Université De Montpellier
Faculté Des Sciences

Niveau : Master 2

Module : Gestion des données au delà de SQL (NoSQL)

HAI914I

Évaluation et Analyse des Performances du moteur KnowledgeGraph RDF

Supervisé par :
M. Federico Ulliana
M. Rodriguez Olivier
M. François Scharffe

Réalisé par :
Allouch Yanis
Kaci Ahmed

2021/2022

Table des matières

1 Préparation des bancs d'essais

Ce sous-projet adjacent au moteur RDF/SPARQL est fournie dans l'archive du rendu.

Afin de réaliser les bancs d'essais, nous avons utilisé WatDiv. Précisément, nous avons créé un sous projet adjacent au moteur RDF/SPARQL que nous avons nommée *watdiv-mini-projet* que nous fournissons dans l'archive de ce projet.
Cependant, la génération des bancs d'essais basée sur les patrons, produits des résultats insatisfaisants. En effet, le banc d'essais ainsi générés contient en **proportion aléatoire** :

— des requêtes sans réponses,

— des requêtes dédoublées un certain de nombre de fois,

— des requêtes avec un même nombre de conditions.

Nous avons constaté cela, en utilisant un calcul programmatique des résultats produits par la génération basée sur WatDiv.
La réalisation de ce calcul programatique est effectué en ajoutant à la classe processeur des requêtes de notre moteur RDF (*QueryProcessor*) les méthodes *getQueriesWithEmptySolutions()*, *getQueriesWithPaterns()* et *getQueriesWithDuplicates()*.

1.1 Requêtes sans réponses

On observe sur l'histogramme de la figure 1 qu'il y a environ cinquante pourcents de requêtes sans réponses sur deux dataset de triples différents générés par WatDiv. Ce pourcentage n'est pas souhaitable, parce qu'il ne correspond pas à une situation réelle selon notre point de vue. Pour cela, nous pensons qu'un pourcentage de 5 ou 10 pourcents est plus réaliste. Donc, on utilisera pour les bancs d'essais une proportion contrôlée de requêtes sans réponses de 10%.

1.2 Doublons dans les requêtes

On peut observer dans le graphique de la figure 2, certaines requêtes ont plus d'un doublon (entre 2 et 13 répétitions). On dénombre au total environ cinquante pourcents de doublons tous confondus dans ce banc d'essai. Ce ratio nous semble approprier puisqu'il pourrait modéliser une base de données ou il n'existe pas plus de deux utilisateurs requérant la même information. Cependant, le caractère aléatoire de cette proportion n'est pas souhaitable pour le banc d'essai, parce que nous avons pu observer des proportions allant de 20% a 90% de doublon par génération. Pour ce faire, on utilisera pour les bancs d'essais une proportion contrôlée de doublons fixe à 50%.

FIGURE 1 – le nombre de réponses aux requêtes sur une instance de 500K et de 2M triples pour 1200 requêtes

