Universités de Montpellier

Faculté des Sciences de Montpellier

Département d'informatique

HAI916I

Gestion des données au-delà de SQL

Moteur KnowledgeGraph RDF

Rapport final

Réalisé Par :

RAMDANE Souhaib (M2 GL)

MEFLAH Wided (M2 IASD)

Encadré par :

Mr. Federico Ulliana

2023/2024

1. Préparation des bancs d'essais

1.1. Création du jeu de test

Nous avons employé WatDiv pour générer deux ensembles de données distincts : un premier ensemble de 500k triplets et un second de 2 millions. En outre, nous avons créé des fichiers de requêtes en nous basant sur le script fourni. Pour la génération des requêtes, nous avons utilisé des templates différents. Deux valeurs distinctes de paramètre ont été utilisés 1k et 10k.

Ensuite, nous avons fusionné les fichiers générés pour chaque paramètre dans un seul fichier. Cela a conduit à la création de deux fichiers de requêtes finaux, L'un contenait 6000 requêtes et l'autre en contenait 100k requêtes.

1.2. Requêtes avec réponses

Dans un premier temps, nous représentons le nombre de requêtes ayant obtenu des réponses en fonction des deux ensembles de données en les testant respectivement sur le premier fichier de requêtes (6000 requêtes) et le deuxième (100k requêtes).

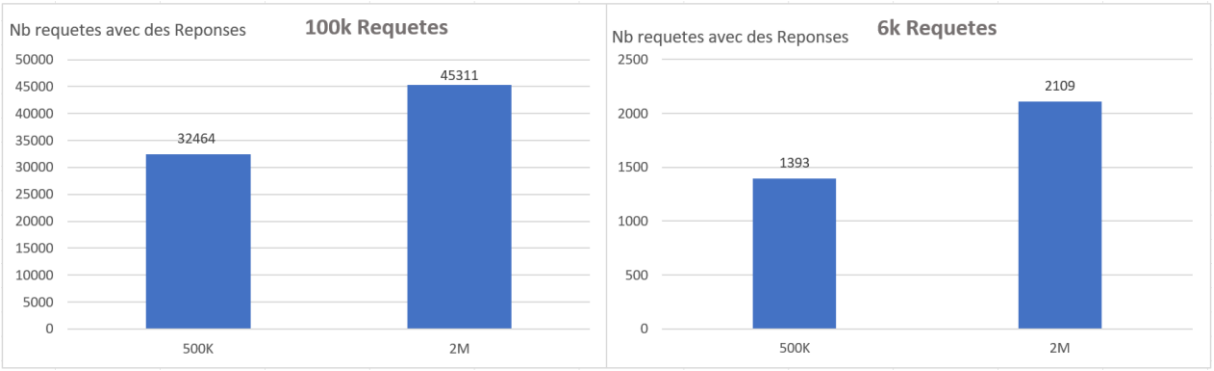

Figure 1 : le nombre de réponses aux requêtes sur une instance de 500K et de 2M

En analysant les deux histogrammes, on remarque tout d'abord qu'en fixant le nombre de requêtes à 6000 ou à 100 000 pour le test, le nombre de requêtes ayant obtenu des réponses augmente avec l'augmentation de la taille des ensembles de données. Dans un second temps, en fixant la taille de l'ensemble de données à 500 000, par exemple, et en calculant le pourcentage des requêtes ayant obtenu des réponses par rapport à chaque nombre de requêtes, on a constaté que pour 6000 requêtes, 23 % ont obtenu des réponses, tandis que pour 100 000 requêtes, ce pourcentage a augmenté à 32 %. Ainsi, le nombre de requêtes obtenant des réponses augmente avec l'augmentation du nombre de requêtes de test.

Dans un second temps, nous représentons le nombre total de réponses obtenues en fonction des deux ensembles de données en les testant respectivement sur le premier fichier de requêtes (6000 requêtes) et le deuxième (100k requêtes).

