Gestion des données au delà de SQL (NoSQL)

HAI914I

Encadré par François SCHARFFE

- Djamel BENAMEUR -

- Antoine BARBIER -

Mini projet :

« Implementation d'un mini moteur de requêtes en étoile »

Département Informatique

Faculté des Sciences - Université de Montpellier

FACULTÉ DES SCIENCES MONTPELLIER

UNIVERSITÉ DE MONTPELLIER

Table des matières

1 Présentation du projet :

Avec l'augmentation de 20% en moyenne des volumes des données dans les dernières années selon des études, la nécessité de traiter, classer, évaluer, analyser ces données est devenu un enjeu majeur dans notre société.

Dans le cadre de l'UE HAI914I, nous avons abordé la thématique de l'interrogation des données avec un mini-projet qui s'articule autour de l'implémentation d'un mini moteur de requêtes en étoile. L'objectif est d'implémenter l'approche hexa-store pour l'intégration des données RDF, ainsi que les procédures nécessaires à l'évaluation des requêtes en étoile (exprimées en syntaxe SPARQL).

Les différentes étapes consistent d'abord à construire un dictionnaire qui associe un entier à chaque ressource de la base RDF puis la création d'un index pour permettre l'évaluation des requêtes. Une fois ses structures créés, il y a l'étape de l'évaluation des requêtes. Pour finir, nous avons effectué des tests sur des valeurs plus au moins importantes pour observer le comportement qui se détache de ces données. Après l'analyse des résultats nous allons pouvoir tirer des conclusions sur le fonctionnement de notre programme.

Le projet est accessible sur gitlab : lien du projet

2 Lancement du projet

Il y a plusieurs possibilités disponibles pour pouvoir exécuter le programme du projet : soit en exécutant le fichier .jar, soit en exécutant le code source avec un IDE comme Eclipse, Visual Studio Code etc.

2.1 Exécution du programme

2.1.1 Avec l'exécutable .jar

Pour exécuter le programme et faire les tests sur celui-ci, la meilleure méthode la plus simple est d'exécuter le fichier .jar du projet. Pour cela, il suffit d'écrire dans le terminal :

```
java -jar <nom-du-fichier.jar> <options ...>
```

2.1.2 Avec VScode

Pour exécuter le code source sous VScode, il faut modifier le fichier .vscode/launch.json qui se situe dans le projet et placer les options du programme souhaité dans "args" comme dans l'exemple suivant :

```
"args": "-queries queries/sample\_query.queryset  -data data/sample\_data.nt"
```

2.1.3 Avec Eclipse

Pour exécuter le code source sous Eclipse, il faut configurer le RUN, en allant dans java application > Main, et placer les options dans les arguments du programme comme dans l'exemple suivant :

```
-queries queries/STAR\_ALL\_workload.queryset  -data data/100K.nt
```

2.2 Les options

Les différentes options de notre programme sont :

— **-queries <queryFile>** : le fichier qui contient les requêtes (option obligatoire)

— **-data <dataFile>** : le fichier qui contient les bases de données (option obligatoire)

— **-output <restultFolder>** : le fichier généré en sortie (option facultative, par défaut le fichier de sortie est *output/*)

— **-type-output <typeFileOutput>** : le type est *txt* ou *csv* (option facultative, par défaut le fichier est de type *csv*)

Nous avons aussi rajouté une option qui est l'option *clean*. Cette option permet de vider le fichier **output-historique.csv** et s'utilise de la manière suivante :

```
java -jar <nom-du-programme.jar> clean
```

3 Conception du programme

3.1 Structure globale

Explication du rôle de chacun des fichiers qui composent notre projet :

— **ConsoleColor.java** : fichier qui contient la classe qui définit de manière statique les couleurs utilisables sur la sortie standard du terminal.

— **DataInformations.java** : la classe DataInformations recense toutes les données statistiques et temps d'exécution du programme.

