INTERROGATION DE DONNÉES DE TYPE GRAPHE

BDLE (BASES DE DONNÉES LARGE ECHELLE)
CAMELIA CONSTANTIN -- PRÉNOM.NOM@LIP6.FR

LANGAGES DE REQUÊTES GRAPHES

EXEMPLES DE REQUÊTES

- Dans un réseau de type transport, alimentation, communication
  - Comment aller de l'adresse a à l'adresse b ?
  - Combien de chemins entre le nœud réseau a et le nœud réseau b ?
  - Le chemin le plus rapide entre l'usine a et le magasin b ?
- Dans un réseau de représentation de connaissance:
  - Une classe (élément) A est elle un sous-classe d'une classe B?
  - Existe-t'il un lien entre l'entité A et l'entité B?
  - Similarité entre l'entité A et l'entité B basé sur le graphe sémantique
- Réseaux de citations
  - Quels auteurs sont le plus cités directement et indirectement ?
  - Quels chercheurs / articles sont important dans un domaine (centralité) ?
- Réseaux sociaux
  - You Might Also Know de Facebook: si on partage beaucoup d'amis, on doit sans doute se connaître
  - Recommandation de lieux dans FourSquare d'après avis des amis: si des amis recommandent un lieu, alors de bonne chance que j'aime aussi
  - Degré de séparation (ex : nombre d'utilisateurs entre deux utilisateurs sur Facebook)

EXEMPLES DE REQUÊTES (SUITE)

- Les voisins de A
- Voisins en commun entre A et B
- Existence / nombre de chemins entre A et B (et nombre, longueur, coût, etc.)
- Existence de chemins entre A et B correspondant à une expression régulière (A B)*(C|D)+
- Recherche du plus court chemin entre A et B, entre tous les nœuds
- Recherche de motifs de graphes : cycles, "motifs de graphes", arbre couvrant, circuit hamiltonien
- Calcul de propriétés : diamètre du graphe, centralité, ..

LANGAGES DE REQUÊTES GRAPHE

- SQL2 : PL/SQL avec boucles et condition d'arrêt suivant la requête (pas de suivant, profondeur voulue, etc.)
- Requêtes hiérarchiques (clause CONNECT BY), Oracle7
- SQL3 : Requêtes récursives (clause WITH), Oracle 11gR2
- DATALOG : modèle théorique base sur les clauses de Horn, des implantations mais pas de produits véritables
- SPARQL : équivalent à SQL pour des données RDF (triplets : sujet, prédicat, objet = arrête étiquetée dans un graphe)
- Cypher : pour la BD graphe Neo4j
- ….....

SQL : EXEMPLE DE REQUÊTE

- Trouver les voisins directs :
  - Le nom des utilisateurs qui suivent 'Marc'?

compte	nom	email	…
N1	Jean	…	
N2	Lucie	…	
N3	Marc	…	

Table Node

follower	followee
N1	N2
N1	N3
N2	N1
N2	N3
…	…

Table Edge

TROUVER LES VOISINS DIRECTS

compte	nom	email
N1	Jean	…
N2	Lucie	…
N3	Marc	…

- Nom des utilisateurs qui suivent 'Marc'?

select B.nom
from node A, node B, edge E
where A.nom='Marc' **and** A.compte=E.followee
and E.follower=B.compte

- Liste des nœuds atteignables depuis le compte de 'Marc' ?
  - Implique d'explorer le graphe depuis un nœud donné avec une profondeur d'exploration fixée

follower	followee
N1	N2
N1	N3
N2	N1
N2	N3
…	…

SQL 3 : REQUÊTES HIÉRARCHIQUES

SELECT select_list
FROM table_expression
[WHERE ...]
[START WITH start_expression]
CONNECT BY [NOCYCLE] { PRIOR parent_expr = child_expr |
child_expr = PRIOR parent_expr }
[ORDER SIBLINGS BY column1 [ASC | DESC] [, column2 [ASC | DESC]] ...
[GROUP BY ...]
[HAVING ...]

