HAI914I
Gestion des données au delà de SQL (NoSQL)

RDF Query Engine

Auteur :

Canta Thomas (21607288)
Fontaine Quentin (21611404)

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

Table des matières

Chapitre 1

Choix des structures

1.1 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">
```

1.2 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]
```

1.3 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.

Chapitre 2

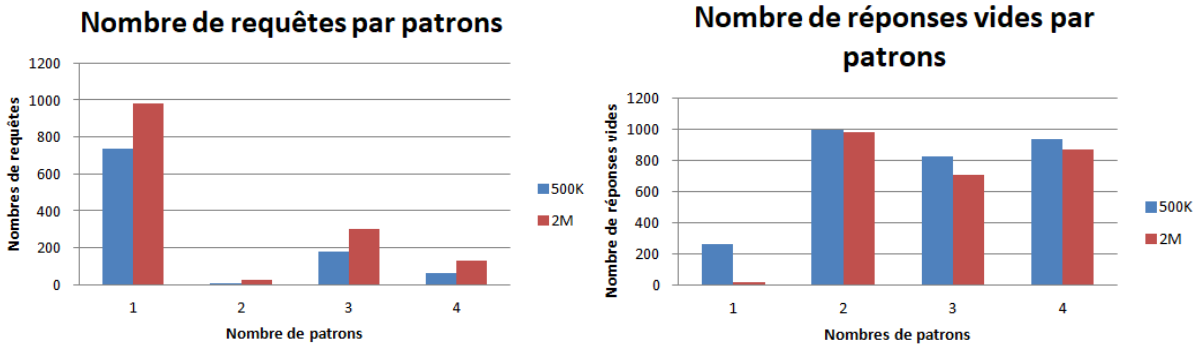
Préparation des bancs d'essais

2.1 Analyse de nos données

Lors de la génération de requête Watdiv sépare les fichier de requêtes en fonction du nombre de conditions (ou patrons) qu'elles composent. Par exemple, le fichier *Q1_nationality.queryset* est un jeu de n requêtes à un patron, *Q2_likes_nationality.queryset* avec 2 patrons et ainsi de suite jusqu'à Q4. Ainsi, pour les test qui suivent nous avons utilisé les fichiers ci-dessous, composée chacun d'eux de 1000 requêtes.

1. Q1_nationality.queryset
2. Q2_likes_nationality.queryset
3. Q3_location_nationality_gender.queryset
4. Q4_location_nationality_gender_type.queryset

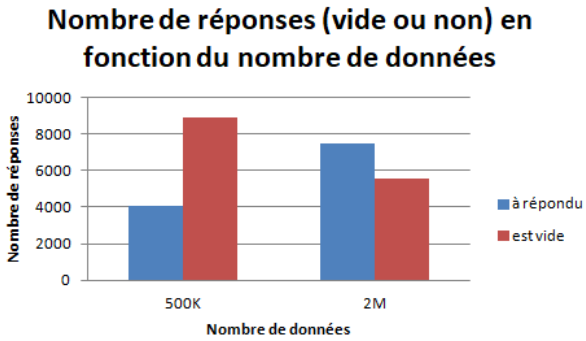
NB : les jeu de données et de requêtes étant générés aléatoirement par Watdiv, les résultats peuvent différer entre d'autre essais.

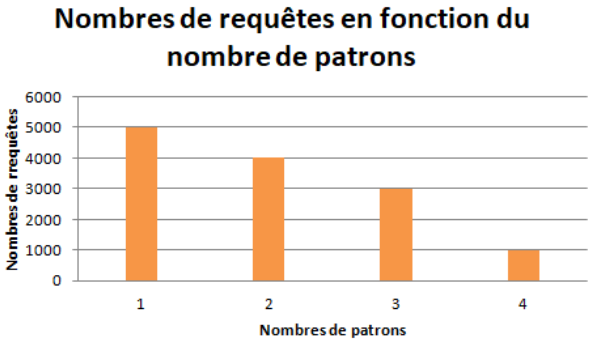


Ce format d'affichage permet de visualiser les lacunes de notre jeu de requêtes et si il est nécessaire d'en créer un nouveau. En effet, il est à constater qu'avec ce jeu de données et de requêtes notre programme obtiens plus de réponse sur des requêtes complexes à 3, voire 4 conditions, qu'à seulement 2 conditions. Cela pourrait provenir d'un jeu de requêtes trop dupliqué par une même requête sans réponse ou encore du jeu de données qui ne possède pas assez d'information.

2.2 Gestion des patrons

Pour cette partie là nous allons donc utiliser tout les jeux de requêtes générés par Watdiv et pas seulement les quatre auparavant choisis. Ainsi, la suite des évaluation qui suit ont été fait avec un jeu de de **13.000 requêtes**.





Pour obtenir des résultats convenables pour notre benchmark, un bon jeu de requêtes contiendrait un nombre quasi équivalent pour chaque nombre de sous-requêtes afin de permettre une bonne représentation de la qualité et de la performance de notre système.