— **Dictionnaire.java** : la classe qui contient la structure de données du dictionnaire

— **Index.java** : la classe qui contient la structure de données de l'index

— **Main.java** : contient la classe main du programme

— **MainJena.java** : contient le programme Jena qui permet d'exporter les résultats jena pour que l'on puisse ensuite les comparer avec notre programme

— **ParserDatas.java** : contient la classe qui se charge de parser les données, de créer le dictionnaire et l'index (en utilisant les classes Dictionnaire et Index)

— **parserQueries.java** : contient la classe qui se charge de parser les requêtes, les évaluer et d'exporter les résultats.

3.2 Dictionnaire :

3.2.1 Structure de données du dictionnaire

Pour la structure de données du dictionnaire, nous avons fait le choix d'utiliser la structure *BiMap* qui permet d'accéder aux données par clef-valeur et par valeur clef avec la même complexité.

3.2.2 Problèmes rencontrés

La première version de notre dictionnaire utilisait la structure de données *HashMap<Integer, String>* afin d'associer un entier à une ressource de la base RDF. Le problème de ce choix est que

l'accès aux données du dictionnaire par clef-valeur et par valeur-clef n'est pas optimisé. En effet, pour accéder à une donnée par valeur-clef, on utilisait une fonction qui parcourrait toute la *HashMap* pour retrouver la donnée. On a donc mis à jour notre code pour utiliser la version actuelle du dictionnaire.

3.3 Index :

Pour l'index, nous avons créé une classe Index dans laquelle on manipule la structure de données suivantes :
HashMap<TypeIndex, HashMap<Integer, HashMap<Integer, ArrayList<Integer> > > >

Cette structure permet d'enregistrer les 6 types d'index (POS, OPS, SPO, PSO, SOP, OSP) et d'accéder très rapidement aux ressources grâce aux HashMap.

Pour créer l'index, on utilise le parseur qui parse le fichier de données. À chaque statement rencontré, on appelle la méthode add de la classe Index. Cette méthode se charge de remplir l'index en y enregistrant les 6 positions possibles pour le sujet, le prédicat et l'objet (soit les différents types d'index).
Pour travailler avec le type d'index, nous avons créée la structure de données suivantes :

Cette structure permet de connaître la position du sujet, du prédicat et de l'objet pour chaque type d'index.

3.4 Le parseur de requêtes et évaluation des requêtes

Pour cette étape, nous avons évalué chaque sous-requête d'une requête et nous avons fait l'intersection entre les différents résultats obtenus pour chaque sous-requête.

Pour évaluer une sous-requête, on recherche les résultats grâce à l'index. Pour que la recherche soit optimale, on utilise le bon type d'index en fonction de la position de la variable *?v0* dans la requête. Par exemple dans le cadre de ce projet on utilisera toujours l'index de type POS ou OPS étant donnée que le la variable *?v0* est forcément le sujet (voir consigne du projet). Ensuite on fait le choix entre un des deux types d'index possible, en prenant de manière arbitraire un index, étant donnée qu'on a qu'une seule variable dans notre projet.

```
enum TypeIndex {
    SPO(0, 1, 2), SOP(0, 2, 1), PSO(1, 0, 2), OSP(1, 2, 0), POS(2, 0, 1), OPS(2, 1, 0);
    public int S, P, O;
    TypeIndex(int S, int P, int O) {
        this.S = S;
        this.P = P;
        this.O = O;
    }
}
```

FIGURE 1 – La structure de données de l'index

3.5 Explication des calcules des temps d'exécutions

Voici la liste des différents temps d'exécution ainsi que l'explication sur leurs méthodes de calculs :

— **temps de création du dictionnaire**

— **temps de création des index**

— **temps de création des différentes exportation** : on calcule le temps pris pour chaque exportation de données vers un fichier.

— **temps de lecture des données** : le temps de lecture des données est calculé en prenant le temps total du parser des données et en enlevant la création du dictionnaire et de l'index, afin d'obtenir seulement le temps de lecture.