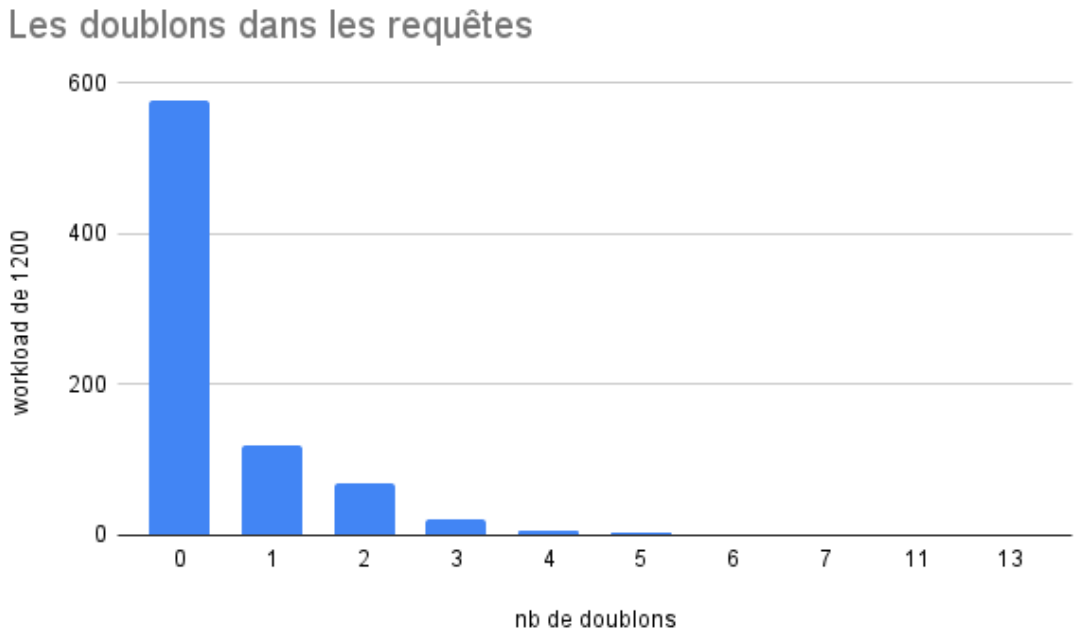

FIGURE 2 – Le nombre de requêtes répété par nombre de répétitions

1.3 Requêtes avec un même nombre de conditions

Dans la figure 3, on observe environ cinquante pourcents de requêtes simples (requêtes avec un seul patron) et cinquante pourcents de requêtes complexes (c.-à-d. requêtes avec plus de deux patrons). Ces pourcentages sont souhaitables pour les bancs d'essai puisqu'il pourrait modéliser un SGBD ou les recherches sont principalement ciblés sur un, deux ou trois critères, ou même plus, de façon aléatoire.

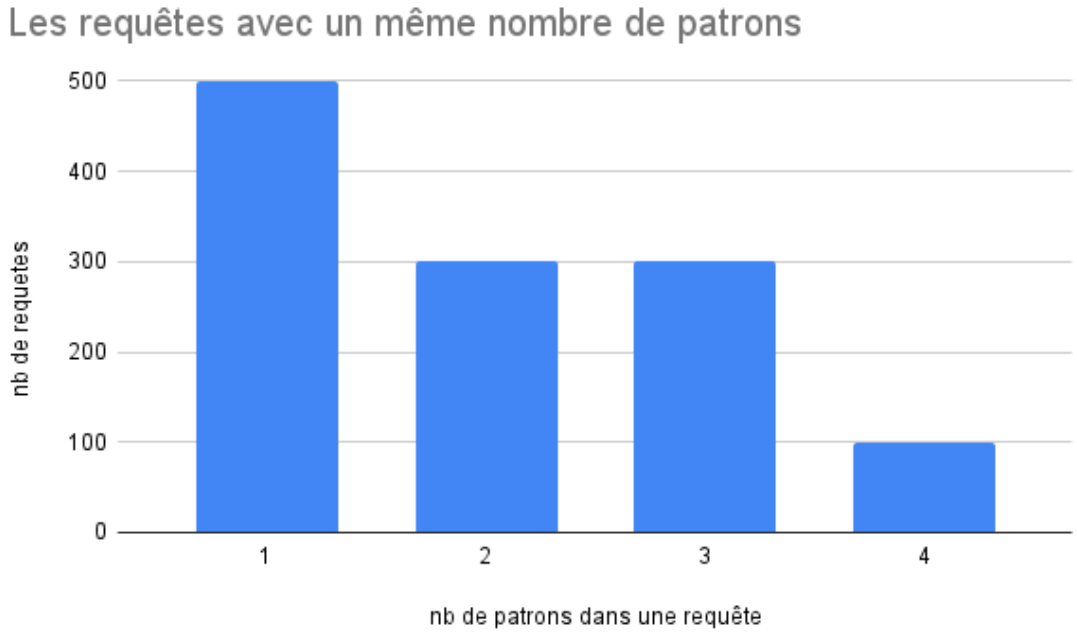

FIGURE 3 – le nombre de requêtes avec un même nombre de conditions (patrons de requêtes)

1.4 Contrôle d'un banc d'essai

Pour qu'un banc d'essai sois satisfaisant, nous devons contrôler deux critères :

1. le pourcentage de réponses vides,
2. et le pourcentage de doublons.