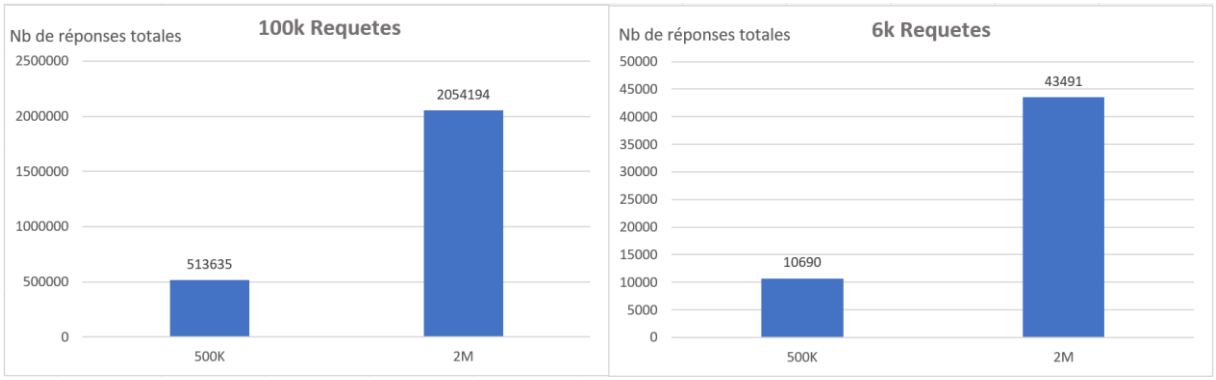

Figure 2 : – le nombre de réponses totales aux requêtes sur une instance de 500K et de 2M

Nous remarquons également que le nombre total de réponses augmente proportionnellement à l'augmentation de la taille des données et du nombre de requêtes.

1.3. Requêtes sans réponse

Nous représentons le nombre de requêtes ayant obtenu des réponses en fonction des deux ensembles de données en les testant respectivement sur le premier fichier de requêtes (6000 requêtes) et le deuxième (100k requêtes).

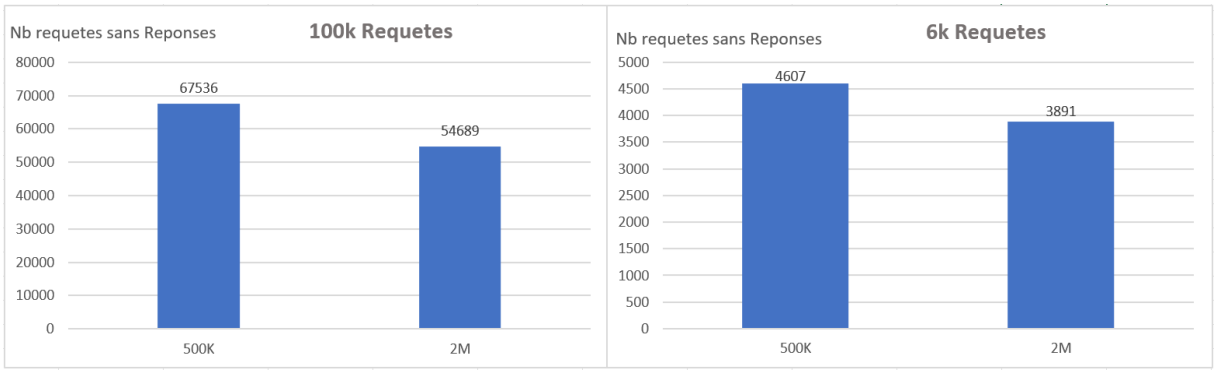

Figure 3 : le nombre de requêtes sans réponses sur une instance de 500K et de 2M

En général, il n'est pas souhaitable d'avoir un grand nombre de requêtes sans réponse. En se basant sur l'analyse effectuée précédemment, on en déduit qu'il faut une quantité plus importante de données et de requêtes pour obtenir de bons résultats.

1.4. Requêtes avec un même nombre de conditions

Nous représentons le nombre de requêtes en fonction du nombre de conditions respectivement sur le premier fichier de requêtes (6000 requêtes) et le deuxième (100k requêtes).