— **temps de lecture des requêtes** : le temps de lecture est calculé en prenant le temps total du parser de requête et lui enlevant le temps pris pour l'évaluation des requêtes, des exports ...

— **temps total d'évaluation du workload** : le temps de workload est calculé en prenant le temps d'exécution de chaque requête et en additionnant ses temps pour obtenir le temps total d'évaluation.

— **temps total (du début à la fin du programme)** : le temps total est l'addition de tous les temps d'exécution précédents (on ne prend pas en compte le reste car c'est négligeable, seulement quelque millisecondes qui concernent quelques print sur la sortie standard, des allocations ...)

3.6 Exportation des résultats

Le dossier où seront exportés les résultats est par défaut le dossier nommé *output*. Ce dossier peut être changé en passant le dossier souhaité en option du programme. Dans ce dossier, un sous-dossier est créé. Ce sous-dossier porte le nom structuré de la manière suivante : la chaîne de caractères "output-" suivie du nom du fichier de données, suivie du nom de fichier de requêtes.

Dans le sous-dossier créé après l'exécution du programme avec un fichier de données et d'un fichier de requêtes, on obtient 4 fichiers après la fin de l'exécution du programme. Les fichiers sont les suivants :

— **Dictionnaire.txt** : ce fichier est le contenue de la structure de données du dictionnaire et permet d'en avoir un aperçu.

— **Index.txt** : ce fichier est le contenue de la structure de données du index et permet d'en avoir un aperçu.

— **project-result** : ce fichier est un fichier de type *csv* ou *txt* (par défaut le type est *csv*). Le fichier *csv* contient sur une seule ligne : le numéro de la requête, la requête, le nombre de résultat de la requête et le résultat. Le fichier *txt* est écrit d'une manière plus visuel que le fichier *csv* et contient les mêmes informations que le *csv*.

— **Informations.csv** : ce fichier regroupe les différentes informations obtenues après l'exécution du programme. Il contient des statistiques et des temps d'exécution.

Dans le dossier *output/* , on retrouve le fichier **ouput-historique.csv** qui regroupe toutes les informations sur l'exécution du programme. Dans ce fichier *csv* on retrouve les informations suivantes :

— **colonne 1** : Date et heure d'exécution du programme

— **colonne 2** : nom du fichier de données

— **colonne 3** : nom du dossier des requêtes

— **colonnes suivantes** : différentes statistiques et temps d'exécution.

Après chaque exécution du programme, le contenu du fichier **Informations.csv** est aussi sauvegardée dans le fichier **ouput-historique.csv**, il garde l'historique de toutes les exécutions du programme et peut-être vidé grâce à l'option *clean* en entrée du programme.

4 Interprétation des résultats et analyse du projet final

Pour effectuer les tests, nous avons d'abord généré des données pour 100K, 500K, 2000K, des requêtes de taille 1200, 5k, 10k. On lance sur les trois machines des tests et on compare les résultats. On va exécuter différents tests avec des données nettoyées, c'est-à-dire que l'on supprimera les requêtes doublons, car elle fausse le nombre de requêtes différentes qui sont testées. On a pu voir que sur certains fichiers de requête, 20% d'entre elles, est présentes plusieurs fois. De plus, on va prendre des jeux de données assez importants afin d'avoir le moins de requêtes possible avec zéro résultats.

4.1 Plan du processus d'analyse des résultats

Pour l'interprétation des résultats, nous allons d'abord récupérer tous les résultats de nos différents tests, les analyser et repérer les différents facteurs qui font varier les temps d'exécution. La création de différents histogrammes nous permettra de faire des analyses sur le choix des systèmes (hardware et software) à utiliser et des différents paramètres à prendre en compte pour l'interprétation des résultats du projet.

4.1.1 Explication du processus

Les étapes du processus sont :

— la récupération des résultats CSV de différentes exécutions du programme

— la création d'un fichier EXEL pour pouvoir générer des histogrammes.

