Université de Montpellier
M2 Génie Logiciel

Mini projet : Un moteur d'évaluation de requêtes en étoile

HAI914I : Gestion des donnees au dela de SQL (NoSQL)

Étudiants :
Denis Beauget
Hayaat Hebiret

Encadrant :
Scharffe François
Rodriguez Olivier

Année : 2021 – 2022

Table des matières

1 GIT

Notre projet est disponible sur le liens suivant : `https://github.com/Beauget/HAI914I_Projet`

2 Création du dictionnaire

Pour la création du dictionnaire nous avons utilisé une structure HasMap :

```
1  //Structure du dictionaire
2  public class DictionaryHashMap extends Dictionary {
3
4      private HashMap<Integer,String> dictionary= new HashMap<Integer,String>();
5
6      ....
7      }
```

Nous avons décidé de **mettre à jour le dictionnaire et les index durant la lecture du fichier rdf**. A chaque données rdf nous vérifions si l'objet, le prédicat et le sujet existe déja dans le dictionnaire. Par exemple :

- L'objet existe : boolean **object** deviens "True" et int **objectIndex** récupère sa position dans le dictionnaire.
- L'objet n'existe pas : boolean **object** reste "False" et int **objectIndex** est définit au moment ou il est ajouté au dictionnaire.

```
1      public Integer[] updateDictionary(Statement st) {
2          double start = System.currentTimeMillis();
3          boolean subject = false;
4          boolean predicate = false;
5          boolean object = false;
6          Integer [] toAdd = {0,0,0};
7          int subjectIndex = 0 , predicateIndex= 0, objectIndex = 0;
8
9           Iterator it = dictionary.entrySet().iterator();
10
11          while (it.hasNext()) {
12
13              HashMap.Entry dic = (Entry) it.next();
14              String value = (String) ((Entry) dic).getValue();
15              Integer key = (Integer) ((Entry) dic).getKey();
16
17              if(value.equals(st.getSubject().toString())) {
18                     subject = true;
19                     subjectIndex= key;
20                 }
```

```
21
22                  if(value.equals(st.getPredicate().toString())) {
23                      predicate = true;
24                      predicateIndex = key;
25                  }
26
27                  if(value.equals(st.getObject().toString())) {
28                      object = true;
29                      objectIndex = key;
30                  }
31          }
32
33
34              if(!subject) {
35                  Integer compteurHm = dictionary.size()+1;
36                  dictionary.put(compteurHm, st.getSubject().toString());
37                  toAdd[0] = compteurHm;
38              }
39  ....
40              return toAdd;
41      }
```

Ici on observe une partie de l'algorithme , on remarque bien le traitement **"à la volé"**. De plus, updateDictionary **renvoie l'index SPO de la donnée en rdf**.

Si on on génère et regarde **dictionary.txt** on obtient bien le format désiré avec une clé unique pour chaque valeur du dictionnaire.

```
1 : http://db.uwaterloo.ca/~galuc/wsdbm/User0
2 : http://schema.org/birthDate
3 : "1988-09-24"
4 : http://db.uwaterloo.ca/~galuc/wsdbm/userId
5 : "9764726"
```

3 Création des index

3.1 Version naïve :

Dans un premier temps nous avions eu une approche "naïve" de la création des index. On n'avait tout simplement ajouté les combinaisons de chaque index sous la forme de simple Triplet. Nous avions posé la question par mail car notre implémentation nous semblait incohérente (cela revenait à parcourir chaque index entièrement dans le pire des cas pour répondre à une requête. M.Rodriguez nous a fait un retour qui a comfirmer nos problèmes. Cette solution ne respectait pas le principe Hexastore et la notion d'arbre binaire.

```java
package deprecated;

// Triplet = 1,2,3 order give by index
public class Triplet {

    public int[] indexing;

    public Triplet(int s, int p, int o) {
        indexing = new int[] {s,p,o};
    }

    public int[] getIn() {
        return indexing;
    }


    public String toString() {
        String s =  indexing[0] + " " + indexing[1] +
        " " + indexing[2] + " " + "\n";
        return s;
    }

}
```

L'intégralité de notre première version naïve est disponible au sein du package "deprecated" de notre projet

3.2 Version optimisée :

Avec le retour nous nous sommes reposés la question de comment créer les index de la façon la plus "optimisée" possible. Nous avons décidé de garder la notion de type à part entière sur nos index avec un attribut pour identfier l'ordre.

