

**Préparé Par**

**Majed ABOU HAMDAN**

**SMB214**

**Responsible : Mr.Pascal Fares**

**2016**

**Introduction:**

Augmentation de a taille moyenne des données :

Taille moyenne des données : Au cours des dernières années, la taille moyenne des données augmentait continuellement. Cette taille, faible en 2007, est actuellement de l’ordre de 2500 Kb. Par exemple la taille moyenne d’une photo est passée de quelques Kb à 2 Mb.

Développement de la connectivité :

A son début, la connexion entre les données était presque inexistante. Il n’y avait pas de liens entre les documents textes. Avec l’évolution du Web au cours des années passant du Web 01, au Web 02 et le Web 03 à nos jours avec XML, RDFa et les réseaux sociaux, la connectivité des données est actuellement très importante et bien développée.

Le problème de la performance de RBDMS :

Le RDBMprésente une perte de performance significative lorsqu’on doit traiter un très grand volume de donner. Elle diminue avec l’augmentation du volume de la taille des domaines. Ainsi, un outil permettant de gérer des données volumineuses tout en conservant un niveau élevé de performance rend très utile cette conception. En effet, l’une des solutions était d’utiliser le NoSQL.

**Le NoSQL:**

Le NoSQL signifie « Not Only SQL » qui est traduit par “pas seulement SQL”. Il désigne une catégorie de base de données apparue en 2009 qui se différencie du modèle relationnel que l'on trouve dans des bases de données connues comme MySQL ou PostgreSQL. Ceci permet d'offrir une alternative au langage SQL.

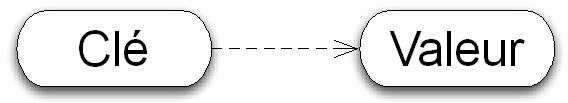
Le NoSQL est apparu afin de contrer la dominance des bases de données relationnelles dans le domaine de l'internet. En effet, un des problème récurrent des bases de données relationnelles est la perte de performance lorsque l'on doit traiter un très gros volume de données. De plus, la multiplication des architectures distribués a apporté le besoin de disposer de solution s'adaptant nativement

Le NoSQL regroupe 4 grandes familles de base de données qui permettent d'offrir une représentation différentes des données :

* Clé – Valeur
* Colonne
* Document
* Base de données Graphe

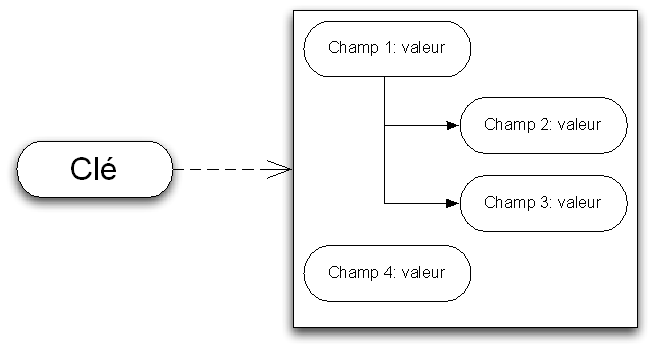
Pour chaque représentation, on dispose de différents acteurs (implémentations)

1. **Représentation en Clé -Valeur :**



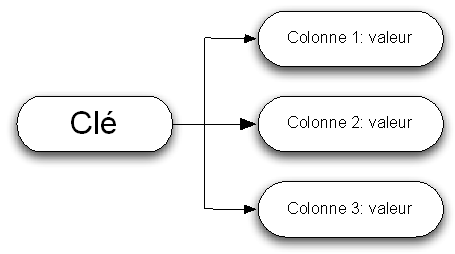
La représentation en clé-valeur est la plus simple et est très adaptée aux caches ou aux accès rapides aux informations. Cette représentation permet en général d’atteindre des performances bien supérieures dans la mesure où les lectures et écritures sont réduites à un accès disque simple. On trouve 3 différentes implémentations: [Riak](http://riak.basho.com/), [Redis](http://code.google.com/p/redis/) et [Voldemort](http://project-voldemort.com/)

1. **Représentation orientée Document :**



La représentation orientée document est plus adaptée au monde de l'internet. Cette représentation est très proche de la représentation clé-valeur à l'exception faite que la valeur est représentée sous la forme d'un document. On peut retrouver dans ce document les données organisées de manière hiérarchique comme ce que l'on trouve dans un fichier XML ou JSON. On trouve 2 types d'implémentations: [CouchDB](http://couchdb.apache.org/) et [MongoDB](http://www.mongodb.org/) ecc….

