Base de données NoSQL – TD4
Neo4j

Master 2 BI&BD - 2020–2021 - 15/11/2021

1 Introduction

La quantité et la nature des données qui nous entourent à l'heure du web 2.0 changent radicalement. D'une part, nous produisons en permanence des quantités phénoménales de données et d'autre part ces données deviennent de plus en plus connectées. La valeur qu'ont les données ne réside plus dans les informations qu'elles contiennent directement, mais de plus en plus dans leurs liens.

1.1 Définition

Les SGBD graphe se basent sur un multigraphe attribué, marqué et orienté. Un graphe marqué a une étiquette pour chaque arête qui est utilisée comme type pour celle-ci. Un graphe orienté autorise des arêtes avec une direction fixée, depuis le nœud source/queue vers le nœud destination/tête. Un graphe attribué autorise une liste variable d'attributs pour chaque nœud et arête, dans laquelle un attribut est une valeur associée à un nom, simplifiant la structure du graphe. Un multigraphe autorise plusieurs arêtes entre deux nœuds. Cela signifie que deux nœuds peuvent être connectés plusieurs fois par différentes arêtes, même si deux arêtes ont la même queue, tête et étiquette.

1.2 Représentation des données

Il existe beaucoup de modèles pour représenter les relations entre les données dans les SGBD graphe. Cependant, un certain effort a été fait pour créer le **Modèle de Graphe Attribué (Property Graph Model)**, unifiant la plupart des différentes implémentations de graphes. Selon celui-ci, l'information dans un graphe attribué est modélisée grâce à trois blocs de base :

— le **nœud** ou sommet (node, vertex)
— la **relation** ou arête (relationship, edge), avec une orientation et un type (orienté et marqué)
— la **propriété** ou attribut (property, attribute), portée par un nœud ou une relation

1.3 Cas d'utilisation

Les graphes sont utilisés par plusieurs entreprises (Google, Payapal, HP, Banques,...). Ils sont très utilisés dans les domaines suivants :

— Recommendation en temps réel
— Détection de fraude
— Recherche d'information
— Administration réseau
— Contrôle d'identité et autorisation d'accès

1.4 Objectifs

L'idée de ce TD n'est pas de découvrir en profondeur les bases de données graphe, mais de réaliser :

- Les avantages que peut apporter un démarche orientée graphe face à des problématiques de massifications de données.
- La pertinence d'utiliser des outils puissants et visuels pour traiter ces problématiques complexes.

Certaines bases graphes : Neo4j, OrientDB et InfiniteGraph. Pour ce TD nous allons utiliser Neo4j.

2 Neo4j

Neo4j est un système de gestion de base de données au code source libre orienté graphes, développé en Java par la société Suédo-Américaine *Neo technology*. Le produit existe depuis 2000, la version 4.3.6 de Neo4j est la version la plus récente (sortie le 14/10/2021).
La démarche utilisée par Neo4j en graphe NoSQL va permettre de traiter :

- Des données massivement interconnectées
- Des données très hétérogènes

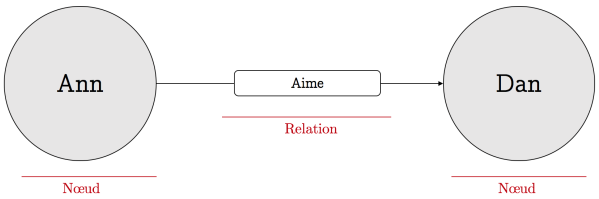
Avantages

- Open source,
- Du fait de sa structure en graphe : modélisation facilitée,
- Un moteur puissant accessible facilement,
- Langage Cypher "assez" visuel,
- Communauté très active.

