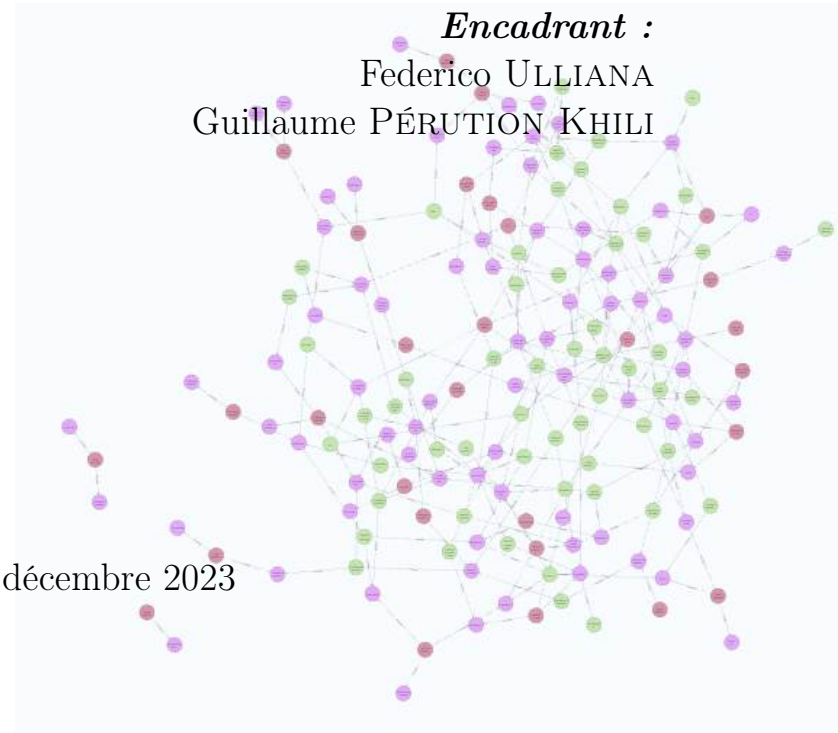
Université de Montpellier

HAI914I : Gestion des données au delà de SQL

Implementation d'un mini moteur de requêtes en étoile

Étudiants :
Awa Seck
Mitra Aelami

Encadrant :
Federico Ulliana
Guillaume Pérution Khili



28 décembre 2023

Table des matières

1 Évaluation des requêtes en étoile

1.1 Chargement de la base de triplets et création d'un dictionnaire pour les ressources

Pour charger les données on utlise la fonction *parseData* qui lit un fichier de données au format N-Triples, ligne par ligne, en utilisant un lecteur (dataReader) et après utilise un analyseur N-Triples (NTriplesParser) pour extraire et traiter chaque triple RDF du fichier. La classe *MainRDFHandler* est utilisée comme gestionnaire pour traiter chaque triple extrait. Les triples sont ensuite stockés dans une liste (statementList) pour pouvoir l'utiliser dans les créations de dictionnaire et de l'index.
Pour créer le dictionnaire on récupére les informations sur les entités RDF à partir du statementList et on les stock dans un Map en ajoutant des valeurs uniques pour chaque sujet, prédicat et objet.
Une illustration est donnée par la figure 1.

```
(Value: "67" , Key: 12215)
(Value: http://db.uwaterloo.ca/~galuc/wsdbm/follows , Key: 12216)
(Value: http://purl.org/ontology/mo/performed_in , Key: 12217)
(Value: http://purl.org/ontology/mo/artist , Key: 12218)
(Value: http://schema.org/actor , Key: 12219)
(Value: http://schema.org/director , Key: 12220)
(Value: http://schema.org/author , Key: 12221)
(Value: http://schema.org/editor , Key: 12222)
```

FIGURE 1 – Visualisation d'une partie de notre dictionnaire

1.2 Création des index

La classe Hexastore représente une structure de données optimisée pour le stockage et l'indexation d'instructions RDF. Pour élaborer cette classe on utilise le dictionnaire ainsi que le statementList créés. On parcours les statements et pour chaque élément de statement (subject, predicat, object) on prend la clé associée via le dictionnaire ce qui nous permettra de le stocker dans une liste contenant une liste de six index différents pour chaque statement(spo, sop, ops, osp, pso, pos) ce qui constitue notre hexastore. La méthode creationIndexHexastore initialise et construit ces index à partir d'une liste d'instructions RDF. Chaque index est organisé de manière à permettre une recherche rapide en fonction des sous-chaînes de sujet, prédicat et objet.
La figure 2 montre un exemple de notre hexastore.

```
Key1: 12224, Key2: 4114, Values: [638]
Key1: 12224, Key2: 4115, Values: [1581]
Key1: 12224, Key2: 4117, Values: [175]
Key1: 12224, Key2: 4118, Values: [292]
Key1: 12224, Key2: 23, Values: [3129, 4952]
Key1: 12224, Key2: 4120, Values: [1036]
Key1: 12224, Key2: 25, Values: [3253, 3458, 3593, 3835, 4964]
```

FIGURE 2 – Visualisation d'une partie de notre l'hexastore

Remarque : la figure 2 est la sortie avec l'index "ops", si pour un même object et predicate on trouve plusieurs subjects alors ils sont stockés dans une liste.