REQUÊTES HIÉRARCHIQUES

- START WITH indique le noeud de départ
- CONNECT BY PRIOR : règle de connexion entre les nœuds, spécifie la relation entre les tuples parent/enfant dans la hiérarchie.
- WHERE supprime les tuples de la hiérarchie qui ne satisfont pas la condition (on n'arrête pas la récursion)
- LEVEL : attribut permettant de retourner la profondeur du nœud par rapport à la racine
- NOCYCLE : ne retourne pas un message d'erreur si un cycle est rencontré
- SYS_CONNECT_BY_PATH : permet de construire le chemin depuis la racine
- CONNECT_BY_ROOT : utiliser le nœud racine dans une condition

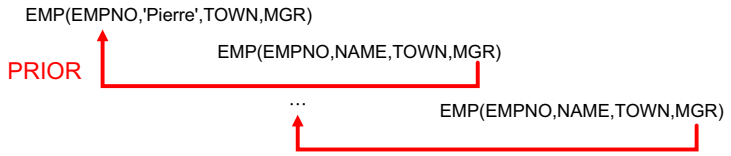
EXEMPLE DE REQUÊTES HIÉRARCHIQUES

Tous les subordonnés parisiens de Pierre?

```
select NAME, LEVEL
from EMP E
where E.TOWN='Paris'
start with E.NAME='Pierre'
connect by E.MGR = prior E.EMPNO;
```

Tous les supérieurs parisiens de Pierre?

```
select NAME,LEVEL
from EMP E
where E.TOWN='Paris'
start with E.NAME='Pierre'
connect by prior E.MGR = E.EMPNO;
```



SQL3 : FACTORISATION AVEC WITH

```
WITH [ RECURSIVE ]
    <vue V1> [ ( <liste_colonne1> ) ] AS [( <requête SQL Q1> )],
    <vue V2> [ ( <liste_colonne2> ) ] AS [( <requête SQL Q2> )], …
<requête SQL Q> ;
```

- Oracle: on ne met pas le mot clé RECURSIVE
- Requête Qi peut utiliser toutes les vues Vj pour j < i
- Récursion : Qi utilise la vue Vi (avec UNION ALL)
  - Qi: **select** … **from** (<requête>) **union all (**select … **from** Vi, …)

EXEMPLE RÉCURSION SQL3

EMP(ENO,NAME,TOWN,MGR)

Tous les subordonnés parisiens de Pierre?

```
WITH subordonne(SUBENO, SUBENAME, TOWN)
AS (SELECT DISTINCT sub.ENO, sub.NAME, sub.TOWN
    FROM EMP sup, EMP sub WHERE sup.NAME='Pierre'
                          AND sup.ENO = sub.MGR
    UNION ALL
    SELECT sub.ENO, sub.NAME, sub.TOWN
    FROM subordonne sup, EMP sub WHERE sup.ENO=sub.MGR)
SELECT * FROM subordonne where TOWN = 'Paris';
```

EXEMPLE RÉCURSION SQL3

EMP(ENO,NAME,TOWN,MGR)

Tous les supérieurs parisiens de Pierre?

```
WITH supérieur(SUPENO, SUPENAME,TOWN)
AS (SELECT DISTINCT sup.ENO, sup.NAME, sup.TOWN
    FROM EMP sup, EMP sub WHERE sub.NAME='Pierre'
                          AND sup.ENO = sub.MGR
    UNION ALL
    SELECT sup.ENO, sup.NAME, sup.TOWN
    FROM supérieur sub, EMP sup WHERE sup.ENO=sub.MGR)
SELECT * FROM supérieur where TOWN = 'Paris';
```

RÉCURSION EN PROFONDEUR OU EN LARGEUR

- Contrôler l'odre d'évaluation :

  SEARCH DEPTH | BREADTH FIRST BY <attr> **SET** <ordre>

```
WITH subordonne(SUBENO, SUBENAME, TOWN)
AS
(
    SELECT DISTINCT sub.ENO, sub.NAME, sub.TOWN
    FROM EMP sup, EMP sub WHERE sup.NAME='Pierre'
                          AND sup.ENO = sub.MGR
    UNION ALL
    SELECT sub.ENO, sub.NAME, sub.TOWN
    FROM subordonne sup, EMP sub WHERE sup.ENO=sub.MGR
)
SEARCH DEPTH FIRST BY SUBENO SET order1
SELECT * FROM subordonne where TOWN = 'Paris'
ORDER BY order1;
```

