UNIVERSITÉ DE MONTPELLIER
FACULTÉ DES SCIENCES

HMIN340 - Mini Projet
Master 2 Informatique

Mini Projet

RDF Engine

Étudiants :
Fabien Feraud
Florian Novellon

Encadrant :
Federico Ulliana

Année universitaire 2018-2019

Table des matières

1 Introduction

Dans le cadre de l'UE HMIN340 Nouvelles Approches pour la Persistance des Données nous avons implémenté une des approches pour la persistance des données RDF vues en cours ainsi que les procédures nécessaires pour l'évaluation de requêtes SPARQL de type "étoile".

Pour atteindre cette objectif, 3 systèmes nous ont été proposés : Hexastore, Columnstore et Graphstore. Notre choix s'est porté sur le système Hexastore car il nous a semblé plus simple à mettre en place et plus adapté au langage utilisé : Java.

Nous allons maintenant voir les différentes étapes que nous avons du réaliser pour implémenter ce systeme.

2 Le dictionnaire

Le dictionnaire permet le stockage de toutes les ressources d'une base RFD en associant chaque élément à un entier unique. De ce fait on peut remplacer toutes les ressources de la base par un entier puis se référer au dictionnaire pour identifier l'élément.

Dans notre système, l'ensemble des ressources est stocké en premier temps dans une HashSet. Un HashSet permet d'avoir la certitude de ne pas avoir de doublon. Cette collection permet le stockage unique des éléments afin d'en faire l'inventaire.

Le dictionnaire est également munie de deux HashMaps. Un HashMap est une structure donnée permettant d'avoir un accès très rapide à un objet grâce à une référence (l'entier unique par exemple). Ici nous utilisons deux HashMaps afin d'avoir un accès à l'association ressource-entier dans les deux sens.

Pour le remplissage du dictionnaire nous utilisons une classe annexe (une spécialisation de RDFHandlerBase) qui dans un premier temps ajoute tous les éléments lu à l'ensemble (le HashSet), puis une fois terminé, remplis les deux HashMaps en associant chaque élément à un entier.

3 L'index

Les index ont pour but de rendre l'évaluation des requêtes plus efficace. Dans notre cas il existe 6 index possibles pour les triplets. Par exemple il existe un index permettant de rechercher un Sujet par rapport à un Objet par rapport à un Prédicat (index POS).

Pour implémenter ce type d'index nous avons utilisé une encapsulation de structures complexes (voir figure 1). Afin d'avoir un accès rapide à tout moment dans la structure

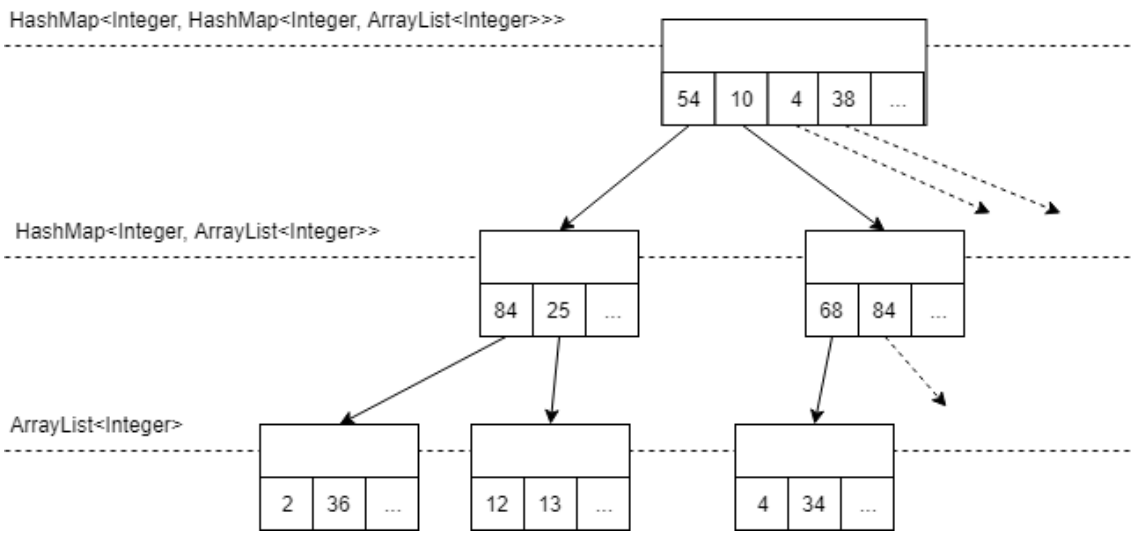

FIGURE 1 – Structure d'un index

nous avons choisi d'utiliser les HashMaps. Nous utilisons une HashMap dans une HashMap pour les deux premiesr niveaux de recherche (les deux niveaux connus dans la requête) puis une ArrayList pour le troisième niveau (la ressource recherchée). Dans une optique d'optimisation, que nous aborderons par la suite, cet ArrayList se doit d'être trié lors de l'utilisation pour une requête.
Pour cela nous avons écrit une fonction permettant l'ajout d'un entier à un ArrayList tout en gardant la structure triée et sans passer par un trie traditionnelle (trie rapide, trie fusion, ...).

Dans le cas du projet, les requêtes recherchent des sujets répondant à plusieurs conditions sur l'objet et le prédicat. Dans un premier temps nous avons donc pensé à implémenter les index POS et OPS, puis grâce aux statistiques on aurait déterminé quel index utilisé pour une requête donnée. Cependant l'utilisation de HashMap permet un temps d'accès constant à tout élément de la structure. Il est donc inutile voire handicapant d'utiliser plusieurs index dans notre cas.

