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

Les indexes sont au coeur de l'approche *hexastore*. Cette approche consiste à définir six indexes. Chaque index contient tout 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 indentifiants des sujet du triplet, la deuxième colonne contient les prédicat et la dernière contient les objets. Les tuples sont ordonnés d'abords sur la première colonne, puis sur la deuxième ensuite sur la dernière.

Cette approche permet de minimiser le temps de recherche de l'existance d'un tuple dans la base. Donc si on recherche un tuple oû le sujet est la variable (du style : *?x habite Montpellier*), alors on utilisera un des deux indexes qui met le sujet en dernier. Ici, on utilisera par exemple l'index suivant l'ordre *Objet* > *Prédicat* > *Sujet* pour trouver les sujets qui habite 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 exmple pour l'ordre *Sujet* > *Prédicat* > *Objet*, l'attribut *order* = *"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'indentifiant du premier terme t_1 et une autre instance de la classe *HashMap*. Cette dernière, quant à elle, est un mapping entre l'indentifiant 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'index se fait par un simple *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 fait 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'interprètation d'une requête se fait par la méthode *parseQuery* de la classe *SPARQLParser* de la bibliothèque *rdf4j*. Elle retourne un objet de la classe *ParsedQuery* qui permet d'acceder à chaque pattern de la requête ainsi que les attributs de ce pattern grâce à différentes méthodes qu'on ne vas pas détailler ici.

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

1.2.2 Évaluation

Afin d'évaluer une requête, on a définit la méthode *processAQuery* qui prends en paramètre une instance de la classe *ParsedQuery* qui se charge d'évaluer la requête. Pour çela, on parcours les patterns (triplets) 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 avec le 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.

1.3 Résultats

Structure La Sortie du Programme est un fichier CSV, contenant deux attributs, le numéro de la requête et les résultats de la requête sous forme d'une liste contenant toutes les réponses de cette dernière. Cette exportation est faite grâce à l'option *exportQueryResult*, cette option évoque la méthode *exportQueryResult()* 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 requête. On a pas exporter la requête en elle même, car on ne stocke pas les requête lors de leurs évaluation. Pour faire ça, il faut les stocker au paravant, ou de relire le fichier des requête lors de l'exportation des résultats, donc on a préféré de mettre le numéro dans le fichier CSV, au lieu de la requête en elle même.

Temps d'exécution Les statistiques de notre programme sont exportés dans un fichier CSV contenant les informations suivantes :

- 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 nécessaire pour la création du dictionnaire (ms).
— temps nécessaire pour la création des indexes (ms).
— nombre d'indexes.
— temps total d'évaluation du workload (ms).
— temps total de l'exécution du programme (ms).

Ces informations sont exportés à l'aide de la méthode *exportStats()* et toutes ces informations sont disponibles.

2 Conclusion

2.1 Utilisation du programme

Pour faciliter les tests de notre programme, il est exécutable via ligne de commande, grâce à notre *Makfile* et la commande **make jar** qui permet de construire un exécutable .jar, aprés la construction du .jar on peut exécuter notre programme avec les options suivantes : java -jar rdfqengine.jar -data "chemin_vers_fichier_données" -queries "chemin_vers_fichier_queries" -output "chemin_vers_dossier_sortie".

Notre Programme reste compilable et exécutable via le *Makfile* avec **make compil** et **make run**, mais il faut changer les options à partir du *Makfile*.

2.2 Perspectives

On a voulu faire une méthode de validation qui compare notre évaluation des requêtes avec une autre évaluation déjà validée (en utilisant Jena API par exemple).