Université De Montpellier - Faculté Des Sciences
Année Universitaire 2022 - 2023
Master Informatique Spécialité Génie Logiciel
HAI914I - Gestion des données au delà de SQL (NoSQL)

Rendu Final :

Implémentation d'un mini moteur de requêtes en étoile

Étudiants :

DABACHIL Ali
RAKKAOUI Merwane

Table des matières

1 Présentation du projet

L'objectif de ce projet dans le cadre du module HAI914I est d'implémenter l'approche hexastore afin de pouvoir interroger des données RDF vues en détail dans le cours ainsi que tout le nécessaire servant à l'évaluation de requêtes en étoile. Ces requêtes sont exprimées dans la syntaxe SPARQL.

Ce projet a été découpé en plusieurs étapes menant jusqu'à son aboutissement final. Ces étapes consistaient à mettre en place les structures de données nécessaires nous permettant de pouvoir évaluer des requêtes en étoile. Un dictionnaire a été créé, celui-ci est un ensemble de couples <clé, valeur> dans lequel la clé (un entier) va associer une valeur (élément d'un patron de triplet RDF). Une fois que ces requêtes sont évaluées, une analyse de performance s'impose pour notre moteur de requêtes afin de mieux comprendre son comportement. Notre système sera par la suite comparé au logiciel libre **Jena**, considéré comme un système Oracle.

Tout le contenu de ce projet est accessible depuis le lien Github suivant : Projet

2 Guide d'exécution du programme

Notre programme peut s'exécuter de plusieurs manières. Vous pouvez ouvrir le contenu de notre projet disponible sur Github via un IDE, comme Eclipse ou IntelliJ IDEA. Plus simple encore, vous retrouverez un fichier exécutable d'extension JAR afin de le lancer via un terminal de commandes sous Windows exclusivement.

2.1 Lancement de notre programme

2.1.1 Via un IDE Java

Pour pouvoir exécuter notre code via un IDE Java, il y a juste à se placer dans le classe Main du projet afin de le lancer. Il faut penser à indiquer des arguments d'entrée de programme en vous rendant dans *Run > Edit Configurations...* si vous êtes sur IntelliJ IDEA ou bien dans *Run > Run Configurations...* si vous expérimentez Eclipse. Vous pourrez ainsi spécifier les arguments d'entrée nécessaires au bon fonctionnement du programme.

Ces arguments en entrée du programme correspondent à ceux qui se trouvent dans la commande permettant d'exécuter le JAR exécutable.

2.1.2 Via un JAR exécutable

Depuis un terminal de commandes, vous devez exécuter la commande ci-dessous :

```
java -jar <nom_du_fichier.jar> [liste options ...]
```

2.2 Options en ligne de commandes

Différentes options en ligne de commandes pour l'exécution de notre programme sont possibles et en voici une liste ci-dessous.

1. **-queries** <dossier_requêtes> : dossier comportant les fichiers de requêtes d'extension queryset
2. **-data** <dossier_données_RDF> : dossier comportant les fichiers de données RDF

③ **-output** <dossier_résultats> : dossier dans lequel va se trouver des données statistiques sous forme d'un fichier .csv

④ **-Jena** : activation de la vérification de la correction et la complétude du système en utilisant Jena comme un oracle

⑤ **-warm** <pourcentage> : un pourcentage de requêtes en entrée sera sélectionné au hasard pour rendre notre système chaud

⑥ **-shuffle** : permutation aléatoire réalisée sur nos requêtes d'entrée

3 Conception et mise en oeuvre du programme

3.1 Structure de notre projet

Notre projet comprend plusieurs packages dans lesquels on retrouve les classes suivantes :

■ Package benchmark :

QueryEntry.java : Classe QueryEntry décrivant la nature que peut avoir un ensemble de requêtes.

Contenu de la classe :

```
1   public class QueryEntry {
2
3   public enum QueryTag {
4       // Qi : requête composée de i patterns
5       // i variant de 1 à 4 ici
6       // DUP : requête apparaissant plusieurs fois dans le jeu de requêtes
7       // NA : requête ne donnant pas de réponses
8       Q1(1), Q2(2), Q3(3), Q4(4), DUP(5), NA(6) ;
9
10      public int numVal;
11
12      QueryTag(int numVal) {
13          this.numVal = numVal;
14      }
15
16  }
17  public String queryString;
18  public Set<QueryTag> tags = new HashSet<>();
19
20 }
```

■ Package dictionary :

Dictionary.java : Classe Dictionary contenant la structure de données qui est un dictionnaire.