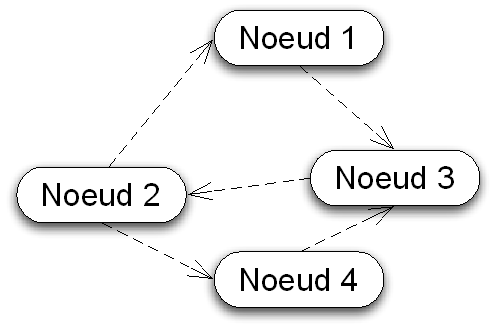
1. **Représentation orientée colonne :**



La représentation orientée colonnes est celle qui se rapproche le plus des tables dans une base de données relationnelles. Elles permettent d'être beaucoup plus évolutive et flexible puisqu'on peut disposer de colonnes différentes pour chaque ligne. On trouve 2 types d'implémentations: [HBase](http://hadoop.apache.org/hbase/), [Cassandra](http://cassandra.apache.org/) ecc…

Mais ces Bases ne sont pas transactionnelles car elles ne sont pas faites pour y stocker des données connectées. Pour cela, on a la **Représentation orientée graph** qui est pour palier à des problèmes impossibles à résoudre avec des BDR.

1. **Représentation orientée graph:**



Les bases de données orientées graphes vous permettent de modéliser, stocker et requêter en temps réel vos données connectées. Le cas d'utilisation typique est bien sur les réseaux sociaux où l'aspect graphe prend tout son sens, mais aussi où des relations complexes entre les acteurs ont besoin d’être d’écrits. Ici, on ne parle plus de table ou de document, mais de nœud et de relation.

On dispose d’une panoplie d’acteurs, dont le plus connu (Bigdata, Cloud Graph, Graph Base, HyperGraphDB, InfiniteGraph, Orient DB, ecc …), le plus utilisé est Neo4J.

**Quand utiliser une BD Graph.**

Les bases de données orientées graphes ne sont pas la réponse à tout, il faut utiliser le bon outil pour le bon besoin. Voici une liste de questions à vous poser avant de partir sur une base de données orientée graphe :

Nous avons les questions suivantes:

* Vos données sont-elles ***dynamiques?***
* Vos données sont- elles ***connectées?***
* Avez-vous besoin d’***un schéma flexible?***
* Devez-vous faire du **temps réel ?**

Si vous répondez oui à au moins deux questions, c'est que vous avez probablement besoin d'une base orientée graphe.

Les cas d'utilisation les plus courants sont :

* la recommandation ;
* la détection de fraude ;
* le Master Data Management ;
* l'analyse de réseaux ;
* l'analyse d'impacts ;
* la gestion des profils et des droits d'accès

**Pourquoi Neo4J ?**

Le Neo4j est une Base de données orientée graphe, il est écrite en Java. C’est un Projet(s) open-source (licence AGPL), développée par Néo Technologie. Il est utilisé par plusieurs entreprises, actives dans différents domaines Comme Adobe, Cisco, Mozilla et autres, qui font confiance à Neo4j.

Il est basé sur des graphes où les BDs graphes manipulent des Données fortement interconnectées et représentent bien un modèle Objet. Dans les BDs graphes, les jointures ne sont pas coûteuses et le modèle est rempli d'algorithmes efficaces et bien connus en Recherche opérationnelle.

Les principales caractéristiques sont les suivantes :

* **transaction :** c'est une base de données transactionnelle, respectueuse des principes ACID ;
* **haute disponibilité :** via la mise en place d'un cluster ;
* **volumétrie :** stocker et requêter des milliards de nœuds et de relations ;
* **Cypher :** un langage de requête graphe déclaratif, simple et efficace ;
* **schemaless :** pas de schéma préétabli.

**Bases de données orientées graphes :**

Les bases de données orientées graphes tournent autour de trois concepts : les nœuds, les relations et leurs propriétés.

Le graphe peut avoir un ou plusieurs nœuds, les nœuds doivent être relies entre eux par une ou plusieurs relations, chaque relation doit avoir une étiquette décrivant la relation et chaque nœud peut disposer d’une ou de plusieurs attributs.

Exemple d'un graphe:

TRAVELS\_WITH

Name: the Doctor

Age: 37

Species: Time Lord

BORROWED

Year: 1963

LOVES

TRAVELS\_WITH

First name: Rose

Late name: Tyler

TRAVELS\_IN

Vehicle: Tardis

Model: Type 40

C'est un graph de plusieurs nœuds, les nœuds sont reliés par un ou plusieurs relations qui ont même une étiquette et pour transformer cette forme en informatique on utilise la matrice Adjacence.