2.3 Duplication

Sur nos 13.000 requêtes, 5596 sont des duplications, soit prêt de 43%. Ce chiffre est beaucoup trop élevé, quasiment la moitié sont des doublons, faussant et biaisant totalement notre benchmark. Pour en savoir plus nous avons visualisé nos résultats en affichant les requêtes dupliquées et le nombre de fois qu'elles le sont. Beaucoup d'entre elles sont dupliquées moins de cinq fois, à l'inverse, d'autres une centaines de fois.

Il n'est pas souhaitable dans notre jeu de requêtes d'avoir un nombre considérable de doublons, mais un faible ensemble peut être autorisé. En effet, trop de doublon biaise et fausse notre benchmark car les résultats obtenus ne sont pas représentatifs pour notre modèle. En revanche, une petite partition de nos requêtes peut l'être, en supposant par exemple que plusieurs utilisateurs ont effectué la même requête, au même moment. Ainsi, nous supposons pour l'instant, qu'une requête peut être dupliquée entre 2 et 5 fois, et que le taux de duplications totales de notre jeu de requêtes ne dépasse pas les 5%.

Chapitre 3

Hardware et Software

Voici les spécifications de la machine utilisée pour les test :

- Ordinateur : Asus R511L (laptop)
- Disque dur : HDD
- Processeur : Intel® Core™ i5-5200U 2.20Ghz x 4
- Ram : 6go
- OS : Pop !_OS 21.10 (64bits)
- Version de GNOME : 40.4.0

Un ordinateur portable, même sur secteur, n'est pas le type de machine le plus adapté pour des test de performances. De plus, le notre a de gros problèmes et commencent à être en fin de vie (problèmes d'accessibilités au HDD, clavier défaillant, crashs réguliers de softwares voire hardwares).

Malgré tout cela, le temps d'exécution de notre programme est raisonnable. Mais, il est, toutefois, recommandé d'avoir une machine avec des composants plus puissants afin d'améliorer la performance des tests.

Chapitre 4

Métriques, Facteurs, et Niveaux

Voici une liste non-exhaustive des métriques permettant d'évaluer les performances de moteurs de requêtes RDF : (Pour 500k données et 13k requêtes)

- Temps de création du dictionnaire (≈4.03s)
- Temps de création des index (≈3.91s)
- Temps d'évaluation des requêtes (≈2.28s)
- Temps moyen de réponse par requête (≈)
- Temps total d'exécution (≈9.96s)

Une fois les métriques déterminées, il est important de connaître les facteurs pouvant affecter l'évaluation du système. Vous les retrouverez listés ci-dessous, dans l'ordre du plus au moins important. De plus nous listerons pour chaque point les niveaux associés :

1. **Nombre de données** (principale) : Le nombre de données devient un facteur conséquent lorsque celui-ci est important. De plus, pour notre projet, il s'agit du seul facteur, avec le nombre de requêtes, que faisons varier.

   Niveau : 500K, 2M données.

2. **Nombre de requêtes** (principale) : Même constat pour les requêtes, un grand nombres de requêtes à évaluer augmentera la durée de notre programme.

   Niveau : 13K, 130K requêtes.

   Le CPU et la RAM sont généralement des facteurs importants, mais pour notre projet, nous n'avons utilisé qu'une seule machine (donc, pas de variations) et le nombre de données et requêtes n'est pas extrêmement important pour nécessiter d'une machine vraiment puissante.

3. **RAM** (secondaire) : La RAM permet de stocker rapidement les calculs effectué par le CPU. En posséder un nombre suffisant minimiseras le temps de d'exécution en facilitant les échanges de données. Surtout en sachant qu'une mémoire surchargé est très coûteux en temps d'exécution.

   Niveau : 4Go, 8Go, 16Go, ...

4. **CPU** (très secondaire) : Le CPU s'occupe de la partie calcul. Un mauvais composant peut réduire considérablement la vitesse et la simultanéité des calculs. Il s'agit d'un facteur que nous varions presque jamais.

   Niveau : unique.

5. **Mémoire allouée à la JVM** (très secondaire) : Une sorte de RAM virtuel pour JAVA. Il est secondaire, car c'est un paramètre que nous ne modifions quasi jamais en cursus scolaire et dépend aussi de notre RAM.

Niveau : 2Go, 4Go, 8Go, ...

6. **Version du système** (très secondaire) : la version de l'OS ou encore la version d'un langage peut affecter légèrement notre système. (ex. une version Java plus récent peut être plus rapide, de même pour une version de linux plus récente)

   Niveau : unique.

NB : Pour rappel, 1K = 1000 et 1M = 1.000.000.

Chapitre 5

Evaluation des performances

5.1 Cold & Warm ?

Cold et Warm sont deux formes d'évaluations pour la mesure de temps d'exécution. Ces deux méthodes permettent d'analyser plus adéquatement nos différentes mesures en fonction de l'utilisation de nos fonctionnalités.