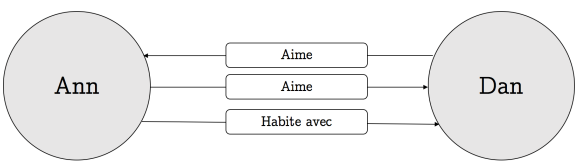
2.1 Modélisation des données

Pour comprendre la modélisation d'une relation dans Neo4j nous allons l'illustrer par des exemples

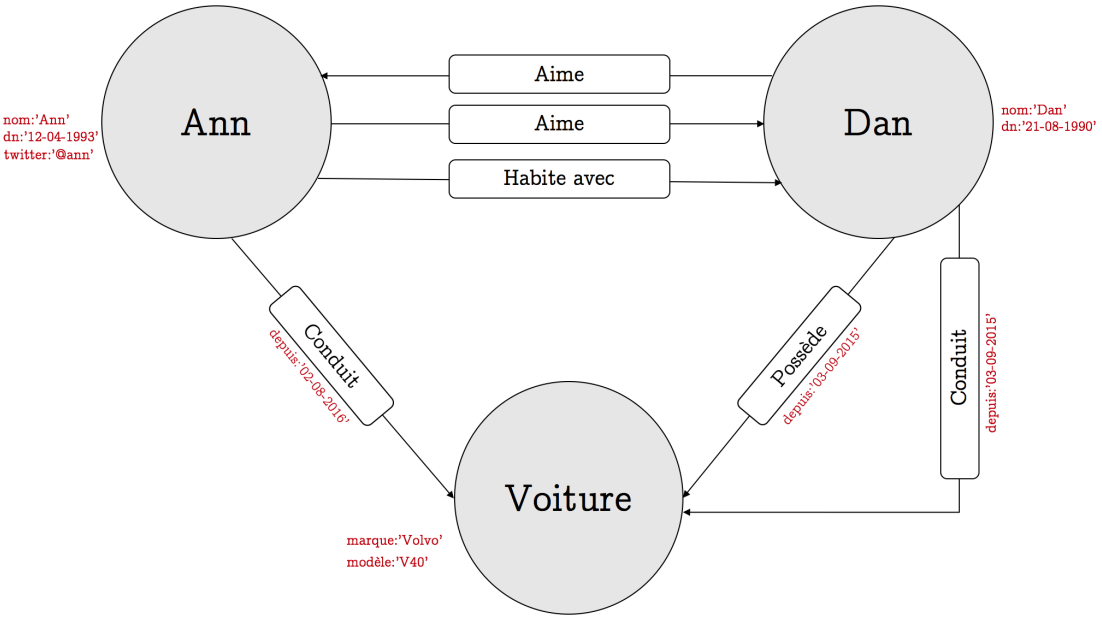
1. Représentation d'une relation
   exemple : `Ann aime Dan`



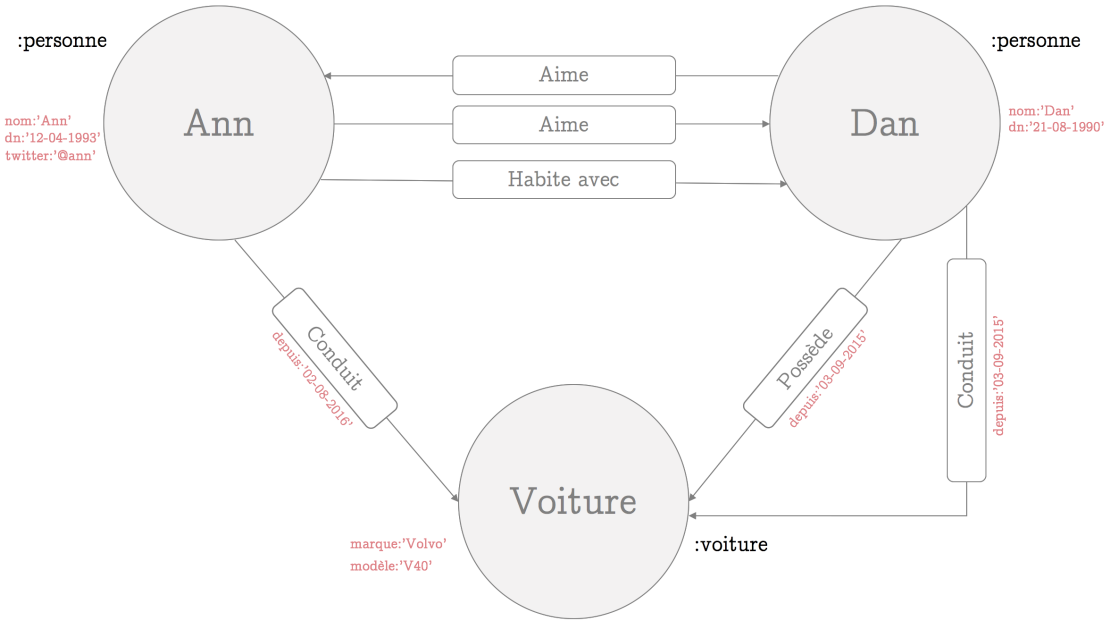
2. Les relations sont orientées
   exemple : `Ann aime Dan ; Dan aime Ann ; Ann habite avec Dan`



