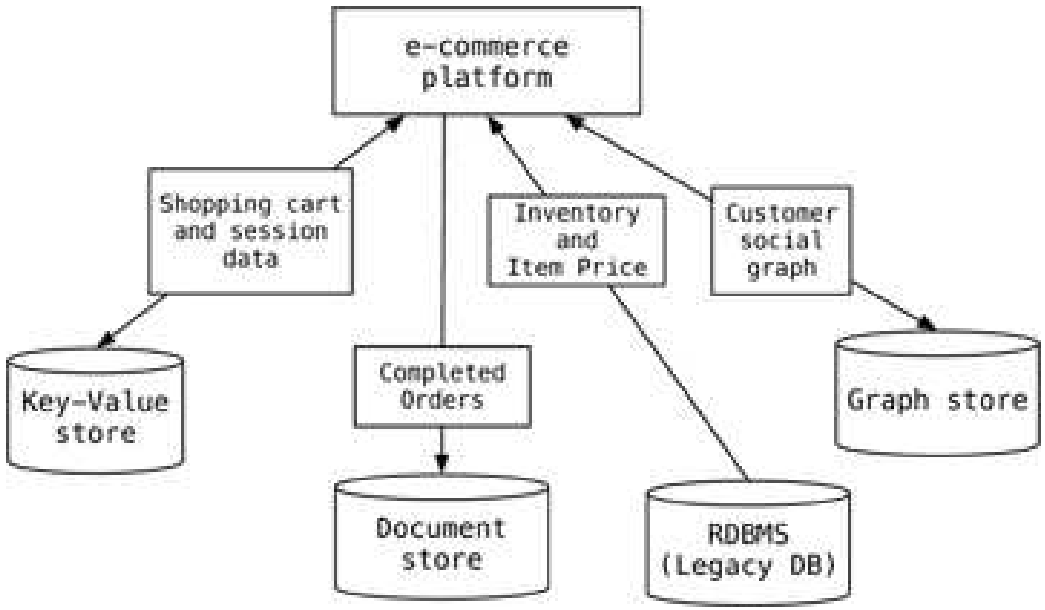


Figure: Persistance dite polyglotte (extrait de NoSQL distilled)

Principe CAP

Constat de Brewer (Towards robust distributed systems, ACM 200) : aucun système distribué n'est à même de satisfaire en même temps les principes C, A et P (au mieux 2 sur les 3)

1. **Consistency (cohérence)** : toute modification de donnée est suivie d'effet pour tous les nœuds du système
2. **Availability (disponibilité)** : toute requête émise et traitée par un nœud du système, reçoit une réponse (même en situation d'échec à produire une réponse)
3. **Partition tolerance (recouvrement des nœuds)** : assurer une continuité du fonctionnement en cas d'ajout/suppression de nœuds du système

Principe CAP

Considérations SGBDR / Systèmes NoSQL

1. **SGBDR** : Cohérence et haute disponibilité (pas ou peu de P, cad petit nombre de nœuds système)
2. **Systèmes NoSQL** : Choix du P (système naturellement distribué) et sélection soit du C, soit du A
   1. abandon du A $\Leftarrow$ Accepte d'attendre que les données soient cohérentes
   2. abandon du C $\Leftarrow$ Accepte de recevoir des données parfois incohérentes