1.3 Lecture des requêtes en entrée

Toutes les requêtes sont stockées dans un tableau global après les avoir lues via la fonction *parseQueries* pour pouvoir les traiter séparément dans une autre fonction nommée *processAQuery* pour une évaluation.

1.4 Accès aux données et visualisation des résultats

Pour le traitement et l'évaluation des requêtes du tableau global, on a créé une fonction *processAQuery* qui prend en entrée une requête SPARQL analysée, extrait les patterns de déclarations associés, et effectue des opérations sur ces patterns. Elle imprime ensuite les prédicats et objets correspondants à chaque pattern, utilise le dictionnaire pour obtenir les clés des prédicats et des objets, et interroge le Hexastore pour récupérer des sujets associés aux paires prédicat-objet spécifiques (la méthode *getSubject*). Pour avoir le résultat final on fait une intersection en utilisant la méthode *retainAll* pour filtrer la liste des résultats en conservant uniquement ceux qui satisfont tous les motifs de déclaration de la requête SPARQL. Ces résultats (Figure 3) sont ensuite affichés dans le terminal et enregistrés dans un fichier nommé *export_ query_ results.csv* dans le repertoire *output*. Au cas où on n'a pas de réponse on affiche qu'on en a pas trouvé et on le stock aussi dans le fichier.

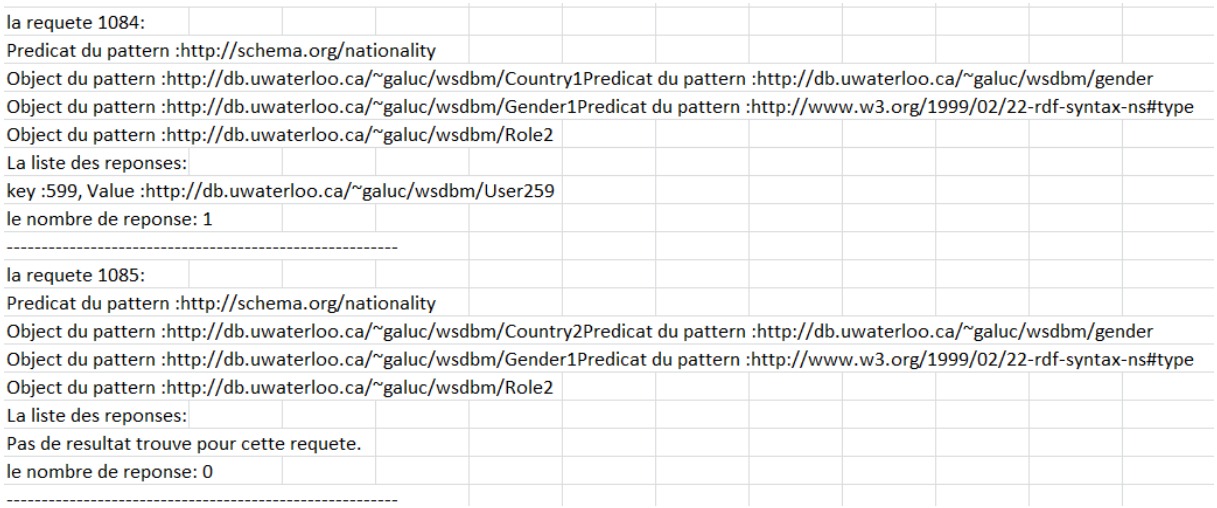


FIGURE 3 – Exemple d'évaluation des requêtes 1084 et 1085 que nous avons sauvegardées dans un fichier CSV nommé *export_ query_ results.csv*