L'utilisation d'index permet certes une grande accélération des procédures de requête mais peut prendre beaucoup de mémoire au programme en contrepartie.

4 Parseur de requête

Nous avons notre dictionnaire et notre index correctement initialisés. Nous pouvons maintenant évaluer nos requêtes mais pour cela nous devons extraire des fichiers chaque requête pour la traiter séparément.

L'utilisateur rentre le chemin d'un dossier où se trouve des fichiers contenant des ensemble de requêtes. Notre parseur va donc récupérer tous les fichiers avec l'extension *.queryset* pour en extraire les requêtes.

```
SELECT ?v0 WHERE {
  ?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/likes> <http://db.uwaterloo.ca/~galuc/wsdbm/Product0> .
  ?v0 <http://schema.org/nationality> <http://db.uwaterloo.ca/~galuc/wsdbm/Country3> . }
```

FIGURE 2 – Exemple de requête SPARQL

La figure 2 nous rappelle la forme d'une requête SPARQL. Dans notre projet une requête est modélisé par une liste de conditions. Ces conditions correspondent aux filtres WHERE de la requête qui sont entre accolades. On récupère grâce à un regex le contenu entre accolade puis chaque conditions en observant qu'elles sont séparés par un point. Enfin on divise chaque élément du triplet séparé par un espace. Notre requête est alors traitée.

On fait ce traitement pour toutes les requêtes de chaque fichier et on obtient notre liste de requêtes prêtes à être évaluées.

5 L'optimisation des requêtes

Reprenons notre liste de requête et regardons comment optimiser leur traitement. Cette optimisation est possible grâce à la sélectivité, c'est-à-dire, le nombre de valeur un index unaire ou binaire.

Dans la partie sur la construction de l'index nous avons vu que nous utilisons qu'un seul index. La sélectivité sur l'index unaire n'est alors pas utilisé dans notre projet car il n'y a pas besoin d'optimiser le choix des index. Cependant la sélectivité sur les index binaires va nous être très utile.
En effet, notre structure d'index ne permet pas de stocker la sélectivité. Cependant celle-ci est extrêmement facile à calculer, il suffit juste de regarder la taille de l'*ArrayList* pour un index binaire donné.

Pour chaque condition de la requête on va récupérer cette sélectivité. On trie alors les conditions afin de joindre les résultats des conditions avec la plus petite sélectivité et d'optimiser la jointure en minimisant le nombre de résultat intermédiaire.

6 Jointure

Dans la partie précédente nous avons parlé d'optimisation de jointure grâce à la sélectivité. Mais les jointures sont aussi optimisées d'une autre manière. Dans la partie index nous avons parlé du fait que l'*ArrayList*, contenant les résultats pour un index binaire, était triée.

Cela permet lors de l'exécution de la jointure de deux *ArrayList* triées d'optimiser le processus. En effet nous cherchons les éléments en commun dans chaque *ArrayList*. Grâce

au tri, on peut effectuer cela en temps linéaire.

7 Temps d'exécution

Les temps d'exécution ont été calculés pour chaque requête. La somme de ces temps nous donne le temps total d'exécution. Il ne prend pas en compte le remplissage du dictionnaire et des index ni le parsage des requêtes. Il représente donc le calcul des statistiques, les jointures et le stockage des résultats en passant des entiers aux chaînes de caractères dans une liste.

8 Écriture des fichiers

Une fois toutes les résultats des requêtes stockés nous pouvons les exporter selon les désirs de l'utilisateur. Nous avons décidé de composer nos différents fichiers de la manière suivante :

- Les temps d'exécution : nous avons regroupé les temps de chaque requête par fichier et les avons stocké dans un fichier csv.
- Les résultats des requêtes : la première colonne de notre fichier csv est la requête en entière, toutes les autres colonnes sont les résultats de la requête.
- Les statistiques : comme pour les résultats, la première colonne correspond à la requête. Ensuite comme on peut voir sur la figure 3 l'*Order Exec* correspond à l'ordre des jointure (le nom est le numéro de la condition de la requête et le nombre entre crochet correspond à la statistique pour la condition). L'*Exec Time* est le temps d'exécution de la requête et *Nb Results* le nombre de résultat pour la requête.

Query	Order Exec	Exec Time (ms)	Nb Results
SELECT ?v0 WHERE {?v0 <http://schema.o	t1[11]	0.81827	11
SELECT ?v0 WHERE {?v0 <http://schema.o	t1[9]	0.14027	9
SELECT ?v0 WHERE {?v0 <http://db.uwater	(t2[11] t1[296])	0.16949	0
SELECT ?v0 WHERE {?v0 <http://purl.org/d	((t1[75] t3[1054]) t2[2308])	0.90009	3
SELECT ?v0 WHERE {?v0 <http://purl.org/d	(((t2[40] t1[332]) t4[1054]) t3[2308])	0.63415	1

FIGURE 3 – Extrait du fichier csv des statistiques

9 Conclusion

Pour répondre à la problématique, c'est à dire de concevoir un système de base de donnée RDF capable de résoudre des requêtes en étoile nous nous sommes orienté

vers le système hexastore. Ce système est relativement simple à implémenter et permet d'avoir une très grande rapidité d'accès aux données mais en contrepartie utilise plusieurs structures de donnée coûteuse en mémoire.

Ce projet nous a permis de mieux comprendre le fonctionnement des systèmes de base de donnée et de voir concrètement les problématiques qui y sont liées.