— prendre les histogrammes les plus pertinents.

— essayer de tirer des conclusions selon les comportements de notre programme en fonction des différents paramètres d'exécutions.

4.1.2 Explication de la préparation des banc d'essais

Pour la génération de données, nous avons utilisé WATDIV qui permet de générer des données en respectant des règles de génération qui est très rapide et très pratique mais qui possèdes aussi certaines limites comme un nombre important de doublons sur des générations de nombres important et aussi le nombre de requêtes sans réponses.

WATDIV nous fournis différents fichiers de requêtes séparés. Pour exécuter nos test de manière rapide, nous avons créé un script python **exec-on-folder.py** qui exécute notre programme avec un fichier de données et qui prend en paramètre un dossier contenant des fichiers de requêtes. Ce script exécute

le programme du projet sur chacun des fichiers requêtes et nous permet de générer tous les fichiers résultats en une seule commande.

4.2 Le système utilisé pour les test

4.2.1 Hardware et software

Au niveau du hardware, nous avons utilisé trois machines différentes pour réaliser nos testes :

Machine 1 :
Mémoire : 15,5 GiB
Processeur : Intel® Core™ i5-7200U CPU @ 2.50GHz × 4

Machine 2 :
Mémoire : 7.59 GiB
Processeur : Intel® Core™ i5-8200U CPU @ 8GHz

Machine 3 :
MacBook Pro (13-inch, 2017, Two Thunderbolt 3 ports)
Mémoire : 8 Go 2133 MHz LPDDR3
Processeur : 2,3 GHz Intel Core i5 double cœur

Au niveau du software, on utilise des machines sous MacOS Monterey et Ubuntu 20.04. Pour les tests nous avons utilisé WatDiv pour générer des bases de données et des jeux de requêtes.

4.2.2 Cold et warm

Les mesures *cold* sont intéressantes pour comprendre les métriques qui ne sont effectués qu'une seule fois, comme pour la création du dictionnaire et la création de l'index.

Les mesures *warm* sont intéressantes dans le cadre des métriques d'évaluation comme les opérations de recherche dans l'index et dans le dictionnaire.

4.2.3 Variations de la mémoire et de la taille des données

Lorsque l'on utilise des fichiers avec un nombre important de données et de requêtes, il est indispensable de faire varier la mémoire. En effet, comme le dictionnaire et l'index sont stocké dans la mémoire RAM, il faut augmenter la mémoire utilisée afin de faire fonctionner le programme.

