Université Paris Dauphine – PSL

Projet Base de données avancées 2022 / 2023

Réalisé par Léa Boussekeyt, Maïrame Ndiaye et Olesia Ved

Table of Contents

INTRODUCTION **3**

MOTIVATIONS DERRIERE LE CHOIX DES MOTEURS **3**

COMMENT INSTALLER ET UTILISER CHAQUE MOTEUR ? **3**

A. Neo4j *3*
B. ArangoDB *4*
C. OrientDB *4*

STRUCTURE DU JEU DE DONNEES ET DESCRIPTION DES CAS D'USAGES **5**

A. Structure du jeu de données *5*
B. Description des cas d'usages *5*

ÉTAPES DE CHARGEMENT DU JEU DE DONNEES SELON LES MOTEURS. **6**

A. Neo4j *6*
B. ArangoDB *8*
C. OrientDB *10*

REQUETES SUR LES DIFFERENTS MOTEURS **11**

I. REQUETES DU POINT DE VUE UTILISATEUR 11
A. Trouver les trois stations les plus proches d'une position donnée *11*
B. Combien d'arrêts y a-t-il entre deux stations de la même ligne ? *13*
C. Quel arrêt a le plus de connexions ? *16*
D. Trouver la meilleure station pour deux amis qui veulent se rejoindre ? *17*
II. REQUETES POINT DE VUE ANALYSTE DE DONNEES 19
A. Classement de la ligne la plus longue a la plus courte *19*
B. Trouver la station avec la plus grande centralité d'intermédiarité *21*

ANALYSE THEORIQUE DES DIFFERENTS MOTEURS ET COMPARAISON **23**

A. Neo4j *23*
B. ArangoDB *24*
C. OrientDB *26*
D. Comparaison des différents moteurs *30*

CONCLUSION **32**

Introduction

Ce projet se porte sur l'utilisation de moteur de base de données orienté graphe. Pour cela, nous nous sommes appuyées sur un jeu de données représentant le réseau métropolitain de Paris. Nous allons vous présenter 3 moteurs :

- Neo4j, implémenté par Léa Boussekeyt
- ArangoDB, implémenté par Maïrame Ndiaye
- OrientDB, implémenté par Olesia Ved

À travers ce projet, nous allons démontrer que toutes nos requêtes choisies peuvent être implémentées à travers ces trois moteurs. Cependant, il est important de préciser que Neo4j et ArangoDB se sont révélés plus faciles en termes d'utilisation qu'OrientDB. Les performances relatives dépendent des requêtes, un point intéressant qui montre que pour choisir au mieux son moteur de base de données, il est important de considérer les cas d'usages attendus.
De plus, du fait de la complexité du modèle graphe qui exige un grand nombre de détails afin d'assurer une bonne compréhension du sujet et des outils, ainsi que la nécessité d'écrire les requêtes d'OrientDB en Java (comme mentionné par email), nous avons été contraintes d'écrire un rapport plus long qu'exigé.

Motivations derrière le choix des moteurs

Ce projet étant basé sur le modèle de type graphe, notre première considération a été de choisir parmi les leaders du marché. Selon le comparateur db-engines.com, voici les bases de données les plus populaires :

Rank			DBMS	Database Model	Score		
Nov 2022	Oct 2022	Nov 2021			Nov 2022	Oct 2022	Nov 2021
1.	1.	1.	Neo4j	Graph	57.30	-1.38	-0.68
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	39.75	-0.67	-1.08
3.	↑ 4.	↑ 4.	Virtuoso	Multi-model	5.84	+0.12	+1.03
4.	↓ 3.	↓ 3.	ArangoDB	Multi-model	5.84	-0.17	+0.74
5.	5.	5.	OrientDB	Multi-model	5.04	+0.06	+0.41

Le classement est très disparate, avec Neo4j (et de moindre mesure Microsoft Azure Cosmos DB) largement plus populaires que leurs compétiteurs. À noter qu'à l'heure où nous avons commencé ce projet (Octobre 2022), ArangoDB se situait au-dessus de Virtuoso. En second lieu, nous souhaitions une version gratuite. Ainsi, puisque Microsoft Azure et Virtuoso ne proposent pas de version gratuite nous avons sélectionné Neo4j, ArangoDB et OrientDB comme logiciels pour tester le modèle graphe.

Comment installer et utiliser chaque moteur ?

A. Neo4j

Neo4j a plusieurs versions, mais nous utilisons ici la version 'Desktop' qui tourne localement et qui est entièrement gratuite. La version 'Desktop' peut se télécharger depuis le lien ici.

B. ArangoDB

Pour télécharger la dernière version d'ArangoDB, il vous faut vous rendre sur ce lien d'ArangoDB. Il existe deux versions pour le moteur, la version Communauté qui est gratuite et celle d'Entreprise qui est payante. Naturellement, nous avons suivi les instructions du site pour télécharger la version en communauté pour notre type de système d'exploitation.
Après le téléchargement, en cliquant sur le logo de l'application, cela démarre le serveur d'Arango. À ce stade, deux possibilités s'offrent à l'utilisateur :

- Soit travailler avec le Shell d'ArangoDB en suivant le chemin donné par les indications.

Pour le mot de passe laisser le champ vide et cliquer sur la touche « Entrée ». Le Shell s'ouvre ensuite et vous pouvez commencer à travailler.

Le Shell d'ArangoDB correspond à un environnement JavaScript dans lequel il est possible d'utiliser les interfaces et modules JS comme les objets de type *db* pour gérer les collections ou exécuter des requêtes. De nombreux modules d'ArangoDB sur les graphes existant sur JS peuvent être utilisés pour manipuler les graphes comme le module sur les *General Graph*.

- Soit travailler avec l'Interface Web d'ArangoDB en ouvrant sur le navigateur le lien : *http://localhost:8529*

Cela vous ouvre ensuite une page vous demandant de vous identifier. Spécifier comme identifiant « root » et laisser le mot de passe vide. Vous devez ensuite déterminer la base de données avec laquelle vous souhaitez travailler, choisissez « _system ».

L'interface web permet de créer et manipuler des collections, de construire des graphes et de les visualiser, d'écrire et exécuter des requêtes etc…

Pour la réalisation du projet, nous avons choisi de principalement travailler avec l'interface web car beaucoup plus agréable pour manipuler et visionner nos graphes. Cependant, pour l'une des requêtes, il a été nécessaire d'utiliser le Shell d'ArangoDB pour accéder au module sur les *General Graph* de JavaScript.

Remarque : Il est également possible d'installer ArangoDB en tant qu'image Docker ou Kubernettes pour ceux qui possèdent ces outils.

C. OrientDB

La dernière version téléchargeable d'OrientDB est disponible sur ce site. Dans le cadre de ce projet, nous allons créer une base de données sur un serveur local et donc travailler sur ce dernier. Pour faire cela, nous devons lancer l'application "server.bat" ou bin "server.sh" (en fonction de votre système) dans le dossier bin d'OrientDB.

Ensuite, il y a plusieurs options d'interactions/ de créations des bases de données.

Premièrement, l'options la plus basique consiste à se rendre sur http://localhost:2480/ afin d'interagir avec OrientDB Studio (l'interface Web). Vous pouvez y créer la base de données, y indiquer la structure de votre base de données, faire les requêtes les plus simples et visualiser les graphes. Il est possible de créer des fonctions sur JavaScript ou SQL.

Deuxièmement, vous pouvez interagir avec les bases de données à l'aide de la console dans le dossier bin. Néanmoins, l'ensemble des requêtes disponibles reste le même que celui du Studio. Finalement, la dernière option d'interaction qui offre le plus de flexibilité est de se connecter depuis Maven (un outil de gestion et d'automatisation de production des projets logiciels Java) à la base de données OrientDB créée préalablement. Vous pouvez voir l'exemple de connexion depuis Maven ci-dessous.

```
OrientGraph g = new OrientGraph("remote:localhost/Paris_Metro", "root", "root");
```

Structure du jeu de données et description des cas d'usages

A. Structure du jeu de données

Notre jeu de données représente le réseau de métros parisiens. Il est donc composé de deux entités d'informations :

- 302 stations qui représentent les nœuds (ou vertices) du graphe. Chaque station a comme propriété : Nom, coordonnées géographiques X et Y représentés en Lamport 1.
- 15 lignes de métros qui représentent 726 arêtes (ou relations/edges) du graphe. Chaque ligne a comme propriété : Numéro de ligne et d'autres attributs non utilisés.

Les données ont été transmises sous forme de deux fichiers Stations.csv et Liaisons.csv téléchargeables à travers ce lien.

Afin de mesurer l'évolution du temps d'exécution en fonction de la taille du jeu de données, nous avons décidé de prendre un nombre de données réduit. Nous avons donc construit un jeu de données supplémentaire comportant seulement les lignes 1-7 et les stations correspondantes. Ce nouveau jeu de données comprend 165 nœuds et 366 arêtes.

B. Description des cas d'usages

Pour ce jeu de données, il y a de nombreux cas d'usages sur lesquels nous nous sommes penchées. Il y a d'une part les cas d'usages du point de vue d'un utilisateur standard et d'autres part ceux du point de vue d'un analyste de données.

Cas d'usages du point de vue d'un utilisateur standard

Beaucoup de raisons peuvent mener un utilisateur à interroger notre système. Dans cette partie, nous allons décrire 4 types d'utilisations d'une personne standard sur ce jeu de données.

1. Contexte : Je suis à un endroit donné et j'aimerais savoir quelles sont les stations de métro les plus proches de là où je me trouve. La requête en langage courant serait donc : *Quelles sont les stations les proches d'une position donnée ?*
2. Contexte : J'aimerais savoir combien d'arrêts il y a entre deux stations d'une ligne sans avoir à faire le compte moi-même et risquer de me tromper… La requête en langage courant serait donc : *Combien d'arrêts y a-t-il entre deux stations données d'une même ligne ?*
3. Contexte : Je veux déménager à côté d'une station bien desservie. La requête en langage courant serait donc : *Quelles sont les stations qui ont le plus de connexions/correspondances ?*

4. Contexte : Je souhaite rejoindre un ami et je cherche la meilleure station pour se rejoindre de telle sorte que l'on réalise la « même » distance en partant de nos positions respectives. La requête en langage courant serait donc : *Quel est le point de rencontre le plus équitable pour deux personnes qui souhaitent se rejoindre ?*

Cas d'usages du point de vue d'un Analyste de données

Du point de vue d'un Analyste de données, de nombreux cas d'usages existent également. Dans cette partie, nous allons décrire 2 types d'utilisations possibles d'un analyste sur ce jeu de données.