Notre programme est également exécutable en ligne de commande. Pour ce faire, on a crée deux fonctions *createOptions* et *handleOptions*. La fonction createOptions configure les options disponibles, telles que le chemin vers le dossier des requêtes, le fichier de données, le dossier de sortie, l'activation de la vérification Jena, l'utilisation d'un échantillon de requêtes pour chauffer le système, et la permutation aléatoire des requêtes.
La fonction handleOptions récupère les valeurs des options spécifiées en ligne de commande et les utilise pour guider l'exécution du programme. Elle mesure et affiche divers éléments tels que le nombre de triplets RDF et le nombre de requête ainsi que des temps d'exécution, qui sont le temps de lecture des données, le temps de création du dictionnaire, le temps de création des index, le temps de lecture des requêtes, le temps total

d'évaluation du workload, et le temps total du programme. Ces temps sont calculés en mesurant le temps d'avant-après pour chaque fonction (*parseData*, *createDictionary*, *creationIndexHexastore*, *parseQueries*,*processAQuery*) dans la méthode *main* en utilisant *System.currentTimeMillis*. Cela va nous permettre de calculer le temps total (du début à la fin) en faisant la différence entre le temps actuel et celui du début c'est à dire le temps de début de la lecture des données (*System.currentTimeMillis() - startTimeParseData*). Ces informations (Figure 4) sont également exportées dans un fichier CSV nommé *output.csv* dans le repertoire *output*

```
Nombre de triplets RDF : 107338
Nombre de requetes : 1200
Temps de lecture des donnees : 479 ms
Temps de lecture des requetes : 295 ms
Temps de creation du dictionnaire : 80 ms
Temps de creation des index : 138 ms
Temps total d'evaluation du workload : 1636 ms
Temps total (du debut a la fin du programme) : 2628 ms
```

FIGURE 4 – Visualisation des résultats en ligne de commande

Remarque :

— Nombre de triplets RDF : nombre de lignes du fichier *100K.nt*

— Nombre de requête : calculé en comptant chaque requête du fichier *STAR_ALL_workload.queryset*

— Compilation en ligne de commande :
java -jar rdfjar.jar -queries "data/STAR_ALL_workload.queryset" -data "data/100K.nt" -output "output/output.csv" -Jena -warm 50 -shuffle
où *rdfjar.jar* représente le fichier jar exporté du programme.

— Le type de CPU qu'on a utilisé est *intel corei7*.

— Le temps d'éxécution des requêtes est plus rapide en ligne de commande qu'en console java sous Eclipse (Figure 5) avec une meilleure performance.

```
Nombre de triplets RDF : 107338
Nombre de requetes : 1200
Temps de lecture des donnees : 667 ms
Temps de lecture des requetes : 363 ms
Temps de creation du dictionnaire : 89 ms
Temps de creation des index : 118 ms
Temps total d'evaluation du workload : 1433 ms
Temps total (du debut a la fin du programme) : 2670 ms
```

FIGURE 5 – Visualisation des résultats en console java

2 Évaluation et Analyse des Performances

2.1 Préparation des bancs d'essais

2.1.1 Génération de données et de requêtes

Pour générer de nouvelles données, nous avons fait cet appel :
bin/Release/watdiv -d model/wsdbm-data-model.txt 5
pour avoir *500K* avec un facteur d'echelle *5* ((5*1) car pour un facteur d'échelle *1* nous avons obtenu *100K* données).
De la même manière nous avons obtenus *2M* de données avec un facteur d'échelle *19*.

Pour la génération de requêtes, nous avons modifié le fichier *regenerate_queryset.sh* en générant d'abord 200 et puis 1000 requêtes par patrons et en renommant nos reqêtes query_Q_i comme ceci :
bin/Release/watdiv -q model/wsdbm-data-model.txt $qt 200 1 > testsuite/queries/query$qt2%.sparql-template.queryset ;
(et *bin/Release/watdiv -q model/wsdbm-data-model.txt $qt 1000 1 > testsuite/queries/query$qt2%.sparql-template.queryset ;*).
Ce qui nous a donné *2600* (respectivement *13K*) de requêtes que nous avons toutes concaténés dans un seul fichier, à l'aide de la commande *cat* (cat *.queryset > 2600.queryset et 13K.queryset), pour l'adapter à notre code.

De ce fait nous disposons de trois fichiers de requêtes pour faire des jeux tests :
STAR_ALL_workload.queryset (100K), *2600.queryset* et *13K.queryset*.

NB : Pour que cette génération de requêtes fonctionne on a eu besoin de modifier chaque fichier de template en ajoutant un retour à la ligne afin que les requêtes aient un accolade fermant.

2.1.2 Jeux de tests et histogrammes des réponses aux requêtes en étoile

Avec les données et requêtes générées, nous avons réalisés des jeux de tests avec lesquels nous affichons un histogramme créé, avec la bibliothèque **JFREE** sous java, à partir des réponses des requêtes.
Ces jeux tests sont :

1. 2600 requêtes sur 500K de données (Figure 6).
2. 13K requêtes sur 500K de données (Figure 7).
3. 1200 requêtes sur 2M de données (Figure 8).
4. 13K requêtes sur 2M de données (Figure 9).

