Université de Montpellier

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

El Houiti Chakib
Kezzoul Massili

26 novembre 2021

Introduction

Objectifs du projet

L'objectifs 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 plusieur jeux de données. L'approche hexastore consiste à utiliser six indexes pour stocker les données. Çela permet d'évaluer des requêtes de façon plus efficace.

La partie d'analyse des performances sera traitée dans le rapport suivant.

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'initialistion 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 meilleur compréhension de l'environnement du projet, voici ci-dessous différentes informations sur les différents fichiers et répertoire 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 toute les sorties du programme.

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

Makefile Fichier qui spécifient les commandes de compilation, initialistion 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 premier 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 indexes. Le sujet, prédicat ainsi que l'objet du triplet sont quand à eux ajouté dans le dictionnaire où un entier unique est associé à chaque URI. Évidemment, les doublons sont ingorés.

1.1.2 Dictionnaire

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

Concrètement, le dictionnaire est implémenté par une instace 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 des 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 Indexes

1.2 Évaluation des requêtes

1.2.1 Structure

1.2.2 Lecture

1.2.3 Évaluation

matchPattern

Intersection de chaque pattern

1.3 Résultats

Structure Fichier CSV en sortie du programme.

Temps d'exécution Calcul des temps d'exécution.

2 Conclusion

2.1 Utilisation du programme

Parler des différents arguments de la ligne de commande ainsi que de leur fonctionnement.

2.2 Perspectives

Évaluer le temps d'exécution des requêtes.