Pour ce faire, nous avons intégré un programme à notre moteur RDF (le package *benchmarkGenerator*) que nous allons décrire dans ce qui suit :

Pour commencer, Le programme utilise notre moteur RDF (le package *gengine.program*) pour évaluer et récupérer les statistiques du pourcentage de réponses vides et du pourcentage de doublons d'un banc d'essai passées en paramètres.
Ensuite, il crée un nouveau fichier de requêtes à partir du banc d'essai selon les pourcentages définis dans ce rapport, plus précisément :

1. On récupère du moteur, un ensemble de requêtes vides,
2. On soustrait ou bien on crée des requêtes sans réponses (ajout de caractères sans aucun sens) pour atteindre la quantité désirée de 10% de requêtes sans réponses,
3. On récupère du moteur, les requêtes en doubles,
4. Nous transformons cet ensemble de requêtes pour satisfaire nos critères d'acceptance d'un banc d'essai (50% de doublon au total),
5. Enfin, on sauvegarde dans un fichier au format *queryset* le nouvel ensemble de requête contrôlé ainsi crée.

Cette partie du processus est implémenté dans la classe *BenchmarkGeneratorProcessor*.

2 Hardware et Software

La réalisation de notre benchmarking sera fait avec le matériel informatique suivant :

- Processeur : AMD Ryzen 5 2600X Six-Core Processor 3.60 GHz,
- RAM Installé : 32,0 GB,
- Système : 64-bit operating system, x64-based processor,
- Hard Drives :
  - SSD : 120 GB SATA

Et le système d'exploitation suivant :

- OS : Ubuntu 20.04 LTS,
- Notre application est codée en Java et sera exécuté en CLI, avec le Java JDK 1.8.

Nous pensons que ce système est adapté a la réalisation du benchmark. En effet, Ubuntu est une distribution, recommandée, souvent utilisée pour faire ce genre d'expériences sur des benchmarks.

3 Métriques, Facteurs et Niveaux

3.1 Métriques

Dans ce TP, notre application est codée en Java. Les limitations de Java vis-a-vis des métriques, nous oblige à ne pouvoir mesurer que des métriques de temps d'exécutions. En effet, en C# ou C++, on aurait pu profiler la RAM et/ou la charge incombé au CPU en plus du temps d'exécution. Malgré ça, nous pouvons dans le cadre ce de TP distinguer plusieurs temps d'exécutions, à savoir :

— Le temps d'exécutions de la lecture des données.

— Le temps d'exécutions de la lecture des requêtes.

— Le temps d'évaluation des requêtes.

— Le temps total d'exécutions du programme.

Nous ne considérons pas les métriques temps de création du dictionnaire et des index, puisque nous n'avons pas ce contrôle au niveau de Jena et par conséquent, on ne pourra pas les comparer avec notre moteur RDF.

La mesure des temps d'exécutions est effectué en ajoutant des chronomètres (nous avons wrapper la classe standard *Stopwatch* dans la classe *Timer*) à certains endroits du code pour mesurer avec justesse les temps d'exécutions.

3.2 Facteurs

Les facteurs qu'on souhaite traiter sont le workload et la RAM allouée a la JVM. On pense observer, que ces deux facteurs aient un impact sur les temps d'exécutions. En effet, le workload a un impact évident sur les temps d'exécutions. Cependant, pour la RAM, comme elle est allouée a la JVM, et qu'il y a plusieurs processus de manipulation de la mémoire par la JVM qui sont parallélisés et indéterministes (voir une courte introduction à la gestion de la mémoire en Java), l'impact de la quantité de RAM sur les temps d'exécutions est moins évident à estimer.

3.3 Niveaux

Les niveaux choisis pour le facteur RAM sont de 2Go et 4Go. Pour ce faire, nous pouvons attribuer ces valeurs en CLI en manipulant les options [1] de la JVM nommé *-Xms* et *-Xmx*. Pour le facteur Workload, nous utilisons un premier workload composé de 5K requêtes et 500K données. Le second workload est composé de 15K requêtes et 500K données. Dans les deux cas, nous contrôlons les critères d'acceptances, tel que les requêtes sont composées à 50% de doublon et à 10% de requêtes vides.