■ Package indexes :

Index.java : Classe Index contenant la structure de données qui est un index.

■ Package qengine.program :

Main.java : Classe Main constituant le point d'entrée du programme.

MainRDFHandler.java : Classe contenant une méthode *handleStatement* utilisée pour le parsing des données RDF issues de notre jeu de données.

QuerySet.java : Classe permettant de filtrer un jeu de requêtes afin d'avoir l'ensemble le plus pertinent pour notre benchmark.

3.2 Présentation des structures de données utilisées

3.2.1 Dictionnaire

Pour définir un dictionnaire, on a opté pour le choix d'une BidiMap. Une BidiMap est une interface générique Java prenant deux paramètres (BidiMap<K,V>) avec K étant le type de la clé et V étant le type de la valeur. C'est un choix intéressant en raison du fait que grâce à une BidiMap on peut accéder à une valeur à partir de sa clé et vice-versa. Le dictionnaire sera construit progressivement dès lorsque l'on lit une donnée RDF.

3.2.2 Index

Nos index sont représentés sous la forme d'une Map<Integer, Map<Integer, ArrayList<Integer>>>. En prenant en compte l'approche hexastore, nous avons créé 6 index exactement (SPO, SOP, PSO, OPS, POS et OSP). Pour être plus précis d'un point de vue structurel, un index est un arbre pour lequel la racine est un sujet (S), un prédicat (P) ou un objet (0) avec des descendants qui eux-même ont des descendants. Ces descendants peuvent être soit des sujets, soit des prédicats ou bien des objets.

Illustration tirée de notre cours *RDF Data Management & SPARQL Query Processing* :

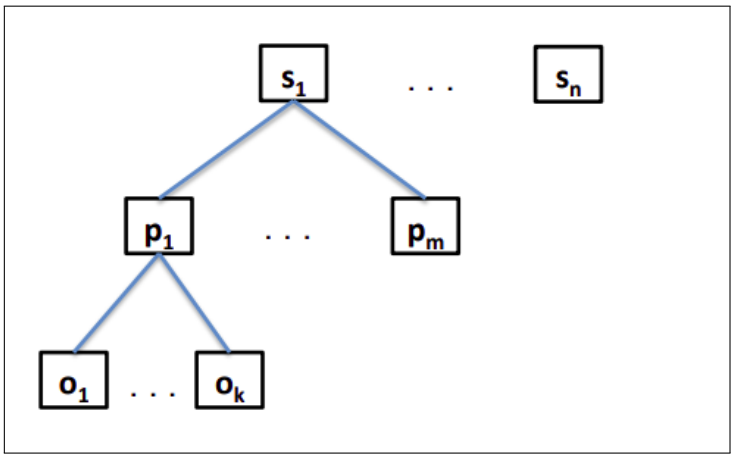


FIGURE 1 – Représentation d'un index SPO sous forme d'arbre

Un constructeur paramétré pour nos index est défini et ce dernier prend en paramètre un dictionnaire et le nom du fichier de données RDF à lire. La construction des index repose sur la construction du dictionnaire. Le parsing est fait deux fois en tout (une fois pour le dictionnaire et une fois pour l'index). Chaque élément d'un patron de triplet (<Sujet, Prédicat, Objet>) va avoir une clé unique qui lui est associé.

Pour construire l'index, on va à chaque fois récupérer la clé à partir de la valeur d'un patron de triplet, présent dans le dictionnaire créé. Ce qui va nous donner à la fin, un ensemble de triplet d'entiers. Une telle structure de données nous permet d'avoir un accès en lecture en temps constant au niveau complexité.

3.3 Évaluation de requêtes en étoile

Pour ce qui est de l'évaluation de requêtes en étoile, on a tout simplement évalué chaque sous-requête d'une requête pour en recueillir les résultats afin d'en faire l'intersection à la fin.

Grâce à notre structure d'index, on est capable de récupérer les résultats pour chaque sous-requête. Pour que la recherche soit efficace, il est nécessaire d'utiliser le bon type d'index selon la position de l'information recherchée dans une requête. Si on souhaite chercher seulement un sujet et que celui-ci se trouve en première position dans un patron de triplet, les index adéquats pourraient être POS ou OPS.

3.4 Informations liées au temps en terme de performance

Pour mesurer les performances de notre système, on a plusieurs données de temps en milliseconde et ils font parti des données statistiques que nous avions dû générer.

