Université De Montpellier
Faculté Des Sciences

Niveau : Master 2

Spécialité : IASD

Module : NoSQL

HAI914I

Projet : Implementation d'un mini moteur de requêtes en étoile

---

Supervisé par :
M. François SCHARFFE

Réalisé par :
AIT ALI YAHIA Yasmine
REHHALI El bachir

2021/2022

Table des matières

1 Création d'un jeu de test

1.1 Type de benchmark

Pour tester notre système, nous avons utilisé le benchmark standard qui est **WatDiv** avec une version pré configurée pour mesurer les performances de notre système sur un large spectre de requêtes SPARQL (SPARQL Protocol and RDF Query Language) avec des caractéristiques structurelles et des classes de sélectivité variables.

1.2 Jeu de test

Pour notre création de données, nous avons généré un grand nombre de données et de requêtes pour conclure avec 2 workload efficaces pour les mesures envisageables.

1. Pour les données, nous avons généré 500K et 100K de triplets.
2. Pour les requêtes, nous avons créé 10 Million de requêtes puis traité la sortie (voir la section : Préparation des bancs d'essai).

2 Facteurs et Mesures

2.1 Hardware et Software

Nous avons deux ordinateurs différents qui possèdent les caractéristiques suivantes :

Ordinateur 1

- **Software :** Windows, Linux.
- **Hardware :**
  - Un processeur AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10Ghz.
  - Une mémoire vive de 8Go.
  - Un disque dur SSD 256Go.

Ordinateur 2

- **Software** : Linux.
- **Hardware** :
  - Un processeur I3 10 ème génération.
  - Une mémoire vive de 8Go.
  - Un disque dur SSD 128Go.

Notre environnement n'est pas parfaitement adapté à l'analyse des performances du système, car nous avons exécuté la majorité de nos tests sur Windows, ce système d'exploitation a des tâches qui tournent en fond et cela fausse le calcul de temps. Il en est de même si nous comparons le temps d'exécution avec l'IDE (Environnement De Développement) Eclipse qui utilise beaucoup la mémoire par rapport à l'exécution directement sur la ligne de commande.

2.2 Métriques

- **Temps de réponse :** c'est le temps écoulé entre le moment où une requête est exécutée jusqu'au moment où elle renvoie une réponse et se termine.
- **Qualité des réponses :** elle consiste à calculer le pourcentage de solutions correctes et fausses par rapport à un système fiable (Jena).
- **Nombre de requêtes par période :** c'est Le nombre de requêtes auxquelles nous pouvons répondre dans une période fixée.
- **Scalabilité :** c'est la variation de la taille des données et des requêtes.

2.3 Facteurs

Ces métriques dépendent de quelques facteurs qui jouent un rôle important dans la variation des résultats de nos mesures. Nous pouvons les classer avec l'ordre suivant :

- **Requêtes :** quantité, nombre de réponses des patterns et échauffement.
- **Performances d'ordinateur :** RAM (Random Access Memory), CPU (Central Processing Unit), Disque Dur et système d'exploitation.

3 Préparation des bancs d'essai

Nous avons créé une instance de données de 500K et une autre de 100K. Nous avons créé également 10M requêtes de type étoile à l'aide du benchmark **WatDiv**. Pour générer ces différentes requêtes, nous avons pris les templates de la version pré-configurée de **WatDiv** et nous avons généré un nombre important de requêtes hétérogènes. C'est-à-dire que notre jeu de requêtes contient des requêtes avec un, deux, trois et quatre patterns.

3.1 Analyse des requêtes générées par Watdiv

Dans cette partie, nous avons créé un programme spécifique pour traiter les 10M de requêtes de **Watdiv** avec 500K pour workload 1 et 100K pour workload 2 en utilisant Jena.

Analyse des doublons

Nous avons traité les 10M de requêtes et nous avons constaté qu'elles sont constituées de 73% de doublons.

Nous avons supprimé les doublons pour bien évaluer le moteur, car les doublons affectent l'analyse puisque le moteur sait déjà comment accéder aux ressources pour construire la réponse et pour notre évaluation nous avons besoin de requêtes complexes.