1. Contexte : Je souhaite classer le réseau de métros parisien de la ligne la plus longue à la plus courte pour pouvoir ensuite y faire des analyses. La requête en langage courant serait donc : *Quel est le classement de la ligne la plus longue à la plus courte ?*
2. Contexte : Je dois déterminer quelle station il faut prioriser lors des réparations. La requête en langage courant serait donc : *Quelle est la station la plus centrale dans le graphe ?* Pour cette requête, nous utilisons la centralité d'intermédiarité, qui est définit comme :

$$C_i(u) = \sum_{x \neq y \neq u} = \frac{\sigma_u(x,y)}{\sigma(x,y)} = \frac{nombre\ de\ plus\ courts\ chemin\ reliant\ x\ à\ y\ passant\ par\ u}{nombre\ de\ plus\ courts\ chemin\ reliant\ x\ à\ y}$$

   Dans notre cas de figure, x, y et u correspondent à des stations. Ainsi, avec cette notion de centralité, la station la plus centrale est celle par laquelle transitent le plus grand nombre de plus courts chemins.

Ainsi, nous avons décrit dans cette partie les différents cas d'usages pour ce jeu de données ainsi que les requêtes en langage courant permettant de les définir. Nous avons chacune implémenté ces requêtes sur nos différents moteurs.

Étapes de chargement du jeu de données selon les moteurs.

A. Neo4j

Étape 1 – Créer une base de données dans Neo4j

Pour commencer, il faut ouvrir Neo4j. Ensuite, il faut créer un projet. Pour cela, cliquer sur '+new', puis 'create from directory', puis sélectionner le fichier où nous avons cloné le jeu de données.
Ensuite, nous allons créer une DBMS. Pour cela, cliquer sur 'add' et sélectionner 'Local DBMS' depuis le menu déroulant.
Les requêtes Neo4j vont aussi utiliser deux plug-ins :

- 'APOC' qui contient de nombreuses fonctions et procédures pour faciliter la fouille des graphes
- 'Graph Data Science Library' qui contient une multitude d'algorithmes de graphes.

Finalement, cliquez sur 'start' qui apparait en survolant le nom de la base de données, cette étape peut prendre un petit peu de temps. Pour la suite, il est aussi nécessaire de copier-coller les liens des deux csv que nous utilisons (Liaisons.csv et Stations.csv). Pour cela, survolez

chaque fichier en bas de l'écran pour faire apparaitre les points de suspension, puis cliquez sur 'Copy url to clipboard' et gardez note de ces deux liens.
Vous pouvez ensuite ouvrir le Neo4j Browser (depuis lequel nous allons faire les queries) en cliquant sur 'Open'.

Étape 2 – Importer les fichiers dans la base de données

i. Créer des contraintes

Bien que cette étape ne soit pas indispensable, ceci est important et fait partie des bonnes pratiques pour s'assurer que la base de données reste cohérente et que nous n'avons pas de duplicata de nœuds en cas de mauvaise manipulation.

```
CREATE CONSTRAINT ON (s:Station) ASSERT s.Station IS UNIQUE
```

ii. Créer les nœuds et les relations

Pour cela, nous allons utiliser la fonction 'LOAD CSV WITH HEADERS' en remplaçant le chemin du fichier avec celui copié à l'étape précédente.
Nous allons d'abord télécharger les nœuds à travers la fonction 'merge'. Cette fonction lit les données existantes avant d'écrire ce qui permet d'éviter les duplicatas.
Ensuite, nous allons matcher les stations de départ et d'arrivée ainsi que requêter le numéro de la ligne par laquelle ces deux stations sont connectées afin de créer les relations.
De plus, nous allons remanier les relations afin que le type de chaque relation indique la ligne par laquelle les stations sont reliées ce qui permettra de faciliter les requêtes. Pour cela, nous allons utiliser la fonction 'refactor' du plug-in APOC.
Les lignes commençant par 'WITH' font partie de la syntaxe de Cypher et sont obligatoires entre les fonctions 'MERGE','MATCH' et 'CALL'.

```
LOAD CSV WITH HEADERS FROM 'http://localhost:11001/project-fd933331-be7d-4eb7-a379-b6bd9122641b/liaisons.csv' AS ROW
WITH ROW WHERE toFloat(ROW.ligne) <= 7 // si besoin de limiter la taille du graph
MERGE (s1:Station{Station:ROW.start})
MERGE (s2:Station{Station:ROW.stop})
WITH ROW
MATCH (s1:Station{Station:ROW.start})
MATCH (s2:Station{Station:ROW.stop})
MERGE (s1)-[rel:LIGNE{LIGNE:ROW.ligne}]->(s2)
WITH rel
CALL apoc.refactor.setType(rel, rel.LIGNE)
YIELD input, output
RETURN count(rel)
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	511 ms	1032 ms

iii. Ajouter les coordonnées géographiques en tant que propriété des nœuds.

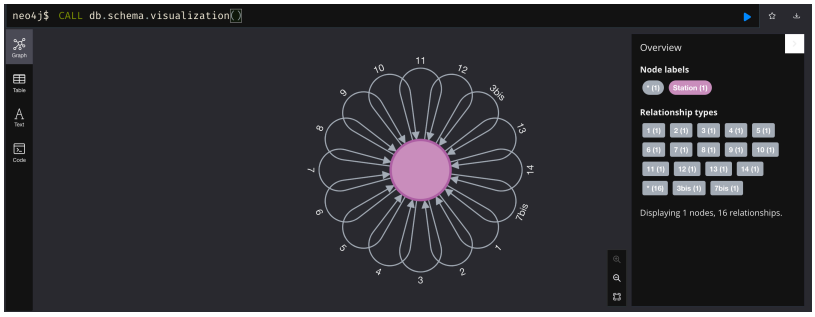
Ceux-ci sont présent dans le fichier 'stations.csv', que l'on va faire correspondre avec les nœuds crées à l'étape précédente. Encore une fois, le chemin du fichier devra être remplacé avec celui copié auparavant.

```
LOAD CSV WITH HEADERS FROM 'http://localhost:11001/project-20b93a0b-a395-4ced-b45d-754c8f2ac0e9/stations.csv' AS ROW
MATCH (s:Station{Station:ROW.nom_clean})
SET s.x_coord = ROW.x
SET s.y_coord = ROW.y
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	147 ms	200 ms

À partir de là, tous les nœuds et les relations devraient être présents et le schéma de la base de données peut être visualisé tel que la figure ci-dessous.

```
CALL db.schema.visualization()
```



B. ArangoDB

Pour le chargement de notre jeu de données dans ArangoDB, nous avons suivi les étapes suivantes :

1. *Conversion CSV/JSON* : L'interface web d'ArangoDB permettant uniquement d'importer des données types JSON, nous avons utilisé un outil en ligne pour convertir les fichiers Liaisons.csv et Stations.csv en Liaisons.json et Stations.json.
2. *Création des collections « Liaisons » et « Stations » :* Une fois la conversion réalisée, nous nous sommes rendues sur l'interface web. Nous avons cliqué sur AddCollection, donné un nom à la collection et choisi comme type « Document ». Les collections « Liaisons » et « Stations » sont désormais créées.
3. *Alimentation des collections « Liaisons » et « Stations »* : Pour charger les données sur les liaisons, nous avons cliqué sur la collection « Liaisons », puis sur le bouton « Upload documents from JSON file » et chargé le fichier Liaisons.json. Pour charger les données sur les stations, nous avons réalisé le même processus.
4. *Création du Graphe sur le réseau de métros parisien* :
    a. Nous avons commencé par nettoyer le jeu de données sur les stations en créant une nouvelle collection de documents nommée « Station » contenant comme clé le nom de la station et ayant comme attributs les position x et y de la station :

```
for s in Stations
    collect nom = s.nom_clean, positionX = s.x, positionY = s.y
    insert {_key:nom, x:positionX , y:positionY} in Station
```

Cette collection sera ensuite utilisée pour créer les nœuds du graphe.

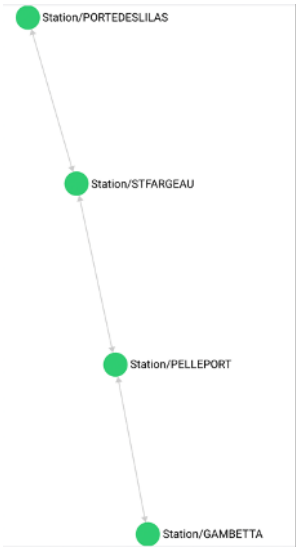
Remarque : Nous avons créé cette collection car si nous avions conservé « Stations » il y aurait eu des duplications dans les noms des stations qui allaient constituer les nœuds de nos graphes. Il était donc nécessaire pour nous en tant que développeur de créer cette unicité. L'opérateur collect du langage AQL permet de créer cette unicité. De plus, les noms des stations n'auraient pas été explicites dans le graphe car lors du chargement des documents JSON, ArangoDB affecte automatiquement des clés numériques pour chaque station, il aurait ainsi été difficile de se repérer dans le graphe avec des clés correspondant à des chiffres. Avec la collection Station, étant donné que les noms de stations sont uniques, il a été possible d'affecter nous-mêmes les clés et de leur donner les noms des stations pour plus de clarté.

b. Nous avons ensuite construit la collection « GStations » de type Edge correspondant aux relations du réseau de métros parisien. Pour l'alimenter, nous sommes parties dans l'onglet « Queries » et avons exécuté la requête suivante écrite en AQL :

```
for l in Liaisons
for s1 in Station
for s2 in Station
    filter l.start == s1._key && l.stop == s2._key
    insert {_from:s1._id, _to:s2._id, ligne:l.ligne} in GStations
```

Les attributs systèmes _from et _to permettent d'indiquer les nœuds de la relation.

c. Une fois la collection « GStations » alimenté, nous avons accédé à la rubrique « Graphs » pour créer le graphe de notre réseau. Nous avons donc cliqué sur « Add Graph » ➔ Name* : GraphStations, Edge definition* : GStations, fromCollections* : Station, toCollections : Station ➔ puis nous avons cliqué sur « Create ». Nous avons donc construit un graphe orienté dont les nœuds sont les stations et les edges portent le numéro de la ligne entre deux stations.