4.3 Analyse des temps d'exécution

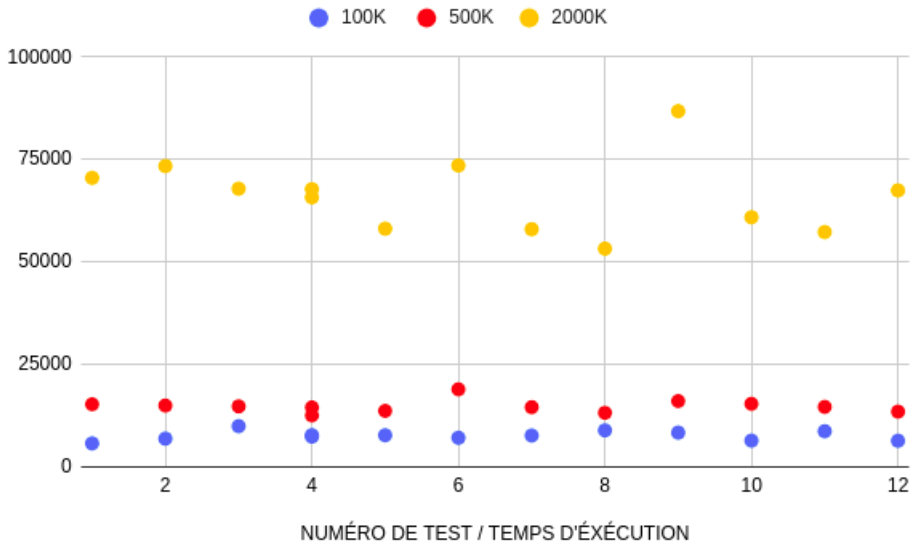

FIGURE **2** – Temps d'exécution selon le numéro de test

On remarque sur le graphe ci-dessus, que sur les fichiers de taille 100k, il existe un nombre important de requêtes sans réponses car la taille de la donnée n'est pas importante pour répondre à ces requêtes et la variation des temps sur les 12 testes reste assez stable.
Sur des fichiers de taille 500k, on remarque que les temps d'exécution varie un peu plus car le nombre de requêtes avec zéro résultat diminue, ce qui augmente le temps global avec forcement plus d'opérations d'écritures des résultats dans le CSV et aussi le temps d'exécution.
Pour finir, sur les fichiers à 2 millions de lignes, comme les données augmentent considérablement pour un même nombre de requêtes, forcément le nombre de requêtes sans réponses divisent par 2 ou 3 selon les situation et donc les temps global explosent par rapport au fichiers de taille 100K, 500K car il parcours plus de données pour essayer de trouver des réponses aux requêtes mais ça reste des temps très abordables à notre échelle avec un pic à 86000 ms car le temps d'évolution du workload est trois fois supérieur à la moyennes des autres testes.

4.3.1 Comparaison entre différentes machine sur notre programme

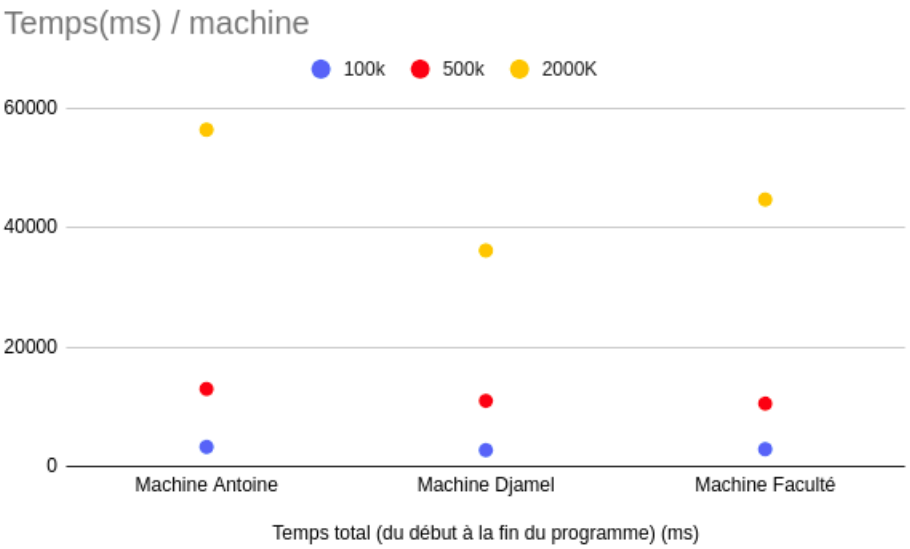

FIGURE 3 – Le temps total d'évolution pour chaque machine selon la taille des données

Sur ce graphe, on remarque que plus la mémoire RAM de la machine est puissante, plus le temps d'exécution est bas. En effet, les données sont stockées en mémoire vive (RAM) et on remarque assez vite que sur la machine d'Antoine que les temps d'exécution sont deux à trois fois plus longs que sur ceux que la machine de la FDS. La RAM joue un rôle important dans la rapidité d'exécution de notre programme.