Pour la structure de données on a donc laissé tombé les Triplet pour des HashMap (pour favoriser l'accès aux branches)

```java
public class IndexOpti extends Index{
    private   String order;
    private   HashMap<Integer,HashMap<Integer,List<Integer>>> index;
...
}
```

L'ajout d'un index deviens alors une peut plus complexe car il faut maintenant s'assurer de ne pas écraser une Hashmap déjà existante lors de sa mise à jour.

```java
public   void add(Integer first,Integer second,Integer third) {
        double start = System.currentTimeMillis();
        //verifie si la premier hashmap existe
        HashMap<Integer,List<Integer>> secondHashMap = index.get(first);
        //si la premiere hash n'existe pas je l'ajoute
        if(secondHashMap==null) {
            index.put(first,new HashMap<Integer,List<Integer>>());
        }
        //verifie si la deuxieme hashMap existe
        List<Integer> currentArray = index.get(first).get(second);
        if (currentArray==null){
            secondHashMap = index.get(first);
            currentArray = new ArrayList<Integer>();
            currentArray.add(third);
            secondHashMap.put(second,currentArray);
            index.put(first, secondHashMap);
        }
        else {
            secondHashMap = index.get(first);
            currentArray.add(third);
            secondHashMap.put(second,currentArray);
            index.put(first, secondHashMap);
        }

        double end = System.currentTimeMillis();

        execIndex += ((end - start) / 1000);

    }
```

Par exemple, si on on génère et regarde **SPO.txt** on obtient bien le format désiré.

```
8198 , 8001 : [8201]
8198 , 7995 : [2318]
8198 , 12011 : [10788]
8198 , 7996 : [8199]
8198 , 12012 : [11592, 11608, 11600, 11621, 11788]
8198 , 7998 : [8200]
8198 , 5535 : [5851]
```

4 Récupération des requêtes

Une fois le dictionnaire et les index crée nous utilisons le **Pattern Processor** qui se charge de faire les requêtes. Dans l'optique de mettre en place **un programme qui peut être maintenue**, le dictionaire et

l'index étende d'une **superClasse**.
Cela aurait permis de faciliter la mise à jour du processeur si nous avions eu plus de temps pour proposer différentes solutions.

```
public class Processor {
      Dictionary dictionary;
      ArrayList<Index> indexes;
      ArrayList<Query> queries;
      ...
}
```

5 Exécution des requêtes

Processor.doQueries() traite toute ses requêtes en utilisant les indexs et le dictionnaire et renvoie un **String contenant le résultat des requêtes** (si celle çi existe).
Si une requête contient plus d'un triplet, nous récupérons le résultat de chaque triplet et faisons une **intersection**.

```
public class Processor {
...
    public String doQueries(){
        double start = System.currentTimeMillis();
        StringBuilder builder = new StringBuilder();
        builder.append("Query , Answer(s) \n");
        for(Query q : queries) {
            List<String> answer = doAQuery(q);
            if(answer!=null) {
                if(answer.size()!=0) {
                    builder.append(q.getRealQuery()+" ,");
                    for(String s :answer) {
                    builder.append(s.toString()+" ");
                    }
                    builder.append("\n");
                }
            }
        }
        double end = System.currentTimeMillis();
        execQuery += ((end - start) / 1000);
        return builder.toString();
    }
    ...
    private   List<Integer> intersectionAnswers(Query q){
        List<List<Integer>> allAnswers = doQueriesInIntegers(q);
        List<Integer> output =null;
        if( allAnswers !=null){
            output = new ArrayList<Integer>(allAnswers.get(0));
```

```
            if(output!=null) {
                for (int i = 1; i < allAnswers.size(); i++) {
                    List<Integer> l1 = allAnswers.get(i);
                    if(l1 !=null) {
                        List<Integer> l2 = intersection(l1,output);
                        output = l2;

                    }
                    else
                        return null;
                }
            }
            else
                return null;
        }
        return output;
    }
    ...
}
```