Exemple visuel en filtrant sur la ligne 3bis du graphe

C'est donc sur la collection des relations et sur ce graphe que nous nous sommes appuyées pour implémenter nos requêtes sur ArangoDB.

C. OrientDB

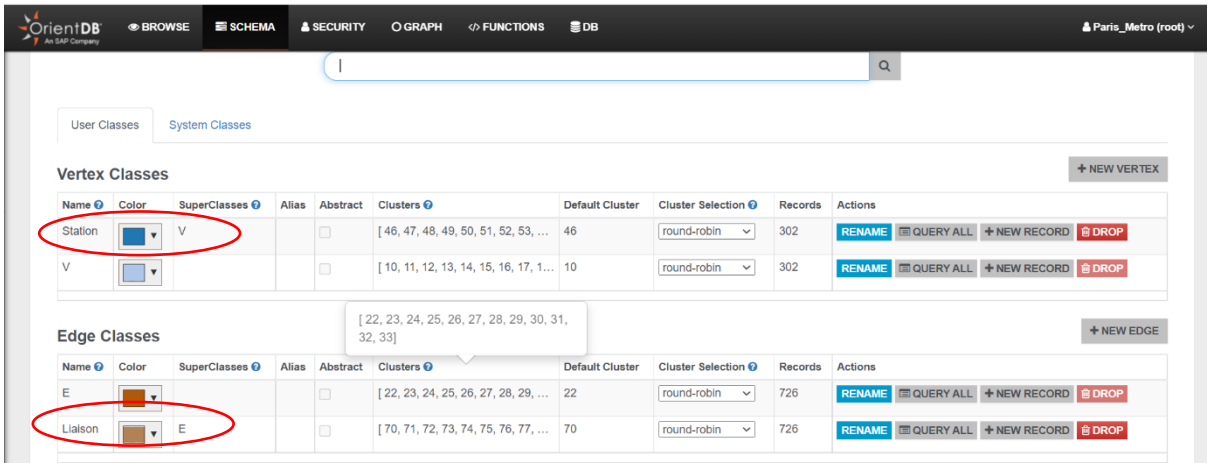
Étape 1 : Création de la base de données et sa structure

Tout d'abord, nous devons créer une base de données. La façon la plus facile est de se rendre sur l'interface web (Studio). Il faut cliquer sur "New DataBase", définir un nom pour la base de données en question, choisir l'identifiant et le mot de passe. Ensuite, nous devons définir les classes correspondantes à notre application. Pour faire cela, nous allons sur l'onglet « Schéma ».

Initialement, il n'existe que trois classes mères :

- V – Classe générique de vertice ou nœud
- E – Classe générique d'edge ou de liaison
- Une classe générique générale

Pour notre cas d'utilisation nous devons créer une classe héréditaire de V et de E, respectivement celles de Station et de Liaison.



Étape 2 : Chargement des données

Dans le cas d'OrientDB, il n'existe pas des fonctions prédéfinies pour charger les données depuis un fichier. Il y a trois façons de faire ceci.

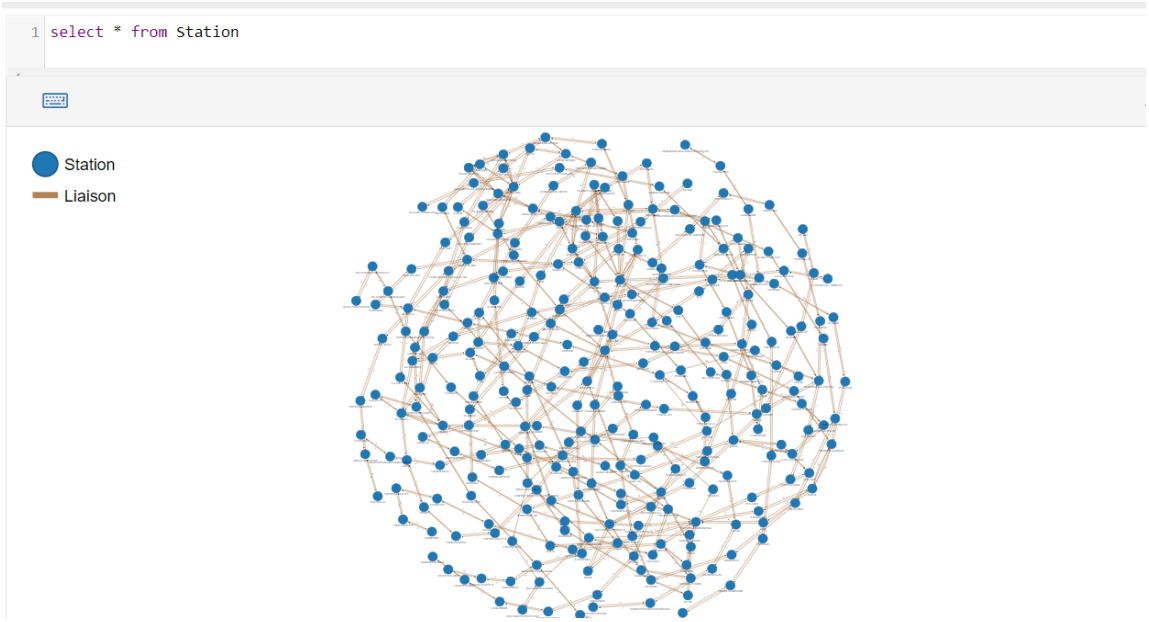
1) Si une application prévoit un flux constant des données avec une structure similaire, il est possible de créer un fichier ETL pour charger les données. Cela permettra de rentabiliser le temps investi dans la création du fichier ETL.
2) Si votre jeu de données est petit, vous pouvez charger vos données une par une grâce à la fonction « CREATE » du langage SQL dans l'interface Web d'OrientDB.
3) Finalement, la meilleure option est d'utiliser le Graph API de Java pour lire les fichiers csv fournies afin de recréer les instances d'OrientDB et de les sauvegarder dans la base de données.

Donc pour la classe Station nous utilisons la ligne ci-dessous

```
graph.command(new OCommandSQL("CREATE Vertex Station set nom='" + nom + "', X=" + X + ", Y=" + Y + ", Lines='" + lines + "'"))
```

Ensuite pour les liaisons nous recherchons deux nœuds en question pour ensuite les relier avec une ligne ci-dessous.

```
graph.command(new OCommandSQL("CREATE EDGE Liaison from" + number(v1Small) + " to " + number(v2Small) + " SET ligne='" + ligne + "'")).execute();
```



Étape 3 : Facilitation du parcours

Les fonctionnalités natives d'OrientDB pour parcourir les graphes ne sont pas très approfondies. Ainsi, pour faciliter les requêtes, étant donnée l'absence de jointure dans le langage basé sur SQL d'OrientDB, nous avons mis en place une propriété supplémentaire qui s'appelle « Lines » pour l'instance Station. Celle-ci représente toutes les lignes qui passent par cette Station. La Station de Père Lachaise aura donc Lines égale à « .3..6. ».

Requêtes sur les différents moteurs

Afin d'avoir une comparaison adéquate entre les différents moteurs, nous avons utilisés les mêmes paramètres pour les mêmes requêtes. Ceci est dû au fait que la traversé d'un graphe plus ou moins long pourrait impacter les performances et ainsi fausser la comparaison. Cependant, chaque moteur a été utilisé sur des ordinateurs différent en local. Nous reconnaissons ainsi que cela peut impacter de façon moindre les performances.

I. Requêtes du point de vue utilisateur

A. Trouver les trois stations les plus proches d'une position donnée

Ici, nous avons décidé de faire notre requête par rapport à la position de Notre Dame.

i. Neo4j

Dans le Neo4j Browser depuis lequel nous avons créé la base de données. Nous allons d'abord lancer une première requête pour enregistrer la position de Notre Dame, puis demander la distance à ces nœuds-là. Ceci se fait en utilisant la fonction point pour mapper les coordonnées, et la fonction point.distance qui donne en sortie la distance géodésique entre ces deux points.

```
:param x_coord => 652109.5385;
:param y_coord => 6861853.2270;
```

```
MATCH (s1:Station)
WITH point({x:toFloat(s1.x_coord), y:toFloat(s1.y_coord)}) AS p1, point({x:$x_coord,
y:$y_coord}) AS p2, s1
RETURN point.distance(p1,p2) AS Distance_en_metre, s1.Station AS Station ORDER BY
Distance_en_metre LIMIT 3
```

Résultat :

	Distance_en_metre	Station
1	9.313225746154785e-10	"CITE"
2	237.8650691737939	"CHATELET"
3	355.4464918559637	"STMICHEL"

	Graphe de taille réduite	Graphe complet
Temps d'exécution	28 ms	66 ms

ii. ArangoDB

Dans le langage AQL, il existe une fonction Distance() prédéfinie permettant de comparer deux points géographiques à l'aide des longitudes et latitudes. Ici, étant donné que les positions géographiques étaient données en Lamport 1 et non en longitude et latitude, nous n'avons pas pu utiliser cette fonction et avons choisi de calculer directement la distance avec la formule de Pythagore (cf ligne 5 requête).

Requête AQL :
Nous calculons la distance entre les stations et Notre Dame, nous trions les stations dans l'ordre croissant, nous récupérons les 3 avec la plus petite distance et renvoyons le résultat.

```
1  LET x = 652109.5385 //Position x lamport de Notre Dame
2  LET y = 6861853.2270 //Position y lamport de Notre Dame
3
4  For s in GStations
5  collect nomStation = DOCUMENT(s._from)._key, dist = SQRT(POW(DOCUMENT(s._from).x - x,2)+POW(DOCUMENT(s
       ._from).y - y,2))
6  SORT dist
7  LIMIT 3
8  return ({nomStation, dist})
```

Résultat :

nomStation	dist
CITE	9.313225746154785e-10
CHATELET	237.8650691737939
STMICHEL	355.4464918559637

	Graphe de taille réduite	Graphe complet
Temps d'exécution	12.337 ms	22.968 ms

iii. OrientDB

Afin d'exécuter la requête 1, il faut lancer la fonction *stationProches()* en lui passant la coordonnée x (dénommée d) et la coordonnée y (dénommée e) de Notre Dame accompagnées de l'instance qui contient le graphe.