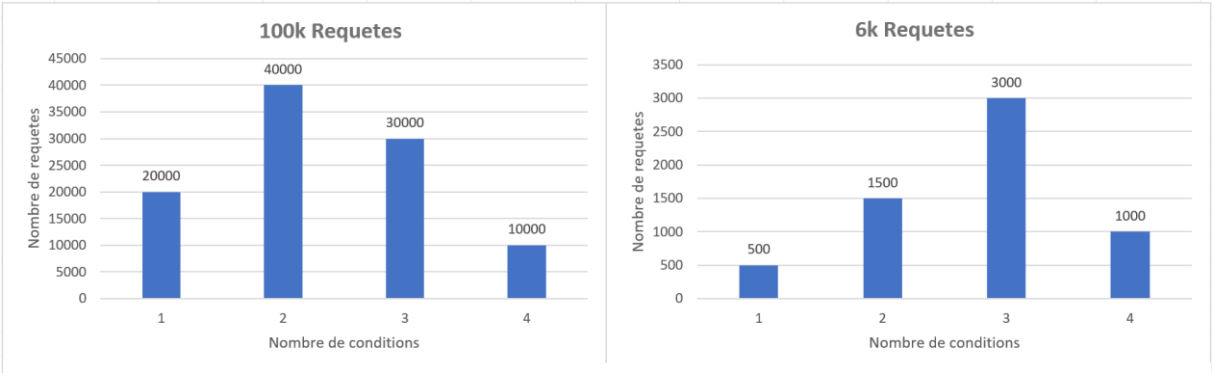

Figure 4 : le nombre de requêtes avec un même nombre de conditions (patrons de requêtes)

Nos deux ensembles de 100k et 6k de requêtes démontre une distribution bien pensée qui reflète une variété de cas d'utilisation réels. L'ensemble de 100k requêtes présente une répartition équilibrée, avec une forte représentation de requêtes à deux et trois conditions, simulant fidèlement les demandes courantes dans un environnement de production. Cette diversité assure une évaluation exhaustive des performances du système à travers un large spectre de complexité des requêtes.

L'ensemble de 6k requêtes, avec sa concentration en requêtes à trois conditions, offre un aperçu précis des performances du système sous une charge moyenne, qui est représentative des conditions opérationnelles typiques. Cette focalisation permet une compréhension approfondie des performances dans des scénarios fréquemment rencontrés, fournissant ainsi des données précieuses sur la robustesse et l'efficacité du système dans les conditions les plus communes.

La diversité incarnée de nombre de conditions dans les deux ensembles de 100k et 6k requêtes est fondamentale pour le benchmark, car elle permet d'assurer une évaluation des performances du système à la fois large et précise. En couvrant un spectre étendu de cas de test, des plus simples aux plus complexes, nous simulons une variété d'interactions que le système pourrait rencontrer en situation réelle, offrant ainsi un aperçu authentique de sa capacité à gérer différents volumes et types de charges de travail. Cette diversité est cruciale pour éviter les biais inhérents aux tests moins variés et pour garantir que notre benchmark ne favorise pas indûment certaines configurations ou types de requêtes. En fin de compte, une telle approche polyvalente assure que notre système est robuste, flexible et bien adapté pour répondre aux exigences

diverses et imprévisibles de l'utilisation en conditions réelles. De ce fait, on peut en déduire que l'efficacité du benchmark utilisé est démontrée.

1.5. Doublons dans les requêtes

Nous représentons le nombre de requêtes en fonction du nombre de répétitions respectivement sur le premier fichier de requêtes (6000 requêtes) et le deuxième (100k requêtes).

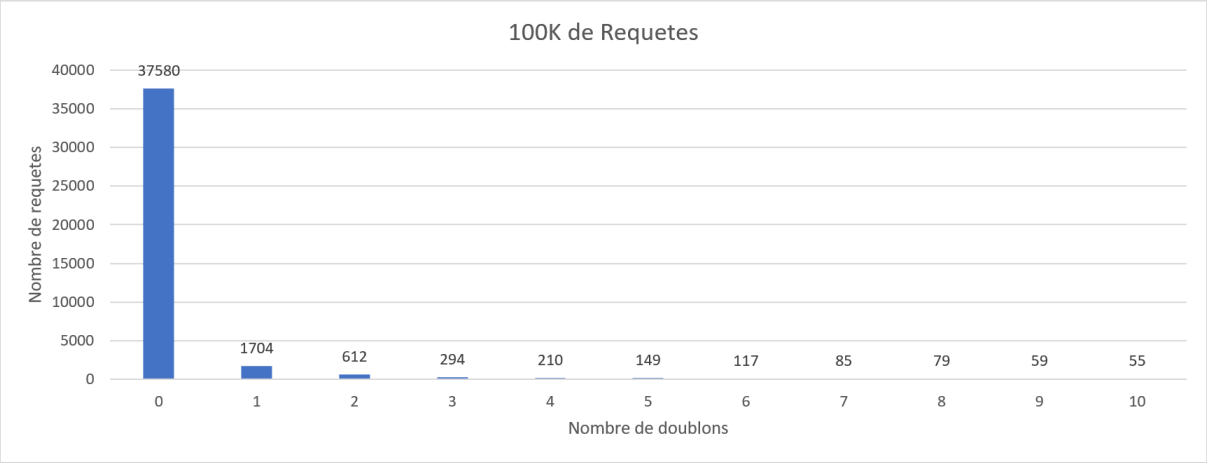

Figure 5 : Le nombre de requêtes répété par nombre de répétitions dans le workload de 100k

