


Rapport 2 HAI914I
Un moteur d'évaluation de requêtes en étoile

Eric GILLES & Morgan NAVEL

22 décembre 2024

1 Introduction

Ce rapport va s'intéresser à l'évaluation de la correction ainsi que de la performance de notre moteur de requêtes en étoile. Il est question ici de mener différents **benchmarks**, pour ainsi comparer sa qualité sur plusieurs volumes de données différentes et sur plusieurs templates de requêtes différentes. En plus de cela, nous comparerons notre moteur à **Integraal**, ainsi qu'à un moteur de requête d'un autre groupe du même projet.

Un premier rapport a été rédigé pour présenter le moteur de requête en étoile et toutes ses fonctionnalités. Il est accessible depuis le lien suivant : **Lien du rapport 1**

Tout le contenu de ce projet est accessible depuis le lien Github suivant :
https://github.com/eric-gilles/StarQueryEngine

2 Benchmarks

2.1 Micro-benchmarks

Durant le développement de notre projet, nous avions établi plusieurs tests unitaires afin de nous guider sur un développement de code correct. Nous avons ainsi écrit plusieurs tests unitaires pour notre classe *Dictionnaire* pour l'ajout et la récupération de l'encodage d'un *Term*, sur différent cas de succès et d'erreur pour ainsi évaluer tous les cas possibles. Nous avons fait de même pour notre **Hexastore**, pour les méthodes d'ajout d'atome RDF, l'évaluation de requête simple avec un unique atome RDF. Cependant, cette approche ne permet pas d'évaluer un système dans sa globalité. En effet, plusieurs tests unitaires corrects n'impliquent pas forcément que 2 fonctionnalités s'intègre bien ensemble.

2.2 Real-life applications

Notre application n'a pas pu être testée sur des données d'application réelles. Comme expliqué dans le cours, il est difficile de trouver des données d'application réelles, car, la plupart du temps, l'application étant propriétaire, ces données sont :

— Trop précieuses
— Trop sensibles

2.3 Standard benchmark

Le **Standard Benchmark** est une solution qui vise à combiner les avantages des deux approches. Il repose sur l'utilisation de modèles de données (templates) qui imitent des données réelles. Cette standardisation est le fruit d'une collaboration entre diverses entités, entreprises et scientifiques, qui s'accordent pour établir des normes communes. Dans notre projet, nous allons utiliser **WatDiv** pour nous permettre de réaliser ce **benchmark**.

WatDiv (ou Waterloo SPARQL Diversity Test Suite) est un outil de **benchmark** qui permet de mesurer les performances d'un système de données RDF. Il est capable de générer des données RDF et des requêtes SPARQL.

3 Préparation des bancs d'essais

Pour évaluer les performances de notre moteur, nous avons préparé plusieurs jeux de données RDF et requêtes SPARQL à l'aide de **WatDiv**. Cette préparation a nécessité un prétraitement rigoureux pour garantir la qualité et la pertinence des bancs d'essais. Nous avons calculé les facteurs d'échelle nécessaires pour obtenir ces fichiers de datasets contenant 500K et 2M triplets (5 et 19). Les commandes utilisées pour les générer sont présentées ci-dessous.

```
bin/Release/watdiv -d model/wsdbm-data-model.txt 5 > ../data/500K.nt
```

Commande de génération de dataset pour 500K triplet

```
bin/Release/watdiv -d model/wsdbm-data-model.txt 19 > ../data/2M.nt
```

Commande de génération de dataset pour 2M triplet

WatDiv génère aussi des requêtes SPARQL basées sur des templates variés. Cependant, ces requêtes doivent être pré-traitées pour garantir leur pertinence dans nos tests. Les étapes de prétraitement incluent :

— **Suppression des doublons :** Les doublons dans les requêtes générées ont été supprimés afin d'éviter les biais dans les mesures de performance.
En tout, sur les tous les fichiers de requêtes du dossier "testsuite", il y a 6 802 requêtes en doublons sur 13 000 requêtes.
— **Réduction des requêtes à zéro réponse :** Les requêtes qui ne retournent aucune réponse ont été identifiées (4 909 requêtes avec 0 réponses). Nous avons limité leur proportion à 5 % du total des requêtes, car un taux élevé de requêtes à zéro réponse peut fausser les résultats des benchmarks (245 requêtes avec 0 réponses conservées).