D'après les différents histogrammes nous constatons un nombre élevé de requêtes avec zéro réponse faisant un grand pic, ce qui n'est pas souhaitable pour le benchmark vu que nous souhaitons minimiser le nombre de réponses zéro, ce qui indiquerait une meilleure couverture des données et une capacité du système à répondre à un large éventail de requêtes.

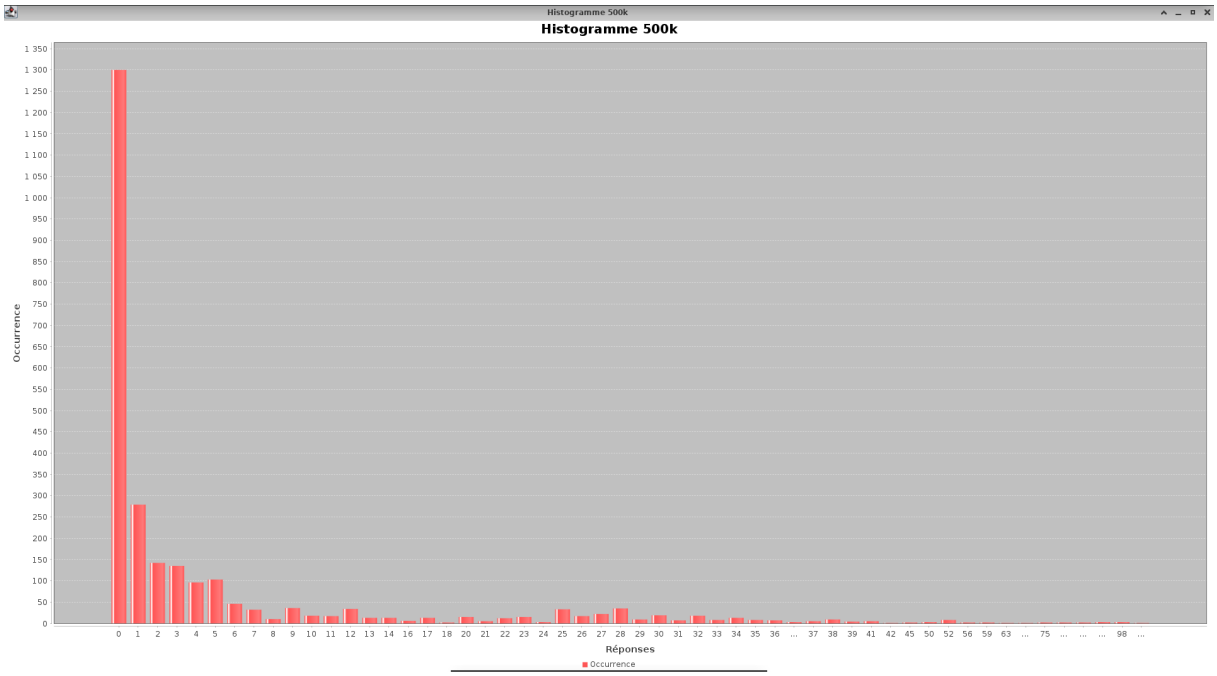


FIGURE 6 – Histogramme des réponses de 2600 requêtes sur 500K de données

Après décompte du nombre de conditions des 2600 requêtes, nous avons obtenu ces résultats :

— 1000 ont un même nombre de condition (1 patron).
— 800 ont un même nombre de condition (2 patrons)
— 600 ont un même nombre de condition (3 patrons)
— 200 ont un même nombre de condition (4 patrons)

Chaque fichier de requête contient des doublons par exemple pour le fichier 13k.queryset nous avons 6129 doublons comme nous pouvons le voir dans la Figure 10. Ce calcul du nombre de doublons est fait dans la fonction *parseQueries* où nous vérifions pour chaque requête lue si elle a été déjà présente ou pas dans l'ensemble des requêtes, si oui, nous incrémentons le compteur du nombre de doublons.
En effet c'est souhaitable pour le benchmark car cela va diminuer le temps d'éxécution, vu que certaines requêtes sont répétées, et le système peut bénéficier de caches.

2.2 Hardware et Software

Pour faire nos jeux de tests nous avons utilisé d'une part un type de matériel comprenant un processeur *Intel Core i7* cadencé à *1,30 GHz*, avec une mémoire vive (RAM) de *8,0 Go*, d'une part un processeur M1 modèle 2020.
Comme Software nous avons traivaillé avec **Eclipse IDE** et **VSCodium**.