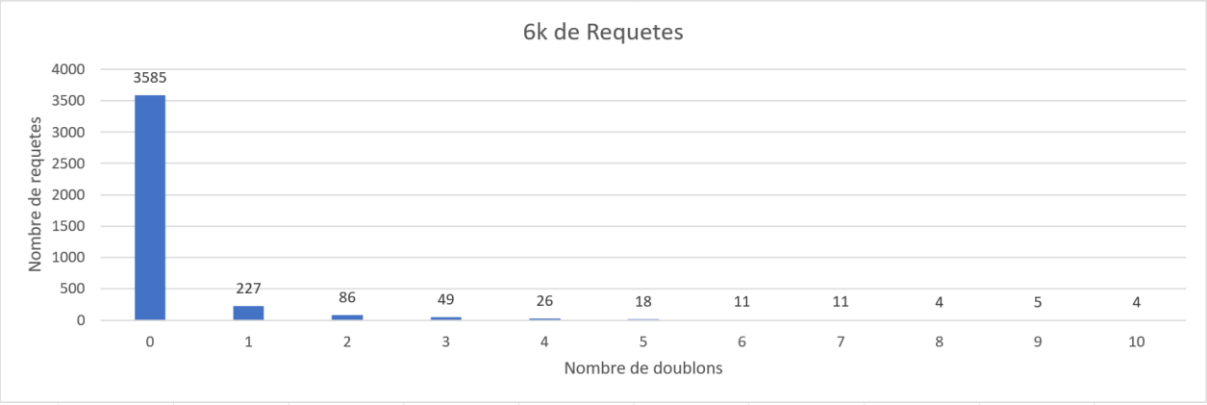

Figure 6 : Le nombre de requêtes répété par nombre de répétitions dans le workload de 6k

Les histogrammes pour les ensembles de 6k et 100k requêtes révèlent une majorité impressionnante de requêtes non répétées, mettant en lumière la diversité et la richesse des tests effectués. La présence de doublons, bien que plus limitée, peut être interprétée avantageusement, car elle permet d'évaluer la constance des performances du système face à des requêtes récurrentes, un aspect crucial dans des contextes réels où des requêtes similaires sont souvent répétées. Cette caractéristique offre une opportunité d'analyser l'efficacité de la mise en cache et des algorithmes d'optimisation. Ainsi, tout en soulignant la nécessité d'affiner la génération des requêtes pour réduire les doublons, le nombre important de requêtes uniques

dans les benchmarks confirme la qualité du test, garantissant que les performances du système sont testées de manière exhaustive et représentative.

un certain nombre de doublons dans les requêtes d'un benchmark peut être acceptable dans la mesure où cela contribue à l'évaluation efficace des performances des systèmes sans compromettre la diversité et la représentativité des requêtes évaluées.

1.6. Amélioration du benchmark utilisé

Afin de tirer parti des statistiques calculées et des analyses effectuées, nous proposons d'améliorer le jeu de tests utilisé en définissant des règles pour la génération des requêtes avec WatDiv : limiter à 10% le nombre de requêtes sans réponse et permettre un seul doublon par requête en augmentant le paramètre de génération de données.

2. Hardware et Software

2.1. Hardware et Software

Nous allons utiliser le hardware suivant pour la réalisation de notre benchmarking :

- Device : Pc Portable :DELL Inspiron 15 5000 Series
- Processor : Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz Intel64 Family 6
- Model 142 Stepping 10 1992
- Processeurs logiques : 8
- Nombre de cœurs : 4
- Mémoire RAM : 16 GB
- System type : Système d’exploitation 64 bits, processeur x64
- Disque dur : SSD 237 GB NVME M2

Nous allons utiliser le software suivant :

- Système d’exploitation : Windows 10
- Langage de programmation : Java
- JDK : 18

2.2. Adaptation de l’environnement à l’analyse des performances

Notre système est adapté à l'analyse des performances, car nous avons pris des mesures pour garantir un environnement de test cohérent et sans interférences. Nous veillerons à ce qu'aucun autre processus ne s'exécute sur l'ordinateur pendant les tests, afin de maintenir des conditions uniformes et de prévenir toute distorsion des résultats. Bien qu'une version de Linux sans interface graphique aurait pu offrir une légère amélioration en termes de ressources systèmes disponibles, la configuration actuelle est suffisamment robuste pour conduire une analyse des performances précise et fiable.