Nous avons réalisé une analyse du prétraitement afin d'obtenir les données ci-dessus (voir en Annexes Tableau 1).

4 Évaluation des performances

L'évaluation des performances est une étape cruciale, pour pouvoir les comparer à l'existant. Dans cette section, nous allons discuter du matériel que l'on a utilisé, de notre méthodologie, des résultats obtenus pour ainsi les comparés dans un premier temps avec **Integraal**, puis avec un autre groupe. Il est possible de faire des comparaisons dans plusieurs configurations : à froid, à chaud, dans une nouvelle JVM à chaque fois, etc.

Dans notre cas, les tests à froid sont possibles uniquement si l'on ne fait pas de prétraitement, c'est-à-dire une étape de nettoyage des données (duplication de requête, requête nulle trop abondante). En effet, si un prétraitement est activé, nous serons obligés d'avoir au préalable exécuté les requêtes sur les données pour nettoyer les requêtes nulle par exemple, des solutions sont possibles : enregistrement des requêtes pour chaque cas. Cependant, cette méthode requiert beaucoup de temps donc nous n'avons pas opté pour celle-ci.

4.1 Matériel et Logiciel

L'ensemble de nos tests ont été effectués sur des machines d'emprunts du département informatique.

Pour l'évaluation des performances, nous avons créé une méthode (voir FIGURE 1) dont l'objectif est de calculer le temps d'exécution d'un ensemble de requêtes.

Au premier abord, il semble naturel de faire une moyenne, cependant ce résultat ne permet une bonne comparaison entre différent moteur de requête. Dans notre cas, on veut savoir lequel est le plus rapide, on a donc juste besoin de faire la somme des temps d'exécution de chaque requête en étoile. Ainsi, nous obtenons un résultat comparable et qui n'inclut pas des erreurs liées à l'approximation et à la conversion de type. Nous avons en plus de cela éviter toutes opérations inutiles, plus particulièrement les logs, durant l'évaluation des requêtes.

```java
public static Map<String, Long> speedTest(List<StarQuery> queries,
RDFHexaStore store, FactBase factBase) {
    Map<String, Long> durations = new HashMap<>();

    long startTimeIntegraal = System.currentTimeMillis();
    for (StarQuery starQuery : queries) {
        FOQuery<FOFormulaConjunction> foQuery = starQuery.asFOQuery();
        FOQueryEvaluator<FOFormula> evaluator =
    GenericFOQueryEvaluator.defaultInstance();
        evaluator.evaluate(foQuery, factBase);
    }
    long endTimeIntegraal = System.currentTimeMillis();
    long durationIntegraal = endTimeIntegraal - startTimeIntegraal;
    if (durationIntegraal == 0) durationIntegraal = 1;
    durations.put("Integraal", durationIntegraal);

    // Mesurer le temps d'exécution de notre HexaStore
    long startTimeHexastore = System.currentTimeMillis();
    for(StarQuery starQuery: queries){
        store.match(starQuery);
    }
    long endTimeHexastore = System.currentTimeMillis();
    long durationHexastore = endTimeHexastore - startTimeHexastore;
    if (durationHexastore == 0) durationHexastore = 1;
    durations.put("Hexastore", durationHexastore);


    // Calculer les moyennes
    System.out.println("Execution time of Integraal: " + durationIntegraal + " ms");
    System.out.println("Execution time of our HexaStore: " + durationHexastore + " ms"
    return durations;

}
```

FIGURE 1 – Méthode de test de rapidité

4.2 Nos performances

Les résultats révèlent un écart de performance notable entre les deux moteurs de requêtes sur tous les jeux de données et dans toutes les configurations testées (voir FIGURE 2 et 4). Bien que cet écart soit considérable, il était attendu, car **Integraal** est un moteur de requêtes développé par une équipe de chercheurs du *LIRMM*, spécialisés dans ce domaine, et optimisé pour ces tâches spécifiques. De notre côté, notre moteur reste encore assez basique, avec seulement quelques optimisations de base, mais offrant de nombreuses pistes d'amélioration.