Positionnement des systèmes / CAP

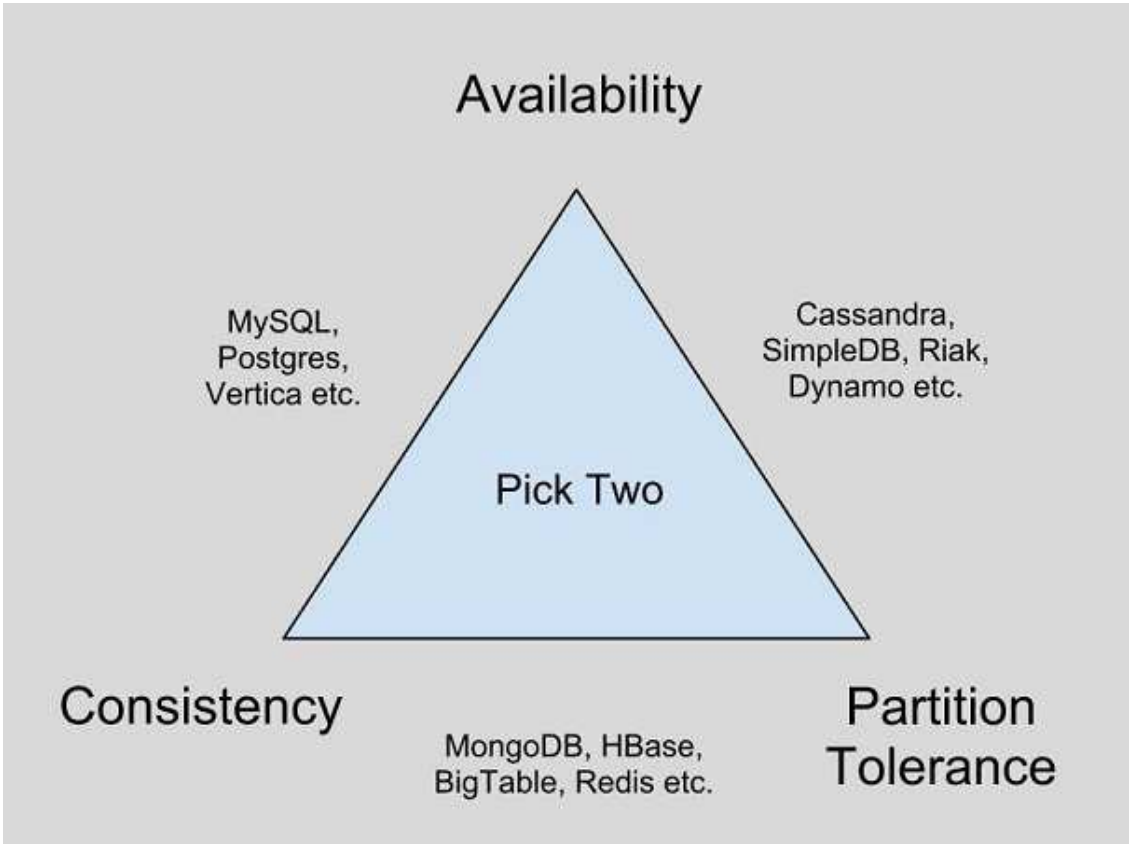


Figure: Synthèse CAP

CAP et architecture distribuée

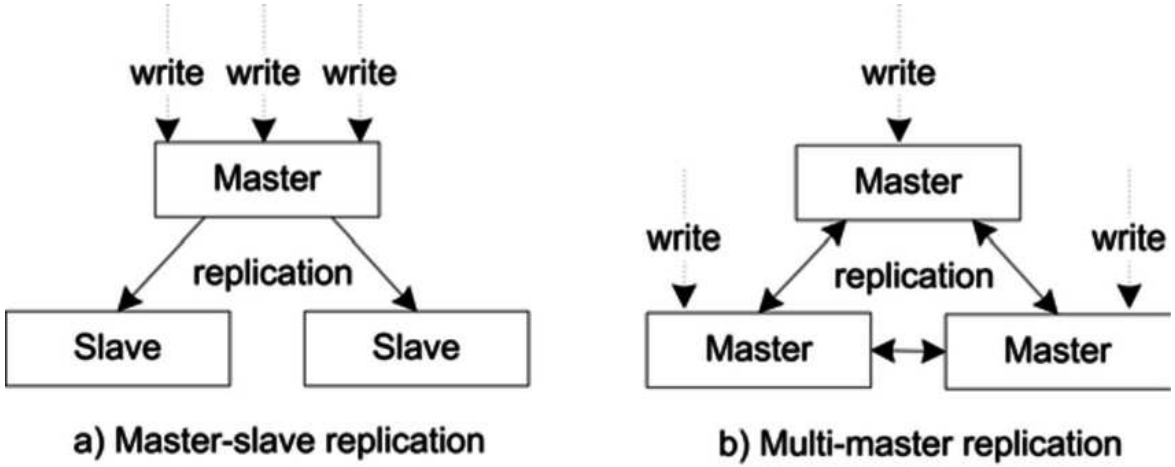


Figure: schémas de distribution

Typologie des systèmes NoSQL

Au regard du mode de représentation choisi

- principe de base : clé/valeur
  - **Systèmes clé/valeur distribués**
  - **Systèmes orientés colonne**
  - **Systèmes orientés document**
- **Systèmes orientés graphe**
- dans une certaine mesure les triplestores et les SGBDOO

Agrégats clé/valeur : unités naturelles pour le distribué

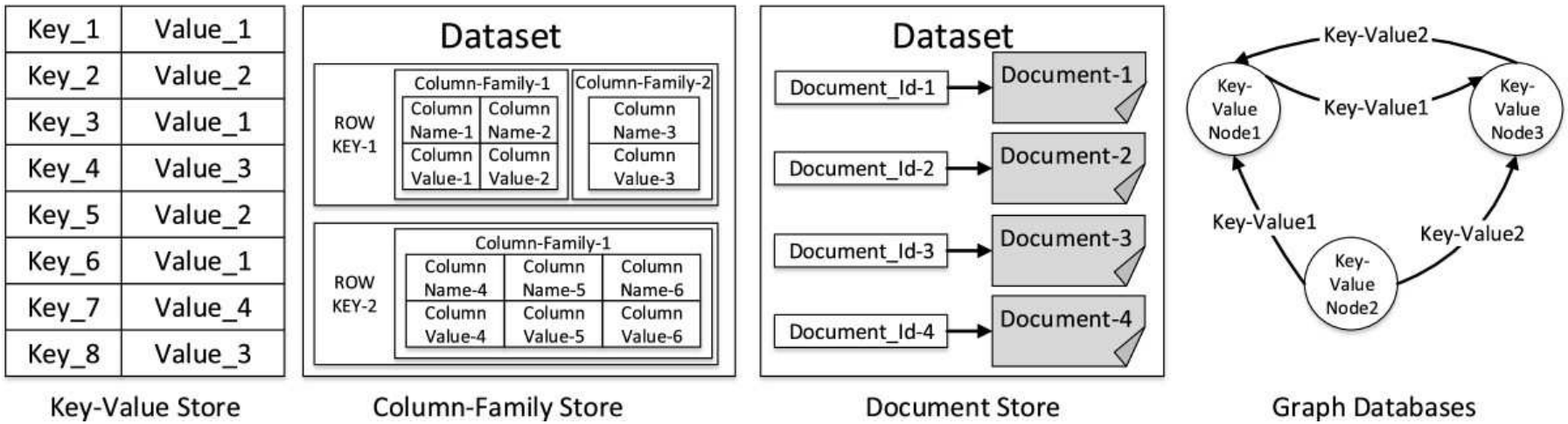


Figure: Extrait de K. Grolinger et al, 2013

Difficulté : absence de standards

Au regard du mode de représentation comme du système choisis

1. **APIs spécifiques**
2. **Terminologies propriétaires**
3. **Mécanismes de requêtage à géométrie variable**
1. **Systèmes ayant fait école ("proofs of concept")**
    1. BigTable
    2. Memcached
    3. Amazon's Dynamo

Systèmes existants

Table: Quelques systèmes et leurs modes de représentation

Name	Mode représentation	CAP
CouchDB	Document	AP
Neo4j	Graph	CA
Hbase	Column	CP
Riak	Key-Value	CP
Project Voldemort	Key-Value	AP
Cassandra	Column	AP
Hypertable	Column	unknown

Systèmes existants

Table: Applications communautaires sur le Web

Name	Système NoSQL	Mode
Google	BigTable, LevelDB	Column
LinkedIn	Voldemort	Key-Value
Facebook	Cassandra	Column
Twitter	Hadoop/Hbase, Cassandra	Column
Netflix	SimpleDB, Hadoop/HBase, Cassandra	Column
CERN	CouchDB	Document
Amazon	Dynamo	Key-Value