- ➢ temps de lecture des données RDF : temps mis pour parser toutes les données
- ➢ temps de lecture des requêtes : temps mis pour lire chaque requête d'un queryset
- ➢ temps de création du dictionnaire
- ➢ temps de création des 6 index
- ➢ temps total d'évaluation du workload : temps mis pour renvoyer toutes les réponses en ayant parcouru entièrement un jeu de requêtes
- ➢ temps total d'exécution : somme de tous les temps listés avant

3.5 Exportation de résultats

Les résultats sont exportés par défaut vers un dossier nommé *output*, dans lequel on retrouve un fichier csv reprenant les informations liées à l'exécution et un fichier au format txt contenant les réponses aux requêtes évaluées. Les informations d'exécution comportent plusieurs colonnes. Suite à l'exécution de l'application permettant de générer des bancs d'essai, on produit un fichier csv contenant différentes stats concernant le jeu de requêtes.

Description du contenu d'un fichier csv obtenu suite à l'évaluation de requêtes :

1. **Colonne 1** : chemin d'accès vers le jeu de données
2. **Colonne 2** : chemin d'accès vers le dossier contenant les jeux de requêtes
3. **Colonne 3** : nombre de triplets RDF
4. **Colonne 4** : nombre de requêtes
5. **Colonnes 5 à 11** : liste des temps énumérés précédemment

4 Évaluation et analyse des performances (Notre système vs Jena)

4.1 Préparation du banc d'essai

Génération de données et requêtes RDF Pour la génération de données RDF, nous avons utilisé l'outil WatDiv qui permet de générer des jeux de données ainsi que des jeux de requêtes en fixant une taille désirée (500K ou 2M par exemple). Cet outil génère un nombre important de requêtes en double et de requêtes ne donnant aucun réponse. WatDiv nous propose différents fichiers de requêtes séparés dans une arborescence simplifiée. Les fichiers de données RDF et les fichiers relatifs aux queryset sont dans des dossiers différents.

Application de manipulation de jeu de requêtes Afin de produire des jeux de requêtes pertinents et de meilleure qualité, nous avons créé une application permettant de modifier certains paramètres :

- ➢ Suppression de manière aléatoire des requêtes afin de réduire le queryset à une taille désirée
- ➢ Équilibrage des proportions de requêtes de Q1, Q2, Q3 et Q4 dans le jeu de données (par suppression)
- ➢ Ajustement de la proportion de requêtes contenant n'importe quel tag (Q1, Q2, Q3, Q4, DUP, NA) pour qu'elle atteigne un pourcentage désiré (par suppression)

Exemple de scénario : Création d'un jeu de données composé de 40000 requêtes à 50% de requêtes NA et mêmes proportions pour chaque requêtes de Q1,Q2,Q3 et Q4

Différentes étapes :

— filterTag(50, NA) : méthode de filtrage de requêtes NA (ne donnant pas de réponses) avec une proportion de 50% du jeu de requêtes initial

— equalize() : méthode permettant l'équilibrage des proportions de requêtes provenant de Q1, Q2, Q3 et Q4

— randomRemove(40000) : méthode effectuant la suppression aléatoire de requêtes garantissant les bonnes proportions pour chaque type de requêtes

Commentaire : Pour que la méthode *randomRemove* fonctionne convenablement, cette application a besoin d'un jeu de requêtes initial assez varié et très grand. La détermination d'un tag pour une requête donnée se fait de la façon suivante :

— Tag NA : L'idée est de lancer une exécution à partir de Jena sur un jeu de données de taille 2M (jeu de données utilisé pour d'autres expériences) afin d'obtenir des résultats d'évaluations de requêtes. Si Jena ne renvoie pas une réponse pour une requête donnée, celle-ci aura un tag NA associé.

— Tag DUP : Si lors du parsing d'un fichier de requête, la requête courante est déjà présente dans le queryset, les prochaines requêtes identiques auront un tag DUP.

— Tag Qi : Défini à partir du nom de fichier d'origine. (l'outil WatDiv génère des fichiers préfixés par Q_(1/2/3/4)).

