HAI914I
Gestion des données au delà de SQL (NoSQL)

Evaluation des requêtes en étoiles

Auteur :

Canta Thomas (21607288)
Fontaine Quentin (21611404)

Master 2 - Génie Logiciel
Faculté des sciences de Montpellier
Année universitaire 2021/2022

Table des matières

Choix des structures

Le dictionnaire

Pour le dictionnaire, nous avons créé une classe Dictionary qui est un singleton, car nous avons besoin de l'instancier qu'une fois.

Notre **:Dictionary** a un attribut de type *HashMap<Integer, String>* qui se nomme *words*. *words* va stocker nos mots sous la forme **key :** id (entier), **value :** "le mot". Un mot sera stocké seulement s'il n'est pas encore dans **:Dictionary** avec un nouvel id.

Nous avons un second attribut, de type *int*, utile pour donner un id à nos mots. Dès qu'un mot sera ajouté, nous incrémentons cet attribut qui sera l'id du prochain mot. Prenons un exemple d'instanciation sous la forme de *spo*.

Supposons les données suivantes :

```
// <subject, predicate, object>
<user0, birthDate, "1998-09-28">
<user0, userId, "69420">
<user1, birthDate, "1998-09-28">
<user2, birthDate, "1998-30-07">
```

Le dictionnaire obtenus :

```
<0, "user0">
<1, "birthDate">
<2, "1998-09-28">
<3, userId>
<4, "69420">
<5, "user1">
<6, "user2">
<7, "1998-07-30">
```

Les index

Notre classe *Indexation* est aussi un singleton est ayant trois attributs de type *TreeMap<Integer, TreeMap<Integer, TreeSet<Integer>>>*. Ces trois attibuts sont *ops, pos* et *spo*, les seuls qui nous serons utiles ici. En effet, seul trois sont actuellement utilisée et nous expliquerons pourquoi par la suite. Si nécessaire, notre code nous permet d'ajouter facilement ceux manquant très facilement.

Le but de cette structuration est de pouvoir stocker les résultats ayant le même chemin de réponse afin d'éviter la recopie et d'améliorer notre temps d'exécution en facilitant la recherche lors de l'évaluation des requêtes. Prenons un exemple d'indexation sous la forme de *ops* avec le dictionnaire vu précédemment.

L'index *ops* obtenus :

```
<"1998-09-28", "birthDate", [user0, user1]>
<"1998-07-30", "birthDate", [user2]>
<"69420", "userId", [user0]>
```

On remarque ici l'intérêt de ce choix de structuration. En effet, si l'on souhaite obtenir tout les utilisateurs ayant pour date de naissance "1998-09-28" il suffira de parcourir une unique branche de notre arbre

```
SELECT ?v0 WHERE {
  ?v0 birthDate "1998-09-28" .}

// response: [user0, user1]
```

Évaluation des requêtes

Pour évaluer les requêtes nous avons commencer à mettre en place une première fonction naïve (*Query::fetchNaive()*) en utilisant une unique index, *pos*. Dans l'optique de ce projet et des requêtes utilisées nous nous sommes alors rendus compte que seul *pos* (resp. *ops*) et *spo* nous suffirait. C'est pourquoi nous avons supprimés dans l'indexation la création des autres indexes (*osp, pso, sop*) afin de minimiser notre temps d'exécution.

Une fois cette première évaluation terminé il fallait optimiser le temps de calculs des requêtes, pour cela nous avons implémentée la méthode merge-join (*Query::fetch()*) vue en cours. Cette méthode requiert d'utiliser deux indexes de recherches, expliquant pourquoi nous conservions uniquement *spo, ops et pos*. En plus de cela, il faut établir un système permettant de choisir le plus rapide pour retourner une réponse. Pour ce faire, nous nous sommes inspirés des système de poids en utilisant un nouveau *TreeMap<String, Integer>* appelée *frequences* dans notre dictionnaire. *frequences* stockera le nombre de fois qu'un mot est apparus sous la forme <"word", nbOccurrence>. Ainsi, si l'occurrence d'un objet dans *ops* est plus grande que l'occurrence d'un prédicat dans *pos* alors on cherchera les sujets en utilisant *pos*, et inversement. Une fois implémentée il suffisait de parcourir nos clauses deux par deux et rejoindre (*join*) les résultats.

Pour tester l'efficacité de notre nouvelle fonctionnalité nous l'avons mise en compétition avec la fonction naïve. Chaque méthode se devait d'évaluer toutes les requêtes et de répéter cela 500 fois. A la fin du processus on peut ainsi récupérer un temps moyen d'exécution. Finalement, les deux méthodes sont quasiment équivalentes, notre fonction merge-join est légèrement plus de rapide de quasiment 20 ms. Sur un plus gros système de donnée et de requêtes elle pourrait montrer plus de potentielles face à la fonction naïve.

Installation & Exécution

JAR

Les consignes d'utilisation sont présentes dans le README fournis avec le rendus du TP. Vous pourrez aussi le consulter sur le github de notre projet présenté ci dessous.

Github

Cliquer ici ou copier coller le lien suivant :
https://github.com/DocAmaroo/MiniStarQueryEngine