3. Métriques, Facteurs, et Niveaux

3.1. Métriques

Les métriques sont les suivantes :

- Temps de réponse
- Qualité des réponses
- Débit : Le nombre de requêtes pouvant être exécutées dans n'importe quelle période de temps.
- Utilisation/Exigences : Ressources consommées par le système (mémoire et taille du stockage)
- Scalabilité : dégradation du temps de réponse avec plus de données/utilisateurs, réduction du temps de réponse avec davantage de ressources.

3.2. Facteurs et leurs niveaux

Les facteurs avec leurs niveaux dans notre cas sont les suivants :

- CPU : i7-8550U, i3-8100
- La mémoire : 16 GB, 4GB
- Disque dur : 237 GB NVME, Hard drive 500GB
- Nombre de requêtes utilisé : 6K, 100k
- Jeu de données : 500k, 2M
- Version du système : windows 1064 bits

3.3. Ordonnancement des facteurs

Après notre analyse, nous avons déterminé que les facteurs principaux par ordre sont :

- Nombre de requêtes utilisées : on a également vu l'impact direct du nombre de requêtes sur l'évaluation du système.
- Jeu de données utilisé : on a vu précédemment l'impact direct de la taille des données utilisées sur l'évaluation du système.

Les facteurs secondaires comprennent :

- Taille de la mémoire
- CPU
- Type de disque dur
- Système d'exploitation

Cette hiérarchisation des facteurs nous a permis de concentrer nos efforts sur les éléments les plus impactants pour les performances, tout en considérant les autres facteurs pour une vue d'ensemble complète.

4. Évaluation des performances

4.1. Métriques cold ou warm

Dans l'évaluation des performances d'un système, il est crucial de choisir le type de démarrage, chaud ou froid, en fonction de la métrique évaluée. Voici notre analyse détaillée pour chaque métrique :

- Temps de réponse : Pour cette métrique, il est judicieux d'utiliser le démarrage chaud pour évaluer le temps de réponse habituel. Cela donne une perspective sur la performance dans des conditions normales.
- Qualité des réponses : La qualité des réponses peut être similaire dans les deux démarrages. Cependant, le démarrage chaud peut offrir une meilleure qualité si des données préchargées améliorent la performance du système.
- Débit : Le démarrage chaud est généralement choisi pour évaluer le débit. Les données préchargées permettent souvent d'atteindre des débits plus élevés par rapport à un démarrage à froid.
- Utilisation/Exigences : Pour évaluer les exigences maximales en ressources, il est recommandé d'utiliser un démarrage froid (cold run). Cela permet de mesurer

l'utilisation maximale de la mémoire et de la taille du stockage lorsque le système démarre à partir de zéro.

— Scalabilité : Pour évaluer la scalabilité du système, les deux démarrages peuvent être utiles. Un démarrage froid peut montrer la dégradation du temps de réponse lorsque le système est surchargé sans préparation, tandis qu'un démarrage chaud peut montrer comment l'ajout de ressources améliore la performance.

4.2. Réaliser ces mesures en pratique

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'exécution 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.

Pour calculer les métriques cold entre chaque test le système sera redémarrer pour être sûr que les tests soient réalisés à froid.

4.3. Vérification de correction et complétude

Dans le cadre de l'évaluation de notre moteur RDF, nous avons mis en place une procédure pour assurer sa correction et sa complétude. Pour ce faire, nous avons utilisé le système oracle Jena Apache comme référence. Cette procédure se décompose en plusieurs étapes clés :

— Préparation des Données pour la Comparaison : Nous avons commencé par générer un fichier CSV contenant, pour chaque requête en étoile, les résultats obtenus à la fois par notre moteur RDF et par Jena. Ce fichier permet une comparaison directe entre les deux ensembles de résultats.

— Parsing des Requêtes et des Réponses : Les données du fichier CSV ont été analysées et structurées. Cette étape nous a permis de stocker les requêtes et leurs réponses respectives de manière organisée, facilitant ainsi les étapes de comparaison ultérieures.

