Rapport HAI914I
Un moteur d'évaluation de requêtes en étoile

Eric GILLES & Morgan NAVEL

21 décembre 2024

1 Introduction

L'objectif de ce projet est de créer un mini-moteur de requêtes en étoiles (exprimées en SPARQL) en utilisant un **Hexastore** pour pouvoir interroger des données RDF. Cela implique la conception d'un dictionnaire compact, l'indexation des données et l'évaluation performante des requêtes. Le prototype sera comparé à **Integraal** pour valider sa correction et analyser ses performances, tout en respectant les bonnes pratiques de développement en Java.

2 Hexastore

La prochaine étape du projet est développée les fonctionnalités de notre **Hexastore**, qui est notre élément nous permettant de faire persister nos données sous formes de triplets RDF (Sujet, Prédicat, Objet). L'**Hexastore** repose sur une multiple indexation, créant six permutations des triplets, ce qui optimise l'accès aux données pour des requêtes en étoile. Cette approche garantit des temps de réponse rapides et une grande flexibilité pour le traitement des requêtes SPARQL.

2.1 Dictionnaire

Pour l'implémentation du dictionnaire de l'**Hexastore**, qui permet d'encoder et de décoder chaque ressource de la base de données, nous avons décidé de créer une classe *Dictionnary* qui contient une **Hashmap** d'entier et de termes RDF (Term) avec les termes en clé et les entiers en valeurs. Nous avons créé dans cette classe différentes méthodes d'ajout et de récupération des données, cela nous a permis d'accéder à une valeur avec sa clé et vice-versa ou d'ajouter un nouveau terme RDF au dictionnaire. Cette classe est ensuite utilisée dans l'**Hexastore** pour l'ajout ou la recherche de données.

2.2 Index

La première chose à faire pour implémenter un **Hexastore** est de construire 6 index. Chaque index est une permutation de nos triplets RDF qui nous permettant d'interroger nos données selon les paramètres données dans la requête, grâce à cela, nous pouvons gagner facilement en performance en choisissant les variables le plus tard possible dans notre recherche pour filtrer les résultats.

Chaque index utilise le même format, mais l'ordre des index changera. Pour notre implémentation en Java, nous avons fait le choix d'utiliser des **HashMap** pour nos deux premières valeurs, et un **Set** pour la dernière valeur.

```
private final HashMap<Integer, HashMap<Integer, Set<Integer>>> spo = new HashMap<>();
```

FIGURE 1 – Exemple d'implémentation d'un des index de l'Hexastore, spo
Tous les autres index (pso, osp, pos, sop et ops) ont le même format.

3 Évaluation de requêtes

Après la construction du dictionnaire et des index terminés, nous avons tous les outils nous permettant de stocker des données, maintenant, il ne reste qu'à restituer ces données. Une requête est construite de différent **RDFAtom** dans notre clause WHERE, représentant nos triplets RDF.

Nous allons ensuite détailler les différentes méthodes complétées de la classe *RDFHexaStore* implémentant l'interface *RDFStorage*, classe Contrat pour un système de stockage de données RDF. Ainsi *RDFHexaStore* représente l'implémentation d'un **Hexastore** pour stocker des **RDFAtom**.

3.1 Match Atom

Nous devons implémenter une méthode, qui à partir d'un **RDFAtom**, récupère les résultats selon les valeurs associées au triplet.

La première chose est d'évaluer quelles sont les variables et les constantes de notre triplet, pour ainsi déduire l'index optimal à utiliser. Pour un triplet RDF, nous pouvons récupérer chaque valeur du triplet représenté par la classe **Term**. Cette classe est très utile, car elle contient une méthode "isVariable" qui répond vrai si c'est une variable sinon renvoie faux. Avec ceci, nous pouvons donc déterminer l'index à utiliser, il ne nous reste donc plus qu'à évaluer la requête selon cet index. Puisque nous connaissons les variables, et donc les index, nous pouvons donc implémenter différentes méthodes en fonction du nombre de variables. Pour chaque méthode, on donne en paramètre les index que nous connaissons, et on peut récupérer toutes les substitutions de variables possibles.

3.2 Match StarQuery

La requête **StarQuery** consiste à effectuer un appariement avec plusieurs RDFAtom. Pour cela, on récupère chaque RDFAtom via l'objet **StarQuery** passé en paramètre. Pour chaque atome, on évalue les substitutions possibles, puis on les compare avec les substitutions précédentes. Cette comparaison se fait simplement par une intersection entre les

deux ensembles de résultats. On vérifie pour chaque élément des deux groupes (substitution précédemment possible, et les résultats de l'atome courrant) si leurs informations sur les mêmes variables sont cohérentes. Si une variable a une valeur différente dans les deux groupes, elles sont incompatibles et ne peuvent pas être combinées. Si toutes les informations sont compatibles, on crée un nouveau groupe qui rassemble ces éléments en une seule association unifiée. À la fin, on obtient les substitutions qui fonctionnent pour tous les atomes de la requête.

3.3 Add Atom, Get Size et Get Atoms

Pour les méthodes suivantes, on se contentera d'expliquer leur fonctionnement principal :

- **Add Atom** : cette méthode permet d'ajouter un triplet RDF dans l'**Hexastore**. Le triplet, s'il est valide, est d'abord encodé à l'aide du dictionnaire, puis chaque index correspondant est mis à jour pour garantir un accès rapide à travers toutes les permutations possibles.
- **Get Size** : retourne le nombre total de triplets enregistrés dans l'**Hexastore**. Elle renvoie un compteur interne (size) mis à jour lors de chaque insertion valide.
- **Get Atoms** : cette méthode retourne tous les triplets RDF stockés sous leur forme d'origine. Elle identifie chaque triplet encodé et décode les indices des triplets pour obtenir les termes RDF associés afin de les convertir en objets RDFAtom.

4 Tests

Dans cette partie, nous allons évaluer notre moteur de requête, afin de vérifier son comportement, sa complétude, sa correction et ses performances.

Nous avons testé et vérifié que le dictionnaire ainsi que les méthodes vu précédemment (*Match(Atom)*, *Match(Query)*, etc) de **Hexastore** fonctionne bien dans tous les cas possibles avec un taux de couverture de 100%.

4.1 Complétude & Correction

Nous avons établi une méthode de test qui utilise Integraal. Dans cette méthode, on y retrouve deux tests qui utilisent l'ensemble des requêtes et atoms des deux datasets (sous le dossier "data"). On obtient donc dans les deux cas des résultats très bons, avec aucun échec, ce qui montre que la complétude de notre moteur de requête.

4.2 Comparatif avec Intregraal

Nous avons, en plus des tests de complétude et de correction, effectué quelques comparatifs de performance. Nous avons pour chaque requête évaluée le temps d'exécution (pour les deux datasets), et établie une moyenne pour **Integraal** et notre moteur. Notre moteur de recherche est en moyenne 2 à 3 fois plus rapide sur les petites instances de données, cependant en ce qui concerne les instances beaucoup plus grosses, **Integraal** est bien plus efficace. En effet, ces résultats sont assez prévisibles, car **Integraal** est un outil développé et optimiser pour les requêtes. Il faudrait alors trouver des optimisations pour améliorer le temps d'exécution. Une solution serait d'exécuter de manière intelligente les atomes des requêtes afin d'être très sélectif dès le début pour réduire les possibilités.