HAI914I - Un moteur d'évaluation de requêtes en étoile

Groupe : Antoine Barbier - Djamel Benameur

Lien du projet : https://gitlab.com/An_toine/implementation-d-un-mini-moteur-de-requetes-en-etoile

---

L'évaluation des requêtes

Nous identifions quatre étapes fondamentales dans l'évaluation des requêtes en étoile.
1. Chargement de la base de triplets et création d'un dictionnaire pour les ressources.
2. Création des index.
3. Lecture des requêtes en entrée.
4. Accès aux données et visualisation des résultats.

Étape 1 : Création du dictionnaire

Problème rencontrée :

La première version de notre dictionnaire utilisait la structure de donnée **HashMap<Integer, String>** afin d'associer un entier à une ressource de la base RDF. Le problème de ce choix de structure est que pour la suite, nous allons devoir accéder au données du dictionnaire de deux façons différentes : par *clef-valeur* et par *valeur-clef*.
Ainsi, dans la première version de notre code, on avait développé une méthode (dans la classe Dictionnaire) qui permettait de trouver la clef à partir de la valeur. Cependant, pour retrouver la clef on faisait un parcours intégrale de la HashMap. Cette méthode n'était pas du tout optimisée et nous aurait fait perdre beaucoup de temps d'exécution pour la suite.

Dans la deuxième version de notre dictionnaire, on a fait le choix d'utiliser une structure de donnée qui permet d'accéder par clef-valeur et par valeur-clef de manière optimale. On a d'abord eu l'idée de créer un dictionnaire avec deux HashMap : **HashMap<Integer, String>** et **HashMap<String, Integer>**. Mais après quelques recherches, on a vu qu'il existait une structure de donnée en JAVA qui faisait exactement cela avec une seule structure de donnée. Cette structure s'appelle : **BiMap** et permet d'accéder aux ressources par *clef-valeur* et par *valeur-clef* de manière optimal.

Étape 2 : Création des index

Pour l'index, nous avons créer une classe Index dans laquelle on manipule la structure de donnée suivante : **HashMap<TypeIndex, HashMap<Integer, HashMap<Integer, ArrayList<Integer>>>>**.

Cette structure permet d'enregistrer les 6 types d'index (POS, OPS, SPO, PSO, SOP, OSP) et d'accéder très rapidement aux ressources grâce aux HashMap.

Pour créer l'index, on utilise le parsing du fichier de données. A chaque statement rencontré, on appelle la *méthode add de la classe Index.* Cette méthode se charge de remplir l'index en y enregistrant les 6 positions possible pour le *sujet*, le *prédicat* et *l'objet*.

Pour travailler avec le type d'index, nous avons créer la structure de donnée suivante :

```
enum TypeIndex {
    SPO(0, 1, 2), SOP(0, 2, 1), PSO(1, 0, 2), OSP(1, 2, 0), POS(2, 0, 1), OPS(2, 1, 0);
    public int S, P, O;
    TypeIndex(int S, int P, int O) {
        this.S = S;
        this.P = P;
        this.O = O;
    }
}
```

Cette structure permet de connaitre la position du sujet, du prédicat et de l'objet pour chaque type d'index.

Étape 3 : Lecture des requêtes en entrée.

Pour la lecture des requête nous avons réutilisé la fonction **parsingQueries** fournis dans le squelette du programme.

Étape 4 : Accès aux données et visualisation des résultats.

Pour cette étape, nous avons évalué chaque sous-requêtes d'une requête et nous avons fait l'intersection entre les différents résultats obtenu pour chaque sous-requête.

Pour évaluer une sous requête, on recherche les résultats grâce à l'index. Pour que la recherche soit optimal, on utilise le bon type d'index en fonction de la position de la variable ?v0 dans la requête. Ensuite on fait le choix entre les deux type d'index possible, en prenant celui qui sera le plus rapide pour la recherche.

Pour la visualisation des résultats, on exporte dans un fichier .txt le résultat de chaque requête.

Informations sur le projet

Lien du projet : https://gitlab.com/An_toine/implementation-d-un-mini-moteur-de-requetes-en-etoile

Lancement du moteur de requêtes :
Pour lancer le programme, il faut lui passer en paramètres les options suivantes :
<queryFile> <dataFile> <restultFolder>

Différents moyen de lancer le programme :
- Avec VsCode : Modifier le fichier .vscode/launch.json, placer les options dans "args"
- Avec Eclipse : Configurer le RUN, java application > Main, et placer les options dans program arguments.

Organisation du projet :
- *Class::DataInformation* : permet de stocker toutes les informations sur les temps d'exécutions et différentes statistiques.
- *Class::Dictionnaire* : stocke le dictionnaire utilisé pour le moteur de recherche
- *Class::Index* : stocke l'index utilisé pour le moteur de recherche
- *Class::ParserDatas* : Parser des données qui initialise la class Dictionnaire et Index
- *Class::ParserQueries* : Parser des requêtes qui trouve le résultat des requêtes.
- *Class::Main* : Point d'entrée du programme