# Rapport: bases de données NoSQL

Christophe Simon

20 novembre 2018

# Table des matières

# Introduction

Dans le cadre de ce travail, l'objectif était de réaliser une base de données NoSQL en rapport avec un contexte concret. Pour ce faire, plusieurs étapes doivent être réalisées :

— Choisir un domaine d'application ;
— Établir l'ensemble des données à sauvegarder dans la base de données ;
— Choisir et spécifier trois fonctions de difficultés différentes (facile, moyenne et complexe) ;
— Choisir le modèle NoSQL adéquat.

Dans ce rapport, vous trouvez les différents points abordés ci-dessus ainsi qu'un bref mode d'emploi et les justifications du modèle, du moteur et de l'utilisation d'une base de données NoSQL pour le cotexte choisi.

# 1 Le domaine d'application

Dans mon cas, ce travail est lié avec celui de « Test Driven Development » que nous avons appelé Barathon. Pour rappel, ce projet consiste à trouver une liste de lieux (bars ou restaurants) sur base de préférences. Ce dernier est codé en Java.

# 2 Les données sauvegardées

Étant donné le fait que sur le projet Barathon, nous utilisons un « système » hybride (NoSQL et JSON), il n'y a vraiment que les données minimales qui sont conservées dans la base de données NoSQL. Il y a trois types d'objets stockés qui sont les utilisateurs (User), les lieux (Place) et les caractéristiques (Caracteristic).

## 2.1 User

Concernant l'utilisateur, les champs sauvegardés sont :

— Pseudo (unique [1]) ;
— Longitude ;
— Latitude.

1. Des contraintes s'appliquent dans la base de données pour s'assurer de l'unicité des champs concernés par la mention « unique ».

### 2.2 Place

Concernant les lieux, les champs sauvegardés sont :
— id (unique) ;
— Name ;
— Longitude ;
— Latitude.

### 2.3 Caracteristic

Concernant les caractéristiques, le seul champ sauvegardé est Name (unique).

## 3 Les requêtes réalisées

Dans cette section, vous trouvez les explications concernant les trois fonctions contenant les requêtes de difficultés diverses.

### 3.1 Requête facile

La première fonction « BarsToEat » contient la requête facile qui consiste à trouver la liste de tous les bars où il est possible de manger.
**Input** : /
**Output** : la liste des bars où il est possible de manger.

```
MATCH (p:Place)-[r:FOLLOWS]-(c:Caracteristic
{name: 'food'}) WHERE r.status='true'
RETURN p.id AS id
```

### 3.2 Requête moyenne

La seconde fonction « NearestBar » contient la requête de diffiulté moyenne. Celle-ci consiste à trouver le bar le plus proche d'un utilisateur peu importe les préférences de ce dernier.
**Input** : l'utilisateur connecté (type User).
**Output** : le lieu le plus proche de cet utilisateur (type Place).

```
MATCH (u:User {pseudo: user.pseudo})-[r:AWAY]-
(p:Place) RETURN min(r.distance) AS distance
MATCH (u:User {pseudo: user.pseudo})-[r:AWAY]-
(p:Place) WHERE r.distance = distance RETURN p.id
AS id
```

### 3.3 Requête complexe

La troisième et dernière fonction « NearbyBars » contient la requête complexe. Elle consiste à trouver les X bars les plus proches classés par ordre croissant de proximité dans un rayon Y donné.
**Input** : l'utilisateur connecté (type User), une distance Y donnée en mètres (type int) et le nombre de bars souhaités X (type int) qui sera la taille de la liste en sortie.
**Output** : la liste de X bars classés par ordre de proximité dans un rayon Y donné.

```
MATCH (u:User {pseudo: user.pseudo})-[r:AWAY]-
(p:Place) WHERE r.distance <= Y
RETURN p.id AS id,
r.distance AS distance ORDER BY r.distance
LIMIT X
```

## 4 Base de données NoSQL

L'utilisation d'une base de données NoSQL peut se justifier pour plusieurs raisons. La première est qu'il s'agit d'un projet en cours de développement et que par conséquent, la flexibilité du NoSQL permet des changements de manière plus aisée. Les données ne doivent pas obligatoirement être stockées de manière uniformes. La seconde est que pour les différentes requêtes que nous devons réaliser, il est plus logique d'utiliser une base de données orientée graphe et par conséquent du NoSQL.

## 5 Modèle de base de données

Comme abordé ci-dessus, le modèle de base de données NoSQL utilisé est le modèle orienté graphe. Les requêtes que nous devons utiliser sont basés sur des relations entre différents objets. Puisque nous sommes à présent dans ce modèle, nous parlerons de noeuds et de relations pour bien comprendre l'utilité de ce dernier dans ce projet. Les différents noeuds sont les utilisateurs (User), les lieux (Place) et les caractéristiques (Caracteristic). Les caractéristiques sauvegardées sont « cheap, music, famousPlace, food, vegetarian, halal, vegan et alcohol ». Un noeud est créé par caractéristique. Différents types de relations sont présentes et elles sont détaillées dans les sous-sections suivantes.