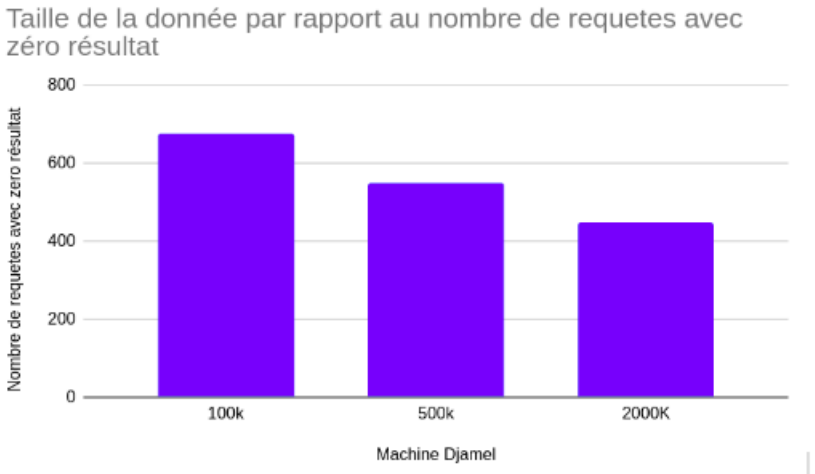

FIGURE 4 – Évolution du nombre de requêtes sans résultat selon la taille des données

Sur ce graphe on remarque que plus on a enrichi notre modèle avec des données et moins on a des requêtes avec zéro résultat.

On peut voir que sur le tableau ci-dessous, les données de taille de 100.000 lignes, le nombre de requêtes avec zéro résultats est de 675 alors que sur les données de taille 500k, on arrive à 549 requêtes avec zéro résultats. Pour 20 millions de données, on arrive à seulement 448 requêtes avec zéro résultats.

Nom du fichier de données	data/100K.nt	data/500K.rdf	data/2000K.rdf
Nom du dossier des requêtes	queries/STAR_ALL_workload.queryset		
Nombre de Requêtes avec zéro résultat	675	549	448

4.4 Vérification des résultats

Dans un premier temps, pour vérifier nos résultats nous avons d'abord effectué des tests sur des données de quelques lignes pour s'assurer d'obtenir des réponses correctes. Nous avons ensuite comparé nos résultats avec le programme d'autres groupes pour s'assurer d'avoir le bon nombre de requêtes sans résultat, de requêtes doublons ... Nous avons aussi comparé aléatoirement quelques réponses à nos requêtes.

Ses tests étaient de simple aperçu pour savoir si on se diriger dans la bonne direction mais n'était pas assez fiable pour montrer que notre programme est correct. Pour vérifier l'exactitude de nos résultats nous avions besoin d'un moyen plus efficaces que la comparaison avec d'autres groupes.

4.4.1 Utilisation de Jena

Pour la vérification des résultats de notre projet, nous avons utilisé jena. Pour cela, nous avons créé un deuxième programme qui utilise jena et exporte les résultats de jena dans le même format de données que notre projet.

Pour vérifier l'exactitude de notre projet nous avons comparé les fichiers de résultats de nos requêtes avec les résultats de jena. Pour faire cette comparaison nous avons utilisé la commande Linux :

```
diff [options] [file1] [file2]
```

Après la vérification sur différentes tailles de données et nombre de requêtes, les fichiers résultats de notre projet et ceux de jena sont identiques.

5 Conclusion

Dans notre projet, nous avons mis l'accent sur la recherche de structures de données optimisées, l'écriture d'un code propre avec une structure correcte afin d'obtenir des temps d'exécution raisonnables dans l'optique d'effectuer des tests de notre moteur de requêtes en étoile sur un nombre conséquent de données et de requêtes. Grâce à l'utilisation de Jena, nous avons pu voir l'exactitude de nos résultats et donc avoir la possibilité de savoir que notre programme est fiable pour les conditions qui nous ont été imposé par la consigne du projet.

Pour aller plus loin dans ce projet, on pourrait améliorer le moteur de recherche pour prendre en compte des requêtes à plusieurs inconnus, et des requêtes où la variable n'est pas seulement le sujet. On pourrait aussi étudier d'autres méthodes d'évaluation des requêtes et les comparer afin de voir lesquelles sont les plus efficaces.