Graphes et persistance des données

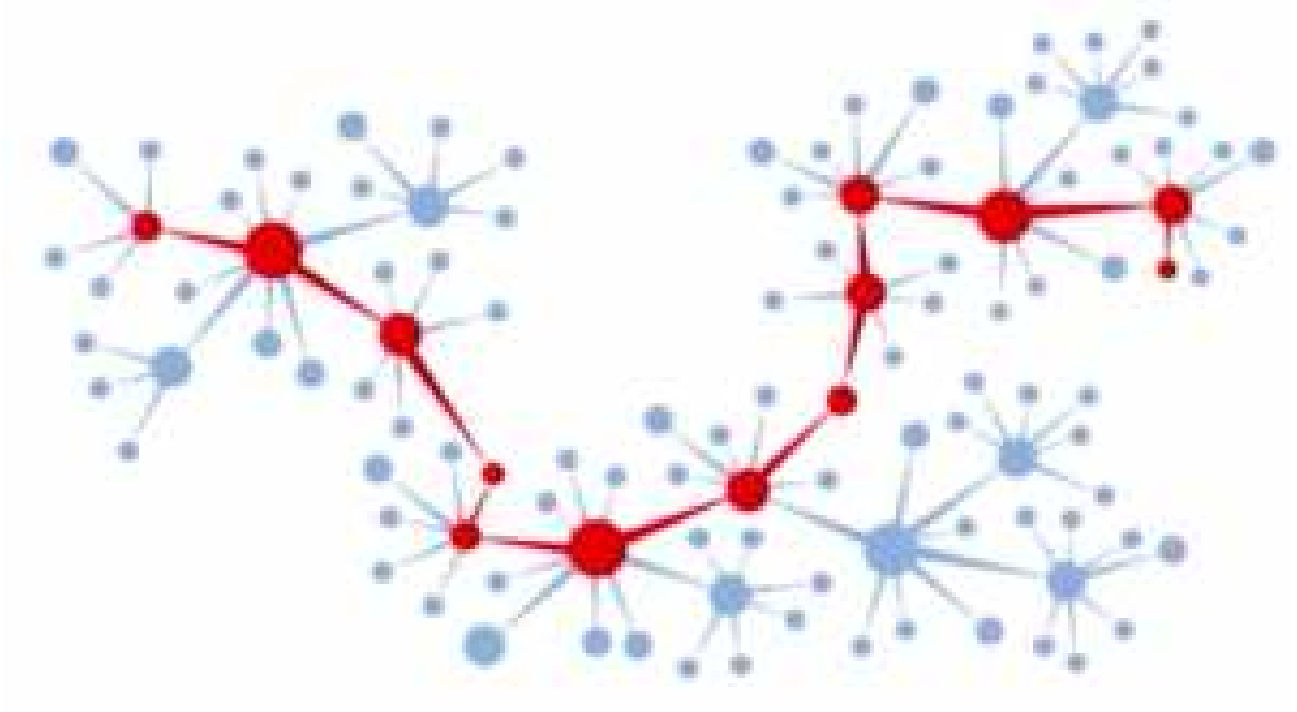


Figure: Illustration graphe, chemin et interaction (doc Neo4J)

Volume de données versus richesse du modèle

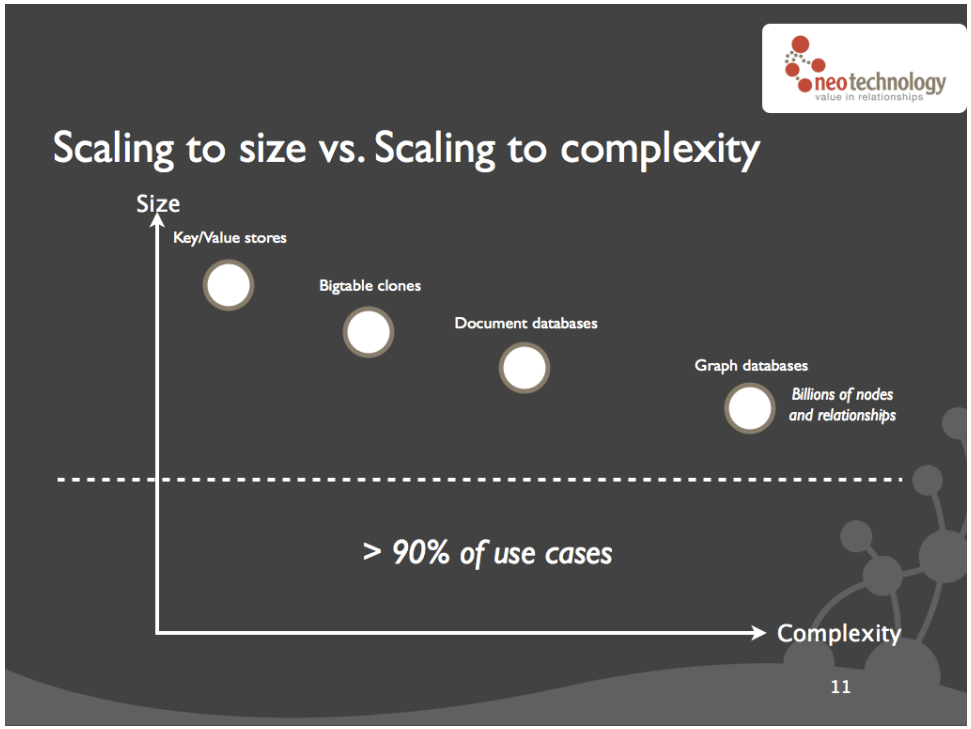


Figure: Une vision très générale (extrait de Neo Technology Webinar)

Adéquation avec le système "mental" humain

Associer et catégoriser : des mécanismes cognitifs[a] naturels

[a] processus psychiques liés à l'esprit

- catégories et associations représentées à l'aide de graphes (que l'on sait traiter efficacement)

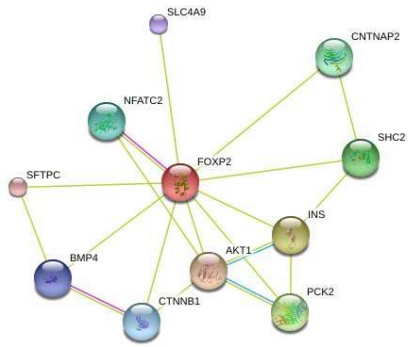


Figure: Gène FoxP2 : implication dans l'acquisition du langage (StringDB)

Le modèle de persistance le plus adapté : une longue histoire

BD : partage et pérennisation de l'information pour différents usages - différents paradigmes[a] de représentations

[a] manières de voir les choses

1960 - système hiérarchique

1960 - système réseau (C. Bachman)

1970 - système relationnel (E.F. Codd)

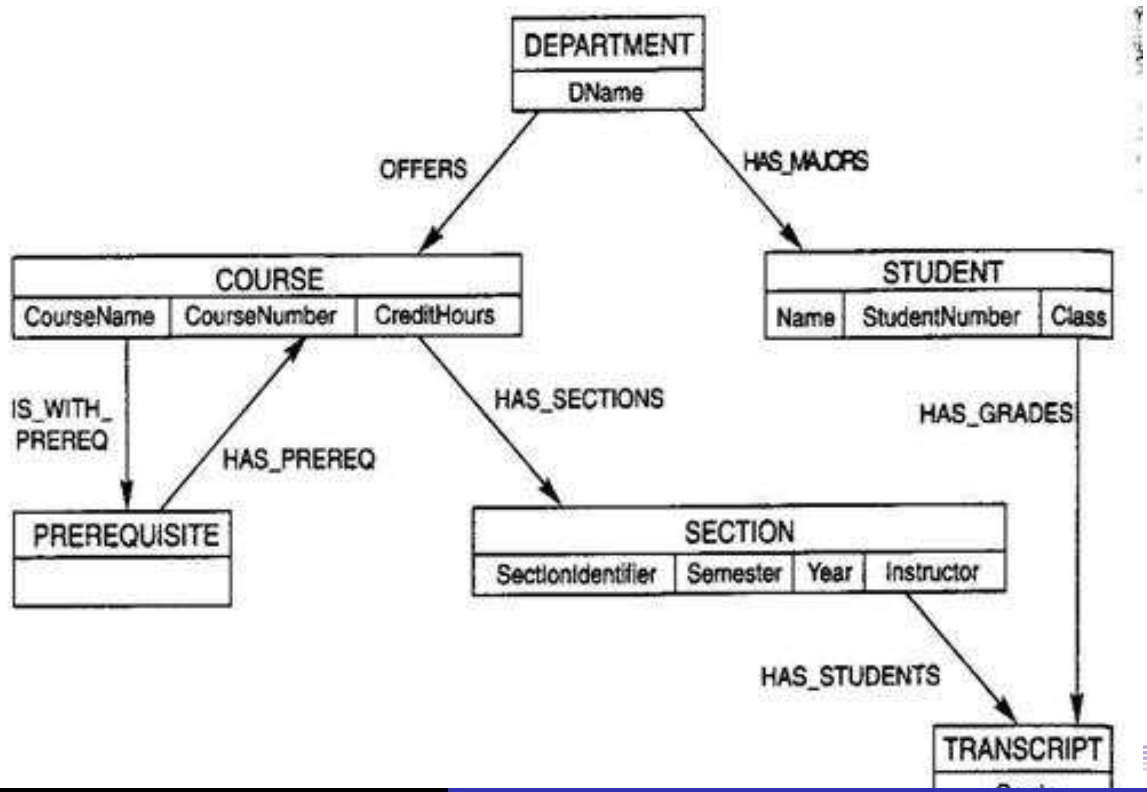
1980 - système objet Objectivity, Objectstore, db4o, Zope Object Database, Caché