— Comparaison de la Complétude : Nous avons d'abord évalué la complétude de notre moteur en comparant le nombre total de ParsedElement générés par notre moteur et par Jena. Ensuite, nous avons comparé le nombre de réponses retournées par chaque requête, pour s'assurer que notre moteur RDF fournit un ensemble complet de résultats pour chaque requête en étoile.

— Vérification de la Correction : Après avoir confirmé la complétude, nous avons procédé à la vérification de la correction. Pour chaque requête, nous avons vérifié si chaque réponse générée par notre moteur RDF était présente dans les résultats de Jena, et vice versa. Cette étape cruciale garantit que notre moteur RDF ne fournit pas seulement l'ensemble complet des réponses, mais aussi que ces réponses sont correctes et fiables par rapport à la référence établie par Jena.

En résumé, cette procédure méthodique nous a permis de valider avec confiance la correction et la complétude de notre moteur RDF, en le comparant de manière exhaustive avec le système reconnu de Jena Apache. Les résultats obtenus confirment l'efficacité et la fiabilité de notre système dans le traitement des requêtes en étoile.

4.4. Importance des facteurs avec un modèle de regression lineaire

On a réalisé une expérience 2 puissance 2 en feasant varier la taille des données et des requêtes avec la taille de la mémoire, on pose les variables suivantes :

La variable XA pour définir le niveau high et low de la RAM. Tel que :

— X_A = -1 si la RAM vaut 4Go,

— X_A = 1 si la RAM vaut 16Go.

La variable XB 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 à la puissance 2 des temps totaux moyen d'exécutions de notre moteur :

	RAM 4 Go	**RAM 16 Go**
Workload 500K-100K	6873 ms	6710 ms
Workload 500K-6K	3484 ms	3436.5ms

on obtient le système d'équations suivants :

$$3484 = q0 - qA - qB + qAB \quad (low,low)$$

$$3436,5 = q0 + qA - qB - qAB \quad (hight,low)$$

$$6873 = q0 - qA + qB - qAB \quad (low,hight)$$

$$6710 = q0 + qA + qB + qAB \quad (hight,hight)$$

En résolvant ce système d'équation on obtient le modèle de régression suivant :

$$y = 5125.875 - 52.625X_A + 1665.625X_B - 28.875X_A X_B$$

Ce modèle de régression, nous apprend que la mémoire améliore les performances de 52.625 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 28.875 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.

4.5. Comparaison de notre système avec Jena

Nous avons mené une série d'expériences comparatives entre notre moteur RDF et Jena Apache, en mettant l'accent sur un workload composé d'une base de données RDF de 500k et un ensemble de requêtes SPARQL(100k et 6k). Deux configurations de mémoire vive, 16GB et 4GB, ont été testées pour évaluer leur influence sur les performances. Nous avons exécuté chaque expérience 5 fois pour obtenir des moyennes robustes des temps d'exécution, comme discuté en cours et en TP.

Notre étude comparative entre le Moteur-RDF et Jena a révélé des aperçus significatifs sur la manière dont la taille du workload influence les performances de chaque système. Les variations de la mémoire vive allouée à la JVM n'ont pas eu d'impact notable sur les temps d'exécution pour aucun des deux systèmes. Cela suggère que les deux moteurs sont optimisés pour fonctionner efficacement dans des environnements à mémoire variable, du moins dans la plage de mémoire que nous avons testée.

Les résultats indiquent que notre moteur a systématiquement surpassé Jena dans la lecture du workload. Cependant, il a montré des temps d'évaluation plus longs. Cette performance pourrait être attribuée à plusieurs facteurs :

- Expertise en Développement : Jena bénéficie de l'expérience et du savoir-faire d'experts en développement logiciel, y compris l'utilisation d'APIs conçues par Google, qui sont mieux adaptées à ces applications que notre équipe d'étudiants qui s'y confronte pour la première fois.
- Temps de Développement : Le moteur Jena a bénéficié d'un temps de développement considérablement plus long comparé à notre projet étudiant, ce qui permet une maturité et une optimisation accrues.

— Optimisation du Code : Notre revue de code a identifié des opportunités d'optimisation qui n'ont pas été exploitées en raison de contraintes de temps.

