HAI914I - Un moteur d'évaluation de requêtes en étoile

<federico.ulliana@inria.fr>
<guillame.perution-khili@inria.fr>

Implementation d'un mini moteur de requêtes en étoile

L'objectif du TD est d'implémenter l'approche hexastore pour l'interrogation des données RDF vue en cours, ainsi que les procédures nécessaires à l'évaluation de requêtes en étoile (exprimées en syntaxe SPARQL) qui seront introduites par la suite.

Le TD suivant sera dédié à l'évaluation des performances du système réalisé. Votre prototype sera comparé avec le logiciel libre Jena.[1] De plus, Jena peut être considéré comme un système "oracle" pour vérifier la correction et complétude de votre moteur, c'est-à-dire, vérifier que votre moteur donne toutes et seules les bonnes réponses à une requête. Celle-ci est évidemment une condition nécessaire à la conduite d'expériences.

Consignes

— Le TD se fait en groupes de 1 ou 2 personnes.
Lorsque vous avez décidé quel sera votre groupe inscrivez-vous dans le tableau correspondant à votre encadrant de TD suivant ce lien :
https://docs.google.com/spreadsheets/d/1bQY-Xba11DNcbCC5gujvPnidhajN-Kcb306PjXveF7M/edit?usp=sharing
— Le langage de programmation est imposé : Java.

Le projet est divisé en 3 rendus :	Dates de rendu
1. dictionnaire et index : rendu du code (pas de rapport)	16 Novembre
2. évaluation des requêtes en étoile : code + rapport 3 pages	2 Décembre
3. analyse des performances : code + rapport 5 pages	17 Décembre

Il n'y aura pas de soutenance. Toutefois, il sera demandé aux groupes d'expliquer le travail réalisé lors des séances de TD. Ainsi, la présence en TD est obligatoire.

Évaluation

Les points suivants seront sujet à évaluation.
— Travail réalisé (code fonctionnel, implémentation des fonctionnalités)
— Qualité du code produit
— Clarté et concision du rapport

1. https://jena.apache.org/tutorials/sparql.html

Point de départ du projet logiciel (il est important de lire toute la page)

On vous met à disposition un squelette du projet à réaliser dans ce dépôt git :
https://gitlab.etu.umontpellier.fr/p00000386189/qengine

Vous pouvez réutiliser ce projet, ou juste l'étudier comme exemple. Ce projet montre simplement comment on peut lire les données et les requêtes depuis des fichiers. Pour en apprendre davantage, il faudra se référer à la documentation de rdf4j [2] ainsi qu'à sa Javadoc [3].

Ce projet est prévu pour utiliser Java 8 afin de s'adapter aux anciennes versions d'Eclipse possiblement installées sur les postes de travail. Libre à vous d'utiliser des versions plus récentes si vous êtes vraiment à l'aise avec l'environnement Java. L'idée est quand même de ne pas perdre de temps avec des considérations techniques.

Ce projet a été testé fonctionnel avec Eclipse Luna 4.4.0 (2014) sur les ordinateurs de la faculté en chargeant le projet avec

`File/New/Jave project`

ainsi qu'avec la dernière version d'Eclipse 4.21.0 (2021-09) en créant un nouveau projet avec le menu

`File/Open Projects from File System...`

Logiciel Eclipse

Si votre poste informatique ne dispose pas du logiciel Eclipse, vous pouvez l'installer en suivant cette procédure

https://moodle.umontpellier.fr/mod/page/view.php?id=131074

Nous avons besoin de la version intitulée *Eclipse IDE for Java Developers*.

Bibliothèques externes

Le projet utilise deux bibliothèques

— rdf4j[2] 3.7.3 pour lire les données rdf et requêtes sparql
— jena [4] 3.9.0 qui sera utilisée pour comparaison dans la phase *d'analyse des performances*

Pour utiliser ces bibliothèques, le projet contient un fichier special `pom.xml` décrivant les dépendances à ces bibliothèques ; il n'est normalement pas nécessaire d'y toucher.
Pour s'assurer que tout fonctionne bien il suffit d'exécuter le programme de la classe `Main`.

2. https://rdf4j.org/documentation
3. https://rdf4j.org/javadoc/latest/
4. https://jena.apache.org/tutorials/sparql.html

Si Eclipse ne reconnaît pas ce fichier `pom.xml` et n'arrive pas à charger les dépendances, c'est qu'il y a un problème avec son plugin maven (m2e).
On peut alors tenter de chercher le menu suivant (il peut ne pas exister) :

`Fenêtre "Package Explorer" (toute à gauche)/clic droit sur le projet/Configure/Convert to Maven Project`.

Dans le cas où ça ne fonctionne pas nous fournissons l'ensemble des librairies externes (`.jar`) à lier au projet suivant ce dépôt :

https://gitlab.etu.umontpellier.fr/p00000386189/qengine-alljars.

Dans Eclipse on peut lier manuellement des `.jar` externes en suivant le menu

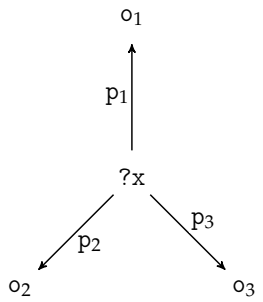
`Project/Properties/Java Build Path/Librairies/Add External JARs...`

Jeu de données et requêtes

Dans le répertoire `data/` du projet, on vous met à disposition des fichiers permettant de réaliser des micro-tests (`sample_data.nt` et `sample_query.queryset`) ainsi que des fichiers contenant plus de données et de requêtes (`100K.nt` et `STAR_ALL_workload.queryset`).