RÉCURSION ET CYCLE

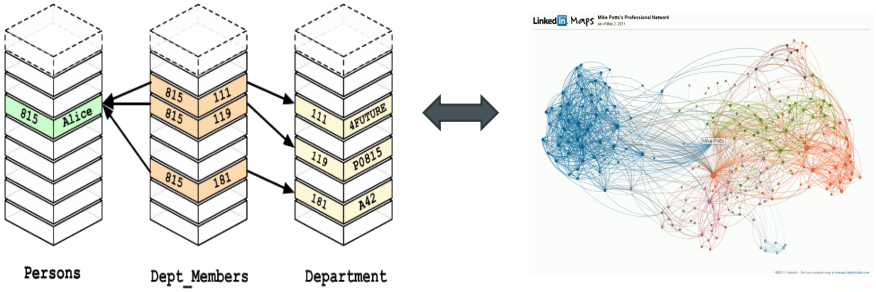
- Détection de cycles :

  CYCLE <attr1> **SET** <attr2> **TO** <val1> **DEFAULT** <val2>

- attr2 = val1 quand un cycle est détecté sur l'attribute attr1 (attre = val2 sinon)
- la récursion s'arrête pour la ligne où le cycle est détécté

```
WITH subordonne(SUBENO, SUBENAME, TOWN)
AS (
    SELECT DISTINCT sub.ENO, sub.NAME, sub.TOWN
    FROM EMP sup, EMP sub WHERE sup.NAME='Pierre' AND sup.ENO = sub.MGR
  UNION ALL
    SELECT sub.ENO, sub.NAME, sub.TOWN
    FROM subordonne sup, EMP sub WHERE sup.ENO=sub.MGR
)
CYCLE SUBENO SET cyc TO 1 DEFAULT 0
SELECT * FROM subordonne where TOWN = 'Paris';
```

LIMITES DES BD RELATIONNELLES

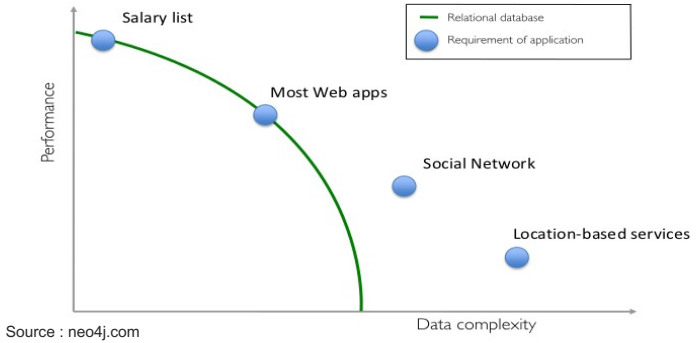


Source : neo4j.com

LIMITES DES BD RELATIONNELLES

- Modèle :
  - Le modèle relationnel est bien adapté pour des données structurées qui peuvent facilement être organisés en plusieurs tables
  - Un graphe est une structuration libre de nœuds divers connectés par des liens
  - Liens entre les noeuds modélisées par des tables supplémentaires → modifier le schéma pour ajouter de nouveaux types de données et de relations
- Requêtes
  - Les requêtes SQL récursives sont complexes, expliciter les liens avec joins
  - Les optimiseurs et structures d'index relationnels ne sont pas adaptés à l'évaluation de requêtes graphes => performances dégradées quand il existe beaucoup de relations

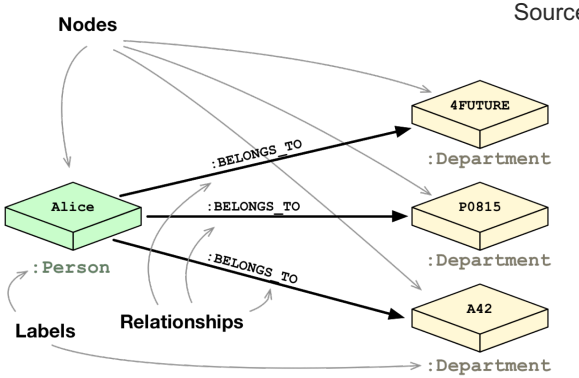
PERFORMANCE DES BD RELATIONNELLES



Source : neo4j.com

Alternative : BD ORIENTÉES GRAPHE

Source : neo4j.com



Remplacer les références logiques clés/clés étrangères par des pointeurs physiques → jointure = suivi de pointeurs