2.3 Métriques, Facteurs, et Niveaux

2.3.1 Métriques

Voici la liste des métriques pour évaluer les performances de moteur de requêtes RDF :

— Temps d'exécution : mesure du temps nécessaire pour traiter une requête RDF.

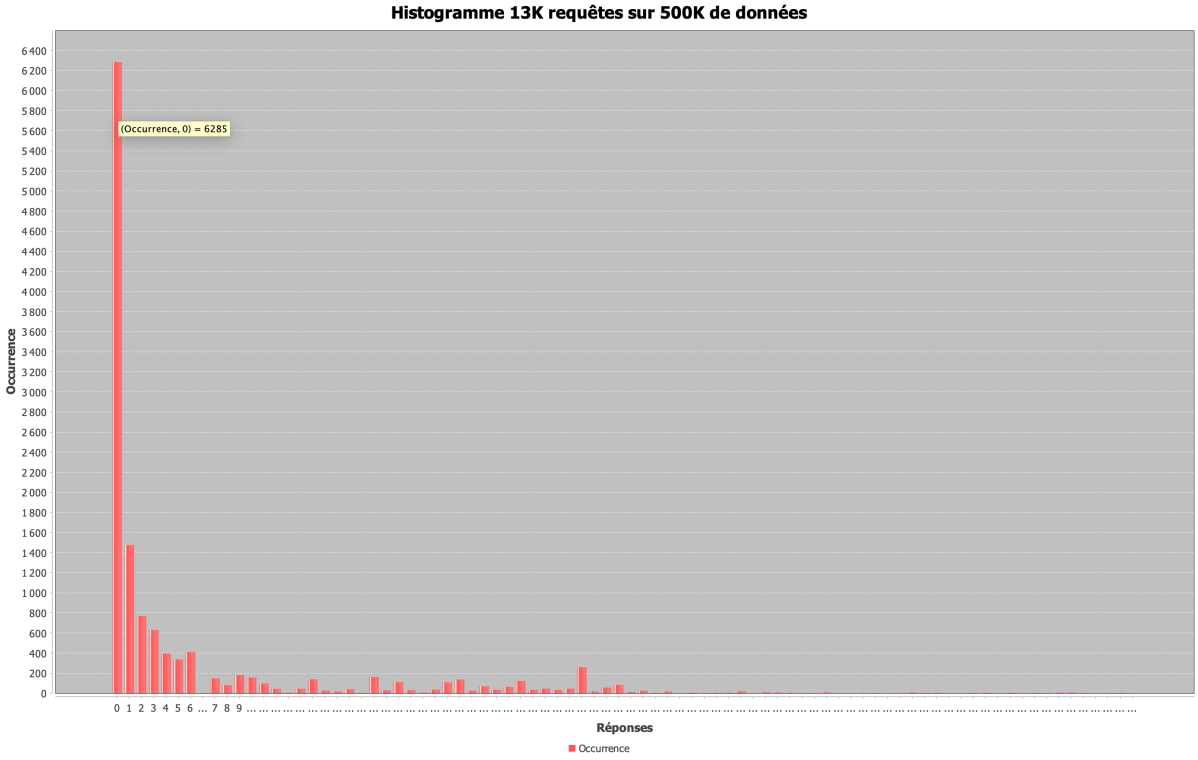


FIGURE 7 – Histogramme des réponses de 13K requêtes sur 500K de données

— Utilisation des ressources : évaluation de la consommation de mémoire et du traitement CPU pendant l'exécution.
— Extensibilité : capacité du moteur à gérer des volumes croissants de données ou de requêtes.
— Précision des résultats : vérification de l'exactitude des résultats renvoyés par rapport à la spécification de la requête

2.3.2 Facteurs et niveaux

— Taille des données :
Niveaux : Petit (100K triples), Moyen (500K triples), Grand (2M triples).
— Nombre de requêtes :
Niveaux : Faible (100 requêtes), Modéré (1000 requêtes), Élevé (10 000 requêtes).
— Complexité des requêtes :
Niveaux : Simple (requêtes basiques), Moyenne (requêtes avec filtres ou options), Complexes (requêtes imbriquées).
— Charge concurrente :
Niveaux : Basse (1 utilisateur), Modérée (10 utilisateurs), Élevée (100 utilisateurs).