Les requêtes en étoile

Les requêtes en étoile constituent la base des requêtes SPARQL (et plus en général des requêtes sur des graphes). Une requête SPARQL en étoile est composée de n patrons de triplet tous partageant une même variable commune. Un exemple de requête en étoile est `SELECT ?x WHERE { ?x` p_1 o_1 `. ?x` p_2 o_2 `. ?x` p_3 o_3 `}` dont la représentation graphique est la suivante.



Voici quelques exemples d'interrogations exprimables par des requêtes en étoile.

Q_1 : *quels écrivains ayant reçu un prix Nobel sont aussi des peintres ?*

Q_2 : *quels sont les artistes nés en 1904 ayant vécu à Paris ?*

Q_3 : *qui sont les amis d'Alice qui aiment le cinéma ?*

Pour Q_1 nous aurons la requête

```
SELECT ?x WHERE { ?x type peintre . ?x won_prize Nobel . ?x type écrivain }
```

Similairement, pour Q_2, nous aurons

p_1 `= type,` o_1 `= artiste,` p_2 `= lives_in,` o_2 `= Paris,` p_3 `= birth_year,` o_3 `= 1904`

Enfin, pour Q_3, nous aurons

```
SELECT ?x WHERE {?x friendOf Alice . ?x likes Movies}
```

Bien entendu, le système résultant du projet doit être capable de gérer des requêtes étoile avec n branches, et n'importe quelle valeur pour les constantes p_i et o_i. Enfin, on assumera que les variables n'apparaissaient que dans les sujets (et jamais au niveau des propriétés ni des objets).

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. Creation des index.
3. Lecture des requêtes en entrée.
4. Accès aux données et visualisation des résultats.

Nous allons maintenant détailler chaque étape.

Le dictionnaire (rendu 16 Novembre)

Le dictionnaire associe un entier à chaque ressource de la base RDF, afin d'en permettre un stockage compact. Par exemple, on voit les triplets `<Bob, knows, Bob>` et `<1,2,1>` indistinctement avec la correspondance {(1,Bob) , (2,knows)}.

L'index (rendu 16 Novembre)

L'index permet une évaluation efficace des requêtes et est adapté au système de persistance choisi. Dans ce projet, on vous demande d'implémenter l'approche hexastore pour l'indexation des données.

L'accès aux données (rendu 2 Décembre)

L'accès aux données se fait par les structures de données mises en oeuvre. Une fois retrouvé l'ensemble des solutions pour le premier patron de triplet, cela nous servira pour filtrer ultérieurement les valeurs récupérées pour le deuxième patron, et ainsi de suite. Il est demandé d'implémenter une méthode spécifique pour l'évaluation des requêtes en étoile.

Consignes :

— Implémenter dictionnaire, indexation, et accès aux données du moteur de requêtes.
— Les données seront stockées en mémoire vive (ram). [5]

5. rien nous empêche de prévoir un système de persistance en mémoire secondaire comme extension possible du travail.

Lecture des entrées et export des résultats :

1. Comme déjà expliqué, vous trouverez sur le dépôt git un squelette du programme à réaliser que vous pouvez réutiliser comme base de votre projet. Vous y trouverez comment lire un fichier de données (`Main.parseData()`) et un fichier de requêtes (`Main.parseQueries()`). Vous pouvez modifier ce projet à votre convenance.
2. Important : votre système doit être orienté vers l'évaluation d'un *ensemble de requêtes* (et non pas d'une seule requête) sur un (seul) fichier de données. Les résultats seront exportés dans un répertoire dédié.
3. Pour réaliser facilement des tests, il sera également important que votre programme soit exécutable en ligne de commande avec les options suivantes.

   ```
   java -jar rdfengine
   -queries "/chemin/vers/dossier/requetes"
   -data "/chemin/vers/fichier/donnees"
   -output "/chemin/vers/dossier/sortie"
   -Jena : active la vérification de la correction et complétude du système
   en utilisant Jena comme un oracle
   -warm "X" : utilise un échantillon des requêtes en entrée (prises
   au hasard) correspondant au pourcentage "X" pour chauffer le système
   -shuffle : considère une permutation aléatoire des requêtes en entrée
   ```

   Il est possible bien évidemment d'ajouter d'autres options à souhait.

4. Les temps d'évaluation des requêtes, ainsi que le temps total d'évaluation du workload, doivent être visualisés dans le terminal et exportés dans un fichier `csv` contenu dans le répertoire indiqué (option `output`). Une ligne du fichier `csv` doit contenir les informations suivantes, si l'information n'est pas disponible écrire NON_DISPONIBLE.
   - nom du fichier de données | nom du dossier des requêtes | nombre de triplets RDF | nombre de requêtes | temps de lecture des données (ms) | temps de lecture des requêtes (ms) | temps création dico (ms) | nombre d'index | temps de création des index (ms) | temps total d'évaluation du workload (ms) | temps total (du début à la fin du programme) (ms)
5. Les résultats des requêtes pourront aussi être exportés dans un fichier `cvs` séparé (option `export_query_results`).