**Requête**:

Pour effectuer une requête, on peut utiliser des outils plus élaborés comme

1. Gremlin :

C'est un langage de traversée de graphe.

Il supporte Neo4J, OrientDB, DEX...

1. Cypher :

C'est un langage déclaratif permettent de requêter et mettre à jour le graph.

Il inspire du SQL, on y retrouve beaucoup de concepts familiers, comme les clauses WHERE, ORDER BY, SKYP, LIMIT ecc…

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

**A Noter:** Les requêtes plus complexes peuvent être effectuées avec le module Cypher qui propose plus d’options

**Parcours de graphes :**

Avant de voir les différents types de parcours, il convient tout naturellement de préciser ce qu'est un parcours ou plutôt quels sont les parcours possibles d'un graphe.

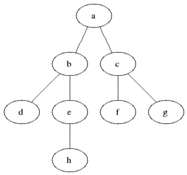
Tout d'abord, il y a deux parcours d'un graphe qui ne reposent pas sur les mêmes éléments. En effet, un type de parcours consiste à  explorer le graphe en passant par tous les sommets. Ce type de parcours est un parcours dit hamiltonien. L'autre type de parcours est un parcours dit Eulérien et consiste à emprunter toutes les arêtes du graphe.

Les parcours peuvent se faire de deux sortes, on a le parcours en largeur et le parcours en profondeur.

1. **Le parcours en largeur** (**Breadth-first search**)**:**

Le parcours en largeur consiste à parcourir le graphe en passant par les sommets directement voisins puis ceux dont il faut passer par un sommet pour les atteindre, et ainsi de suite.

Dans ce parcours, les nœuds seront vérifiés niveau par niveau.



Propriétés intéressantes:

Complexité: (n) où n étant le nombre de nœuds.

Niveau 1

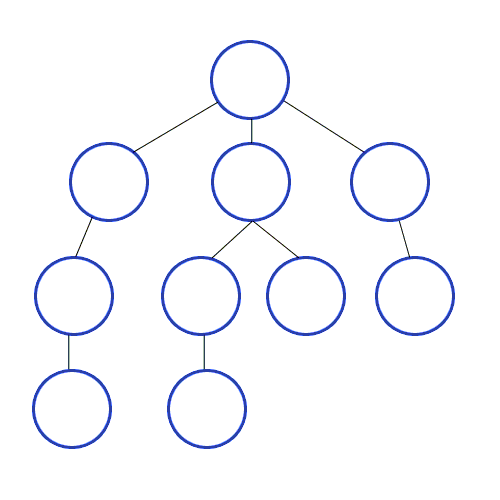
Niveau 2

Niveau 3

1. **Le parcours en profondeur** (**Depth-first search**)**:**

Il s'agit d'un parcours où l'on explore un chemin jusqu'à ce que l'on ne puisse plus avancer dans le graphe. Ce cas arrive lorsque l'on a soit parcouru tout le graph, soit lorsque l'on est arrivé sur un sommet qui n'a pas de successeurs. Dans ce cas, il faut rebrousser chemin dans notre exploration et partir d'un sommet qui n'a pas été visité.

Dans ce parcours, les nœuds seront vérifiés branche par branche



Propriétés intéressantes:

Complexité: (n) où n étant le nombre de nœuds.

Branche 1

Branche 2

Branche 3

Branche 4

Dans les 2 types de parcours notre but est de répondre à une requête formulée par un utilisateur. Pour cela, la première étape est de parcourir le graphe pour trouver le ou les nœuds qui correspondent aux propriétés demandes. Donc, il faut parcourir tous les nœuds pour répondre à une requête, d’où la complexité (**n** : étant le nombre de nœuds).