1990 - système objet/relationnel Oracle, PostgreSQL

Plus récent - NoSQL regroupant différentes approches dont les systèmes à base de graphe

BD réseau représentée à l'aide d'un graphe des types

Les sommets représentent les types d'articles ; et les arcs les types d'ensembles



BD objet : état + comportement

SQL3 et ODL/OQL (ODMG) : décrire et interroger les BDOO

Listing 1: Un exemple ODL (source tech. ingenieur)

```
class DIPLOMES
tuple (Intitule : string, Cycle : integer,
  Detenu : set (ETUDIANTS) inverse Detient )
end;
class ETUDIANTS
tuple (Numero : integer,
  Nom : string, Prenoms : list(string),
  Detient : set(DIPLOMES),
  Est-inscrit : set( tuple (Mod : MODULES, inverse
      Inscrits Note : real )) )
end;
```

Quelques rappels : théorie des graphes

Eléments de vocabulaire

- graphe G=<V;E> : où V, ensemble des sommets et E, ensemble des arêtes,
- graphe orienté : les arêtes sont des arcs
- sous-graphe G'=<V';E'> de G=<V;E> est un graphe tel que V' $\in$ V et E' $\in$ E
- chemin C entre 2 nœuds v1 et v2 : séquence de nœuds et d'arêtes permettant de rejoindre v2 à partir de v1
- un graphe est dit connecté si il existe un chemin reliant toute paire de nœuds
- un cycle est un chemin fermé (C(vi;vi))
- un arbre est un graphe connecté et acyclique

Quelques rappels : théorie des graphes

De nombreux algorithmes

parcours en largeur ou en profondeur

recherche du plus court chemin (e.g. Dijkstra)

mesures de centralité (e.g. Eigenvector) : mise en avant d'indicateurs structurels

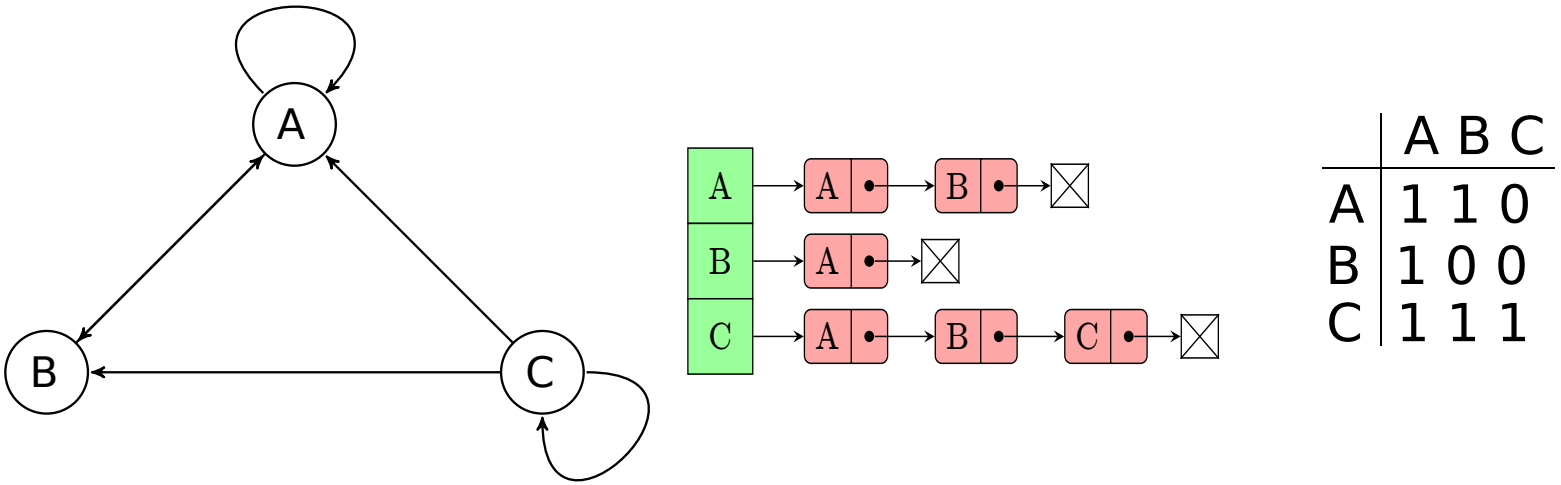
partitionnement

coloration

recherche de composantes connexes

...

Les structures support



	A	B	C
A	1	1	0
B	1	0	0
C	1	1	1

Figure: Liste et matrice d'adjacence

Cas d'utilisation

Tout domaine qui se visualise naturellement sous forme de graphe : système NOSQL connecté (à la différence des systèmes à base d'agrégats)

1. réseaux sociaux
2. réseaux biologiques (cascades signalétiques, voies métaboliques, . . .)
3. réseaux structurant les territoires (géomatique)
4. web de données (LOD), systèmes de recommandation en ligne . . .

Grandes forces

Complexité des données : connectivité + volume + structuration partielle

Atouts des systèmes graphes

1. requêtes topologiques : produits d'expression de gènes interagissant en cascade, amis d'amis d'ennemis, meilleure façon de rallier Paris à partir de Montpellier . . . . . .