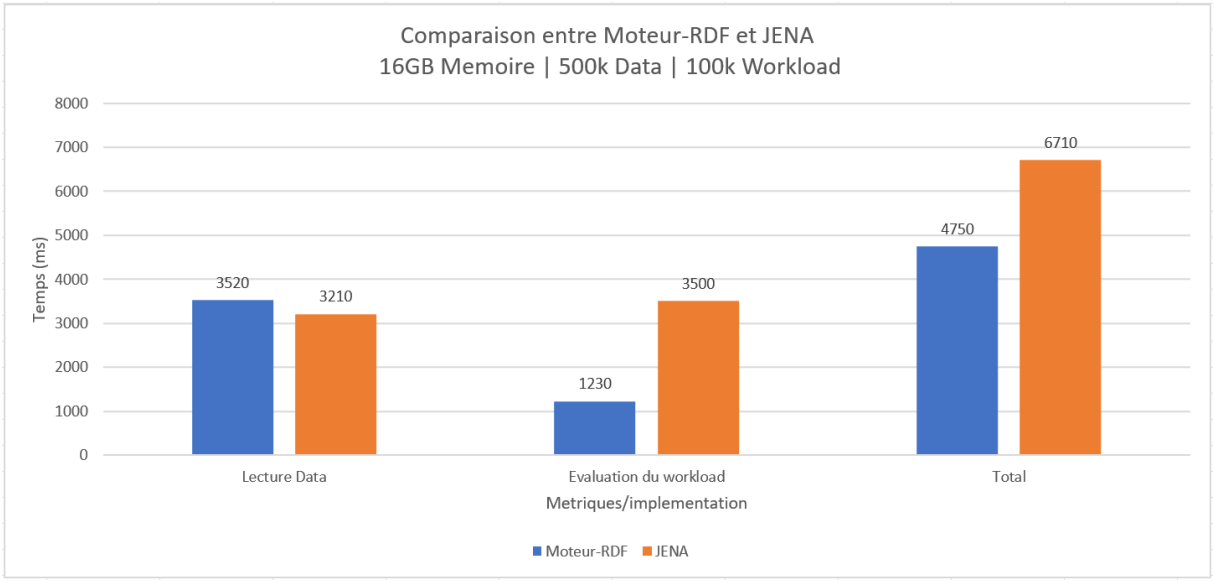

Figure 7 : 1er Histogramme de comparaison de notre moteur avec Jena sur les

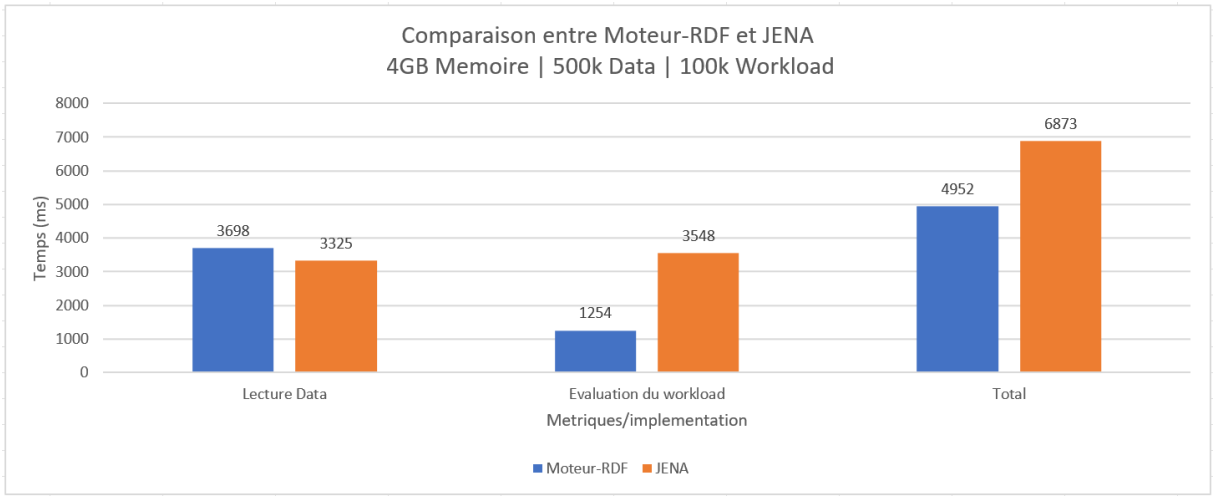

Figure 8 : 2eme Histogramme de comparaison de notre moteur avec Jena sur les