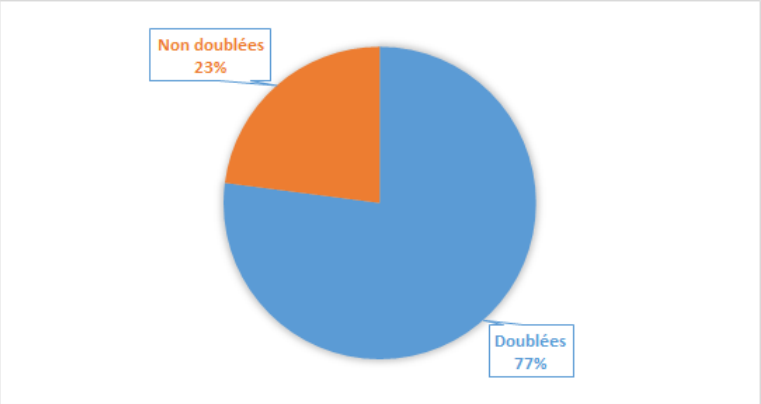

FIGURE 1 – Analyse des requêtes doublées.

Analyse des résultats vides

Après la suppression des doublons, nous avons gardé 278108 requêtes puis nous avons analysé celles avec des résultats vides par rapport à 500K et 100K. Notre fichier final contient que 5% de requêtes vides.

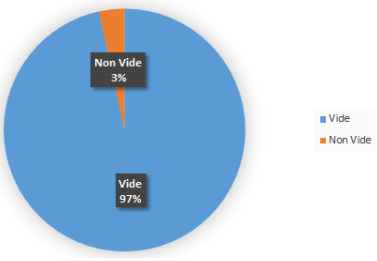

FIGURE 2 – Pourcentage de vide avant

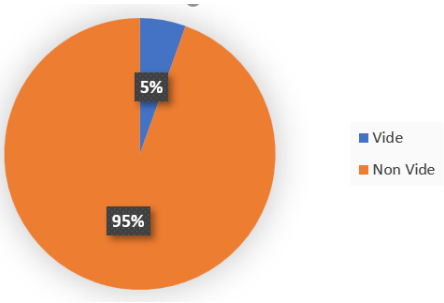

FIGURE 3 – Pourcentage de vide après.

3.2 Workload final

Nous avons deux workload pour évaluer notre moteur :

Workload 1

Le premier banc d'essai contient 500K avec 10000 requêtes.

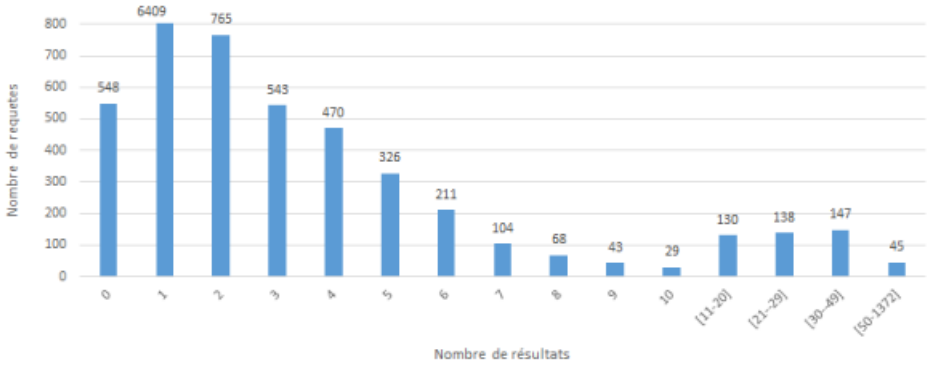

FIGURE 4 – Analyse des requêtes doublées.

Workload 2

Le deuxième banc d'essai contient 100K avec 2100 requêtes.

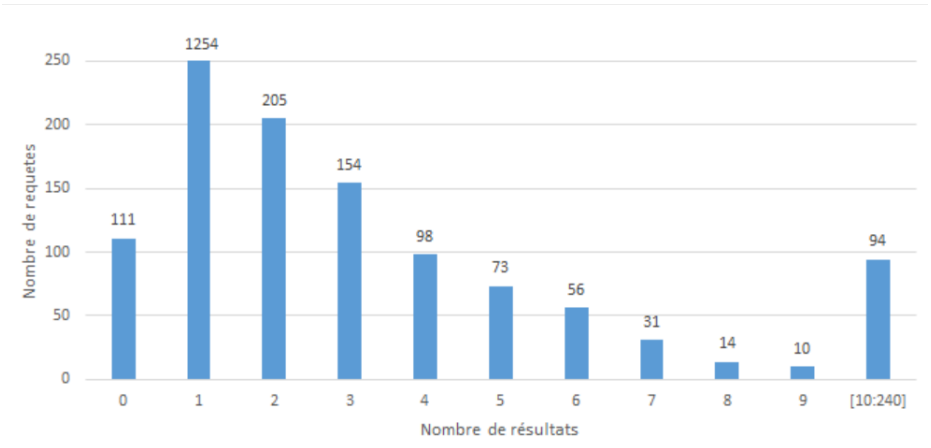

FIGURE 5 – Analyse des requêtes doublées.