Enfin, dans cette expérience, le facteur principal est le workload. La RAM étant un facteur secondaire.

3.4 Métriques cold et warm

Nous avons abordé en cours, la façon de déterminer si une métrique était respectivement, cold ou warm. Dans ce TP, parmi les temps d'exécutions, on distingue d'une part les métriques cold suivantes :

1. Voir la documentation oracle et baeldung.com

1. temps d'exécutions de lecture du workload.

D'autre part, les métriques warm suivantes :

1. temps d'exécutions de l'évaluation du workload,
2. temps d'exécutions total du moteur.

Pour calculer les métriques warm, on doit adapter l'exécution du moteur RDF pour chauffer le système avec 30% de requêtes choisies au hasard sans compter le temps d'exécutions puis d'exécuter sur le banc d'essai au complet. Pour ce faire, nous avons implémenté un sous-programme s'occupant d'encapsuler cette responsabilité dans le package *metrics*. Le reste de l'exécution ne diffère pas d'une exécution normale.

4 Vérification de la correction et de la complétude

Avant de se lancer dans le benchmarking, on doit s'assurer que notre moteur RDF est correcte et complet. Pour ce faire, nous avons choisis le système oracle Jena Apache.
Nous avons intégré à notre application, dans le package *engineComparator*, le processus de vérifications implémentant les étapes suivantes pour vérifier la correction et la complétude de notre moteur RDF :

1. On doit disposer à l'entrée du programme deux paramètres à savoir : un fichier résultat de notre moteur RDF, qui contient des requêtes et leurs réponses. En deuxième paramètre, un fichier résultat de l'exécution de Jena que nous avons implémenté dans le package *qengineWithJena*,
2. Puis, nous parsons les requêtes et les réponses d'un fichier pour les sauvegarder dans une collection de *ParsedElement* que nous avons créée,
3. Ensuite, nous comparons la complétude en distinguant tout d'abords le nombre de *parsedElements* total, puis le nombre de réponses par requêtes,
4. Enfin, pour vérifier la correction, on vérifie tout d'abord la complétude (optimisation de temps de calcul) puis on vérifie si chaque réponse de chaque requête de notre moteur est dans Jena, et vice versa.

À la fin de ce traitement, nous avons deux réponses possibles :

— True : notre système est correct et complet sur les données fournies,

— False : notre système est incorrect et/ou incomplet sur les données fournies.

5 Protocole de test

Nous définissons le protocole de test de la manière suivante :

1. Générer avec WatDiv, un dataset d'une taille de 500K,
2. Générer avec les scripts nommé *regenerate_ queryset_ XYZ.sh* fournit, un queryset d'une taille arbitraire assez grande (pour nous faciliter la tâche de contrôle du queryset servant aux bancs d'essai),
3. Exécuter le benchmarkGenerator avec le workload et permettant de contrôler les critères d'acceptance du workload. Il produit en résultat un workload répondant aux critères d'acceptance,
4. Utiliser le workload contrôlé comme input pour l'implémentation de Jena que nous avons effectué et notre propre implémentation du moteur RDF,
5. Par la suite, il faut faire 10 exécutions en CLI avec chaque système pour obtenir assez de résultats permettant de calculer la moyenne robuste tel que nous l'avons abordés en cours,
6. Ensuite, nous avons implémenté un programme qui va automatiquement parser les 10 exécutions de chaque système et calculer la moyenne robuste pour chaque métrique et sauvegarder le résultat dans un fichier CSV final par système analysé,
7. Enfin, on peut récupérer ce fichier résultat final pour comparer les résultats des métriques entre Jena et notre implémentation.

Pour ce faire, nous avons créé un script bash permettant d'automatiser le processus, que nous fournissons dans le rendu du projet. ce script est intitulé *benchmark.sh*

6 Importance des facteurs avec un modèle de régression linéaire

Pour calculer l'importance des facteurs choisis dans ce TP (workload et mémoire vive) sur notre moteur RDF, on pose les variables suivantes :

— La variable X_A pour définir le niveau high et low de la RAM. Tel que :
  — X_A = -1 si la RAM vaut 2Go,
  — X_A = 1 si la RAM vaut 4Go.