2.3.3 Ordonnancement des facteurs selon leur importance

1. Taille des données
2. Nombre de requêtes
3. Charge concurrente
4. Complexité des requêtes

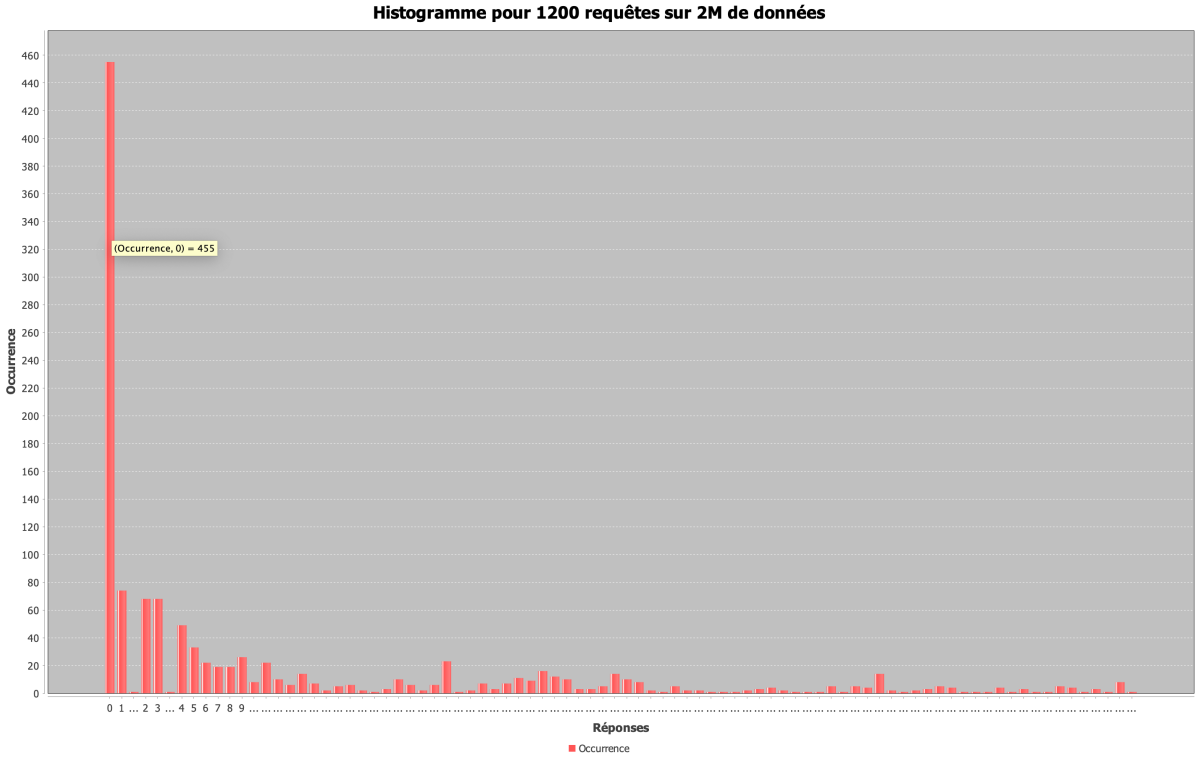


FIGURE 8 – Histogramme des réponses de 1200 requêtes sur 2M de données

5. Extensibilité
6. Temps d'exécution des requêtes
7. Utilisation des ressources
8. Précision des résultats

2.4 Évaluation des performances

— Métriques les mesures "cold" ou "warm" :
Pour les métriques temps d'exécution des requêtes il préférable d'effectuer des mesures "cold" et "warm" .
Pour la précision des résultats (efficacité de la réponse) il préférable d'effectuer des mesures "warm" pour examiner comment l'efficacité de la réponse évolue au fur et à mesure que le système traite un plus grand nombre de requêtes.

— Protocole de réalisation des mesures en pratique : Cold :
Mesures du temps de lecture des données (dataReadTime) : Mesuré dès le début du programme lors de la lecture des données RDF. Mesures du temps de création du dictionnaire (dictionaryCreationTime) : Mesuré après la lecture des données RDF et la création du dictionnaire.
Warm :
Mesures du temps de création des index (hexastoreCreationTime) : Mesuré après la création du dictionnaire et l'initialisation de la classe Hexastore. Mesures du temps de lecture des requêtes (queryReadTime) : Mesuré lors du traitement des requêtes SPARQL. Mesures du temps total d'évaluation du workload (totalEvaluationTime) : Mesuré après le traitement de toutes les requêtes. Mesures du temps