4 Évaluation

4.1 Cold et Warm

Cold

Nous avons utilisé cold pour mesurer le pire des cas et évaluer la consommation des ressources, la mesure utilisée est le temps de réponse des requêtes.

Nous utilisons également cold pour mesurer le temps de création du dictionnaire/index et pour faire une comparaison avec Jena.

Warm

Nous avons utilisé warm pour mesurer le temps de réponse réelle de la requête, ainsi que, le temps moyen de réponse.

Pour utiliser Warm, il suffit d'ajouter -warm {pourcentage d'échauffement} pour que le programme démarre en mode warm.

Comparaison du temps de réponse cold vs warm

Temps de Réponse(ms)	Warm	Cold
500K	117	152
100K	15	16

4.2 Évaluation factorielle

Nous avons varié les données (500K/100K) et la taille de la mémoire (JVM 2G/4G) :

Création du dictionnaire et de l'index

Pour avoir un temps rapide de création, nous avons utilisé la création au moment du parsing dans MainRdfHandler.

Temps de Réponse(ms)	2G	4G
500K	598895	960407
100K	13685	15058

Temps de réponse sur toutes les requêtes

Temps de Réponse(ms)	2G	4G
500K	214	103
100K	17	16

4.3 Comparaison avec Jena

Évaluation de la complétude

Nous avons comparé les résultats produits par notre moteur avec les résultats de Jena qui sont fiables.

Création d'objet QuerySave
Nous avons utilisé une classe pour modéliser la sauvegarde de la requêtes et de toutes les informations nécessaires pendant le traitement.

Cette approche nous a aidé à faciliter la vérification des résultats avec jena ainsi que, l'utilisation de warm (pour randomizer les requêtes).

Création des identifiants pour chaque requête
Pour identifier les requêtes, nous avons généré un identifiant unique pour chacune de ces requêtes.

L'id nous a beaucoup aidé même dans le traitement des doublons, cet id est stocké dans QuerySave.

La création de l'id se fait par le hachage des patterns :

$$condition = P + Hashcode(Predicat) + O + Hashcode(Object)$$

Vérification du résultat
L'idée est de stocker les résultats dans un objet QuerySave pour comparer les résultats de notre moteur à ceux de jena (après la transformation sous la même forme que notre moteur). Les requêtes fausses sont exportées dans un fichier pour améliorer notre programme.

Résultat de complétude
Notre moteur arrive à avoir 100% de complétude après la correction des erreur que nous

avions dans la construction d'index (stockage de valeurs doublées dans quelques triplets).

Évaluation du temps

Création des structures de données

Temps de Réponse(ms)	Moteur	Jena
500K	5490	1000308
100K	1611	13893

Réponse sur toutes les requêtes

Temps de Réponse(ms)	Moteur	Jena
500K	214	163
100K	16	16

Nous Remarquons que notre moteur est beaucoup loin de Jena en terme de création du modèle de données, par contre la structure Hexastore permet l'accès rapide aux données avec index, ce qui permet d'avoir un temps de réponse meilleur.

5 Création du cache

Nous avons créé un cache de données pour améliorer le temps de réponse de la requête dans le cas où une requête similaire est déjà passée.

L'idée est d'enregistrer des (idReq,Resultats) dans un fichier que nous importons au début du programme.

Cet id est unique, ce qui fait que dés que nous trouvons que l'id est déjà exécuté, nous récupérons directement le résultat au lieu d'utiliser le traitement.

Temps de Réponse(ms)	Sans Cache	Avec Cache
500K	214	66
100K	16	13

6 Conclusion

Nous avons bien réussi à construire un moteur qui utilise l'approche hexastore pour répondre aux requêtes en étoile.

L'inconvénient de notre moteur c'est la création du modèle de données qui prend beaucoup de temps (l'une des causes que nous avons remarqué c'est la vérification de l'existence de la donnée dans la structure avant l'insertion).
Par contre, notre moteur a un temps moyen de réponse réel qui est pertinent ($\simeq 10\mu s$).