Ensuite, nous récupérons l'attribut X et Y et nous effectuons le calcul de ***dist,*** qui est la distance euclidienne. Nous mettons les paires Station/Distance dans un dictionnaire et ensuite nous affichons les trois stations avec la distance la plus courte.

```
public static List<Vertex> stationsProches(double d, double e, OrientGraph graph) {

    long startTime = System.nanoTime();
    Double xStation;
    Double yStation;
    Double dist;
    Map<Vertex, Double> distance = new HashMap<Vertex, Double>();
    for (Vertex v : (Iterable<Vertex>) graph.command(new OCommandSQL("SELECT FROM Station")).execute()) {
        xStation = v.getProperty("X");
        yStation = v.getProperty("Y");
        dist = Math.sqrt(Math.pow((d - xStation), 2) + Math.pow((e - yStation), 2));
        distance.put(v, dist);
    }
    List<Vertex> keys = distance.entrySet().stream().sorted(Map.Entry.<Vertex,Double>comparingByValue()).limit(3).map(Map.Entry::getKey).collect(Collectors.toList());

    graph.shutdown();
    long elapsedTime = System.nanoTime() - startTime;
    System.out.println("Total execution time in Java in millis: " + elapsedTime / 1000000);

    return keys;
}
```

Résultat :

```
CITE
CHATELET
STMICHEL
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	58 ms	73 ms

B. Combien d'arrêts y a-t-il entre deux stations de la même ligne ?

i. Neo4j

Nous allons utiliser une fonction de APOC qui permet de mettre d'avantages de paramètres sur la recherche des chemins. Ceci permet de forcer à ce que tous les chemins passent sur la ligne commune aux deux stations (et ainsi limiter la taille de notre recherche) ainsi que le paramètre 'uniqueness:'NODE-GLOBAL'' qui assure que nous n'avons que un seul chemin en sortie.

```
MATCH (s1:Station {Station: 'PORTEDAUPHINE'}) ,(s2:Station {Station: 'PIGALLE'})
WITH s1, s2
// Obtenir la ligne de métro que les deux stations ont en commun
MATCH (s1)-[r1]-(), (s2)-[r2]-() WHERE r1.LIGNE = r2.LIGNE
WITH distinct(r1.LIGNE) + '>' as Ligne_metro, s1, s2
// Obtenir le path entre les deux stations qui passe par la ligne trouvé
CALL apoc.path.expandConfig(s1,{
 relationshipFilter: Ligne_metro, uniqueness: 'NODE_GLOBAL', terminatorNodes: s2
}) YIELD path
// Obtenir le nombre de relations parcouru et les stations sur l'itineraire
WITH path as path, length(path) AS hops, nodes(path) as nodes
UNWIND nodes AS node
Return hops – 1 as Nombre_Arrets, collect(node.Station) as Stations
```

Résultat :

Nombre_Arrets	Stations
9	["PORTEDAUPHINE", "VICTORHUGO", "CHARLESDEGAULLEETOILE", "TERNES", "COURCELLES", "MONCEAU", "VILLIERS", "ROME", "PLACEDECLICHY", "BLANCH

	Graphe de taille réduite	Graphe complet
Temps d'exécution	202 ms	485 ms

ii. ArangoDB

Nous récupérons la première ligne en commun entre les deux stations puis nous recherchons un chemin entre les deux stations pour la ligne en commun trouvée.

Requête AQL:

```
1  let stationA = "Station/PORTEDAUPHINE"
2  let stationB = "Station/PIGALLE"
3
4  //Récupération d'une ligne en commun des deux stations
5  let ligne = (
6  For g1 in GStations
7  For g2 in GStations
8  filter g1._from == stationA && g2._from==stationB && g1.ligne==g2.ligne
9  collect l = g1.ligne
10 return(l))[0]
11
12 //Recherche chemin entre les deux stations pour la ligne donnée
13 FOR v, e, p IN 1..10 OUTBOUND stationA GRAPH 'GraphStations'
14      FILTER UNIQUE(p.edges[*].ligne)==APPEND([],ligne) && v._id == stationB && UNIQUE(p.vertices[*]._key
            )==p.vertices[*]._key
15      RETURN { vertices: UNIQUE(p.vertices[*]._key), NbStations: length(p.vertices[*]._key)-2}
```

Détail ligne 13 requête : On récupère dans v, e, p respectivement les vertices, edges et paths en partant de stationA pour une profondeur allant jusqu'à 10. Le mot clé OUTBOUND précise que l'on parcourt le graphe en suivant les voisins sortant des nœuds. Cette ligne de code permet donc de traverser le graphe à partir de stationA et de récupérer tous les chemins à partir de cette station.

Détail ligne 14 requête : Cette ligne de code nous permet de filtrer les chemins contenant uniquement des stations de la ligne concernée (condition 1 filtre), se terminant par stationB (condition 2 filtre) et n'ayant pas de boucle (condition 3 filtre).

Résultat:

vertices	NbStations
["PORTEDAUPHINE","VICTORHUGO","CHARLESDEGAULLEETOILE","TERNES","COURCELLES","MONCEAU","VILLIERS","ROME","PLACEDECLICHY","BLANCHE","PIGALLE"]	9

	Graphe de taille réduite	Graphe complet
Temps d'exécution	21.433 ms	54.281 ms

iii. OrientDB

Afin exécuter la requête 2 nous devons exécuter la fonction *nbArret()* à laquelle nous passons en paramètres le nom de la station du départ et d'arrivée ainsi que l'instance du graphe.
En vert : Nous récupérons à l'aide de la requête SQL «Select * from Station » tous les nœuds du graphe. Ensuite grâce aux fonctionnalités de Java, nous recherchons les objets vertex dans le graphe qui portent les noms des nœuds donnés.
En bleu : Nous recherchons la ligne commune entre les deux nœuds grâce à l'attribut supplémentaire «Lines » d'une station.

En orange : Nous partons dans les deux sens depuis un nœud donné puisque nous ne savons pas dans quelle sens se trouve l'autre nœud. Par conséquent, nous exécutons en boucle *While* deux fois la fonction *findNextOLine2()*. L'idée de cette fonction est de passer la requête SQL «Select expand(in()) FROM #VertexX » à la base de données OrientDB. Cette requête trouve les voisins les plus proches d'un nœud et ensuite nous les filtrons par leur numéro de ligne. Ensuite, nous éliminons le voisin que nous avons déjà rencontré, pour ne pas aller en arrière. À chaque itération de cette boucle nous incrémentons la variable count qui correspond à la réponse pour notre requête.

```
29 public class requete2nombreArretMemeLigne {
30     public static void nbArret(String depart, String arrivee, OrientGraph graph) {
31         long startTime = System.nanoTime();
  Station")).execute();
33         List<Vertex> listaVertex = new ArrayList<Vertex>();
34         CollectionUtils.addAll(listaVertex, result.iterator());
35         String name = null;
36         Vertex departVertex = null;
37         Vertex arriveeVertex = null;
38         int numberarr = -1;
39         //Getting the vertex from the vertex list
40         for (int v = 0; v < listaVertex.size(); v++) {
41             name = listaVertex.get(v).getProperty("nom");
42             if (name.equals(depart)) {
43                 departVertex = listaVertex.get(v);
44             }
45             if (name.equals(arrivee)) {
46                 arriveeVertex = listaVertex.get(v);
47             }
49         String s1 = arriveeVertex.getProperty("Lines");
50         String[] lines1 = s1.split("\\.");
51         String s2 = departVertex.getProperty("Lines");
52         String[] lines2 = s2.split("\\.");
53         String lineNumber = null;
54         //finding the common line
55         for (int i = 0; i < lines2.length; i++) {
56             if (lines2[i] != "") {
57                 boolean contains = Arrays.stream(lines1).anyMatch(lines2[i]::equals);
58                 if (contains) {
59                     lineNumber = lines2[i];
60                     break;
61                 }
62             }
63         }
64         System.out.println("the line is " + lineNumber);
66         int count = 0;
67         Vertex minus10 = arriveeVertex;
68         Vertex minus11 = arriveeVertex;
69         List<Vertex> firstNeighbors = findNextOnLine1(graph, arriveeVertex, lineNumber);
70         //traversing the graph common line to find the arrive node
71         if (!firstNeighbors.contains(departVertex)) {
72             while (!end) {
73                 List<Vertex> firstWay = findNextOnLine2(graph, firstNeighbors.get(0), minus1
  lineNumber);
74                 List<Vertex> secondWay = findNextOnLine2(graph, firstNeighbors.get(1),
  minus11, lineNumber);
75                 minus10 = firstNeighbors.get(0);
76                 minus11 = firstNeighbors.get(1);
```

Résultat :

```
the line is 2
the number of stations between is 9
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	169 ms	207 ms

C. Quel arrêt a le plus de connexions ?

i. Neo4j

Ici, il suffit de matcher tous les nœuds et relations et ainsi compter le nombre de relation pour chaque nœud. Nous demandons ensuite les résultats par ordre décroissant.

```
MATCH (n)-[r]->()
WITH n.Station as Station, collect(distinct(r.LIGNE)) as Connections, count(distinct(r.LIGNE)) as Compte
RETURN Station, Connections, Compte ORDER BY Compte DESC
```

Résultat :

	Station	Connections	Compte
1	"CHATELET"	["11", "4", "1", "7", "14"]	5
2	"REPUBLIQUE"	["11", "5", "3", "8", "9"]	5

	Graphe de taille réduite	Graphe complet
Temps d'exécution	39 ms	56 ms

ii. ArangoDB

Requête AQL:

```
1  //On associe chaque station aux lignes auxquelles elle appartient
2  let stationsEtcorrespondances = (
3  for s in GStations
4  collect stationMetro = s._from, ligne = s.ligne
5  return({stationMetro, ligne}))
6
7  //On calcule le nombre de correspondance par station
8  let Nbcorrespondances= (
9  for s in stationsEtcorrespondances
10 collect stationM = s.stationMetro WITH COUNT INTO numCorr
11 sort numCorr DESC
12 return(numCorr))
13
14 //On trouve le max
15 let maxCorrespondance = Max(Nbcorrespondances)
16
17 //On retourne les stations qui ont le max de correspondances
18 for s in stationsEtcorrespondances
19 collect stationM = s.stationMetro WITH COUNT INTO numCorr
20 filter numCorr == maxCorrespondance
21 return({stationM, numCorr})
```

Remarque : L'utilisation de With Count Into est équivalent à un group by + count(*) en SQL.
Exemple : À la ligne 10 du code, on associe à chaque station le nombre de fois qu'elle apparait avec une ligne différente (c'est-à-dire son nombre de correspondances).