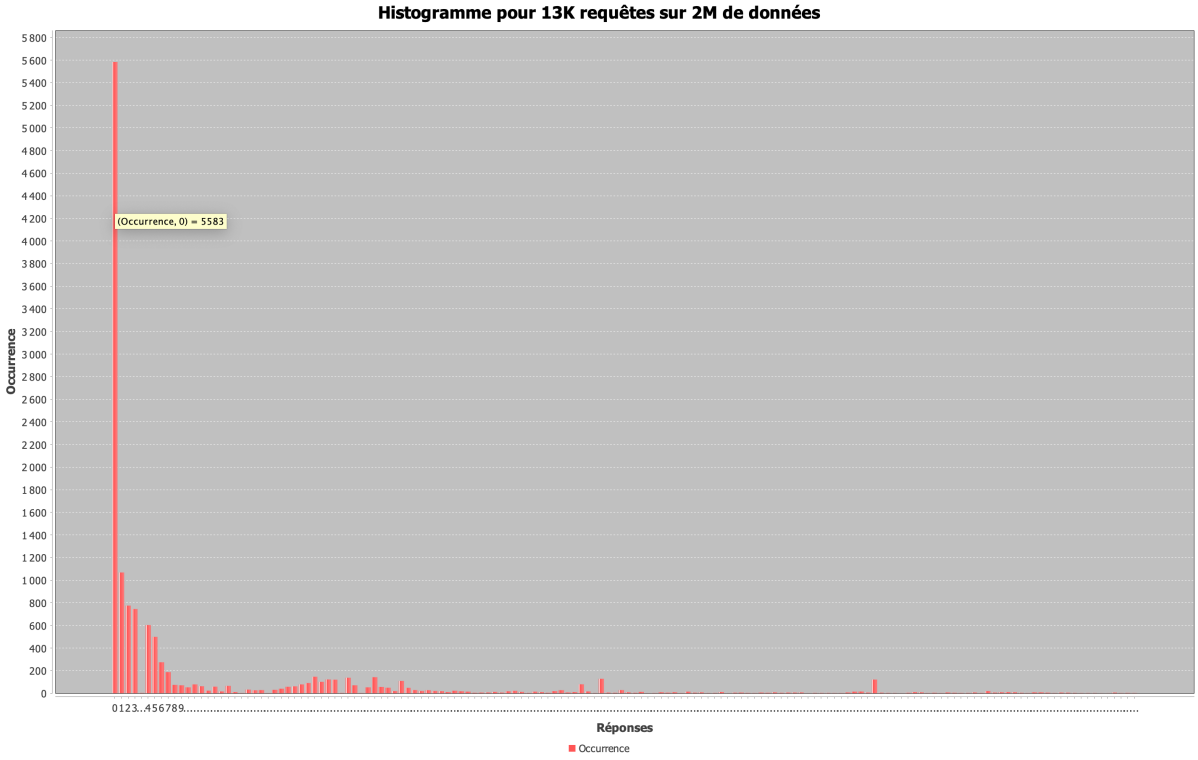


FIGURE 9 – Histogramme des réponses de 13K requêtes sur 2M de données

total (totalTime) : Mesuré du début à la fin du programme.

Le temps de chaque étape est enregistré à l'aide de marqueurs temporels (timestamps) dans le code de la classe Main. Les résultats sont ensuite affichés et exportés dans un fichier CSV.

— Procédure pour la vérification de la correction et de la complétude de votre système : En éxécutant les requêtes en étoile à la fois sur notre système et sur Jena, nous obtenons les mêmes réponses avec un temps plus grand sur notre système.

— En augmentant la taille des données dans différentes expériences, nous constatons une incrémentation des métriques comme le temps d'éxécution et les performances de notre système, visible dans les fichier csv : *output.csv, output500k.csv, output1M.csv, output2M.csv.*

2.5 Representation graphique des résultats

— 6500 requêtes sur 500K de données (Figure 11).
— 1200 requêtes sur 1M de données (Figure 12).
— 2600 requêtes sur 1M de données (Figure 13).
— 1200 requêtes sur 2M données (Figure 14).

L'analyse des performances de traitement de requêtes NoSQL sur des données RDF de tailles variées indique une relation significative entre le volume de données et les résultats obtenus. En observant l'augmentation du nombre de données RDF, il est apparent que le nombre de réponses zero diminue, ce qui suggère une amélioration de l'efficacité du traite-

1	Nom du fichier de donnees: data/2M.nt
2	Nom du dossier des requetes: data/13k.queryset
3	Nombre de triplets RDF: 2036619
4	Nombre de requetes: 13000
5	Temps de lecture des donnees: 5309
6	Temps de lecture des requetes: 1193
7	Temps de creation du dictionnaire: 868
8	Temps de creation des index: 1144
9	Temps total d'evaluation du workload: 30418
10	Temps total (du debut a la fin du programme): 38932
11	Nombre total de doublons dans les requetes : 6129

FIGURE 10 – Visualisation du nombre de doublons avec 13K de requêtes sur 2M de données sauvegardé dans le fichier CSV nommé *output2M.csv*

ment des requêtes à mesure que la base de données s'enrichit. Cependant, il est essentiel de noter que cette amélioration s'accompagne d'une augmentation du temps d'exécution. Cette corrélation entre la taille des données et le temps d'exécution suggère que, bien que le système puisse mieux répondre aux requêtes avec un ensemble de données plus vaste. Cette observation souligne l'importance d'optimiser les performances du programme pour gérer efficacement des volumes croissants de données RDF tout en maintenant des temps d'exécution acceptables.