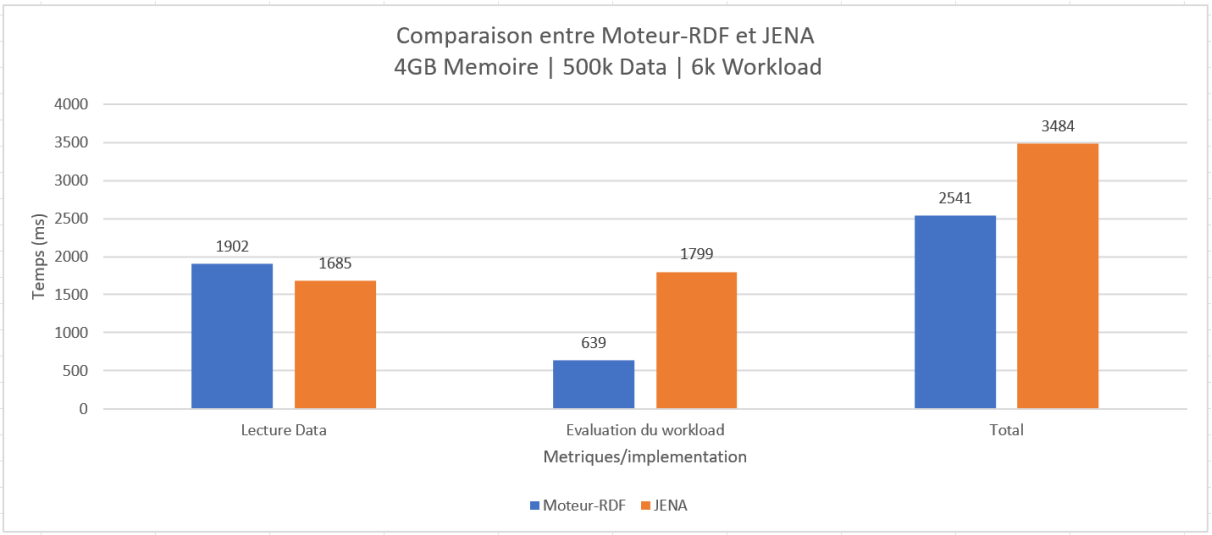

Figure 9 : 3eme Histogramme de comparaison de notre moteur avec Jena sur les

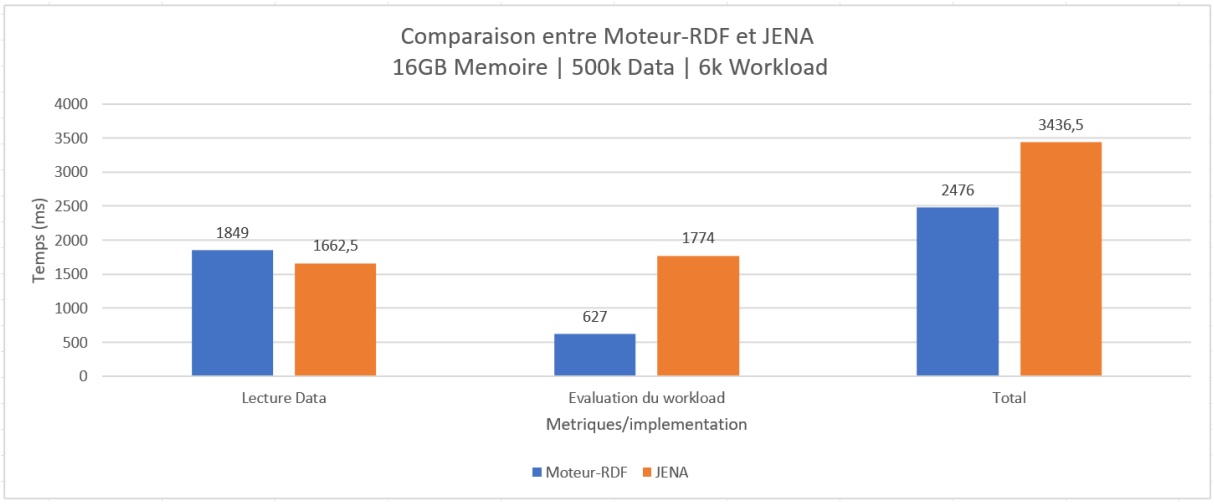

Figure 10 : 4eme Histogramme de comparaison de notre moteur avec Jena sur les