Lors des tests à chaud, l'écart entre les deux moteurs est particulièrement marqué, et il s'intensifie à mesure que le volume de données augmente. Cependant, une amélioration significative est observée, avec des performances doublées, lorsque l'étape de prétraitement est activée. Nous avions précédemment constaté que nos jeux de données, bien que vastes, contenaient de nombreux défauts, tels que la duplication de requêtes et la présence de requêtes vides. Ces requêtes, qui représentent souvent une grande majorité de nos tests, entraînaient une perte de données et expliquent en grande partie les gains de performance obtenus grâce à ce prétraitement.

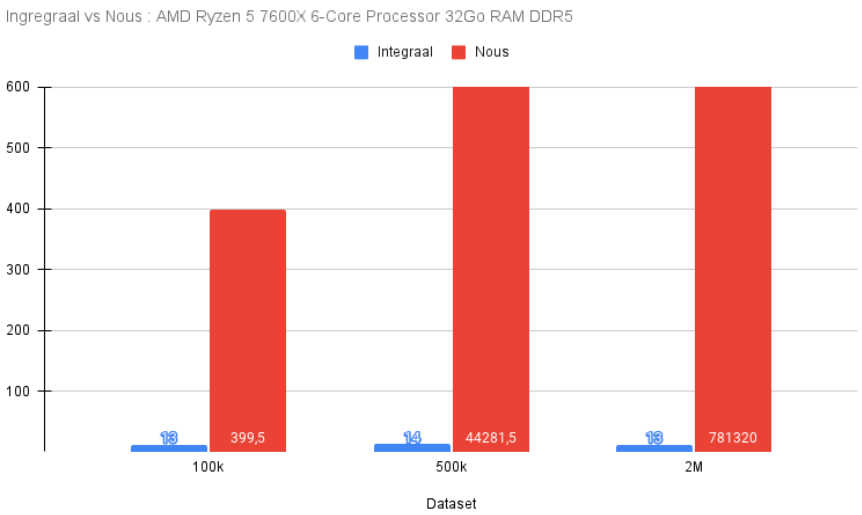

FIGURE 2 – Histogramme de comparaison de performance avec prétraitement (à chaud)

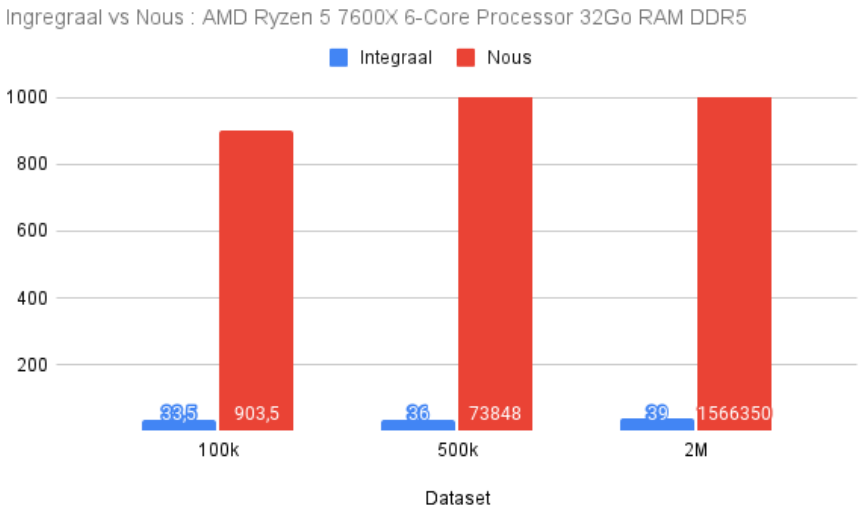

FIGURE 3 – Histogramme de comparaison de performance sans prétraitement (à chaud)