Modèle général

Les éléments clés

1. nœuds pour décrire des entités
2. propriétés pour en enrichir la description
3. arcs pour mettre en relation des entités avec d'autres entités ou encore connecter des nœuds avec leurs propriétés
4. patterns : dégager du sens à partir des connexions entre les éléments du graphe

Modèle de graphe plus ou moins riche en fonction du système considéré

Graphe attribué (Property Graph) : le plus souvent exploité

- un ensemble de nœuds souvent typés (LPG : Labeled Property Graph)
  - chaque nœud a un identifiant unique, un ensemble d'arcs entrants et sortants, et possède une collection de propriétés
- un ensemble d'arcs
  - chaque arc a un identifiant unique, une extrémité sortante (queue) et une extrémité entrante (tête), un label indiquant le type de relation entre les deux nœuds, et possède une collection de propriétés (paires clé/valeur)
- ensemble de propriétés : paire clé/valeur définie comme un tableau associatif (valeur : type primitif et tableau de types primitifs)

multi-graphe attribué et orienté : illustration Neo4J

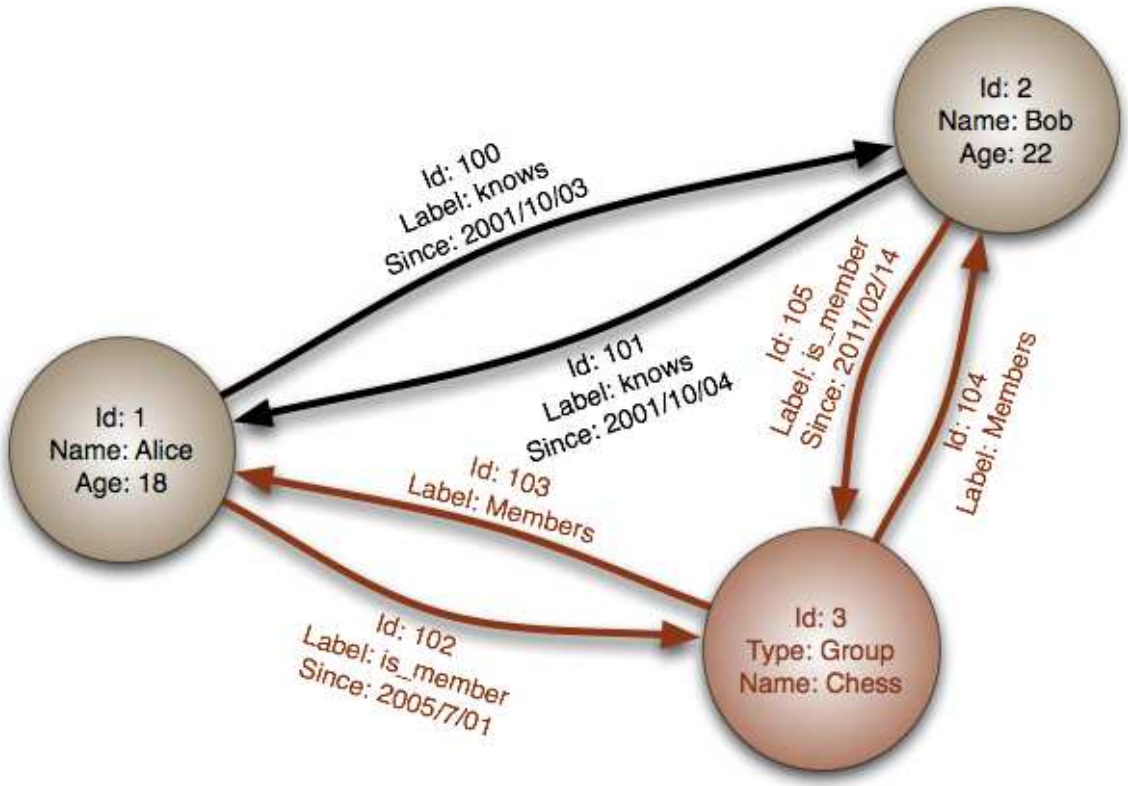


Figure: Illustré (source : Documentation Neo4J)

BD graphes (pas toujours graphe attribué)

Non exhaustif

- Neo4J
- FlockDB (Twitter)
- Pregel (décisionnel)
- InfiniteGraph
- DEX
- OrientDB
- HypergraphDB
- et les solutions adossées à RDF (triplestores) à l'exemple de Stardog ou Sesame

Quelques spécifications Neo4J (le J pour Java)

Différents supports pour l'accès et la manipulation des données

- différents stratégies de parcours de graphes (Traversal Java API)
- langages de requête Gremlin et Cypher (OpenCypher), et un standard en cours de définition GQL (Graph Query Language)
- index pour un accès performant aux nœuds et arcs
- mécanismes transactionnels (ACID)
- architecture "clustérisée" pour version payante (la distribution est un exercice difficile dans les BD graphes)
- pensé pour le web : Java EE (framework Spring et Spring Data), web de données (SAIL et SPARQL), API et interface REST

Schema-less : type d'entité = label et type de relation = type

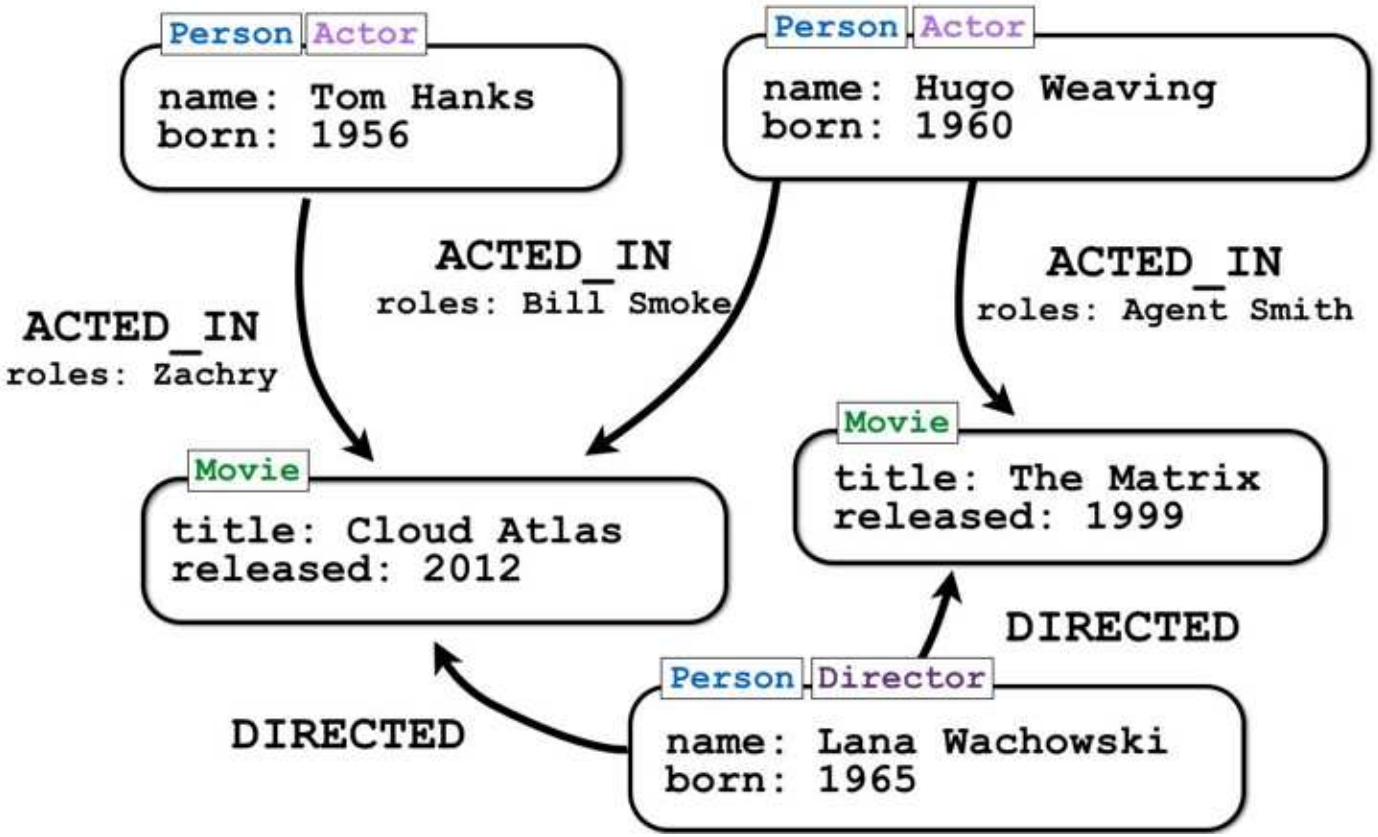


Figure: Illustré (source : Documentation Neo4J)