Description du contenu d'un fichier csv obtenu suite à l'éxécution de l'application filtrage de jeu de requêtes :

1. **Colonne 1** : chemin d'accès vers le jeu de données
2. **Colonne 2** : chemin d'accès vers le dossier contenant les jeux de requêtes
3. **Colonne 3** : nombre de triplets RDF
4. **Colonne 4** : nombre de requêtes
5. **Colonne 4** : nombre de requêtes avec le tag NA
6. **Colonne 5** : nombre de requêtes avec le tag DUP
7. **Colonne 6** : nombre de requêtes avec les tags NA et DUP
8. **Colonne 7** : nombre de requêtes Q1
9. **Colonne 8** : nombre de requêtes Q2
10. **Colonne 9** : nombre de requêtes Q3
11. **Colonne 10** : nombre de requêtes Q4
12. **Colonne 11** : pourcentage de requêtes avec le tag NA
13. **Colonne 12** : pourcentage de requêtes avec le tag DUP
14. **Colonne 13** : pourcentage de requêtes avec les tags NA et DUP
15. **Colonne 15** : pourcentage de requêtes Q1
16. **Colonne 16** : pourcentage de requêtes Q2
17. **Colonne 17** : pourcentage de requêtes Q3
18. **Colonne 18** : pourcentage de requêtes Q4

4.2 Plan d'analyse des résultats

Afin de pouvoir interpréter nos résultats d'exécution, on doit d'abord récupérer les résultats de nos différents tests qui vont pouvoir être analysés. À partir de cela, nous pourrons relever les différents facteurs pouvant influencer les temps d'exécution.

4.2.1 Détails du plan d'analyse

1. Choix des métriques et paramètres à analyser
2. Pour chaque paramètre à analyser, produire un jeu de requête permettant de mesurer son impact.
3. Produire un jeu de requêtes standard permettant d'évaluer par comparaison l'impact des paramètres associés aux jeux de requêtes créés précédemment.
4. Pour chaque jeu de requêtes, lancer les tests d'éxécution pour notre moteur et Jena.
5. Pour chaque paramètre à analyser, comparer les statistiques de résultats d'éxécution de : notre moteur sur le jeu de requêtes lié à un paramètre, notre moteur sur le jeu de requêtes de référence, Jena sur le jeu de requêtes lié à un paramètre, Jena sur le jeu de requêtes de référence.
6. Interpréter et analyser les résultats de cette manière : comparer pour chaque système les résultats d'éxécution du jeu de requêtes lié à un paramètre et celui de référence pour révéler l'impact du paramètre choisi. Ensuite faire des comparaisons entre notre moteur et Jena pour savoir quel moteur est le plus sensible au paramètre.

4.3 Informations sur la machine utilisée pour les tests

4.3.1 Présentation de ses caractéristiques (Hardware et Software)

Configuration matérielle :

- **Processeur** : AMD Ryzen 7 3800X 8-Core Processor, 3893 MHz, 8 cœur(s), 16 processeur(s) logique(s)
- **RAM** : G.Skill Trident Z F4-3600C16D-16GTZNC, capacité 32 GB et timings CL16 19-19-39.
- **Stockage** : Force Series Gen.4 PCIe MP600 1TB NVMe M.2 SSD
- **OS d'exploitation** : Microsoft Windows 10 Professionnel
- **Environnement JAVA** : Java 11

La configuration donnée ci-dessus a été choisie pour notre moteur de requêtes et celui de Jena sur les queryset générés.

4.3.2 Mémoire vive à faire varier avec le volume de jeu de données

Lorsque l'on utilise des fichiers contenant des données RDF et des requêtes volumineuses, il est important de pouvoir faire varier la capacité de la RAM. Étant donné que le dictionnaire et l'index sont accesibles depuis la mémoire vive.

4.4 Métriques, Facteurs, et Niveaux

Liste de métriques permettant d'évaluer les performances de moteurs de requêtes RDF :

- Temps de création du dictionnaire
- Temps de création des index
- Temps total d'évaluation du workload
- Comportement du système à l'exécution (cold ou warm)

Remarque : Notre programme a été fait de façon à ce que même si l'application est lancée avec l'option **-Jena**, les index et le dictionnaire seront quand même générés. Or Jena a son propre système pour évaluer les requêtes et n'utilise pas nos index et notre dictionnaire. Pour Jena, le temps d'évaluation du workload correspond donc quasiment au temps d'éxécution total.

Facteurs rentrant en jeu pendant l'évaluation du système :

- Taille du jeu de données (dataset)
- Taille du jeu de requêtes (queryset)
- Proportions de requêtes (NA, DUP, Q1, Q2, Q3 et Q4)
- Configuation matérielle de la machine utilisée (hardware)

Pour des mesures cold et warm, on peut s'intéresser au temps total d'évaluation du workload comme métrique.