— La variable X_B pour définir le niveau high et low du workload. Tel que :
  — X_B = -1 si le workload est de 500K données et 5K requêtes,
  — X_B = 1 si le workload est de 500K données et 15K requêtes.

Voici les résultats de l'expérience 2^2 des temps totaux moyen d'exécutions de notre moteur :

Tel que nous l'avons vu en cours, nous procédons à la résolution du système d'équation nous donnant le modèle de régression linéaire suivant :

$$y = 9005.44 - 205.75X_A + 4199.56X_B - 257X_AX_B$$

	RAM 2Go	RAM 4Go
Workload 500K-5K	4754.63 (ms)	4857.13 (ms)
Workload 500K-15K	13667.75 (ms)	12742.25 (ms)

TABLE 1 – Impacte de la quantité de RAM et du Workload sur le temps d'exécution du moteur RDF

Ce modèle de régression linéaire, nous apprends que la mémoire améliore les performances de 205.75 millisecondes (ms). En outre, plus le workload est important plus il impacte négativement le temps de réponse. Enfin, l'interaction entre nos deux facteurs (workload/RAM) réduit de 257 ms le temps de réponse.

On peut conclure que le workload est bien le facteur principal et que la RAM est un facteur secondaire dans l'évaluation des performances. Par ailleurs, leur interaction est plus ou moins faible.

7 Comparaison de notre système avec Jena

Nous présentons une comparaison des métriques sélectionnés dans ce rapport afin d'observer lequel des deux systèmes est le plus rapide à évaluer un ensemble de requêtes SPARQL sur une base de donnes RDF.

Pour ce faire, nous avons réalisé une expérience 2^2 avec comme facteur principale, un workload composer d'une base de donne RDF et un ensemble de requêtes SPARQL. Comme facteur secondaire, on a choisi la mémoire vive accorder à la JVM. Nous avons effectué 10 exécutions par expériences afin de calculer des moyennes robustes des temps d'exécutions tel que nous l'avons abordés dans le cours et en TP.

Les figure 4, figure 5, figure 6 et figure 7 nous permettent d'observer que dans chaque expérience réalisée, notre moteur a été le plus rapide pour la lecture du workload mais plus lent dans l'évaluation de ce dernier.

Nous pouvons expliciter cette différence des temps d'exécutions par les raisons suivantes :

— La conception logicielle réalisée dans Jena est effectué par de nombreux experts (avec l'utilisation d'API développé par Google) qui maitrise mieux les modèles de développement Java adapter à ce type d'application que des étudiants qui font cela pour la première fois,

— Le temps de développement du moteur Jena et plus important que celui qui peut être fourni pour un projet effectuer par des étudiants,

— Après avoir fait une revue de code nous avons détecté plusieurs calculs qui auraient pu être optimisé, cependant par manque de temps nous n'avons pas pu tous les faire,

— Enfin, Jena Apache est un projet open-source complet, c'est pourquoi il peut exécuter des traitements supplémentaires à la lecture du workload et ainsi être plus lent que notre moteur qui effectue cette tâche de manière plus spécifique.

De plus, on observe sur la figure 4 et figure 6 que l'augmentation de la charge de travail n'impacte pas énormément le **temps d'évaluation du workload** de notre moteur RDF, alors que le temps d'évaluation du workload de Jena Apache est doublé. C'est-à-dire que notre moteur RDF est a priori scalable verticalement. Cependant, notre moteur reste toujours plus lent dans l'évaluation du workload.

Pour expliquer cela, nous savons[2] qu'il existe des mécanismes de découpages de lecture de la donnée par quantité de ligne, colonne ou taille de données lues. De ce fait, nous pensons qu'une des raisons du a cette augmentation importante du temps d'évaluation du workload provient du fait qu'il y a ce mécanisme implémenté dans Jena Apache.