BD ORIENTÉES GRAPHES

- Organisées selon des modèle complexes et flexibles
- Stockage optimisé pour des structures fortement connexes (stockage dédié aux nœuds et aux arcs) et pour la lecture et le parcours du graphe
- Utilisation des algorithmes graphe optimisés avec des API intégrées
  - plus court chemin, centralité, etc
- Requêtes :
  - en temps linéaire (exponentiel pour SQL) lorsque le volume/connectivité/profondeur de parcours de données augmentent
  - traversent le graphe efficacement
  - adjacence entre les éléments voisins sans indexation : pointers physiques (enregistrés comme part entière de la donnée), évitent les jointures coûteuses

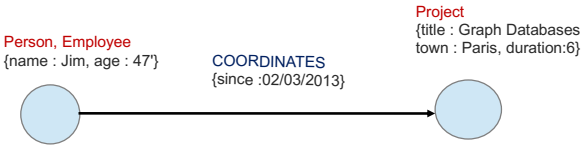
BD ORIENTÉES GRAPHES

Exemples de BD orientées graphe :

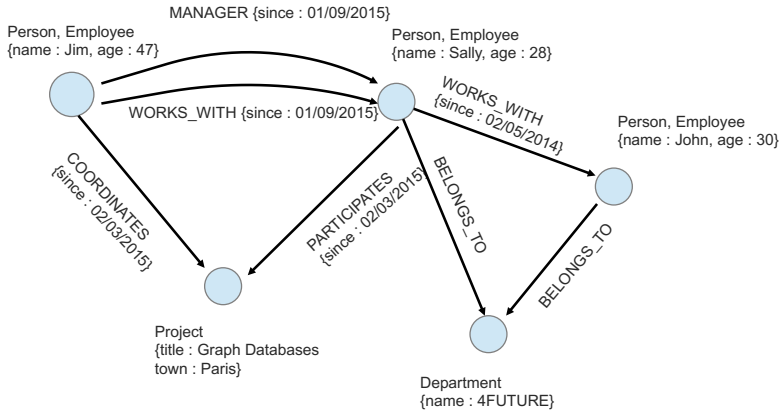
- AllegroGraph, ArrangoDB, GraphEngine, Sones, HypergraphDB
- Neo4j :
  - BD transactionnelle / ACID (Atomicité, Cohérence, Isolation, Durabilité)
  - haute disponibilité (mise ne place d'un cluster)
  - scalabilité : stocker et interroger des milliards de noeuds et de relations
  - utilisateurs : Viadeo, ebay, National Geographic, Cisco, Adobe, Meetic, SFR, Voyages-SNCF, etc

NEO4J : GRAPHE DE PROPRIÉTÉS

- Noeuds :
  - étiquettes pour différencier les nœuds
    - donnent un rôle/type à un nœud, plusieurs étiquettes possibles pour un nœuds
  - propriétés clef/valeur
- Relations :
  - nœud de départ, nœud d'arrivée
  - type de relation
  - propriétés clef/valeur



NEO4J : GRAPHE DE PROPRIÉTÉS



NEO4J : CYPHER

- Langage de requêtes déclaratif (inspiré de SQL)
- Motifs de chemins / graphes

Syntaxe :

- spécification de nœuds : (p : Person), (p : Person:Employee), (p:Person {name:Jim}), ()
- spécification de relations : (u) → (v), (u) -[r] → (v),

  (u) – [r : MANAGER|WORKS_WITH] → (v),

  (u) - [:COORDINATES {since:"02/03/2015"}] ->(p)
- motifs de chemins :
  - (u)-->(z)<--(v), (u)-->()<--(v), (u)--(v)
  - (u) - [*2] → (v) équivalent à (u) → () → (v)
  - (u) - [*3..5] → (v) : longueur entre 3 et 5 (relations)
  - (u) - [*3..] → (v) : chemin de longueur minimum 3
  - (u) - [*..5] → (v) : chemin de longueur maximum 5
  - (u) - [*] → (v) : n'importe quelle longueur