6 Résultat obtenu

Tout les fichiers sont disponible dans le dossier "output" de notre projet à conditions que la demande d'écriture a été faite.
Extrait du fichier **Answers.txt**

```
SELECT ?v0 WHERE {
?v0 <http://schema.org/eligibleRegion>
<http://db.uwaterloo.ca/~galuc/wsdbm/Country137> . }
http://db.uwaterloo.ca/~galuc/wsdbm/Offer195
http://db.uwaterloo.ca/~galuc/wsdbm/Offer311
http://db.uwaterloo.ca/~galuc/wsdbm/Offer642
http://db.uwaterloo.ca/~galuc/wsdbm/Offer642

...

SELECT ?v0 WHERE {    ?v0 <http://purl.org/dc/terms/Location>
<http://db.uwaterloo.ca/~galuc/wsdbm/City0> .
?v0<http://schema.org/nationality>
<http://db.uwaterloo.ca/~galuc/wsdbm/Country0> .
?v0 <http://db.uwaterloo.ca/~galuc/wsdbm/gender>
<http://db.uwaterloo.ca/~galuc/wsdbm/Gender1> . }
http://db.uwaterloo.ca/~galuc/wsdbm/User178
http://db.uwaterloo.ca/~galuc/wsdbm/User831
```

Pour le calcul des métriques on peut remarqué que **TempsDeLectureDonnee** est NON_DISPONIBLE. En effet la lecture est intimement lié à la création du dictionaire et de l'index. De plus comme le fichier contenant les données rdf n'est jamais enregistré dans une structure, il n'y a pas de sens de compter son temps de lecture.
Extrait du fichier **option_1_data_output.csv**

```
nomDuFichierDonnee,nomDuFichierRequete,
NombreDeTriplet,NombreDeRequete,
TempsDeLectureDonnee,TempsDeLectureRequete,
TempsDeCreationDico,NombreIndex,
TempsCreationIndex,TempsTotalEval,
TempsTotal
data/100K.nt,data/STAR_ALL_workload.queryset,
107338,1200,
NON_DISPONIBLE,389.0,
37604.0,6,
325.0,39649.0,
39651.0
```

7 Bilan

```
java -jar qengine-0.0.1-SNAPSHOT-jar-with-dependencies.jar
Veuillez entrer le path du fichier contenant les requêtes sparql(si vous souhaitez tester le code avec STAR_ALL_workload.qu
eryset et 100K.nt écrire "defaut")
defaut
--- Bienvenue dans notre moteur de requête RDF ---

Options disponible (taper le chiffre correspondant à l'option) :

1 : Création du dictionnaire, index et réponses aux requêtes et du .csv contenant les temps de calculs et des informations
sur les fichier
2 : Création du dictionnaire et temps d'exécution (SANS ECRITURE /output)
3 : Création du dictionnaire et temps d'exécution (AVEC ECRITURE /output)
4 : Création des indexs et temps d'exécution (SANS ECRITURE /output)
5 : Création des indexs et temps d'exécution (AVEC ECRITURE /output)
6 : Chargement + exécution des requêtes et temps d'exécution (SANS ECRITURE /output)
7 : Chargement + exécution des requêtes et temps d'exécution (AVEC ECRITURE /output)
8 : Toute les données des options précédentes (SANS ECRITURE)
9 : Toute les données des options précédentes (AVEC ECRITURE)
0 : Quittez l'application
```

FIGURE 1 – Lancement du programme depuis le terminal

Pour ce projet, notamment en raison de la deuxième partie du projet nous avons essayé d'extraire un maximum de métrique de notre code. D'une part pour avoir un retour concret du travail que nous avions effectué (un visuel sous un format textuel) et d'autre part pour connaitre les limites de notre projet (quel métrique nous ne pouvions pas obtenir par exemple). Nous avons donc permis l'extraction du dictionnaire, des index (un fichier par index) et des réponses aux requêtes sous un format .txt pour avoir un format brut de tout nos résultats. Pour faciliter l'identification des fonctionnalités nous avons rajoutés un petit client CLI ainsi que le projet au format .jar.

Bien que nous pensons que notre projet soit conforme aux attentes nous aurions aimé parcourir plus en détail les possibilités d'optimisé celui ci.
Nous avons vu en cours qu'il est possible de mettre un plaçe un dictionnaire dont **l'ordre des données**

permettait un parcours plus efficace de celui çi, notamment d'ignorer certaine positions car on est sûr qu'elle n'y est pas (par ordre alphabétique par exemple).
Il aurait également étais intéressant **d'identifier la structure des requêtes** afin de ne pas répéter les solutions des triplet et de ne pas essayer d'obtenir une solution en passant par tout les indexes.