Jeux de données plus large : OpenStreetMap ou MusicBrainz

Gestion pouvant aller jusqu'à plusieurs milliards de nœuds (2^{32} identifiants possibles)

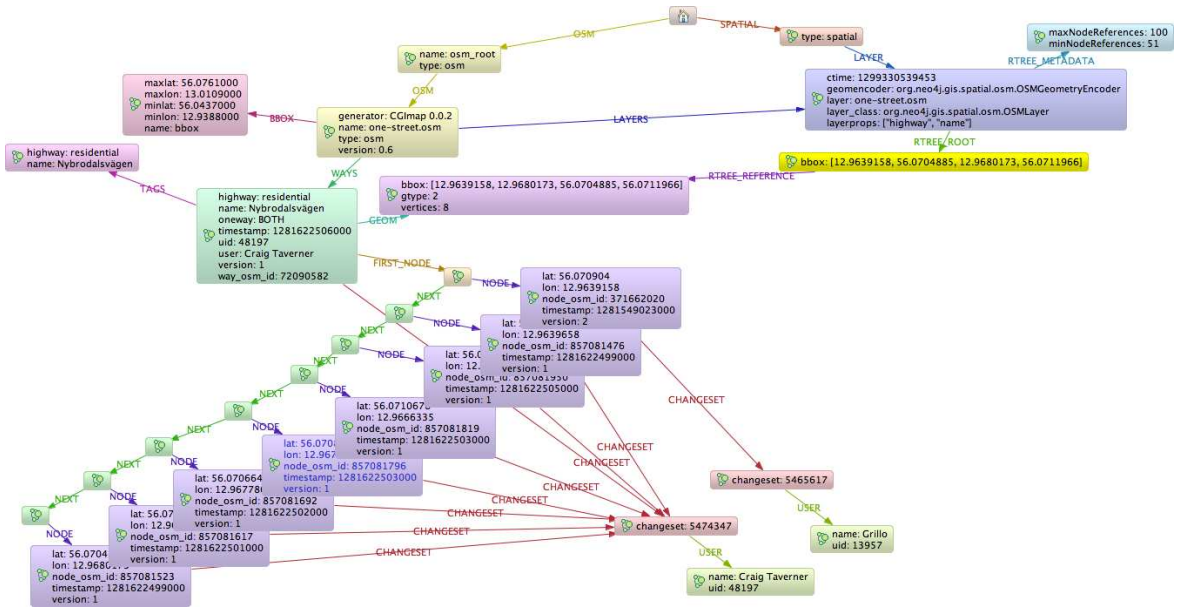
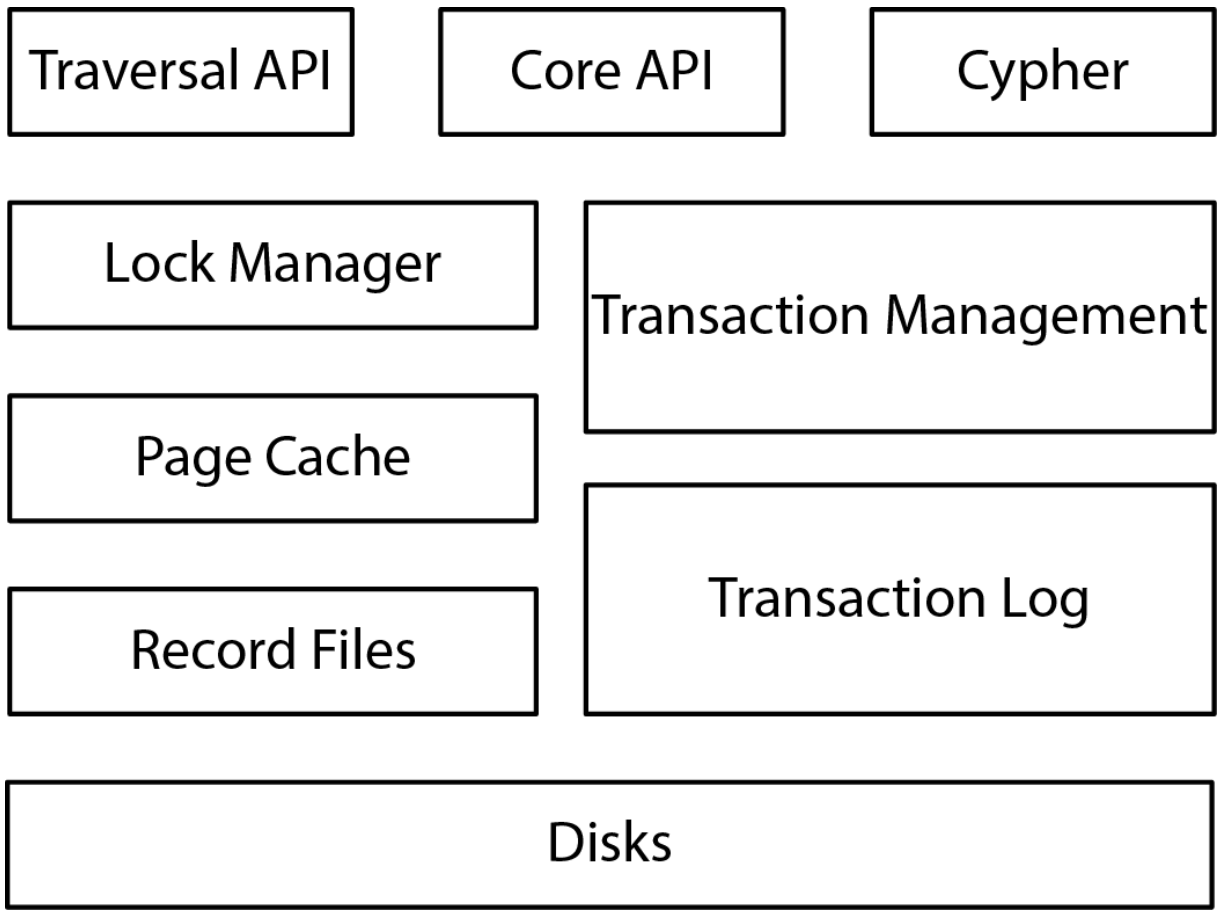


