Université de Montpellier

Rapport de projet NoSQL
Partie I - évaluation des requêtes en étoile

El Houiti Chakib
Kezzoul Massili

28 novembre 2021

Introduction

Objectif du projet

L'objectif du projet est, dans un premier temps, de développer un mini-moteur de requête en étoiles en utilisant l'approche *hexastore*. Puis dans un second temps, analyser les performances de cette approche en fonction de plusieurs jeux de données. L'approche *hexastore* consiste à utiliser six index pour stocker les données. Cette méthode permet d'évaluer des requêtes de façon plus efficace.

La partie d'analyse des performances sera traitée dans un futur rapport.

Environnement de développement

Le projet a été développé sur notre propre environnement de travail. Nous avons choisi de ne pas utiliser l'environnement *Eclipse*, car ce dernier est très lourd et peine à s'exécuter correctement sur nos machines. Afin de palier au problème des dépendances, nous avons utilisé *Maven* qui est un outil de gestion et d'automatisation de production des projets logiciels *Java*. Nous avons aussi utilisé le logiciel *Make* dans le but de faciliter au mieux l'initialisation de l'environnement (téléchargement des dépendances), la compilation des fichiers sources, l'exécution du programme ainsi que la création du fichier *jar* qui sera ensuite utilisé pour l'exécution du programme. Aussi, afin de faciliter la coopération avec mon binôme, nous avons utilisé *Git* à travers le service *Github* pour la gestion du code.

Le programme est écrit en *Java*[1]. Le projet utilise deux bibliothèques externes :

— *rdf4j* pour lire les données rdf et requêtes sparql

— *jena* qui sera utilisée pour comparaison dans la phase d'analyse des performances

Structure du projet

Pour une meilleure compréhension de l'environnement du projet, voici ci-dessous différentes informations sur les différents fichiers et répertoires du projet.

src/ Répertoire contenant les fichiers sources du projet.

target/ Répertoire contenant les fichiers générés (les *.class*) durant la production ainsi que les dépendances lier au projet.

data/ Répertoire contenant les différents jeux de données.

Jeu de données Les *fichiers.nt* contiennent les données au format N3.

Requêtes Les fichiers.queryset contiennent plusieurs requêtes au format SPARQL.

output/ Répertoire contenant toutes les sorties du programme.

README.md Fichier expliquant la manière d'utiliser le programme (initialisation, compilation et exécution). Referez-vous à la section *Utilisation* de ce dernier pour plus d'informations.

Makefile Fichier qui spécifient les commandes de compilation, initialisation et autres.

pom.xml Fichier qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, etc.).

1. La version de java utilisé lors du développement du projet est la version 11.0

1 Modèlisations et implémentation

1.1 Chargement des données

1.1.1 Jeu de données

La première étape consiste à charger les données en mémoire vive. Les données sont dans le format N3. Pour les charger, nous utilisons la méthode *parseData()* de la classe principale qui, pour chaque triplet RDF du fichier, fait appel à la méthode *handleStatement(Statement)* de la classe *MainRDFHandler*. Cette méthode ajoute le triplet dans les six index. Le sujet, prédicat ainsi que l'objet du triplet sont quant à eux ajouté dans le dictionnaire où un entier unique est associé à chaque URI. Évidemment, les doublons sont ignorés.

1.1.2 Dictionnaire

Le dictionnaire associe pour chaque URI un entier unique. Cet entier est utilisé pour identifier les URI dans les index. Cela permet, d'une part, de ne pas avoir à comparer les URI en tant que chaîne de caractère. Opération qui est plus coûteuse. Mais aussi, puisque la taille d'un entier est inférieure à la taille d'une chaîne de caractère, de réduire la taille finale des index.

Concrètement, le dictionnaire est implémenté par une instance de la classe *HashBiMap<String, Integer>*. La classe *HashBiMap* est une classe de mapping entre deux types de données. À la différence de la classe *HashMap*, la classe *HashBiMap* ajoute une contrainte d'unicité sur les valeurs de ses éléments. Cette contrainte permet d'activer une *vue inverse* du dictionnaire. C'est-à-dire, à partir de l'entier unique associé à une URI, on peut retrouver l'URI associée en un temps constant.

```
0  public static HashBiMap<String, Integer> dict = HashBiMap.create();
```

1.1.3 Index

Les index sont au cœur de l'approche *hexastore*. Cette approche consiste à définir six index. Chaque index contient tous les triplets de la base de données suivant un certain ordre sur ces termes. Ainsi l'index suivant l'ordre *Sujet* > *Prédicat* > *Objet* peut-être représenté par un tableau ou la première colonne contient les identifiants des sujet du triplet, la deuxième colonne contient les prédicats et la dernière contient les objets. Les tuples sont ordonnés d'abord sur la première colonne, puis sur la deuxième ensuite sur la dernière.

Cette approche permet de minimiser le temps de recherche de l'existence d'un tuple dans la base. Donc si on recherche un tuple où le sujet est la variable (du style : *?x habite Montpellier*), alors on utilisera un des deux index qui met le sujet en dernier. Ici, on utilisera par exemple l'index suivant l'ordre *Objet* > *Prédicat* > *Sujet* pour trouver les sujets qui habitent à Montpellier.