3. Les proriétés des nœuds et des relations sont des ensembles de `clé:valeur`
   exemple : `Ann aime Dan ; Dan aime Ann ; Ann habite avec Dan ; Dan possède et conduit une voiture volvo v40 depuis 03-09-2015 ; Ann conduit la voiture de Dan depuis 02-08-2016`



4. La représentation des classes se fait à travers des `étiquettes` de type `clé:valeur`
   exemple : `Ann est une :personne ; Dan est une :personne; et la volvo v40 est une :voiture`



2.2 Résumé

nœud

— un objet du graphe
— peut avoir des propriétés
— peut être étiqueté

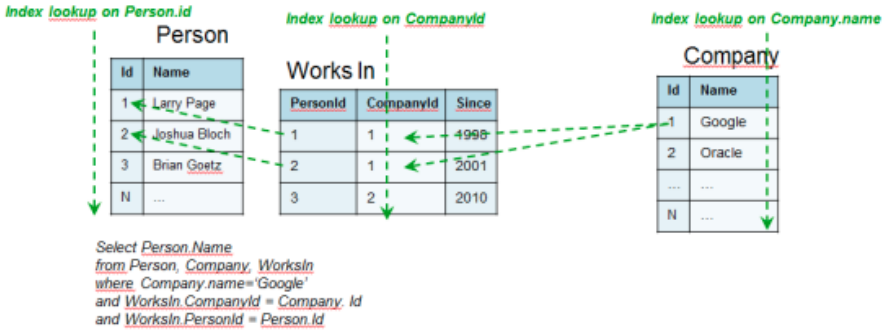
Relation

— un objet du graphe
— relie les nœuds par un `type` et une `direction`
— peut avoir des propriétés

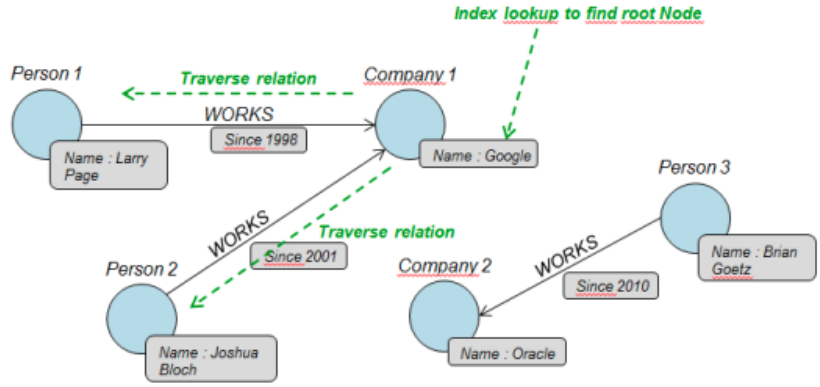
2.3 Pourquoi les graphes

Intuitif : facile à imaginer et à dessiner

Rapide : par rapport au modèle relationnel, on calcule moins de jointures, car on connait directement le lien entre les données. Dans un cas classique avec des relations entre deux tables on effectue une jointure :



En graphe, on ne parcourt que les nœuds qui nous intéressent :
— On retire la table qui contenait les clefs étrangères.
— On lie les données (qui deviennent des nœuds) directement à l'aide d'une relation.