Figure: Un exemple plus complet : cartographie en Norvège avec OSM

Principales briques du système



Modèle de données Neo4J

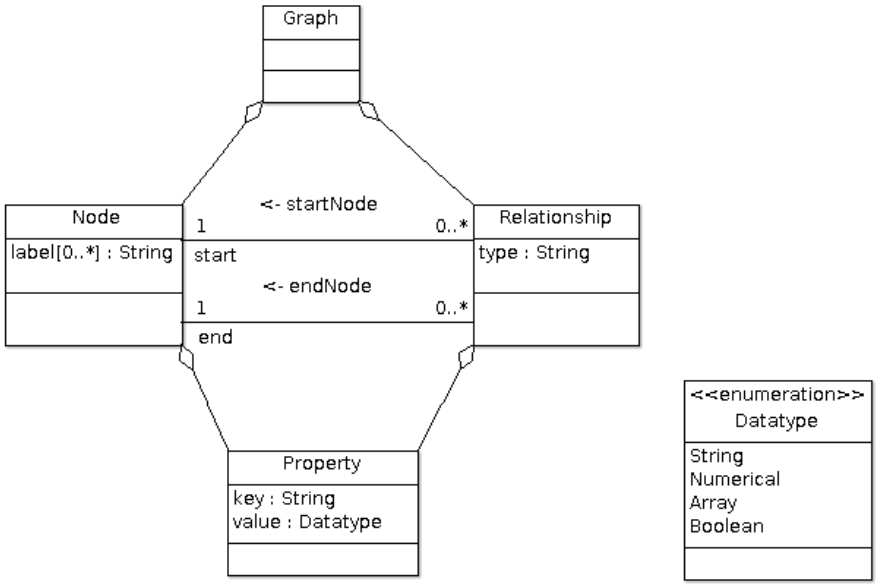


Figure: Diagramme de classes UML illustratif

Cypher : expressions pour poser des filtres sur le graphe

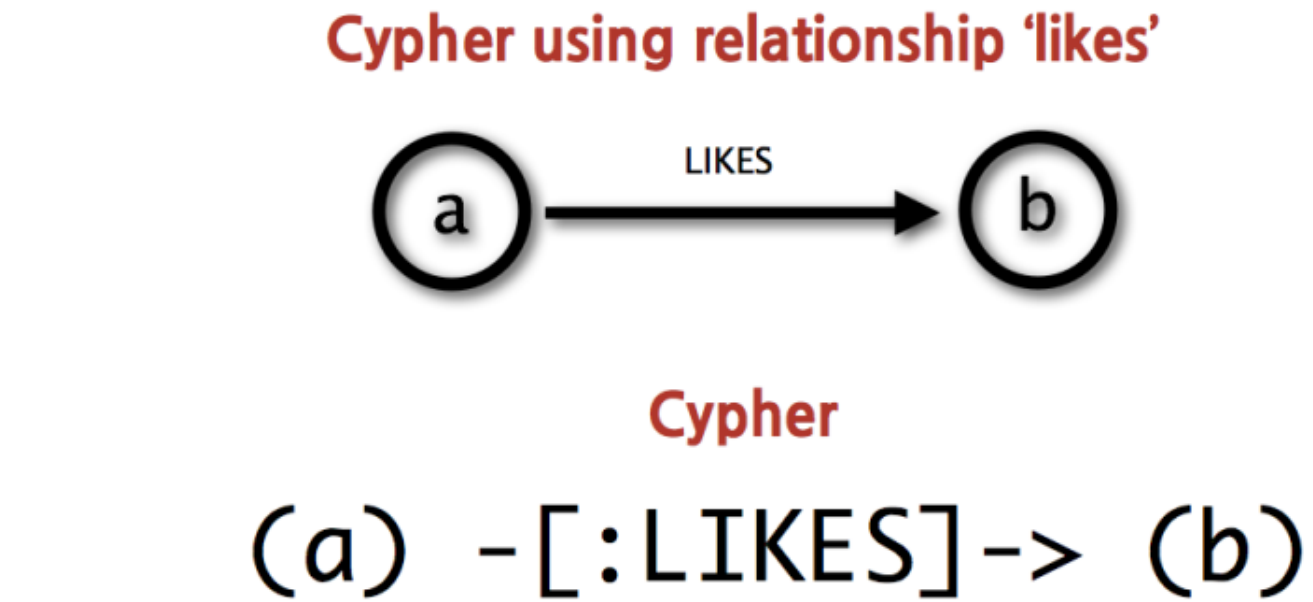


Figure: Illustré (source : Documentation Neo4J)

Cypher : principales clauses

- CREATE - création de nœuds et d'arêtes
- DELETE - REMOVE : suppression de nœuds, d'arêtes et de propriétés
- SET - mise à jour de valeurs de propriétés
- MATCH - rechercher des points d'entrée dans le graphe
- MERGE - combinaison de MATCH et CREATE
- WHERE - poser des sélections
- RETURN - nœuds et arêtes à retourner
- UNION - combiner les résultats de plusieurs requêtes
- WITH - sorte de pipe en commande Unix . . .

Exemple Cypher

Ordres de création

```
CREATE (m:Commune:Ville {nom:'MONTPELLIER',
    latitude:43.610769, longitude:3.876716,
    codeinsee:'34172'})
RETURN m

CREATE (m:Commune {nom:'NIMES', codeinsee:'30189'})
    -[:WITHIN]-> (h:Departement {nom:'GARD',
    numero:'30'})
```

Listing 2: CREATE

Exemple Cypher

Clauses dans une grammaire déclarative à rapprocher de SQL : MATCH, WHERE, RETURN

```
MATCH (d:Departement {nom:'GARD'}) <-[p:WITHIN]-
    (n:Commune)
RETURN d, n, p

MATCH (d:Departement) <-[p:WITHIN]- (n:Commune)
WHERE d.nom = 'GARD'
RETURN d.nom, n.nom
```

Listing 3: MATCH

Exemples génériques Cypher

Listing 4: infos sur le "schema"

```
match n
return distinct labels(n)

match n-[r]-()
return distinct type(r)

match n-[r]-()
return distinct labels(n), type(r)

MATCH ()-[r]->()
RETURN TYPE(r) AS rel_type, count(*) AS rel_cardinality
```

Exemple de partitionnement

Listing 5: Compter les communes

```
MATCH (:Commune)-[:WITHIN]->(d:Departement)
WITH d, count(*) as nC
WHERE nC > 8
RETURN d.nom as dep, nC as communes
```

La clause MERGE

Ne créer que ce qui n'existe pas et éviter les doublons (à la différence de CREATE)

Listing 6: Usage de la clause MERGE

```
-- Alicante existe : juste ajout du pays - avec create
    : doublon
MERGE (c:City {name:'Alicante'}) SET c.pays='Spain'
    RETURN c
-- Lunel n'existe pas : ajout de l'ensemble du noeud
MERGE (c:City {name:'Lunel'}) SET c.pays='France'
    RETURN c
```