Une évaluation à froid (Cold) sera utilisée pour les métriques que nous faisons qu'une fois, sans préchargement ou mise en cache antérieure, tels que le chargement des données. Dans le cas de notre étude, ceci inclut le temps de création du dictionnaire, le temps de création des index ou encore le parsing.

En revanche, une évaluation à chaud (Warm) est utilisée pour des mesures dont les informations ont été préchargées et/ou que le système à été, au préalable, échauffé afin d'obtenir une évaluation plus concrète. Notamment utilisés pour des opérations effectuées régulièrement, dans notre cas, à nos requêtes. Il faudra donc mettre en place un système d'échauffement pour notre modèle avant d'évaluer correctement le temps de réponse des requêtes.

5.2 Mise en pratique

Comme énoncé auparavant, la méthode à froid ne nécessite pas de pré disposition, a contrario de celle à chaud, de ce fait, nous ne traiterons que l'évaluation à chaud dans cette partie, et notamment l'échauffement de notre jeu de requêtes. Deux aspects s'offrent à nous, le nombre et la distribution des patrons à choisir.

En effet, nous visons ici à échauffer notre système, il faut donc déterminer un nombre de données assez conséquent pour l'entraîner convenablement, mais trop de donnée serait inutile et synonyme de perte de temps. De même, un échauffement trop simpliste, en utilisant des requêtes avec 1 ou 2 patrons, pourrait ne pas être suffisant. Il faut donc distribuer nos patrons efficacement pour un obtenir un échauffement optimal.

Le graphe ci-dessous (cf. Figure 5.1) représente une répartition possible, mais la plus convaincante dans le cadre de notre étude. Nous supposons que 20% de notre jeu de requêtes est un bon compromis afin d'assurer un bon échauffement et un temps minimum d'exécution. Il faut dorénavant diversifier nos patrons afin de le rendre optimal. Une diversification assez simpliste sera de diviser nos requêtes extraites par le nombre de patrons, ici nous en avons 4, ce qui revient à 5%.

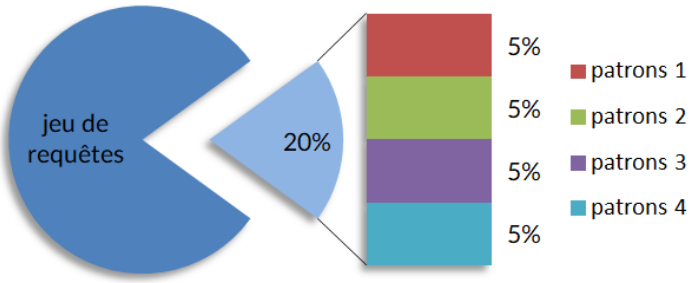


FIG. 5.1 – Répartition de l'échauffement

Après avoir mis en place notre système d'échauffement nous pouvons maintenant le comparer à une exécution à froid. Avant cela, nous avons généré un jeu de requêtes plus conséquent contenant 50.000 requêtes (sans doublons) afin de constater les bénéfices notre nouvelle fonctionnalité. En effet, sur un jeu trop faible il se peut que les deux méthodes aient des temps d'exécution quasi similaire, c'est ce que nous constatons dans nos résultats obtenus ci-dessous (cf. Figure 5.2) .

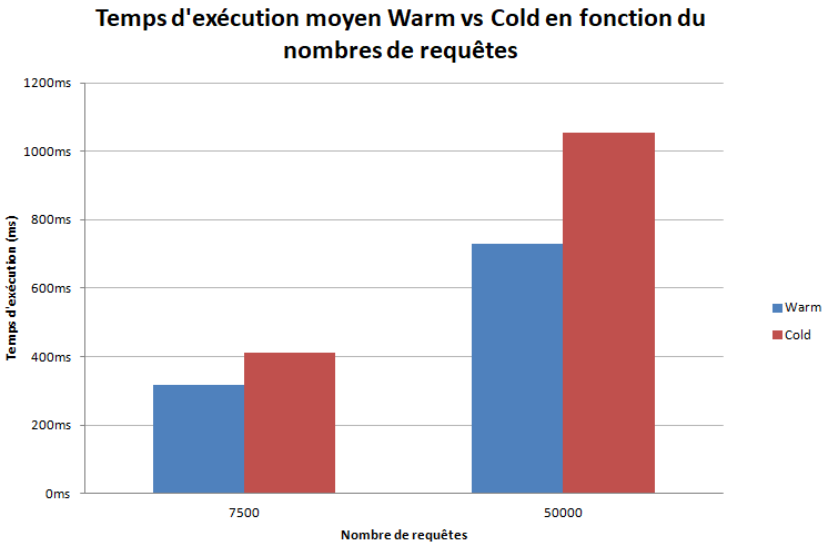


FIG. 5.2 – Comparaison des temps d'exécution

5.3 Vérification de notre système avec Jena

Pour vérifier la complétude de notre système nous pourrions comparer le résultat d'une requête à la réponse obtenue par un autre système de requête fiable, notamment Jena.

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