Agilité : le modèle peut s'adapter rapidement aux besoins, un langage de requête facile à apprendre.

3 Syntaxe du langage Cypher

Cypher est un langage déclaratif permettant de requêter et mettre à jour le graphe. Inspiré du SQL, on y retrouve beaucoup de concepts familiers comme les clauses WHERE, ORDER BY, SKYP, LIMIT...

Son objectif est de permettre à l'utilisateur de définir des motifs (pattern), qui seront par la suite recherchés dans tout le graphe.

Ainsi, notre **exemple** `Ann aime Dan` se crée en Cypher par :

```
CREATE (:Personne{name:"Ann"}) -[:AIME]-> (:Personne{name:"Dan"})
```

— `(:NOEUD)` : est utilisé pour définir le nœud et ses propriétés.

— `-[:RELATION]-` : est utilisé pour définir la relation entre les nœuds et ses éventuelles propriétés.

— < ou > : avant ou après la relation est utilisé pour définir l'orientation de la relation.

Pour interroger Neo4j avec Cypher on utilise `MATCH` qui est l'équivalent du `SELECT` en SQL.

```
MATCH (a:personne}) -[AMIS_FB]- (b:Personne}) RETURN *
```

N.B. : Dans Neo4j, toutes les relations sont orientés. Cependant, vous pouvez avoir la notion non orientés au moment de la requête. Il existe des cas où l'orientation de la relation n'est pas nécessaire. Dans l'exemple précédent la relation `[AMIS_FB]` soit on est ami soit on ne l'est pas.

3.1 Les nœuds

Les nœuds sont représentés avec des parenthèses. Si vous avez besoin d'identifier le nœud dans votre requête, il suffit de lui donner un nom : `(monNœud)`

Pour spécifier un label, il suffit de l'ajouter comme ceci : `(monNœud:monLabel)`

Voici quelques exemples :

```
CREATE (personne1:Personne:{name:"Ann"}) -[AIME]-> (personne2:Personne{name:"Dan"})
CREATE (personne2)-[POSSEDE]->(voiture1:Voiture:Diesel{Marque:"Volvo",Modele:"V40"})
```

— `()` : n'importe quel nœud ;

— `(personne1:Personne)` : un nœud identifié dans la variable `personne1` avec le label Personne ;

— `(voiture1:Voiture:Diesel)` : un nœud identifié dans la variable `voiture1` avec le label `Voiture` et `Diesel`.

3.2 Les relations

Les relations sont représentées par deux tirets avec un '> ou <', ce qui ressemble à une flèche.

Si vous avez besoin d'identifier la relation, vous pouvez lui donner un nom comme ceci : `-[maRelation]->`

Pour spécifier le type de la relation, il suffit de l'ajouter comme ceci : `-[maRelation:MON_TYPE]->`

Voici quelques exemples :

— `(a)-(b)` : n'importe quelle relation entre le nœud a et b (peu importe la direction) ;

— `(a)-[:AMI]->(b)` : relation de type `AMI` depuis le nœud `a` vers le nœud `b` ;

— `(a)-[r:AMI|CONNAIT]->(b)` : relation identifiée dans la variable `r` de type `AMI` ou `CONNAIT` depuis le nœud `a` vers le nœud `b`.

3.3 Les requêtes

Pour illustrer l'utilisation de Cypher on va utiliser un exemple. Ajouter un acteur (Type du nœud : Actor) nommé "James Earl Jones" (attribut name) :

3.3.1 Insert

Comme vu précédemment, pour insérer des données avec Cypher on utilise `CREATE`.

```
CREATE (james:Actor {name:'James Earl Jones'})
CREATE (episode6:Movie{title : 'Return of the Jedi'})
CREATE (james)-[:ACTED_IN{role : 'Darth Vader (voice)'}]->(episode6)
```

Insertion à la volée :