Résultat:

stationM	numCorr
Station/CHATELET	5
Station/REPUBLIQUE	5

	Graphe de taille réduite	Graphe complet
Temps d'exécution	10.086 ms	15.163 ms

iii. OrientDB

Afin de trouver les stations avec le plus grand nombre de connexions, nous devons lancer la fonction *stationPlusConnexion()*. Dans cette fonction, nous parcourrons les nœuds du graphe et comptons les connexions pour chaque nœud grâce à l'attribut « Lines »

```
public static List<Vertex> stationPlusConnexion(OrientGraph graph) {
    long startTime = System.nanoTime();
    List<Vertex> maxConnection = new ArrayList<Vertex>();
    int nbMaxConnection = 2;
    int connection = 0;
    String lines;

    for (Vertex v : (Iterable<Vertex>) graph.command(new OCommandSQL("SELECT FROM Station")).execute()) {
        lines = v.getProperty("Lines");
        connection = lines.split("\\.").length;
        if (nbMaxConnection < connection) {
            nbMaxConnection = connection;
            maxConnection.clear();
            maxConnection.add(v);
        } else {
            if (nbMaxConnection == connection) {
                maxConnection.add(v);
            }
        }
        for (Vertex v1 : maxConnection) {
        }
    }
    graph.shutdown();
    long elapsedTime = System.nanoTime() - startTime;
    System.out.println("Total execution time in Java in millis: " + elapsedTime / 1000000);
    return maxConnection;

}
```

Résultat :

```
REPUBLIQUE
CHATELET
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	35ms	63 ms

D. Trouver la meilleure station pour deux amis qui veulent se rejoindre ?

i. Neo4j

Pour cela, nous allons utiliser le plug-in 'Graphe Data Science' de Neo4j. Pour utiliser ses fonctions, il est nécessaire de créer une projection du graphe que l'on appellera ici 'Metro' sur laquelle on appellera l'algorithme du plus court chemin. La requête donne en sortie le chemin de chaque personne et le point moyen du chemin le plus court.

```
CALL gds.graph.project.cypher( 'Metro', 'MATCH (s:Station) RETURN id(s) AS id',
 'MATCH (s:Station)-[r]->(s2:Station) RETURN id(s) AS source, id(s2) AS target')
YIELD graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipQuery, relationshipCount AS
rels;
MATCH (s1:Station {Station: 'CHATELET'}) ,(s2:Station {Station: 'PORTEDAUPHINE'}) WITH s1, s2
CALL gds.shortestPath.dijkstra.stream(
 'Metro',{sourceNode:s1, targetNode:s2}) YIELD path
WITH nodes(path) as nodes, length(path) as len
UNWIND nodes as Arrets
WITH Arrets.Station as Arrets, len
RETURN collect(Arrets)[len/2] as Point_de_rencontre, collect(Arrets)[..len/2+1] as Chemin_A,
reverse(collect(Arrets)[len/2..]) as Chemin_B
```

Résultat :

Point_de_rencontre	Chemin_A	Chemin_B
"CHAMPSELYSEESCLEMENCEAU"	["CHATELET", "PYRAMIDES", "MADELEINE", "CONCORDE", "CHAMPSELYSEESCLEMENCEAU"]	["PORTEDAUPHINE", "VICTORHUGO", "CHARLESDEGAUI

Temps d'exécution	Graphe de taille réduite	Graphe complet
Créer la projection de graph	288 ms	483 ms
Requête	17 ms	117 ms

ii. ArangoDB

Requête AQL:

```
let stationA = "Station/PORTEDAUPHINE"
let stationB = "Station/CHATELET"

//Calcul du plus court chemin entre A et B
let pcch = (FOR v
  IN OUTBOUND SHORTEST_PATH
  stationA TO stationB
  GStations
  return(v))

//Calcul de l'indice de la station du milieu du chemin
let station_milieu_chemin = FLOOR(LENGTH(pcch)/2)

//Retourne la station à laquelle doivent se rejoindre les deux amis
return(pcch[station_milieu_chemin])
```

Le langage AQL fournit le mot clé SHORTHEST_PATH qui renvoie le plus court chemin entre deux nœuds donnés d'un graphe. Le mot clé OUTBOUND précise que l'on parcourt le graphe en suivant les voisins sortant des nœuds.

Résultat:

_key	_id	_rev	x	y
CHAMPSELYSEESCLEMENCEAU	Station/CHAMPSELYSEESCLEMENCEAU	_fFwdu4K--Q	649643.8307	6863316.2783

	Graphe de taille réduite	Graphe complet
Temps d'exécution	0.821 ms	2.078 ms

iii. OrientDB

Afin d'exécuter cette requête, il faut faire appel à la fonction *showPath()*. Cette fonction récupère d'abord les nœuds du graphe afin de retrouver les deux nœuds indiqués en paramètre de la fonction. Ensuite, la fonction envoie la requête : «Select expand(shortestPath(vertex1, vertex2)) ». La fonction shortestPath est prédéfinie dans OrientDB. Elle retourne les nœuds qui constituent le chemin le plus court entre les deux nœuds donnés. Enfin, nous choisissons le nœud au milieu de ce trajet pour le communiquer aux utilisateurs.

```java
public static void showPath(String depart, String arrivee, OrientGraphNoTx g) {
    long startTime = System.nanoTime();
    Iterable<Vertex> result = g.command(new OSQLSynchQuery<Vertex>("SELECT FROM Station")).execute();
    List<Vertex> listaVertex = new ArrayList<Vertex>();
    CollectionUtils.addAll(listaVertex, result.iterator());
    String name = null;
    Vertex v1 = null;
    Vertex v2 = null;
    for (int v = 0; v < listaVertex.size(); v++) {
        name = listaVertex.get(v).getProperty("nom");
        if (name.equals(depart)) {
            v1 = listaVertex.get(v);
        }
        if (name.equals(arrivee)) {
            v2 = listaVertex.get(v);
        }
    }
    String V1 = number(v1); // Vertex A
    String V2 = number(v2); // Vertex D
    String s = "select expand(shortestPath(" + V1 + ", " + V2 + "))";
    result = g.command(new OSQLSynchQuery<Vertex>(s)).execute();
    listaVertex.clear();
    CollectionUtils.addAll(listaVertex, result.iterator());
    List<String> edgePath = new ArrayList<String>();
    String singleIn = "";
    String singleOut = "";

    System.out.println("SHORTEST PATH (vertex):");
    for (int v = 0; v < listaVertex.size(); v++) {
        System.out.print(listaVertex.get(v).getProperty("nom") + " ");
    }
    System.out.println("");
    int stationChoice=Integer.valueOf(Math.round(listaVertex.size()/2));
    System.out.println("The Station to meet is "+listaVertex.get(stationChoice).getProperty("nom"));
    long elapsedTime = System.nanoTime() - startTime;
     System.out.println("Total execution time in Java in millis: " + elapsedTime / 1000000);
    g.shutdown();
}
```

Résultat:

```
SHORTEST PATH (vertex):
CHATELET LOUVRERIVOLI PALAISROYALMUSEEDULOUVRE TUILERIES CONCORDE CHAMPSELYSEESCLEMENCEAU FRANKLINDROOSEVELT GEORGEV CHARLESDEGAULLEETOILE VICTORHUGO PORTEDAUPHINE
The Station to meet is CHAMPSELYSEESCLEMENCEAU
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	35 ms	49 ms

II. Requêtes point de vue analyste de données

A. Classement de la ligne la plus longue a la plus courte

i. Neo4j

Nous allons faire la requête sur les relations (qui contiennent comme type le numéro de la ligne) pour ensuite les compter afin d'obtenir un classement de la plus longue à la plus courte ligne.

```
MATCH ()-[r]->()
WITH TYPE(r) AS Numero_de_Ligne, COUNT(r) as amount
RETURN Numero_de_Ligne, amount/2 + 1 as Longueur ORDER BY amount DESC
```

Résultat (affichage seulement des 4 premières lignes) :

	Numero_de_Ligne	Longueur
1	"7"	38
2	"8"	38
3	"9"	37
4	"13"	32

	Graphe de taille réduite	Graphe complet
Temps d'exécution	9 ms	11 ms

ii. ArangoDB

Requête AQL:

```
//On associe chaque ligne aux stations auxquelles elle est liée
let lignesEtstations = (
for s in GStations
collect ligne = s.ligne, stationMetro = s._from
return({stationMetro, ligne}))


//On calcul le nombre de stations par lignes et on les tries

for l in lignesEtstations
collect ligneM = l.ligne WITH COUNT INTO nbStations
sort nbStations DESC
return({ligneM,nbStations})
```

Résultat (affichage seulement des 4 premières lignes) :

ligneM	nbStations
7	38
8	38
9	37
13	32

	Graphe de taille réduite	Graphe complet
Temps d'exécution	1.837 ms	4.765

iii. OrientDB

Afin d'exécuter cette requête nous devons appeler la fonction *ClassementLignes()*. Cette fonction récupère tous les nœuds et les parcours afin de compter les stations appartenant à chaque ligne. La fonction retourne un dictionnaire Ligne/Nombre.

```
public static Map<String, Integer> ClassementLignes(OrientGraph graph) {
    long startTime = System.nanoTime();
    String s;
    Map<String, Integer> classement = new HashMap<String, Integer>();
    for (Vertex v : (Iterable<Vertex>) graph.command(new OCommandSQL("SELECT FROM Station ")).execute()) {
        s =  v.getProperty("Lines");
        String[] lines = s.split("\\.");
        List<String> stringList = new ArrayList<String>(Arrays.asList(lines));
        for (String i : stringList) {
            int count = classement.containsKey(i) ? classement.get(i) : 0;
            classement.put(i, count + 1);
        }
    }
    graph.shutdown();
    long elapsedTime = System.nanoTime() - startTime;
    System.out.println("Total execution time in Java in millis: " + elapsedTime / 1000000);
    return classement;
}
```

Résultat :

```
13=32
9=37
7=38
8=38
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	28 ms	39 ms

B. Trouver la station avec la plus grande centralité d'intermédiarité

i. Neo4j

Pour cela, nous allons utiliser l'algorithme de centralité d'intermédiarité ('betweeness centrality') sur la projection de graphe déjà créée auparavant. La requête donnera en sortie la station avec la plus grande centralité d'intermédiarité.