4.5 Protocole de test

Pour chaque expérience (variation du jeu de données et de requêtes, nature du queryset, système cold/warm..), on met en route le programme permettant l'évaluation des requêtes sur notre moteur et on fait ceci également avec Jena. Pour un queryset donné, on exécute notre programme 5 fois ce qui nous produira 5 fichiers de résultats. Pour chaque métrique, on fait la moyenne des 5 données produites à travers les résultats obtenus (moyenne des temps de création du dictionnaire, des index, etc..). Dans le cas où l'on souhaite faire ces expériences en cold, un redémarrage de la machine est effectué avant chaque exécution du programme.

Afin de vérifier la correction et la complétude de notre approche, nous nous sommes appuyés sur un jeu de données contenu dans le fichier *100K.nt* fourni de base dans le projet et sur le queryset se trouvant dans le fichier *STAR_ALL_workload.queryset*. Ensuite, nous avons lancé le programme servant à l'évaluation des requêtes afin de recueillir les réponses pour notre système et pour Jena. Une fois cela effectué, une comparaison des réponses obtenus s'est faite et nous avons constaté que les ensembles de réponses obtenus sont les mêmes.

4.6 Modèle de régression

Tableaux utilisés pour définir le modèle de régression pour chacun des systèmes :

NS	0,5gb	2gb	24gb
500k	6044	6272	4874
2m	20032	14163	14339

FIGURE 2 – Variations de la RAM et de la taille du jeu de données - Notre système

Jena	0,5gb	2gb	24gb
500k	12704	12395	12803
2m	66260	64816	67186

FIGURE 3 – Variations de la RAM et de la taille du jeu de données - Jena

Équations de régression obtenues pour chacun des systèmes :
Notre système : $y = 11322 - 1715.75xA + 5863.25xB - 1130.75xAxB$
Jena : $y = 39739.3 + 256.25xA + 26984.8xB + 206.75xAxB$
xA : taille de la mémoire | xB : taille de la donnée | xAxB : relation entre xA et xB

Pour les deux systèmes, les valeurs utilisées pour le calcul de régression étaient les niveaux de mémoire 0.5gb et 24gb. La niveau 2gb n'est présent qu'à titre indicatif.

INTERPRÉTATION :

Notre système : Une augmentation de la mémoire semble être intéréssante pour une réduction du temps d'éxécution. Comme attendu, le facteur le plus impactant est le jeu de données en entrée.

Jena Le coefficient de xA est négligeable face au base response time qui est de 39739.3 ms. La taille de la donnée a une grande influence sur le temps d'exécution.

4.7 Représentation graphique des résultats

Voici les histogrammes construits à partir des résultats de plusieurs expériences réalisées. Avant chaque présentation d'un histogramme, on retrouve le détail du jeu de données utilisé.

4.7.1 Le jeu de données REF

nom du fichier de données	nom du dossier des requêtes	nombre de triplets RDF	PERCENTAGE NA			
qengine-master\data\watdiv2M.nt	watdiv-mini-projet\testsuite\queries\10000	2147303	51.1425			
PERCENTAGE DUP	PERCENTAGE BOTH NA AND DUP	PERCENTAGE Q1	PERCENTAGE Q2	PERCENTAGE Q3	PERCENTAGE Q4	
60.4425	36.895	25.0	25.0	25.0	25.0	

FIGURE 4 – Jeu de données - REF

4.7.2 Impact du paramètre DUP

PRÉSENTATION DU JEU DE DONNÉES DUP :

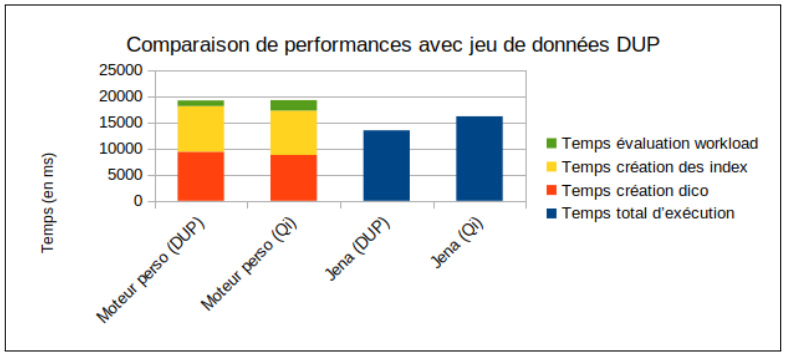
nom du fichier de données	nom du dossier des requêtes	nombre de triplets RDF	nombre de requêtes	PERCENTAGE NA		
qengine-master\data\watdiv2M.nt	watdiv-mini-projet\testsuite\queries\10000	529580	40000	0		
PERCENTAGE DUP	PERCENTAGE BOTH NA AND DUP	PERCENTAGE Q1	PERCENTAGE Q2	PERCENTAGE Q3	PERCENTAGE Q4	
100	0	100	0	0	0	