Pour tous les acteurs dont le nom est 'James Earl Jones' ajouter un film (Movie) nommé 'Rogue One : A Star Wars Story', et ajouter une relation entre l'acteur et le film de type `ACTED_IN`, de l'acteur vers le film :

```
MATCH (actor:Actor)
WHERE (actor.name='James Earl Jones')
CREATE (swstorie:Movie{title:'Rogue One : A Star Wars Story'})
CREATE (actor)-[:ACTED_IN{role : 'Darth Vader (voice)'}]->(swstorie)
```

Ici, `actor` et `movie` sont des alias, valident le temps de la transaction en cours.
La relation est représentée via : `-[:ACTED_IN]->` que l'on décrit avec `->`, à l'intérieur duquel on décrit `[alias:TYPE]`. Avec un type `ACTED_IN`, et sans alias, on obtient bien une relation entre les deux nœuds désignés par leur alias.

Et si on veut être sûr de ne pas ajouter de doublon sur la relation ?

```
MATCH (actor:Actor)
WHERE (actor.name='James Earl Jones')
CREATE UNIQUE(actor)-[:ACTED_IN{role : 'Mufasa (voice)'}]->
(lionking:Movie{title:'The Lion king'})
```

3.3.2 Schema

Pour afficher les étiquettes et les relations entre étiquettes.

```
call db.schema.visualization()
```

3.3.3 Update

Les opérations de mise à jour se fait en utilisant `MATCH ... WHERE ... SET`

```
MATCH (movie:Movie)
WHERE movie.title = 'Return of the Jedi'
SET movie.year = 1983
CREATE (george:Person{name:'George Lucas'})-[:WROTE]->(movie)
RETURN movie.title, movie.year;
```

3.3.4 Read

`MATCH` est utilisé pour rechercher des informations. Retrouver les acteurs nommés 'James Earl Jones' :

```
MATCH (actor:Actor)
WHERE actor.name='James Earl Jones'
RETURN actor;
```

Récupérer les titres de films, avec une limite de 25 lignes retournées :

```
MATCH (m:Movie) WHERE EXISTS(m.title) RETURN m.title, m.year LIMIT 25
```

Récupérer toutes les relations `[:ACTED_IN]`

```
MATCH p=(Actor)-[r:ACTED_IN]->(Movie) RETURN p LIMIT 25
```

Afficher le graphe en entier

```
MATCH p=()-[r]->() RETURN p LIMIT 25
```

Afficher toutes les relations

```
MATCH p=()-[r]->() RETURN type(r) LIMIT 25
          OU les relations distinctes
MATCH p=()-[r]->() RETURN distinct type(r) LIMIT 25
```

3.3.5 Count

```
MATCH (actor {name: 'James Earl Jones'})-[:ACTED_IN]-(Movie)
WITH actor, count(Movie) AS moviesCount
WHERE moviesCount > 2
RETURN actor.name, moviesCount
```

3.3.6 Delete

1. Supprimer un nœud et toutes ses relations

```
MATCH (m{title: 'The Lion king'})
DETACH DELETE m
```

Vérifier le graphe

```
MATCH p=()-[r]->() RETURN p LIMIT 25
```

2. Supprimer tous les nœuds et toutes les relations

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

La page de la documentation officielle de Cypher est :

https://neo4j.com/docs/developer-manual/current/cypher/

4 Prise en main

4.1 Activité 1

Installez Neo4j Desktop (installable disponible sur Moodle) et prenez en main Neo4j avec les exemples décrits dans la section **3.3 Requêtes** (de **Insert** à **Delete**).
Supprimez toutes les relations et tous les nœuds avant de passer à l'activité 2.

4.2 Activité 2

Un beau matin, vous vous réveillez à Winterfell en Westeros... Sans aucune connaissance historique... Avant de vous mettre à réfléchir, il va falloir intégrer les informations du royaume [1] *!*

Récupérez le fichier qui modélise la base *graphe game of thrones* sur Moodle. Observez la création de nœuds et de relations. Créez une nouvelle base de données puis exécutez les instructions pour créer le graphe.