REQUÊTES CYPHER

- Interrogation des données :
  - MATCH (m) : retourne les instances (bindings de variables) de motif m
  - WHERE : prédicats pour filtrer les résultats
  - RETURN [DISTINCT]: formatage des résultats sous la forme demandée:
    - valeurs scalaires, éléments de graph, chemins, collections ou même documents.
    - DISTINCT : élimination de doublons
  - LIMIT : restriction sur le nombre de résultats retournés
  - ORDER BY [DESC] : ordonnancement

REQUÊTES CYPHER

- Modification des données :
  - CREATE (DELETE) : créer (effacer) des nœuds et des relations
  - MERGE (m) : cherche le motif m en le créant s'il n'existe pas (MATCH+CREATE)
  - SET (REMOVE) : ajoute (enlève) des propriétés et étiquettes

Exemple CREATE/DELETE

- Créer un nouveau nœud :

```
CREATE (jim:Person  {name: 'Jim', age: 47})
```

  - () : spécification de nœud
  - jim : nom de variable pour le nœud, 'Person' son étiquette/rôle
  - {} : spécification des propriétés (couples clé:valeur)

- Effacer le nœud (et ses éventuelles relations) :

```
MATCH (n) DETACH DELETE n
```

  - DETACH : pour effacer des nœuds avec des relations attachées

Requête

- Voir le nœud créé :

```
MATCH (n:Person) WHERE n.name='Jim' RETURN n
```

- MATCH : cherche un motif (un nœud avec l'étiquette 'Person' désigné par la suite par la variable n)
- WHERE : condition supplémentaire sur la propriété name (peut être spécifiée aussi dans le motif)
- RETURN : retourne le nœud avec toutes ses propriétés et étiquettes

Exemple SET

- Ajouter une étiquette supplémentaire 'Employee'

```
MATCH (n:Person {name:'Jim'})
SET n:Employee RETURN n.name, labels(n) as labels
```

- Ajouter une propriété supplémentaire (la modifier si elle existe) :

```
MATCH (n:Person {name:'Jim'})
SET n.job='programmer' RETURN properties(n)
```

- Pour les enlever : REMOVE

```
MATCH(n) WHERE NOT (EXISTS(n.emploi))
SET n.emploi=n.job REMOVE n.job RETURN properties(n)
```

  - Alternative : … WHERE n.emploi IS NULL

EXEMPLES DE REQUÊTES

- Tous les nœuds du graphe

```
MATCH(n) RETURN n
```

- Tous les nœuds qui on une relation avec un autre noeud

```
MATCH(n) → () RETURN n
```

- Toutes les relations de John (indépendamment de la direction)

```
MATCH (john {name :"John"}) – [r ] - ()
RETURN TYPE(r)
```

- Tous les noms d'employés avec les noms de leur département :

```
MATCH (e: Employee) → (d: Department)
RETURN e.name, d.name
```

EXEMPLES DE REQUÊTES

- Trouver Jim et ses subordonnés :

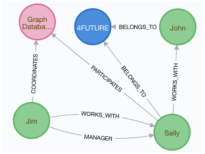
```
MATCH (jim :Person {name : 'Jim'}) - [:MANAGER]->(sub)
RETURN jim, sub
```

- Depuis quand participe Sally au projet "Graph Databases" ?

```
MATCH (sally:Employee { name : 'Sally' })
MATCH (projet:Project  { title :'Graph Databases' })
MATCH (sally)-[r:PARTICIPATES] → (projet)
RETURN r.since
```

- Tous les projets à Paris dans lesquels a travaillé Sally

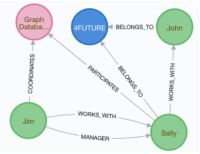
```
MATCH (sally:Employee)-[:PARTICIPATES]->(project)
WHERE sally.name="Sally"   AND project.town ="Paris"
RETURN project.title
```



EXEMPLES DE REQUÊTES

- Pour chaque employé le nom des chefs des projets dans lesquels il travaille :

```
MATCH (e)-[:PARTICIPATES]->(p)<-[:COORDINATES]-(c)
RETURN e.name, c.name
```



- Qui est le plus âgé parmi Jim et Sally ?

```
MATCH(p : Person)
WHERE p.name='Jim' OR p.name='Sally'
RETURN p.name as oldest
ORDER BY p.age DESC  LIMIT 1
```