FIGURE 5 – Jeu de données - DUP

PRÉSENTATION DU JEU DE DONNÉES QI :

nom du fichier de données	nom du dossier des requêtes	nombre de triplets RDF	nombre de requêtes	PERCENTAGE NA		
qengine-master\data\watdiv2M.nt	watdiv-mini-projet\testsuite\queries\10000\Q1	2147303	40000	1.605		
PERCENTAGE DUP	PERCENTAGE BOTH NA AND DUP	PERCENTAGE Q1	PERCENTAGE Q2	PERCENTAGE Q3	PERCENTAGE Q4	
88.7475	0.25	100.0	0.0	0.0	0.0	

FIGURE 6 – Jeu de données - Qi



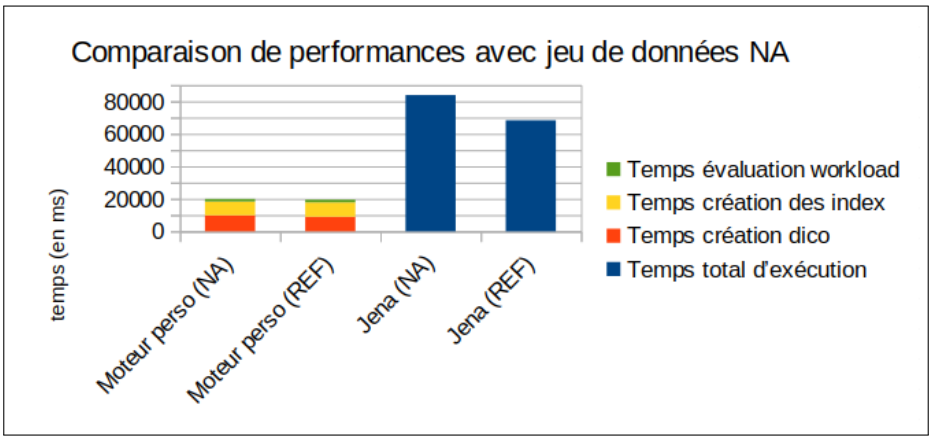
INTERPRÉTATION : Dans un jeu de requêtes contenant uniquement des requêtes DUP, le système Jena est plus rapide que notre système. Au vu des résultats, il semblerait que Jena ait un mécanisme particulier permettant de ne pas ré-évaluer une requête et donc de directement donner le résultat sans le recalculer.

4.7.3 Impact du paramètre NA

Présentation du jeu de données NA :

nom du fichier de données	nom du dossier des requêtes	nombre de triplets RDF	nombre de requêtes	PERCENTAGE NA
qengine-master\data\watdiv2M.nt	watdiv-mini-projet\testsuite\queries\10000	2147303	40000	99.997505
PERCENTAGE DUP	PERCENTAGE BOTH NA AND DUP	PERCENTAGE Q1	PERCENTAGE Q2	PERCENTAGE Q3	PERCENTAGE Q4
28.945	71.055	25.0	25.0	25.0	25.0

Figure 7 – Jeu de données - NA



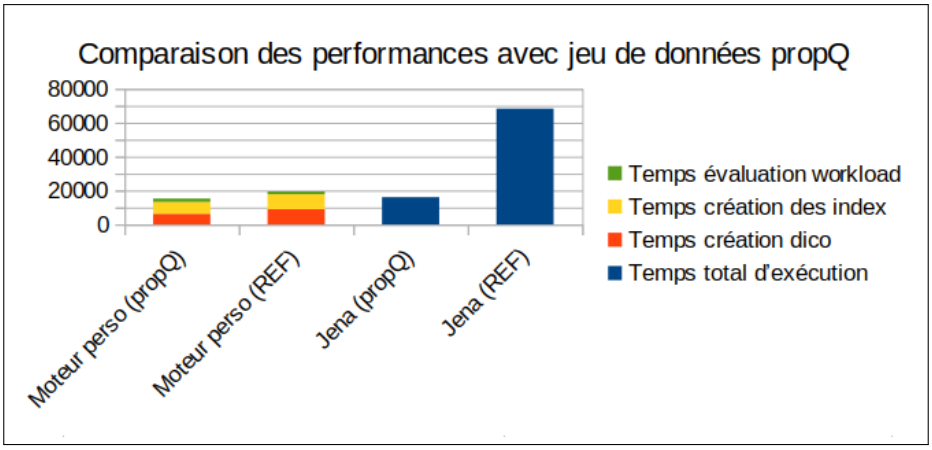
Interprétation : Dans un jeu de requêtes contenant uniquement des requêtes NA (donnant aucune réponse), il n'y a aucune différence de performance visible entre notre système et son jeu de données de référence. On constate que Jena éprouve plus de difficultés à pouvoir évaluer ce genre de requêtes par rapport à son jeu de données de référence. Notre système est plus rapide que Jena pour l'évaluation des requêtes NA. Jena a un mécanisme moins adapté pour ces requêtes là.