```
CALL gds.betweenness.stream('Metro')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).Station AS name, score ORDER BY score DESC LIMIT 1
```

Résultat :

	name	score
1	"MADELEINE"	29960.38674615682

	Graphe de taille réduite	Graphe complet
Temps d'exécution	0.022 secondes	0.158 secondes

ii. ArangoDB

Pour cette requête, il est nécessaire d'utiliser le shell Arangosh pour réaliser la première partie de la requête. En effet, pour calculer la centralité d'intermédiarité, on utilise le module d'ArangoDB en javascript sur les General Graph afin d'accéder à sa fonction betweenness(). Nous avons ensuite stocké le résultat dans une collection nommée collectionCentraliteInter. À la suite de cela, il est possible de retourner sur l'interface web pour poursuivre la requête en filtrant sur la station ayant la plus grande centralité.

Étape 1 - Côté ArangoSh:

```
[127.0.0.1:8529@_system> var graph_module = require("@arangodb/general-graph");

[127.0.0.1:8529@_system> var g = graph_module._graph("GraphStations");

[127.0.0.1:8529@_system> centralite_inter = g._betweenness();
[127.0.0.1:8529@_system> for (var i in centralite_inter){ db.collectionCentralite
Inter.save({kle:i, centralite:centralite_inter[i]}) };
```

Étape 2 – Côté Interface Web:

```
//Association noeuds graphes aux centralités récupérées

let resultat1 = (For s in GStations
For i in collectionCentraliteInter
filter s._from == i.kle
collect station = s._from, centralite = i.centralite
return({station, centralite}))

//Tri du plus grand au plus petit et recuperation du premier
for elem in resultat1
sort elem.centralite desc
limit 1
return elem
```

Résultat requête:

station	centralite
Station/MADELEINE	1

	Graphe de taille réduite	Graphe complet
Temps d'exécution étape 1	7.57 secondes	10.34 secondes
Temps d'exécution étape 2	57.224 ms	87.068 ms

iii. OrientDB

Dans le cas d'OrientDB, il n'y a pas de fonction prédéfinie pour calculer la centralité d'intermédiarité. Nous avons donc dû calculer cette mesure à partir de sa formule. Nous créons donc deux boucles « for » et nous calculons le plus court chemin entre tous les nœuds. Ensuite, nous faisons la somme du nombre de fois que chaque nœud fait partie du chemin le plus court.

```
public static void mostImportantnode(OrientGraph g) {
    long startTime = System.nanoTime();
    Iterable<Vertex> result = g.command(new OSQLSynchQuery<Vertex>("SELECT FROM Station")).execute();
    List<Vertex> listaVertex = new ArrayList<Vertex>();
    CollectionUtils.addAll(listaVertex, result.iterator());
    String name;
    Map<String, Integer> shortest = new HashMap<String, Integer>();
    for (int v = 0; v < listaVertex.size(); v++) {
        for (int u = v; u < listaVertex.size(); u++) {
            String nameV = listaVertex.get(v).getProperty("nom");
            String nameU = listaVertex.get(v).getProperty("nom");
            String V1 = number(listaVertex.get(v)); // Vertex A
            String V2 = number(listaVertex.get(u)); // Vertex D
            String s = "select expand(shortestPath(" + V1 + ", " + V2 + "))";
            result = g.command(new OSQLSynchQuery<Vertex>(s)).execute();
            List<Vertex> listaVertexShortPath = new ArrayList<Vertex>();
            CollectionUtils.addAll(listaVertexShortPath, result.iterator());
            for (int k = 0; k < listaVertexShortPath.size(); k++) {
                int count = shortest.containsKey(listaVertexShortPath.get(k).getProperty("nom"))
                        ? shortest.get(listaVertexShortPath.get(k).getProperty("nom"))
                        : 0;
                shortest.put(listaVertexShortPath.get(k).getProperty("nom"), count + 1);
            }

        }
```

Résultat:

```
MADELEINE
```

	Graphe de taille réduite	Graphe complet
Temps d'exécution	15 secondes	74 secondes

Analyse théorique des différents moteurs et comparaison

A. Neo4j

Neo4j sont à l'origine de ces modèles graphe ainsi que le leader dans cette catégorie. Cependant, il ne supporte aucun autre type de modèle. Neo4j a beaucoup développé son architecture au fur et à mesure des années, et ce rapport se base sur les derniers développements en date.

Bien que ce projet ait été effectué sur la version Desktop de Neo4j qui est gratuite, Neo4j propose également deux autres options :

- AuraDB qui est une version managée de base de données graphes dans le cloud.
- Neo4j entreprise qui est version conçue pour les déploiements commerciaux où le passage à l'échelle est important.

Neo4j est un moteur de base de données de graphes natif car il utilise l'adjacence sans indice. Ceci permet d'accélérer le traitement grâce au fait que chaque nœud est stocké avec une référence directe à ses nœuds adjacents et aux relations. Les nœuds, les relations et les propriétés (comme la position géographique des stations dans le cas de ce projet) sont stockés séparément sur disque. Chaque nœud et relation a une clef unique, attribué automatiquement par Neo4j à la création, de sorte que le logiciel puisse retrouver le nœud ou la relation approprié dans son système de fichier.

Neo4j a été développé en Java. Afin d'effectuer les requêtes, les développeurs de Neo4j ont développé le langage de programmation NoSQL Cypher sur lequel nous avons fait les requêtes. Celui-ci est un langage inspiré par SQL, intuitif et open-source. Les procédures de bases sont complémentées par des fonctions avancées obtenu à travers l'installation de librairies complémentaires telle que :

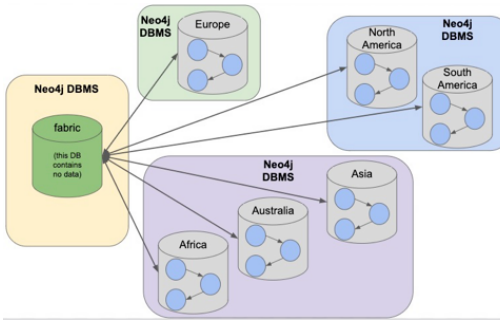
- APOC ('Awesome Procedure On Cypher') qui contient de nombreuses fonctions utilitaires pour des procédures courantes et pour manipuler les graphes.
- 'Graph Data Science' qui contient plus de 50 algorithmes de graphes optimisés, tel que pour calculer la centralité ou encore la détection de communautés.

Une API existe aussi afin de traduire des requêtes Gremlin. De plus, Neo4j peut aussi s'intégrer officiellement avec .Net, Java, Javascript, Go et Python. D'autres langages sont aussi disponibles grâce aux contributions de la communauté Neo4j.

Lorsque l'on utilise un cluster, la réplication est effectué à travers une architecture maitre-esclaves en suivant le protocole de Raft qui gère le problème de consensus distribué. Le maitre informe régulièrement les esclaves de son existence à travers un 'heartbeat message', la fréquence pouvant aussi être paramétré et est par défaut toutes les 5 secondes. Si le message n'est pas reçu par un esclave, alors il deviendra 'candidat' et enverra une demande de vote pour qu'une élection ait lieu. Si celui qui envoie la demande reçoit la majorité des nœuds il devient alors maitre et tous les changements dans la base de données passeront par lui. Chaque modification demandée par le serveur sera rajoutée dans le log du maitre. Afin de commettre la modification, le maitre répliquera d'abord la demande aux autres nœuds et dès lors que la majorité des esclaves ont écrit l'entrée alors celle-ci sera dites comme 'commis' dans le nœud maitre. Pour finir le maitre préviendra par la suite les esclaves de ce changement de statut et enverra un message au client pour confirmer la transaction. Le cluster aura ainsi achevé un

consensus. Pour des transactions larges, il est recommandé d'utiliser la fonction native 'Call' ou la fonction 'Periodic Commit' de APOC qui permet de commettre une transaction en lot, ce qui permet une mise-à-jour plus fréquente ainsi que d'éviter une surcharge de la mémoire active. La fréquence à laquelle les esclaves seront mis à jour peut être réglé dans le fichier 'configuration' de Neo4j, et est par défaut 10 secondes. Il est également à noter que bien que cette architecture permette de respecter les propriétés ACID, ces restrictions peuvent aussi créer un goulot d'étranglement si le client fait de nombreuses opérations d'écriture.

Si le volume de données devient trop grand pour tenir sur un serveur, il faudra alors utiliser la plateforme Fabric de Neo4j qui permet le 'sharding' (partitionnement) de la donnée, une option ajoutée en Janvier 2020 et améliorée avec la version 5 de Neo4j. Cette option peut aussi servir à des cas où un utilisateur nécessite seulement une partie du graphe et un autre le graphe entier. Un autre cas d'usage est lorsqu'il y a des différences en autorisation (certains membres auront accès à certaines partitions mais pas à d'autres). La base de données Fabric contiendra alors pas de données mais agira comme le point d'entrée pour les autres graphes (qui eux contiendront les données). Neo4j est très flexible sur la façon dont ce partitionnement est effectué, les décisions selon comment séparer la base de données retombant entièrement sur l'utilisateur selon ses cas d'usages. Neo4j autorise aussi la création de multiple SGBD contenant chacune une ou plusieurs partitions, comme ci-dessous :



Source: *Sharding graph data with neo4j fabric - developer guides*), Neo*4j Graph Data Platform*

Cela permet par exemple de séparer l'Europe des autres SGBD afin de respecter des lois RGPD. Si les utilisateurs d'Amérique du Nord et d'Amérique du Sud utilisent régulièrement des données de l'un ou l'autre, cela permet également d'optimiser les temps de requêtes. Finalement, si on fait beaucoup de requêtes en Europe et moins en Asie, Australie et Afrique alors mettre l'Europe dans sa propre SGBD et les derniers à trois permet aussi d'équilibrer la charge des différentes SGBD.

B. ArangoDB

ArangoDB, c'est quoi ?

ArangoDB est une base de données NoSQL multi-modèles native. Cela signifie qu'elle prend en charge plusieurs types de données de manière native. Les modèles NoSQL qu'elle supporte sont :

- Clé/Valeur
- Document
- Graphe

ArangoDB possède une hiérarchie de stockage comme d'autres bases de données. Au plus haut niveau de la hiérarchie, nous avons les bases de données qui contiennent des collections (de type document ou edge) qui elles-mêmes contiennent des documents JSON dont les noms d'attributs sont des chaînes de caractères et les valeurs peuvent être des null, booléens, nombres, chaînes de caractères, tableaux, objets imbriquées etc… Les documents sont également composés d'attributs système :

- _key, _id et _rev pour les collections de types document, ces attributs permettent d'identifier chaque document.
- Pour les collections de types edges, il y a en plus les attributs systèmes _from et _to qui indiquent les _id des nœuds appartenant à une relation. Ce sont ces attributs qui permettent de rendre ArangoDB natif pour les graphes. En effet, les attributs _from et _to permettent le parcours des graphes suivant les relations, ce qui accélère le parcours des graphes par rapport aux modèles relationnels.

Pour réaliser des requêtes sur les différents modèles, ArangoDB met à notre disposition leur propre langage AQL (ArangoDB Query Langage). Le résultat de chaque requête peut être un tableau associatif, des documents JSON ou si lié à un graphe, la visualisation de ce dernier.
De plus, ArangoDB offre la possibilité de combiner les 3 types de modèles de données dans une même requête. Enfin, le système permet de combiner des jointures, des traversées, des filtres, des opérations géo-spatiales et des agrégations dans les requêtes.

Analyse théorique d'ArangoDB :

ArangoDB laisse le choix à l'utilisateur de choisir parmi plusieurs architectures en fonction de ses besoins et de ce qu'il recherche. 3 types d'architectures sont proposées par le système :

1. *« Single instance »* : Il s'agit de la configuration la plus simple et qui est installé par défaut sans nécessité de réaliser des procédures spécifiques. Dans cette architecture, le serveur d'ArangoDB est autonome, sans réplication, sans possibilité de basculement vers un système secondaire en cas de panne. Il s'agit donc de la version la moins sécurisée en termes d'architecture. Il est possible toutefois avec cette architecture d'exécuter plusieurs processus d'ArangoDB côte à côte sur la même machine à condition que ces processus soient sur des dossiers et branchés sur des ports différents.

2. *« Active Failover »* : Cette architecture utilise la technologie multi-nœud et s'appuie sur le principe de l'architecture maître/esclave. Au sein de cette architecture, on a une « Single instance » d'ArangoDB qui est accessible en lecture et écriture, qu'on appelle « Leader » et une ou plusieurs instances d'ArangoDB à serveur unique, seulement accessible en lecture qu'on appelle « Followers » et qui répliquent de manière *asynchrone* les données du Leader. La cohérence de la base de données est donc à terme et la disponibilité des données est garanti. Cette architecture proposée par ArangoDB, possède en plus un tiers actif (nommé « Agency supervision ») qui fait office de témoin et qui a pour rôle de déterminer de façon dynamique quel serveur devient le leader en cas de panne. L'avantage de cela est que le basculement en cas de panne est automatique. Cette architecture fournit donc une haute disponibilité aux petits projets avec une réplication rapide mais seulement cohérente à terme.

3. *« Cluster »* : Cette architecture d'ArangoDB est un modèle maître/maître CP sans aucun point de défaillance. « Maître/maître » signifie que contrairement à la précédente

architecture, les clients peuvent envoyer des requêtes à n'importe quel nœud et avoir la même vue sur la base de données. « CP » fait référence au C et P du théorème CAP. En cas de partitionnement du réseau, la base de données préfère la cohérence à la disponibilité. Enfin, « aucun point de défaillance » signifie que si une machine tombe en panne, le cluster pourra toujours servir les demandes.

Un cluster ArangoDB est composé de plusieurs instances ArangoDB qui communiquent entre elles via un réseau. Chaque instance joue un rôle spécifique. On a d'une part, les « Agents » qui forment l'Agence du cluster qui correspond à l'endroit où est stockée la configuration du Cluster. L'Agence ce charge d'élire les leaders, priorise les opérations qui arrivent et fournit des services de synchronisation du cluster. Étant au cœur du cluster, l'Agence doit avoir une forte tolérance aux pannes. Pour cela, les agents utilisent l'algorithme Consensus Raft pour garantir une gestion de la configuration sans conflit au sein du cluster. Ensuite, on a les « Coordinators » qui jouent le rôle de point d'entrée entre le client et la donnée. Ils coordonnent les tâches du cluster comme l'exécution des requêtes. Ils savent où sont stockées les données et optimisent l'endroit où exécuter les requêtes. Enfin, on retrouve dans le cluster les « DB-Servers » qui ont la charge de l'écriture et lecture des données. Les DB-Servers sont soit Leader ou Follower pour un shard (partition) choisit par l'Agence. Les opérations sur les documents sont d'abord appliquées sur le leader de chaque partition puis répliquées de manière *synchrone* sur tous les Followers. La réplication synchrone implique donc la cohérence de la base de données pour une disponibilité amoindri étant donné que le client est bloqué jusqu'à que la mise à jour soit réalisée chez tous les nœuds.

Dans le cadre de l'architecture Cluster, ArangoDB utilise le « Sharding » (Partitionnement) pour faciliter le passage à l'échelle et d'utiliser plusieurs machines. Cela ne change rien pour l'utilisateur, il peut toujours s'adresser à n'importe quel coordinateur qui déterminera automatiquement où les données sont stockées (en lecture) ou doivent être stockées (en écriture). Tous les coordinateurs qui utilisent l'Agence partagent les informations sur les différents shards. Pour déterminer dans quel tiroir stocker les données, ArangoDB utilise une fonction de hachage. Par défaut, le hachage est calculé à partir de l'attribut _key du document. Le partitionnement réalisé est donc horizontal. La tolérance au partitionnement et la résistance à la montée en charge est donc forte au sein de cette architecture proposée par ArangoDB.

À propos du moteur de stockage : Le moteur de stockage d'ArangoDB est responsable de la persistance des documents sur le disque, du maintien des copies en mémoire, de la fourniture d'indexes et de caches pour accélérer les requêtes. Le moteur de stockage est basé sur RocksDB de Facebook qui est optimisé pour les grands ensembles de données. Les indexes sont stockées dans les disques et les caches sont utilisés pour accélérer les performances. Le moteur utilise des verrous au niveau des documents pour permettre des écritures simultanées et gérer les conflits. Ainsi, les écritures ne bloquent pas les lectures et inversement.

C. OrientDB

Introduction

OrientDB est un SGBD multi-modèle crée en 2010 par la société Orient Technologies. Il supporte les modèles clef/valeur, document, graphe et objet. OrientDB est développé en Java.

Il est supporté sur les OS Linux (architecture ARM incluse), Windows, Solaris, HP-UX, AIX et Mac OS X et utilise au minimum Java 1.6.

Couches