Nous nous sommes interrogés sur l'impact potentiel de Java sur nos tests, notamment en raison d'un éventuel stockage en cache ou d'autres optimisations. Pour évaluer cela, nous avons décidé de comparer les résultats obtenus avec et sans les optimisations de Java. À cet effet, nous avons utilisé la commande suivante :

```
java  -XX:-UseCompiler -jar target/qengine-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Commande pour exécuter le JAR sans les optimisations de Java

Cette commande désactive le compilateur **JIT** (Just-In-Time) de la JVM. Nous avons exécuté cette configuration sur plusieurs de nos datasets, et les résultats montrent une diminution significative de la vitesse d'exécution.

Malgré la désactivation des optimisations, **Integraal** reste relativement rapide, tandis que notre moteur de requêtes subit un ralentissement de plus en plus marqué, surtout sur les grands datasets. La différence de rapidité entre les exécutions avec et sans optimisations est frappante, en particulier sur les gros volumes de données.

Bien que les tests confirment un ralentissement significatif sans les optimisations, il est difficile d'en tirer des conclusions définitives. La commande utilisée désactive clairement la compilation **JIT**, mais nous ne savons pas avec certitude si d'autres mécanismes, comme le cache, sont également affectés.

Pour tirer des conclusions valides, il serait nécessaire de répéter tous les tests dans cette configuration sur l'ensemble des datasets. Cependant, cette approche n'est pas réaliste dans notre contexte, car elle nécessiterait un temps considérable, surtout sur les gros datasets où l'exécution devient extrêmement lente, voire impossible. Même avec un ordinateur aussi puissant que celui que nous avons utilisé, cela a pris un temps considérable pour pouvoir faire nos tests.

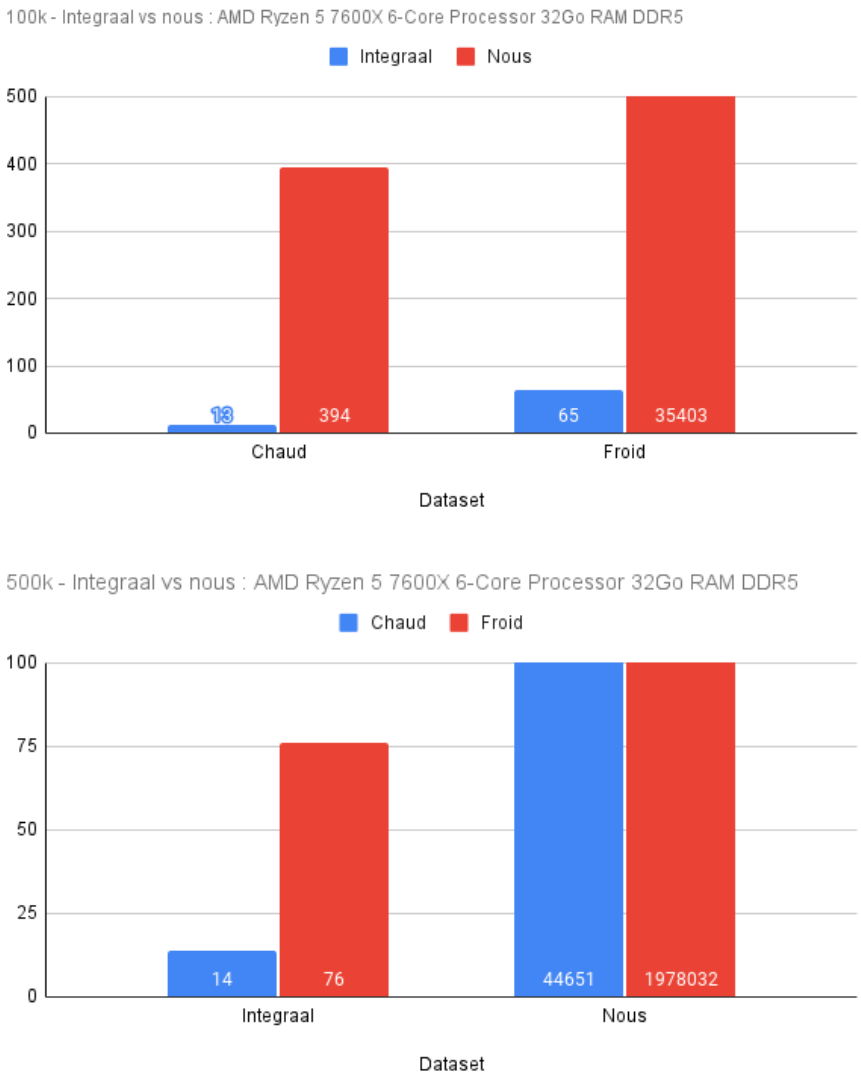

FIGURE 4 – Histogramme de comparaison de performance à froid et à chaud

4.3 Comparaison avec un autre groupe