4.7.4 Impact du paramètre Qi

Présentation du jeu de données Qi :

nom du fichier de données	nom du dossier des requêtes	nombre de triplets RDF	nombre de requêtes	PERCENTAGE NA
qengine-master\data\watdiv2M.nt	watdiv-mini-projet\testsuite\queries\10000\Q1	2147303	40000	1.605
PERCENTAGE DUP	PERCENTAGE BOTH NA AND DUP	PERCENTAGE Q1	PERCENTAGE Q2	PERCENTAGE Q3	PERCENTAGE Q4
88.7475	0.25	100.0	0.0	0.0	0.0

Figure 8 – Jeu de données - propQ



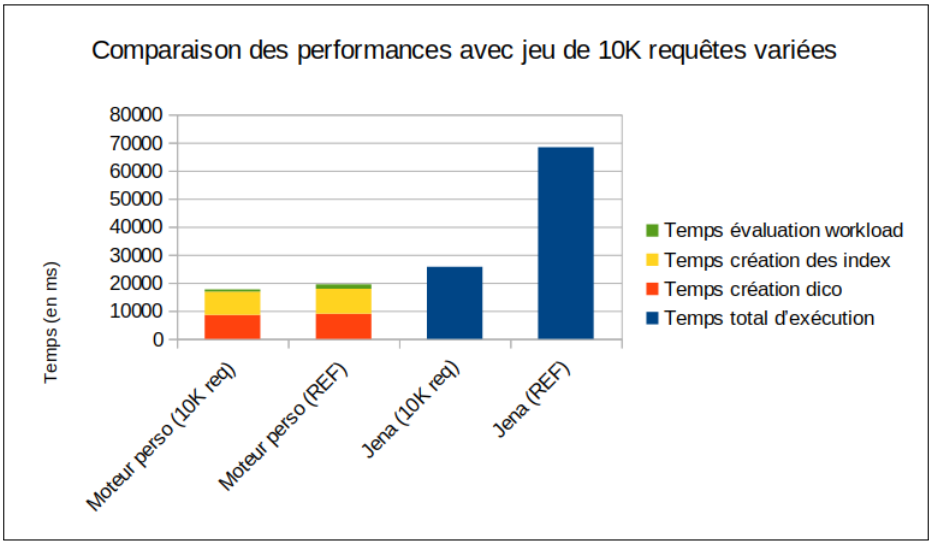
Interprétation : Dans un jeu de requêtes contenant uniquement des requêtes Q1, le système Jena est efficace pour évaluer ces requêtes.

4.7.5 Impact du paramètre nombre de requêtes

Présentation du jeu de 10K requêtes variées :