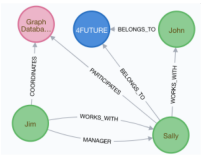
- Les 5 personnes les plus âgées :

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.age DESC LIMIT 5
```

EXEMPLES DE REQUÊTES

- *Les projets dans lesquels travaille Sally et dans lesquels ne travaille pas John*

```
MATCH (sally:Person {name:"Sally"})-[:PARTICIPATES]->(project)
MATCH (john:Person {name:"John"})
WHERE NOT (john)-[:PARTICIPATES]->(project)
RETURN DISTINCT project.title
```



- Utilisation COUNT :
  - Nombre total de nœuds du graphe :

```
MATCH(n) RETURN COUNT(n)
```

  - Nombre total de relations dans le graphe :

```
MATCH () → () RETURN COUNT(*)
```

Agrégation

- *Les 10 employés qui ont travaillé dans le plus de projets :*

```
MATCH (e:Employee)-[:PARTICIPATES]->(p)
RETURN e.name, count(p)
ORDER BY count(p) DESC  LIMIT 10
```

→ Autres fonctions d'agrégation : min, max, sum, collect

- Agrégation :
  - Similaire au GROUP BY en SQL
  - e.name présent dans RETURN → utilisé pour partitionner les données, count sera appliqué sur chaque partition
  - Utilisation DISTINCT (count (DISTINCT p)) pour compter uniquement les valeurs uniques
- Pour chaque nœud ses étiquettes et le nombre total de nœuds auquel il est connecté

```
MATCH (n) -- ()  RETURN n, labels(n), count(*)
```

- Pour chaque type (étiquette) de relation le nombre de relations de ce type

```
MATCH () -[r] ->() RETURN type(r), count(*)
```

EXEMPLES DE REQUÊTES

- *Le collègue de Jim et le collègue de son collègue*

```
MATCH (jim)-[:WORKS_WITH*1..2]-(foaf)
WHERE jim.name="Jim"
RETURN foaf.name
```

- *Les employés qui travaillent avec les employés qui travaillent avec Jim et avec lesquels Jim n'a jamais travaillé:*

```
MATCH (fof)-[:WORKS_WITH*2]-(jim:Person)
WHERE jim.name = "Jim" and NOT (jim)-[:WORKS_WITH]-(fof)
RETURN DISTINCT fof.name
```

Plus court chemin

- *La longueur du plus court chemin entre "Jim" et "John" :*

```
MATCH p=shortestPath( (jim)-[*1..10]-(john) )
WHERE jim.name="Jim" and john.name = "John"
RETURN length(p)
```

- *Afficher le nom des personnes sur le plus court chemin :*

```
MATCH p=shortestPath( (jim)-[*1..10]-(john) )
WHERE jim.name="Jim" and john.name = "John"
RETURN  EXTRACT ( n in nodes(p) | n.name)
```

- *Tous les plus courts chemins entre 'Jim' et 'John' :*

```
MATCH (jim : Person {name : 'Jim'})
MATCH (john : Person {name : 'John'}),
p=allShortestPaths((jim)-[*1..10]-(john))
RETURN p
```

Clause WITH

- Pour des requêtes avec un pipeline d'opérations, relie les différentes opérations entre elles .
- Exemples :
  - *Appliquer un filtre sur le résultat d'une fonction d'agrégation*

```
MATCH (sally {name : 'Sally'})-->(other)-->(x)
WITH other, count(DISTINCT x)+1 AS nbfoaf
WHERE nbfoaf > 2
RETURN other, nbfoaf
```

  - *Trier les résultats avant de les retourner dans une liste*

```
MATCH (n) WHERE n.name is not null
WITH n
ORDER BY n.name DESC LIMIT 4
RETURN collect(n.name)
```

Clause WITH

- Faire des traitements successifs : *En partant de 'Graph Databases', trouver les voisins, les classer par nom, en garder seulement le premier et afficher ensuite ses voisins*

```
MATCH (p {title: 'Graph Databases'})--(other)
WITH other, p
ORDER BY other.name DESC LIMIT 1
MATCH (other)--(neigh) WHERE neigh <> p
RETURN distinct neigh.name
```