Nous avons comparé les performances de notre Hexastore à celles de celui du groupe 3 (voir Figure 5). Sur le dataset de 100 000 entrées, notre implémentation s'est avérée plus rapide. Cependant, en raison de la durée élevée des tests pour les datasets de 500 000 et 2 millions d'entrées, nous n'avons pas pu les exécuter dans un délai raisonnable.

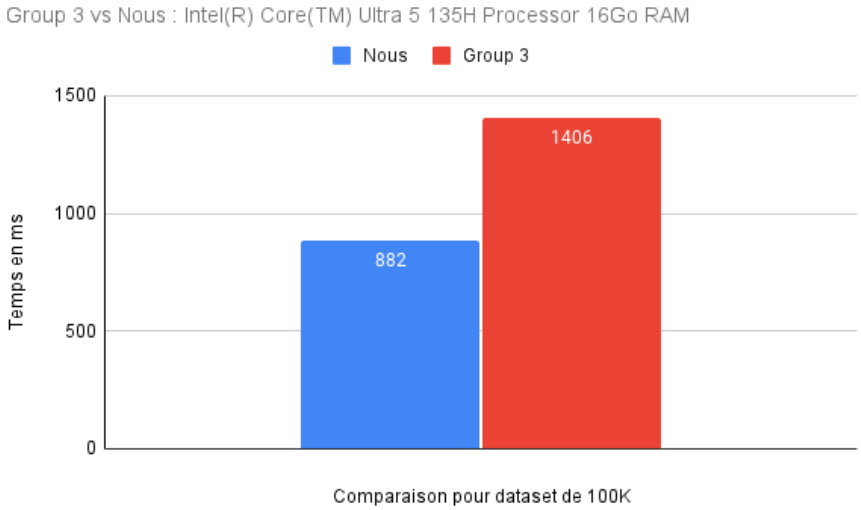

FIGURE 5 – Histogramme de comparaison de performance entre nous et le Group 3

5 Conclusion

En conclusion, ce projet nous a permis d'explorer en détail les différentes étapes nécessaires à la réalisation de benchmarks, tout en identifiant les erreurs courantes à éviter, telles que l'influence des caches ou l'importance de simuler un départ à froid pour garantir des résultats fiables. Malgré ces avancées, nos benchmarks présentent encore des axes d'amélioration. Notamment, une partie significative des données initiales s'avère inutile et est éliminée lors de la phase de prétraitement, ce qui réduit le volume des données réellement utilisées et peut biaiser l'évaluation des performances. Des ajustements futurs devront se concentrer sur une gestion plus efficace des données pour maximiser la pertinence des tests.

6 Annexes

Prétraitement - Statistiques générales

Nombre total d'atomes RDF analysés :
— Fichier 100K : 107 338 atomes
— Fichier 500K : 530 990 atomes
— Fichier 2M : 2 052 345 atomes
Nombre total de doublons : 6 802 / 13 000
Nombre de requêtes à 0 réponses : 4 909 / 13 000

Prétraitement - Analyse des requêtes détails

Fichiers de Requêtes	**Doublons (/1000)**	**Pourcentage de requêtes vides**		
		100K	**500K**	**2M**
Q_1_subscribes	950	0.0 %	0.0 %	0.0 %
Q_1_nationality	770	61.1 %	24.8 %	1.4 %
Q_1_eligibleregion	770	10.9 %	0.0 %	0.0 %
Q_1_includes	770	1.8 %	2.2 %	2.7 %
Q_1_likes	770	4.9 %	1.7 %	0.0 %
Q_2_likes_nationality	772	97.4 %	87.3 %	50.0 %
Q_2_subscribes_likes	48	83.2 %	94.7 %	84.2 %
Q_2_includes_eligibleRegion	14	93.8 %	96.7 %	87.5 %
Q_2_tag_homepage	49	98.5 %	97.8 %	100.0 %
Q_3_location_nationality_gender	196	96.4 %	94.2 %	28.6 %
Q_3_location_gender_type	768	77.9 %	62.1 %	4.2 %
Q_3_nationality_gender_type	773	76.8 %	78.9 %	13.3 %
Q_4_location_nationality_gender_type	202	94.7 %	97.6 %	40.0 %

TABLE 1 – Résultats détaillés des requêtes sur les différents jeux de données