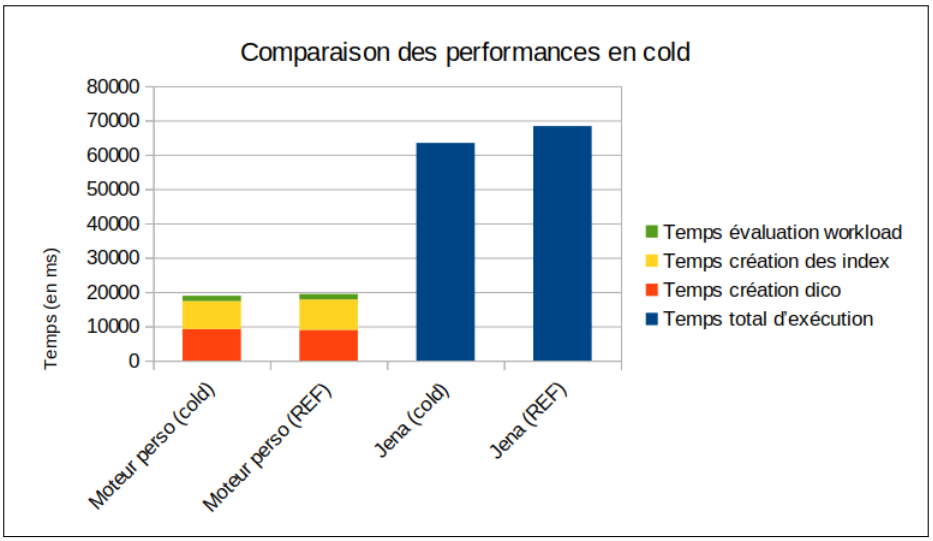
nom du fichier de données	nom du dossier des requêtes	nombre de triplets RDF	nombre de requêtes	PERCENTAGE NA	
qengine-master\data\watdiv2M.nt	watdiv-mini-projet\testsuite\queries\10000	2147303	10000	50.42	
PERCENTAGE DUP	PERCENTAGE BOTH NA AND DUP	PERCENTAGE Q1	PERCENTAGE Q2	PERCENTAGE Q3	PERCENTAGE Q4
61.17	36.359997	25.5	23.880001	25.5	25.119999

Figure 9 – Jeu de requêtes variées



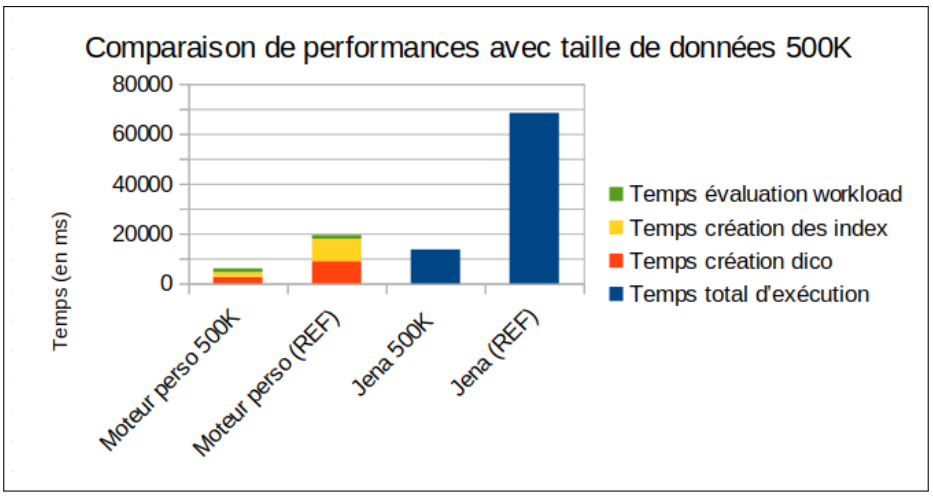
Interprétation : Pour notre système, le temps d'évaluation du workload ne représentant pas une partie majeure du temps total, une réduction du nombre de requête n'a donc pas un effet significatif. La réduction du temps d'évaluation du workload semble proportionnelle cependant. Pour Jena, la réduction du temps total d'exécution semble proportionnelle au nombre de requêtes à évaluer.

4.7.6 Impact du paramètre cold/warm



INTERPRÉTATION : Aucune différence significative n'est à relever.

4.7.7 Impact du paramètre taille du jeu de données



INTERPRÉTATION : Il apparaît que la taille du jeu de données en entrée est de loin le paramètre le plus influent sur le temps total d'éxécution quel que soit le système.

5 Conclusion

Ce projet nous a permis de mieux comprendre les différentes facettes de l'approche hexastore à partir de l'implémentation d'un moteur de requêtes en étoile. Son implémentation a nécessité de faire de bons choix au niveau des structures de données à utiliser, pour évaluer des requêtes sur tout jeu de données RDF de manière performante.

Des bancs d'essai ont été préparés pour ainsi simuler plusieurs expériences évoquées précédemment et en comprendre davantage sur le comportement de notre système.

L'analyse des performances de notre moteur s'appuie sur sa comparaison avec l'outil Jena, selon différentes métriques. À partir de cette analyse, on peut savoir si notre système est fiable ou non.

Des améliorations pourraient être apportées à notre système pour la prise en charge d'autre types de requêtes.

Nous tenons à adresser nos remerciements à M.Ulliana pour son aide apportée tout au long de ce projet.