**Programmation du graphe:**

Pour programmer le graph, on utilise le langage java:

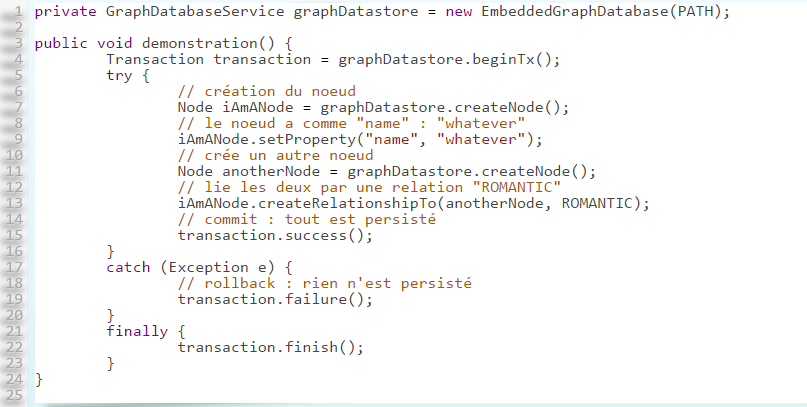
1. Création d'un nœud (vertex, node) qui est la forme la plus simple d'un graphe.
   * ***Node iAmANode = graphDatastore.createNode();***

(**Req:** *iAmANodeest le nom du* nœud*)*

1. Création d'une relation (relationship, edge) qui permet d'organiser les nœuds.
   * ***iAmANode.createRelationshipTo(anotherNode, ExampleRelationships.ROMANTIC);***
2. Création d'un propriété (property, key/value pairs) qui enrichit les noeuds et les relations.
   * ***iAmANode.setProperty("having", "a property!");***

**Exemple 1 (Programmation de 2 nœuds):**

Dans cet exemple on doit créer un graph des 2 nœuds, le premier ayant une propriété, la 2eme sans propriétés et les nœuds sont reliés par une relation « ROMANTIC ».



**Exemple 2 (Traversée d’un graph):**

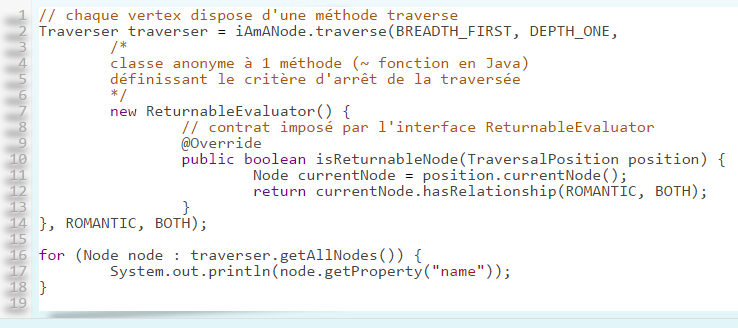
Dans cet exemple la requête est de chercher les nœuds reliés au nœud initial par la relation ROMANTIC, et d'imprimer à la fin la propriété “Nom” des nœuds correspondant à la requête.

En java cette requête se traduit par:

Pour choisir le parcours à suivre : on utilise la commande …#traverse # …

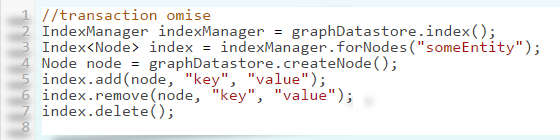
* Si le parcours choisi est en largeur, ce sera: traverse « BREADTH\_FIRST »
* Si le parcours en profondeur ce sera: Traverse « DEPTH\_FIRST “

Ensuite “pour réponde aux requêtes” qui sont les conditions à vérifier, c.à.d. la relation « ROMANTIC » dans notre exemple, on utilise la fonction ReturnableEvaluator.



**Exemple 3 (index d’un graph):**

Pour accélérer la recherche des nœuds, il est souhaitable d’indexer les nœuds en utilisant les fonctions prédéfinies en Neo4j

****

**Mise en pratique**

**Typée les relations:**

En Neo4j, chaque relation doit être typée.

Ex:

Création d’une énumération qui représentera tous les types de notre graphe.



1. **Création d’une classe:**

Pour créer et démarrer la base de données Neo4j en mode embarqué, il suffit de:

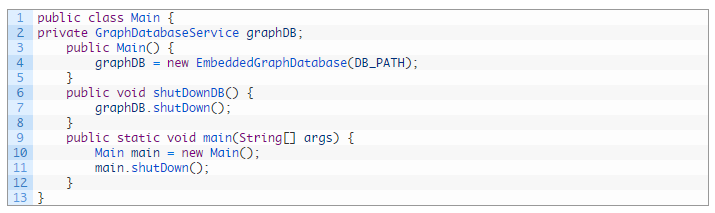
- Déclarer un objet de type GraphDatabaseService.

- instancier avec la classe EmbeddedGraphDatabase en passant comme paramètre au constructeur le chemin de la base.

- La méthode shutdown de notre base de données permet de l’éteindre correctement.