### 5.1 User à Place

Les relations de User à Place sont nommées « AWAY » et ont un attribut distance qui est un nombre à virgule représentant la distance (en mètres) qui sépare l'utilisateur des établissements auxquels il est relié. Cette propriété sera mise à jour lorsque l'utilisateur se servira de l'application puisque sa position changera et par conséquent la distance.

### 5.2 User à Caracteristic

Les relations de User à Caracteristic sont nommées « WANTS » et ont un attribut status qui est un booléen signifiant si oui (true) ou non (false) l'utilisateur souhaite les caractéristiques auxquelles il est relié.

### 5.3 Place à Caracteristic

Les relations de Place à Caracteristic sont nommées « FOLLOWS » et ont également un attribut status qui est un booléen signifiant si oui (true) ou non (false) l'établissement respecte les caractéristiques auxquelles il est relié.

### 5.4 Place à Place

Les relations de Place à Place sont également nommées « AWAY » et ont elles aussi un attribut distance qui est un nombre à virgule représentant la distance (en mètres) qui sépare l'établissement concerné des autres auxquels il est relié.

Cette relation se justifie par le fait qu'elle nous fera gagner du temps dans la recherche d'un itinéraire car la distance séparant les bars ne devra pas être recalculée à chaque fois.

## 6 Moteur de base de données

Dans le modèle orienté graphe qui a été choisi, deux choix de moteurs étaient possibles parmis ceux que nous avons vu au cours. J'ai privilégié neo4j par rapport à orientDB pour diverses raisons.

Le moteur orienDB ne propose pas uniquement de la base de données orientée graphe et au vu de l'objectif de notre application, ce n'était pas nécessaire d'alourdir cette partie.

neo4j propose plus de services pratiques tels que l'interface avec le browser web qui est très pratique.

Pour finir, d'après mes connaissances, neo4j est plus majoritairement utilisé qu'orientDB donc il me semblait préférable de commencer par apprendre ce moteur de base de données.

# 7 Mode d'emploi

Le code dans le dossier zip associé est une version du projet Barathon afin qui contient un certain nombre de classes inutiles pour ce travail-ci mais permet de tester avec de vrais objets. Puisqu'il s'agit de fonctions qui vont être utilisées depuis d'autres classes, j'ai réalisé un fichier « Test_DBAcces.java » qui va permettre de lancer ces différentes fonctions et d'observer leur résultat pour ce travail en particulier.

La classe DB contient les fonctions de création de la base de données ainsi que les fonctions contenant les trois requêtes à réaliser.

La classe DBAccess, contient les fonctions pour se connecter, créer un lieu, un utilisateur, ... Il est primordial de changer le nom d'utilisateur, le mot de passe et l'adresse pour se connecter à sa base de données neo4j.

Les autres classes utiles sont dans des fichiers séparés.

### Lancement de « l'application »

Tous les fichiers présents dans le sous dossier java doivent être compilés en utilisant la commande suivante dans java :
*javac -classpath . :./../resources/neo4j-java-driver-1.4.4.jar :./../resources/gson-2.8.5.jar :./../resources/org.apache.commons.io.jar :./../resources/commons-io-2.6.jar :./../resources/org.json.jar nom_du_fichier.java*

Logiquement, la commande suivante devrait suffire puisque les dépendances compileront automayiquement :
*javac -classpath .. :./../resources/neo4j-java-driver-1.4.4.jar :./../resources/gson-2.8.5.jar :./../resources/org.apache.commons.io.jar :./../resources/commons-io-2.6.jar :./../resources/org.json.jar Test_DBAccess.java*

Pour lancer l'exécution en se trouvant dans le sous dossier java, la commande est :
*java . :./../resources/neo4j-java-driver-1.4.4.jar :./../resources/gson-2.8.5.jar :./../resources/org.apache.commons.io.jar :./../resources/commons-io-2.6.jar :./../resources/org.json.jar Test_DBAccess*

Pour la version finale du projet Barathon, il y aura un auntfile afin faciliter les deux étapes précédentes.

# Conclusion

Le premier point sur lequel nous pouvons discuter est que concernant les bases de données non SQL (non relationnelles), il est impératif de faire des tests sur ce qui entre et ce qui sort de la base de données du côté de l'application. Concernant mon application, les tests sont assez sommaires voir inexistants car nous avons le contrôle sur ce qu'il se passe. Les utilisateurs ne rentrent jamais de données eux-mêmes dans la base de données.

Un second point sur lequel il est possible de discuter est qu'il est important de ne pas penser en tables (comme en relationnel) pour ensuite passer en NoSQL. Cela ferait perdre toute la flexibilité du non relationnel sans profiter des avantages du relationnel et par conséquent, il ne serait absolument pas justifié.

En relation avec le point précédent, nous pouvons nous douter qu'il est important de bien choisir son modèle de base de données et par la suite son moteur. Un mauvais choix ne serait également qu'une perte de temps et d'énergie. Il n'y a pas de solution miracle mais bien une solution adaptée à chaque problème.

Comme nous l'avons vu au cours, le NoSQL n'est pas là pour remplacer SQL mais bien pour compléter certaines lacunes. Les deux « familles » ont leurs avantages et inconvénients.

Ce travail m'a permis de découvrir un nouveau modèle de base de données et par conséquent d'étoffer ma connaissance dans ce vaste domaine.