Une force : les appels récursifs

Listing 7: parcourir le graphe

```
(A)->()->()->()->(B)
(A)-[*]->(B)

MATCH (c1:Commune)-[:NEARBY]->()<-[:NEARBY]-(c2:Commune)
RETURN c1, c2

MATCH (m:Commune {nom:'MONTPELLIER'}) -[:NEARBY*]-
    (n:Commune)
RETURN m, n

MATCH (m:Commune {nom:'MONTPELLIER'}) -[:NEARBY*1..2]-
    (n:Commune)
RETURN m, n
```

Cypher : requête de navigation

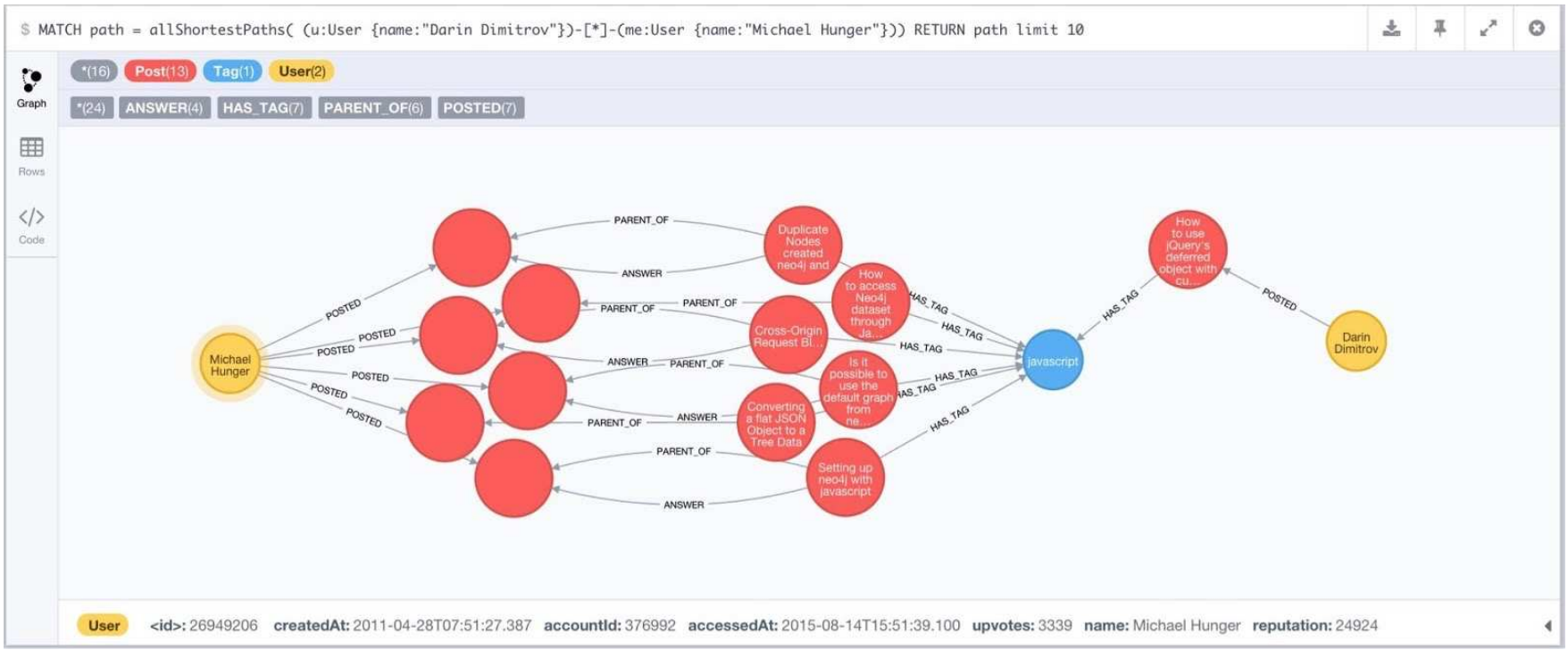


Figure: Recherche des chemins les plus courts

Exemple Cypher

Nouvelle consultation sur les chemins

Listing 8: taille du/des chemin(s) le(s) plus court(s) entre MONTPELLIER et NIMES

```
MATCH p=shortestPath(
    (m:Commune)-[:NEARBY*]-(n:Commune) )
WHERE m.nom='MONTPELLIER' and n.nom = 'NIMES'
RETURN length(p) as taillePLusCourtChemin
```

Exemple Cypher

Détailler les nœuds du chemin

Listing 9: Donner les communes entre MONTPELLIER et NIMES

```
MATCH p=shortestPath(
    (m:Commune)-[:NEARBY*]-(n:Commune) )
WHERE m.name='MONTPELLIER' and n.name = 'NIMES'
RETURN extract (n in nodes(p) | n.name) as
    communesSurLeChemin
```

Exemple Cypher

Donner le nom et le code insee des communes présentes dans les plus courts chemins entre MONTPELLIER et NIMES

Listing 10: Donner les communes entre MONTPELLIER et NIMES

```
MATCH p=allshortestPaths(
    (m:Commune)-[:NEARBY*]-(n:Commune) )
WHERE m.name='MONTPELLIER' and n.name = 'NIMES'
RETURN extract (n in nodes(p) |
    {nom:n.name,code:n.codeinsee}) as communesSurLeChemin
```

Exemple Cypher

Demander à ce que le chemin ne passe pas par un nœud

Listing 11: Plus court chemin entre MONTPELLIER et NIMES sans Lunel

```
MATCH p=shortestPath( (m:Commune
    {name:'MONTPELLIER'})-[:NEARBY*]-(n:Commune
    {name:'NIMES'}) )
where not ('LUNEL' in (extract (n in nodes(p)|
    n.name))) return p
```

Exemple Cypher

Autres opérations CRUD

Listing 12: Suppression dans la base

```
MATCH (n)
OPTIONAL MATCH (n)-[r]-()
DELETE n,r

MATCH (n)
DETACH DELETE n

MATCH(m:Personne)
REMOVE m:Acteur
RETURN m
```

Exemple Cypher

Consulter à partir des identifiants

Listing 13: Fonction ID

```
MATCH (s) WHERE ID(s) = 245 RETURN s

MATCH (n:Commune) where ID(n) >=20 RETURN n
```

Exemple Cypher

Exemples autour de la négation

Listing 14: Usage du NOT

```
MATCH (i:Commune)
WHERE NOT (i) -[:NEARBY]-> (:Commune)
RETURN i

MATCH (i:Commune)
WHERE SIZE((i) -[:NEARBY]-> (:Commune)) = 0
RETURN i
```

Exemple d'utilisation d'index

Performances d'accès : définir un index sur un ou plusieurs attributs (BTree)

Listing 15: index non unique

```
CREATE INDEX ON :City(name)
DROP INDEX ON :City(name)
```

Listing 16: index unique

```
CREATE CONSTRAINT ON (c:City) ASSERT c.name IS UNIQUE
DROP CONSTRAINT ON (c:City) ASSERT c.name IS UNIQUE
```