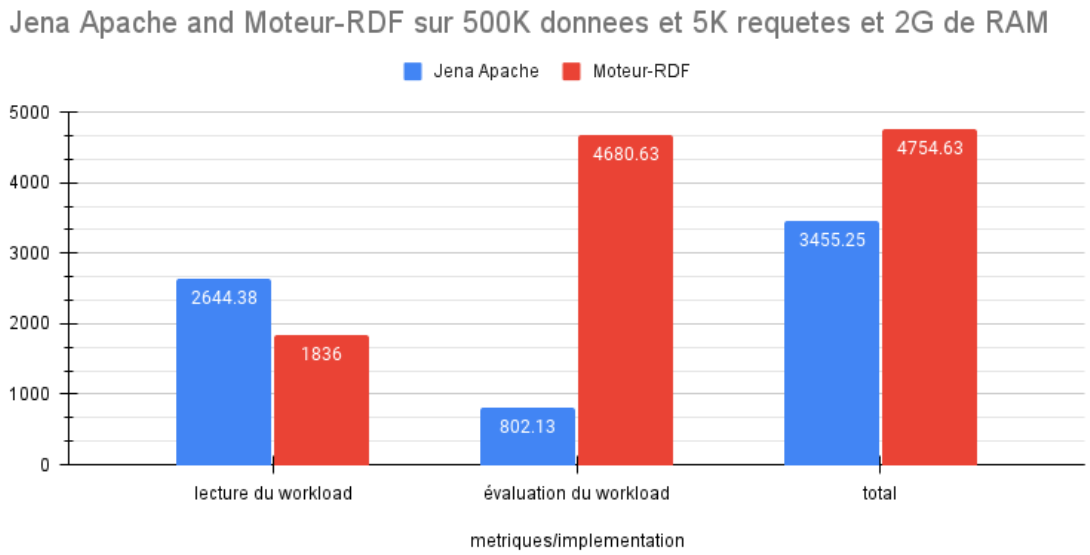

FIGURE 4 – 1er Histogramme de comparaison de notre moteur avec Jena sur les temps d'exécutions (ms)

2. Vu dans le module de M1, intitulé : HMIN122M - Entrepôts de Données et Big-Data

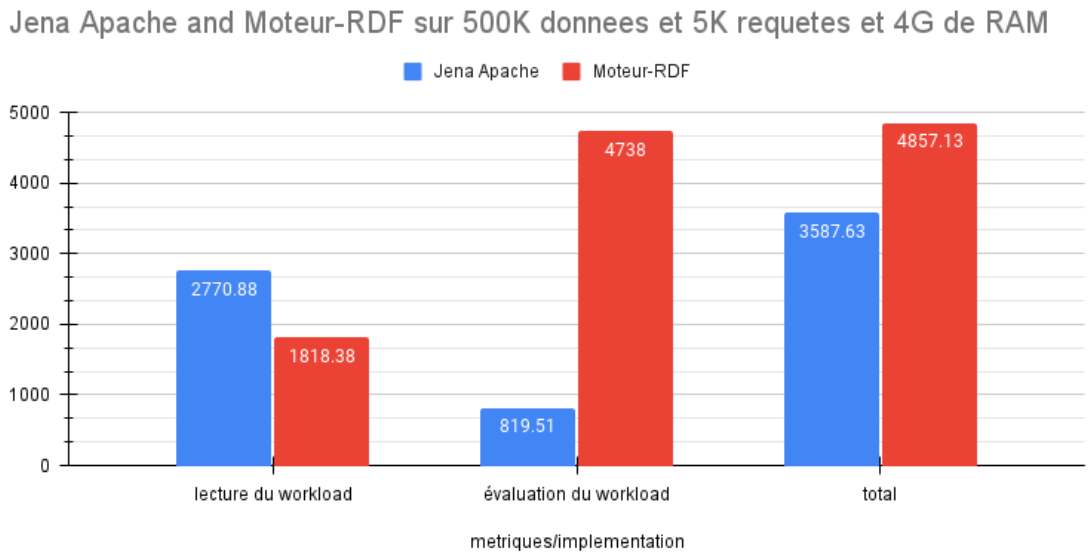

FIGURE 5 – 2eme Histogramme de comparaison de notre moteur avec Jena sur les temps d'exécutions (ms)