Affichez le graphe :

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

Modélisez en Neo4j les nœuds et relations décrites ci-dessous :

- Rickard Stark (décédé), ancien Gouverneur du Nord et Seigneur de Winterfell,
- Son fils ainé, Brandon Stark (décédé)
- Son deuxième fils, Eddard Stark Gouverneur du Nord exécuté par Joffrey Baratheon (décédé)
  - Sa femme, Lady Catelyn Stark (décédé)
    - leur fils ainé Robb Stark (décédé)
    - leur fille ainée Sansa Stark
    - leur fille cadette Arya Stark
    - leur second fils Bran Stark
    - leur dernier fils Rickon Stark (décédé)
- Sa fille, Lyanna Stark (décédée)
  - Son fils, Jon Snow (ressuscité), Lord commandant de la Garde de nuit
- Son fils cadet, Benjen Stark, patrouilleur de la Garde de nuit

Requêtes

1. Affichez le graphe en entier
2. Affichez les personnages féminins
3. Affichez les personnages qui ne sont pas décédés

4.3 Activité 3

Dans cet exercice nous allons utiliser un jeu de données fourni par Neo4j. Lancez *Movie DBMS* à partir de *Example Project* sur Neo4j Desktop.

1. Exécutez la requête suivante pour vérifier votre graphe

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

1. exercice game of thrones de Stéphane Crozat

Requêtes à faire

1. Affichez tous les acteurs (`nom`) qui ont joué ("ACTED_IN") dans des films (`titre du film`).
2. Affichez tous les films dans lesquels 'Tom Cruise' a joué.
3. Listez les acteurs et actrices qui ont joué dans "Stand By Me".
4. Affichez les acteurs qui ont joué dans au moins 4 films.
5. Affichez combien de films ont été dirigés par 'Tom Hanks'.
6. Affichez les films sortis après 2002.
7. Affichez la liste des acteurs (qui ont joué dans un film) dont leurs années de naissance (`"born"`) n'ont pas été renseignées.
8. Rajoutez l'année de naissance de l'actrice "Naomie Harris" qui est née en 1976.
9. Affichez la liste d'acteur, leur âge actuel, le titre du film dans lequel l'acteur a joué, l'année de sortie du film et l'âge de l'acteur quand le film est sorti.
10. Qui ont dirigé le film "The Matrix", affichez les noms des personnes et le `type` de relation.
11. Listez les personnes qui ont au minimum 3 types de relation avec un film.
12. "Tom Cruise"
    (a) Listez les acteurs qui ont joué un rôle avec "Tom Cruise"
    (b) Listez les acteurs qui ont joué dans 2 films ou plus avec "Tom Cruise"
13. Listez les personnes et leurs relations avec le film "Cloud Atlas"
14. Affichez le plus court chemin pour relier "Al Pacino" à "Meg Ryan". Utilisez la fonction `shortestPath()`

```
MATCH p=(shortestPath(REQUETE)) RETURN p
```

15. Affichez la liste des films dans lesquels les acteurs de "The Matrix" ont joué un rôle

5 Mise en Oeuvre sous Docker

Les activités de ce TD peuvent être réalisées avec le conteneur *Docker* de Neo4j qui est disponible sur http ://hub.docker.com.

5.1 Docker et Neo4j

1. Rechercher si l'image de Neo4j est disponible dans votre docker :

```
docker images
```

2. Si l'image n'existe pas, télécharger la dernière version de Neo4j :

```
docker pull neo4j:latest
```

3. Créer un répertoire de travail local à vous :

```
mkdir -p ~/Developpement/VotrePrenom/env/neo4j/data
```

4. Lancer votre un conteneur et associer votre répertoire de travail : Deux ports sont ouverts pour Neo4j 7474 pour l'accès http et 7687 pour accéder à l'API Neo4j.

```
$ docker run -p 7474:7474 -p 7687:7687 --name neo4j_td4_g1
-v ~/Developpement/VotrePrenom/env/neo4j/data:/data -d neo4j:latest

$ docker ps
```

Une fois le conteneur Neo4j lancé, allez sur `http://localhost:7474/` et prenez la main sur Neo4j avec les exemples décrits dans la section **3.3 Requêtes** (de **Insert** à **Delete**).