**Déclarer un objet de type**

**Instancier avec la classe**



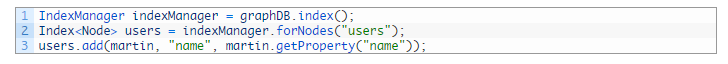
**Permet de l’éteindre correctement**

1. **Création des nœuds:**

Il suffit de déclarer pour chaque nœud un nouvel objet de type Node créé à partir de notre objet graphe :

Pour lui affecter une propriété, comme le nom par exemple, il faut appeler la méthode setProperty qui prend en paramètres une clé et une valeur.

Pour indexer le nœud « Martin » afin de le récupérer facilement Pour indexer le nœud « Martin » afin de le récupérer facilement



1. **Création d’une relation:**

La relation prend en paramètre le nœud à lier et un type. Pour créer une relation entre deux nœud une méthode « **createRelationship** » est disponible dans chaque objet de type Node. Elle prend en paramètre le nœud à lier et un type.

(Voici un exemple qui lie comme ami Martin avec Romain )



1. **Création de plusieurs nœuds:**

Pour créer plusieurs nœuds et plusieurs relations.

Nous devons les encapsuler dans une transaction.

Car si une erreur se produit pendant une insertion il faut invalider les présentes.



**Encapsuler plusieurs noeud et relations**

**Création de plusieurs nœuds**

1. **Création de la traversée:**

Nous allons maintenant créer la traversée qui permettra de connaitre les amis de Martin qui ont au moins deux amis en commun avec lui.

**Récupérer le nœud grâce à l’index**

1. Mais avant il faut récupérer le nœud grâce à l’index.



1. Ensuite la traversée partira de Martin et pour chaque nœud traversé nous allons récupérer ceux qui ont au moins deux relations sortantes de type « Friend ».

Pour effectuer cette opération il faut créer un objet de type Traverser récupéré par le nœud de départ qui est Martin :

Les paramètres de la méthode traverse sont :

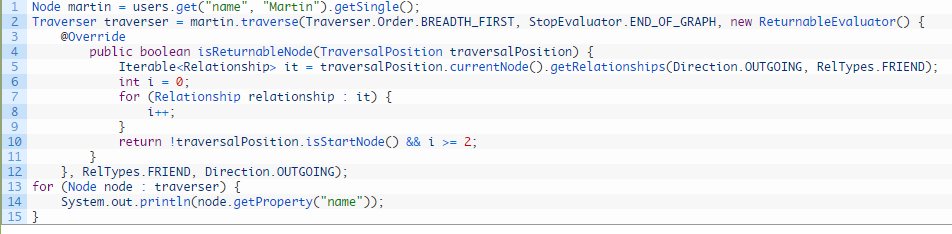
1. Traverser.Order : Ce paramètre peut prendre deux valeurs :
   * Order. BREADTH\_FIRST qui force en premier la traversée de chaque relation du nœud courant
   * Order.DEPTH\_FIRST qui force d’abords la traversée des nœuds enfant du nœud courant.
2. StopEvaluator : Ce paramètre définit la portée de la traversée dans le graphe.

(Ici dans l’exemple nous traversons tout le graphe.)

1. ReturnableEvaluator : Ce paramètre prend un objet de type ReturnableEvaluator qui contient une méthode qui définit si le nœud traversé doit être récupéré.

Dans l’exemple les nœuds validés doivent avoir au moins deux relations sortantes et on exclut le nœud de départ.

1. RelationshipType : Ce paramètre définit le type de relation à parcourir.
2. Direction : Ce paramètre définit la direction des relations à parcourir.



But est de connaitre les amis de Martin qui ont au moins deux amis en commun avec lui.

Résultat: 

Alors il est très simple de faire des requêtes dans ce graphe grâce à Neo4j et les requêtes plus complexes peuvent être effectuées avec le module Cypher qui propose plus d’options.

**Conclusion:**

Comme les autres systèmes de gestion de bases de données, Neo4J est très complet et possède beaucoup de fonctions et autres mots-clés permettant un contrôle très poussé des données. L’avantage dans un système orienté graphe réside dans sa très grande lisibilité pour l’homme, ce qui permet de comprendre rapidement l’architecture mise en place pour ces données, leur organisation et donc, un traitement humain plus vif.

Neo4j est un graphe qui «stocke » des données dans des nœuds possédant des propriétés, les nœuds sont organisés par des relations qui présentent eux aussi des propriétés et une traversée navigue dans le graphe à partir d’un nœud et identifie les chemins ou les sous-chemins avec les nœuds ordonnés en fonction d’options.