OrientDB est construit par un empilement des couches. Elles sont les fondations et implémentent l'interface de stockage de données. Le figure ci-dessous met aussi en évidence le rôle central de la couche de gestion Document sur laquelle se reposent les autres modèles de données. Par défaut, les applications clientes peuvent attaquer n'importe quelle couche de données et ont accès à l'intégralité des informations stockées. Cette structure laisse entrevoir un risque sur l'intégrité des informations, comme la modification directe des données de graphes en passant par l'API Document : les fonctionnalités fournies dans les couches supérieures, comme le mécanisme transactionnel des nœuds et des arêtes des graphes, ne seraient alors pas utilisées. On se trouverait alors dans un état inconsistant.

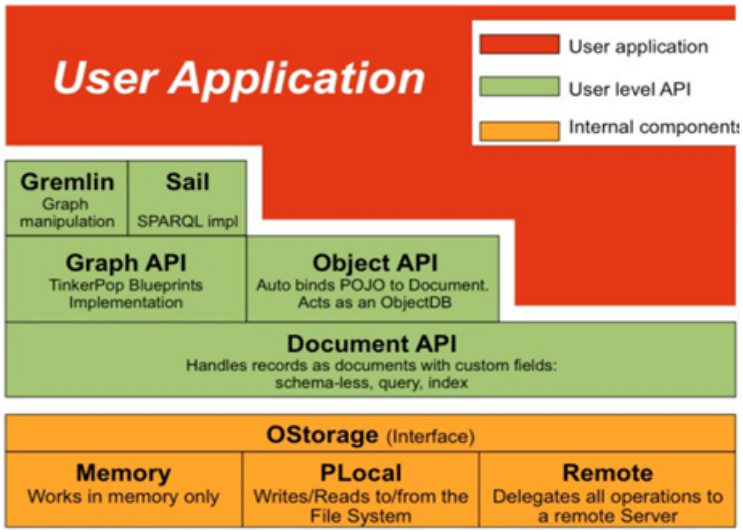


Image prise sur « Bases de données graphes : comparaison de NEO4J et OrientDB » Par Nicolas Vergnes

Architecture Distribuée

OrientDB est un modèle Multi Master replication. La réplication multi-maîtres est une architecture pour la réplication des bases de données permettant la lecture et l'écriture à tous les serveurs. Le système de réplication multi-maître est responsable de propager les modifications de données faites par chaque membre et résoudre les conflits provoqués par des modifications concurrentes faites sur des membres différents. Cela permet la scalabilité horizontal sans les goulots d'étranglements comme pour la plupart des solutions NoSql.

À partir de la version v2.1, OrientDB propose l'utilisation des serveurs REPLICA. Ce sont des serveurs qui n'effectuent que la lecture des données. À partir de la version 2.2, les nœuds REPLICA ne participent pas dans les WriteQuorum.

Par default, OrientDB crée un cluster par classe. Dans ce cas-là toutes les données de cette classe sont stockées dans le même cluster qui porte le nom de la classe en question. Il est possible de créer jusqu'à 32,767 (2^15-1) clusters dans une base de données.

Tous les serveurs d'une grappe communiquent entre eux de façon décentralisée grâce à l'application distribuée HazelCast. Un nouveau serveur se connecte d'abord sur le bus de

communication puis synchronise les replicas des clusters qu'il doit héberger avant de devenir actif. Les serveurs informent leurs clients de cet ajout.

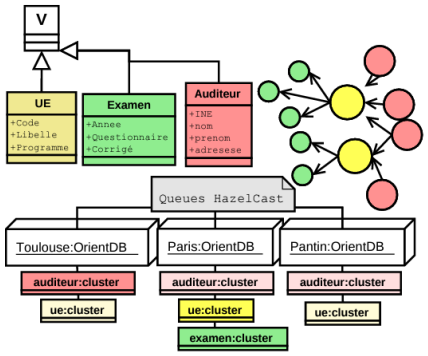


Figure 12: OrientDB : Les 3 classes de nœuds sont répliquées sur 3 instances

Image prise sur « Bases de données graphes : comparaison de NEO4J et OrientDB » Par Nicolas Vergnes

Lorsqu'un serveur tombe, ses clients se redirigent automatiquement vers d'autres serveurs actifs. Le serveur pourra ensuite rejoindre le cluster en resynchronisant ses clusters obsolètes aux autres. Cela est fait de la manière suivante : lorsque la base de données n'est pas alignée, alors OrientDB utilise Conflict Resolution Strategy chain. Cette stratégie est aussi utilisée lorsque le nombre de serveurs est pair. L'importance du nombre de serveurs est expliquée dans la rubrique « Limitations de l'Architecture Distribuée **d'OrientDB»**
Le Conflict Resolution Strategy est appelé en chaine jusqu'à ce que la version "gagnante" soit élue. Dans sa configuration par default :

- Premièrement, s'il y a de la majorité stricte selon les versions retenues, la version gagnante est donc élue.
- S'il n'y a pas de majorité stricte en termes de version, OrientDB procède à l'analyse du contenu. Les deux versions dont le contenu est similaire sont assimilées à une seule version.
- Si la majorité stricte n'a pas été retrouvée avec le contenu, alors la version la plus récente gagne.

Routage

Définition de la méthode de routage d'une classe vers ses clusters :

Exemple: ALTER CLASS Entreprise CLUSTERSELECTION ROUND-ROBIN

Les différentes manières de routage décrites dans la documentation d'OrientDB[1] :

- Choix par défaut : le logiciel choisit le cluster en utilisant l'attribut defaultClusterId de la classe. Jusqu'à la version 1.7, celle-ci c'était la méthode par défaut.
- Round-Robin : le logiciel organise les clusters configurés pour la classe suivant l'ordre établi et affecte chaque nouvel enregistrement au cluster suivant l'ordre établi.
- Balanced : le logiciel vérifie le nombre d'enregistrements dans les clusters configurés pour la classe et attribue le nouvel enregistrement à celui qui est le plus petit à chaque

[1]https://orientdb.com/docs/2.2.x/

instant. Pour éviter les problèmes de latence lors des insertions de données, OrientDB calcule la taille du cluster toutes les cinq secondes ou plus.

- Local : Lorsque la base de données est exécutée en mode distribué, elle sélectionne le cluster maître sur le nœud actuel. Cela permet d'éviter les conflits et de réduire la latence du réseau avec les appels à distance entre les nœuds
- Custom : En plus des stratégies de sélection de cluster répertoriées ci-dessus, vous pouvez également développer vos propres stratégies de sélection via l'API Java.

Limitations de l'Architecture Distribuée d'OrientDB

Certaines actions pour OrientDB peuvent causer des incohérences pour la base de données. Ci-dessous sont les plus fréquents :

- hotAlignment:true peut apporter une incohérence à la base de données. Ce paramètre permet de réeffectuer à nouveau les opérations à un nœud qui était en panne.
- La création de la base des de données sur plusieurs nœuds peut causer les problèmes de synchronisation quand les clusters sont créés automatiquement.
- Un nombre pair des nœuds peut causer une incohérence. Lors de la séparation et la reconstruction de la base de données, la version finale peut être fausse puisqu'il n'y a pas eu de majorité stricte.

Sharding

Le sharding est une fonctionnalité en architecture distribuée disponible sur OrientDB. Aussi appelée partitionnement horizontal en français, elle désigne la possibilité de morceler une classe de données sur plusieurs nœuds. La classe possède son cluster par défaut sur un des nœuds qui, contrairement à une simple réplication, est le seul point d'entrée pour traiter les manipulations de données dans son périmètre et mettre à jour ses répliquas. Une application cliente recevra une erreur si elle essaye explicitement d'attaquer un répliqua avec une requête.

Cache

Chaque serveur OrientDB possède son propre cache logique indépendant. En architecture distribuée, un deuxième niveau de cache logique vient s'ajouter, c'est un cache commun à tous les nœuds. Un dernier niveau de cache physique existe pour les bases de données stockées en local.

D. Comparaison des différents moteurs

Comparaison	Neo4j	ArangoDB	OrientDB
Type de modèle authorise	Graphe	Multi Modèle : Document, Clé/Valeur, Graphe	Multi Modèle
Langage Natif	Cypher	AQL	SQL sans jointure
Langage supportés	Gremlin	Gremlin	Gremlin
Sharding	+	+	+
Gestion des transactions	ACID, Protocole de Raft	ACID, Multiversion Concurrency Control	ACID
Fonctionnalités de traversée	+	+	-
Tolérance de panne	Si cluster	Si ActiveFailover ou Cluster	+
Réplication	+	Si ActiveFailover ou Cluster	+
Langage de développement du moteur	Java	C++, JavaScript	Java
Format de stockage données	Enregistrement de taille fixe liée	Documents JSON	Enregistrement liée

Autres considérations :

Accès à l'information

Un point important à considérer est la facilité d'accès à l'information par rapport à un moteur utilisé. Ceci comprend à la fois la qualité de la documentation, et la communauté autour de ce moteur. Cette dernière permet de l'entraide afin de se débloquer rapidement. La documentation doit être précise et complète. Pour ceci, Neo4j a un avantage majeur, ayant de loin la plus grande communauté (grâce à laquelle la librairie APOC a été développé) ainsi qu'une documentation accessible et à jour. Du côté d'ArangoDB, leur communauté concernant les graphes n'est pas très grande et il n'est pas facile de retrouver les informations recherchées à travers des forums. Cependant, la Documentation fournit sur le site est d'une grande qualité et offre à un nouvel utilisateurs toutes les ressources nécessaires pour utiliser au mieux l'outil.
De plus, pour ces deux moteurs, il existe une version entreprise payante qui offre du support technique, à l'instar d'OrientDB qui est complètement open source. Pour cette dernière, l'accès à l'information est difficile notamment sur la façon de requêter sur le moteur ainsi que pour la prise en main initiale. Le fait qu'OrientDB ait une communauté faible rend le processus de développement plus difficile.

Prise en main de l'interface

L'interface Web d'ArangoDB ainsi que le Neo4j Browser sont très ergonomiques et faciles d'utilisation. De plus, Neo4j propose un outil Bloom, qui est très utile dans le cadre professionnel où certaines équipes ont peu de compétences techniques. Celui-ci permet des

visualisations, afin d'examiner des nœuds et des relations dans la base de données et accéder à des requêtes préprogrammées. OrientDB offre une Interface Web appelée Studio. Elle est dédiée principalement à la visualisation des graphes et assez lente lors de l'utilisation.

Traversée des graphes :

Il est important de noter que sur certaines traversées de graphes ArangoDB rencontrait plus de difficultés que les autres moteurs. En effet, le moteur force l'utilisateur à saisir la profondeur de la traversée. En testant l'outil sur notre graphe complet de réseau de métros parisien, dès lors que la profondeur spécifiée dépasse 20 nœuds, le système peut se bloquer. Ainsi, si le cas d'usage est de faire de longues traversées de graphes, ArangoDB ne sera pas le moteur le plus adaptée si utilisation de l'architecture *« Single instance ».* Les autres moteurs ne rencontrent pas ce problème. Pour le cas d'OrientDB la fonction Traversée est peu utile. Premièrement, elle force l'utilisateur à saisir la profondeur, Deuxièmement, il est impossible d'appliquer les filtres pour la traversée afin de limiter les chemins explorés. Finalement, le résultat de cette fonction est très difficile à exploiter lors de la programmation en Java.

Donc, quel moteur utiliser ? :

Nous avons présenté à travers ce rapport, ce que chacun permet de réaliser pour les modèles graphes. Il est certain que pour certains cas d'usages il est nécessaire d'utiliser certains moteurs plutôt que d'autres. Si pour notre cas d'usage, nous avons besoin de plusieurs types de modèles, il est alors préférentiel d'utiliser ArangoDB ou OrientDB qui ont la spécificité d'être multi-modèle. Si nécessité de faire des traversées profondes dans le graphe alors il est préférable de privilégier Neo4j. Si nous souhaitons avoir plusieurs points d'accès physique à la base de données en mode maitre, il faut alors choisir ArangoDB ou OrientDB, puisque ce sont des modèles maitre/maitre. Si nous recherchons un moteur dont la prise en main est simple alors nous privilégierons Neo4j ou ArangoDB. Ainsi, nous dirions donc qu'il faut choisir le moteur en fonction du contexte. Il est important cependant de préciser que Neo4j est de loin la base de données graphe la plus populaire. Créateur des bases de données graphes, ils sont les plus développés dans ce type de modèle.

Conclusion

Ainsi, nous vous avons présenté trois moteurs de bases de données orientées graphes et avons mis en lumière leurs avantages et leurs inconvénients. Nous allons conclure en revenant brièvement sur notre expérience lors de la préparation de ce projet.

Pour la répartition du travail, nous avons réalisées des parties en commun et des parties de façon individuelle. Dans un premier temps, nous nous sommes réunies pour choisir le jeu de données et les moteurs adaptés à celui-ci. Nous nous sommes aussi mises d'accord sur les requêtes à effectuer. À partir de là, chacune s'est formée sur l'utilisation de son moteur et a implémenté les différentes requêtes. Pour la rédaction du rapport, chacune a rédigé les parties concernant son moteur et les parties communes ont été faite ensemble. Ce projet est le résultat de plusieurs semaines de travail car nous étions désireuses de découvrir les outils dans toutes leurs fonctionnalités et que nous avons rencontrés quelques difficultés.

À propos des difficultés, de manière générale, les bases de données graphes sont très différentes de tout ce que nous avons pu voir auparavant. Nous avons dû donc prendre une grande partie de notre temps à nous former sur les requêtes et le modèle graphe qui était nouveau pour nous. De plus, nous n'avons pas choisi des requêtes simples que ce soit du point de vue utilisateur ou analyste car nous voulions des cas d'usages qui soient intéressants et réalistes. Enfin, OrientDB a été particulièrement difficile à utiliser mais nous avons décidé de le garder pour mettre en lumière les différences entre les leaders dans les modèles graphes.

Ce projet a été un challenge pour nous mais a été très enrichissant et nous sommes contentes d'avoir pu expérimenter les modèles de graphes en détail qui aurons beaucoup à nous offrir dans les années à venir. En effet, Gartner estime que d'ici 2025, les technologies de graphes seront utilisées dans 80% des innovations dans le domaine des données et des analyses[2].

[2] *Gartner « Market Guide : Graph Database Management Solutions »* Merv Adrian, Afraz Jaffri 30 August 2022, disponible ici