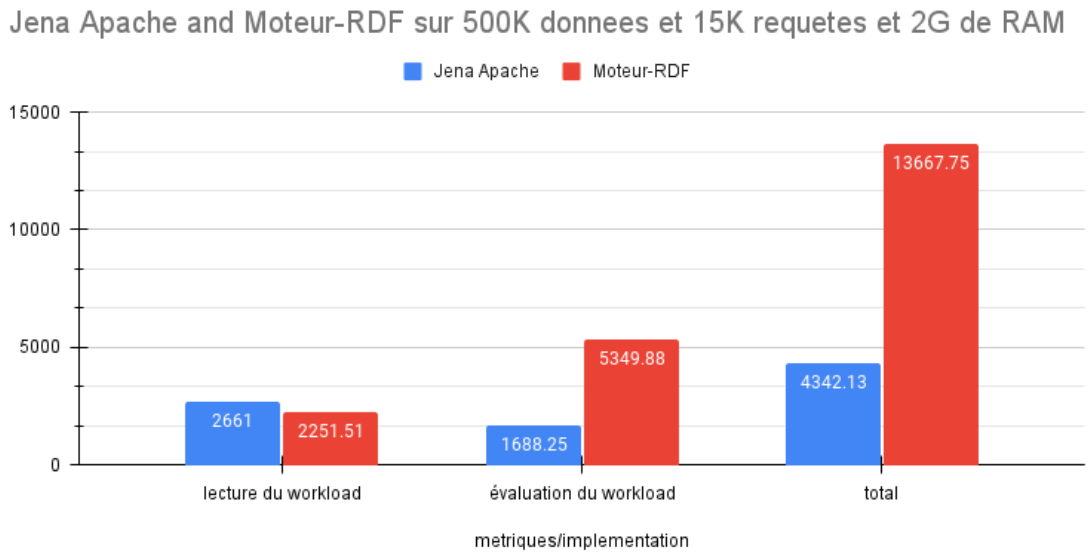

FIGURE 6 – 3eme Histogramme de comparaison de notre moteur avec Jena sur les temps d'exécutions (ms)

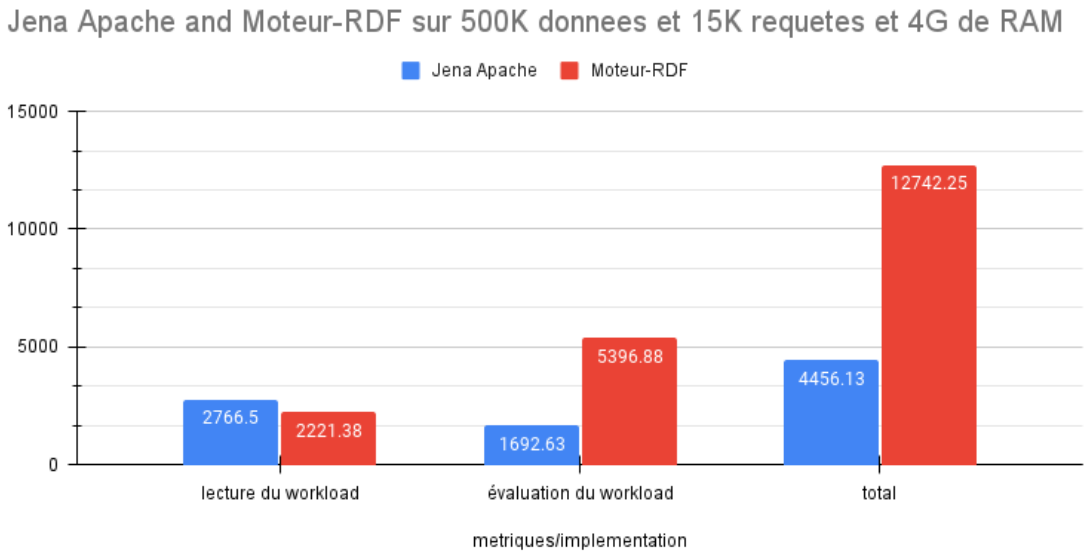

FIGURE 7 – 4eme Histogramme de comparaison de notre moteur avec Jena sur les temps d'exécutions (ms)