Dans notre implémentation, on a commencé par définir une classe *Index* qui représente un seul index. Cette classe a comme premier attribut *order*, une chaîne de caractère contenant l'ordre dans lequel sont stockés les termes de chaque triplet. Par exemple pour l'ordre *Sujet* > *Prédicat* > *Objet*, l'attribut *order* vaut : *"spo"*. Cela nous permet de savoir dans quel type d'index on est.

L'index, à proprement parler, est définit comme une imbrication de deux instances de la classe *HashMap* suivit d'une instance de la classe *TreeSet*. La première instance de la classe *HashMap* est un mapping entre l'identifiant du premier terme t_1 et une autre instance de la classe *HashMap*. Cette dernière, quant à elle, est un mapping entre l'identifiant du second terme t_2 et une instance de la classe *TreeSet*. Cette instance contient l'ensemble ordonné des derniers termes du triplet composé de t_1 et t_2.

```
0  private HashMap<Integer, HashMap<Integer, TreeSet<Integer>>> indexMat;
```

En utilisant cette structure, la recherche au sein d'un index se fait par un simple *TreeSet indexMat.get(int, int)* qui a une complexité constante.

1.2 Évaluation des requêtes

Le programme prend en paramètre un fichier contenant toutes les requêtes à exécuter. Dans cette partie, nous détaillerons la partie d'évaluation des requêtes. La lecture et l'évaluation des requêtes se font de façon parallèle. C'est-à-dire que le programme interprète une requête puis l'évalue et donne son résultat avant de passer à la suivante.

1.2.1 Lecture

La lecture et l'interprétation d'une requête se font par la méthode *parseQuery* de la classe *SPARQLParser* de la bibliothèque *rdf4j*. Elle retourne un objet de la classe *ParsedQuery* qui permet d'accéder à chaque pattern de la requête ainsi que ses attributs grâce à différentes méthodes qu'on ne va pas détailler ici.

Les requêtes considérées ici sont des requêtes en étoiles où la variable est toujours le sujet. Donc on a décidé de n'utiliser que l'index *"pos"*.

1.2.2 Évaluation

Afin d'évaluer une requête, on a défini la méthode *processAQuery* qui prends en paramètre une instance de la classe *ParsedQuery* et se charge d'évaluer la requête. Pour cela, on parcourt les patterns (Triplets qui sont de la forme *?x pred obj*) de la requête un à un et on recherche, au sein de notre index, les sujets apparaissant avec l'objet et le prédicat de ce pattern.

On réalise une intersection entre le résultat du premier pattern et le résultat du second, ainsi de suite jusqu'à la fin de la requête. Puisque nous manipulant à ce stade des ensembles ordonnés, cette intersection est faite grâce à un *sort-merge-join* afin d'optimiser le temps d'évaluation finale.

```
0  SortedSet<Integer> merge = new TreeSet<Integer>();
1
2  while (!listA.isEmpty() && !listB.isEmpty()) {
3    if (listA.first() == listB.first()){
4      merge.add(listA.first());
5      listA.remove(listA.first());
6      listB.remove(listB.first());
7    } else if (listA.first() < listB.first()) {
8      listA.remove(listA.first());
9    } else {
10     listB.remove(listB.first());
11   }
12 }
13 return merge;
```

Listing 1 – "Sort merge join de deux listes"

1.3 Résultats

Les fichiers résultant de l'exécution du programme sont mis dans le répertoire spécifié avec l'option *-output*.

Réponses du moteur La sortie du programme est un fichier CSV nommé *queryResult.csv*, contenant deux attributs : le numéro de la requête et les résultats de la requête sous forme d'une liste contenant l'ensemble des réponses de cette dernière. Cette exportation est faite si l'option *-export_query_results* est spécifié. Cette option invoque la méthode *exportQueryResults()* qui permet de transformer la liste des réponses de toutes les requêtes en un fichier CSV.

Le numéro de la requête i dans le fichier CSV représente la requête numéro i dans le fichier des requêtes. On n'a pas exporté la requête en elle-même, car on ne stocke pas les requêtes lors de leurs évaluations. Donc on a préféré mettre le numéro dans le fichier CSV, au lieu de la requête en elle-même.

Temps d'exécution Les statistiques de notre programme sont exportées dans le fichier *stats.csv* contenant les informations suivantes au format CSV :

- Nom du fichier des données.
- Nom du fichier contenant les requêtes.
- Nombre de triplets RDF
- Nombre de requêtes.
- Temps de lecture des données (ms)
- Temps de lecture des requêtes (ms)
- Temps de création du dictionnaire (ms).
- Temps de création des index (ms).
- Nombre d'index.
- Temps total d'évaluation du workload (ms).
- Temps total de l'exécution du programme (ms).

2 Conclusion

2.1 Utilisation du programme

Afin de faciliter les tests de notre programme, il est exécutable via ligne de commande. Après la construction du fichier *jar*, on peut exécuter notre programme avec les options suivantes :

```
java -jar rdfqengine.jar
    -data "chemin/vers/fichier/données.nt"
    -queries "chemin/vers/fichier/queries.queryset"
    -output "chemin/vers/dossier/sortie/"
```

Notre Programme reste compilable et exécutable via le *Makefile* avec **make compil** et **make run**. Les arguments sont paramètrable au sein même du *Makefile*.

NB : N'oubliez pas d'initialiser l'environnement en exécutant le commande **make init**.

2.2 Perspectives

Dans la seconde partie du projet, un travail de validation des résultats et d'évaluation des performances sera réalisé.