De plus, en augmentant le nombre de requêtes avec un ensemble de données constant, on observe une dispersion accrue des réponses sur l'axe horizontal (abscisses) de l'histogramme. La proportion de réponses zero tend à augmenter, représentant environ la moitié du nombre total de requêtes. Cette observation suggère que l'augmentation du nombre de requêtes entraîne une diversification des réponses et une tendance à augmenter le nombre de résultats zero, ce qui se reflète également par une augmentation de l'axe vertical de l'histogramme.

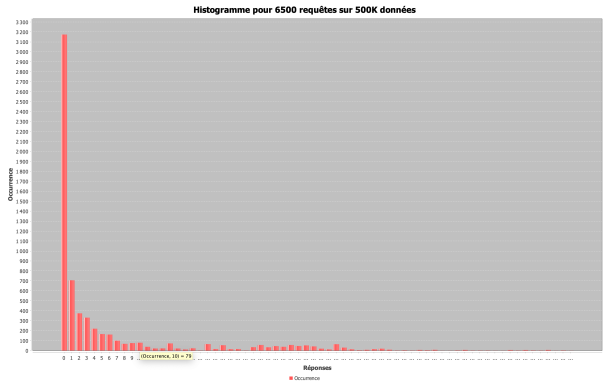


FIGURE 11 – Histogramme des réponses de 6500 requêtes sur 500K de données

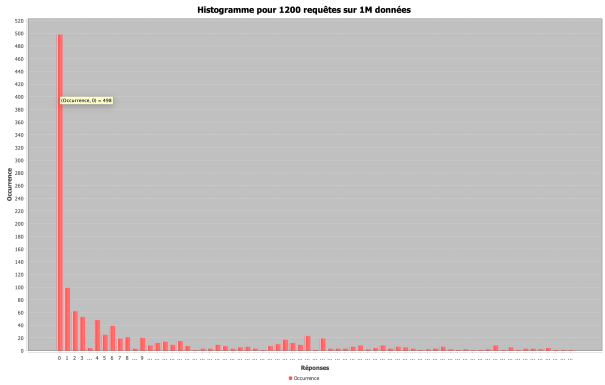


FIGURE 12 – Histogramme des réponses de 1200 requêtes sur 1M de données

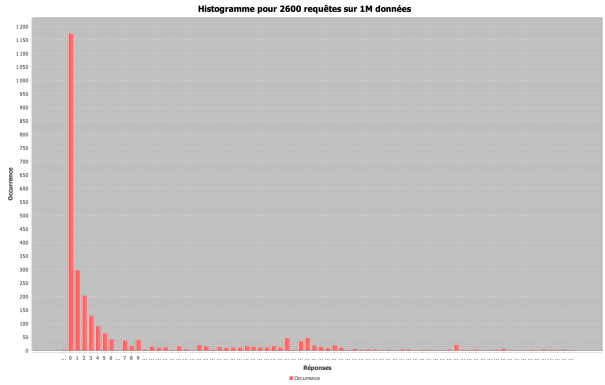


FIGURE 13 – Histogramme des réponses de 2600 requêtes sur 1M de données

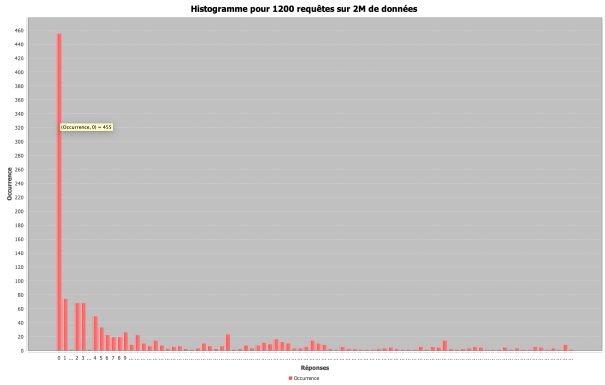


FIGURE 14 – Histogramme des réponses de 1200 requêtes sur 2M de données