HMIN 340 - NoSQL

Projet Mini Moteur

Rendu final

Réalisé par:
DIDIER Adrien
MARTIN Loïc

Sommaire

1. Préambule

Ce rapport va vous présenter l'ensemble du travail réalisé pour mener à bien ce projet en termes de développement et de réalisation des tests.

Il a tout d'abord été demandé de se concentrer sur la lecture et le traitement des requêtes. Le moteur développé doit être capable de prendre en charge n'importe quels types de patterns de requêtes dont ceux en étoiles.

Pour la partie concernant les requêtes en étoile, il était demandé de réaliser une méthode spécifique pour l'évaluation de ces dernières.
De plus, nous vous présenterons la méthode permettant de prendre en compte n'importe quel ensemble de patterns de triplets, ainsi que la création d'un jar et l'utilisation de différents arguments.

Après avoir développé notre programme, il est demandé de calculer les performances de notre système. Cette partie va donc se consacrer à l'analyse des benchmarks ainsi qu'à la mise en place des tests à réaliser qui vont nous permettre de comparer notre prototype à celui de nos camarades ainsi qu'à Jena. WatDiv sera utilisé pour mettre en place les différents jeux de données et de requêtes pour mener à bien nos expérimentations qui sont présentées dans la dernière partie de ce rapport.

2. Dictionnaire - Index

Pour la partie concernant le dictionnaire et les index, nous avons essayé de minimiser le plus possible les temps d'exécution pour leur création.
Nous avons testé deux méthodes pour effectuer un ensemble de tests concernant le temps d'exécution et l'espace pris en mémoire.
Pour ce travail, nous avons utilisé les trois fichiers RDF proposés dans le dossier "Matériel". Nous avons calculé le temps nécessaire pour créer le dictionnaire et les six index suivant l'approche hexastore (spo, pso, osp, sop, pos, ops). Ensuite, nous avons stocké ces résultats sous forme de fichiers texte afin d'obtenir la taille des fichiers pour vérifier si ces méthodes étaient différentes en termes de coûts de stockage.

Nous allons donc vous présenter les deux méthodes sélectionnées et les résultats obtenus. Pour terminer, nous vous présenterons la solution retenue pour mener à bien ce projet.

2.1. HashMap

2.1.1. Dictionnaire

Nous nous sommes en premier lieu orientés vers l'utilisation d'un HashMap pour la création du dictionnaire. Ce choix nous semblait logique étant donné que c'est une structure de données permettant une association clé-valeur. Cette approche nous permet en effet de stocker nos valeurs en leur attribuant un index.
Pour procéder à la création du dictionnaire, nous avons récupéré les ressources présentes dans le fichier RDF donné en argument. Ensuite, pour chaque triplet "subject, predicate et object", nous ajoutons leurs valeurs avec une clé auto-incrémentée dans la table de hachage. Pour éviter les doublons nous n'avons aucune action à effectuer, car la méthode add de HashSet vérifie d'elle-même que la valeur donnée n'est pas déjà présente.

Nous obtenons les résultats suivants :

Datasets	100K	500K	1M
Nombre de couples (clé-valeur)	12 225	51 043	
Taille en mémoire	671 Ko	3.06 Mo	
Temps d'exécution	9 secondes 265 millisecondes	7 minutes 14 secondes 197 millisecondes	

Nous pouvons constater que le deuxième fichier demande 7 minutes de traitement pour obtenir une liste de 51 043 clé-valeur.
Il n'a même pas été nécessaire de tester le troisième fichier étant donné le temps requis pour le second.

2.1.2. Index

L'utilisation du HashMap pour la création du dictionnaire s'est voulue peu concluante. Nous avons tout de même réalisé les tests pour la création des différents index.

Pour l'index, nous avons remplacé l'ensemble des valeurs par leur clé à l'aide des méthodes proposées par la classe HashMap.

Datasets	100K	500K	1M
Nombre de triplets	107 338	529 580	
Taille en mémoire	6 * 2.08 Mo 12.42 Mo	6 * 11.1 Mo 66.6 Mo	
Temps d'exécution	4 secondes 227 millisecondes	3 minutes 37 secondes 781 millisecondes	

Nous pouvons observer que le programme demande moins de temps de traitement que la création du dictionnaire. Mais n'est tout de même pas assez performant pour l'utilisation souhaitée. Il nous semble peu judicieux d'attendre 10 minutes pour obtenir un dictionnaire et les index pour seulement le deuxième fichier de données. Le troisième demanderait trop de temps de traitement.

2.2. HashSet

Nous nous sommes donc orientés vers une nouvelle méthode qu'est la classe HashSet.
Cette classe implémente l'interface Set, en utilisant une table hachée. Un HashSet est implanté comme une HashMap, dont les clés sont les éléments du HashSet.

2.2.1. Dictionnaire

Datasets	100K	500K	1M
Nombre de couples (clé-valeurs)	12 225	51 043	98 974
Taille en mémoire	611 Ko	2.78 Mo	5.20 Mo
Temps d'exécution	697 millisecondes	1 seconde 660 millisecondes	3 secondes 317 millisecondes

Cette fois-ci, le HashSet s'est montré extrêmement performant pour la création du dictionnaire. En effet, HashSet est une classe fournissant des opérations de définition de performances élevées. De plus, la valeur étant elle-même la clé, il est extrêmement rapide de vérifier l'existence de cette dernière dans le dictionnaire pour éviter les doublons.

2.2.2. Index

Datasets	100K	500K	1M
Nombre de triplets	107 338	529 580	
Taille en mémoire	6 * 2.06 Mo 12.36 Mo	6 * 11.1 Mo 66.6 Mo	
Temps d'exécution	5 secondes 407 millisecondes	3 minutes 13 secondes 483 millisecondes	

Toutefois, nous avons rencontré de nouveau un problème lors de la création des index en termes de temps d'exécution. En effet, étant donné que le HashSet a pour clé sa valeur, il n'existe pas de méthode pour obtenir la position et ainsi l'index de la valeur.
Pour remédier à ce problème, il nous a fallu cloner le dictionnaire dans un array et effectuer un indexOf sur les valeurs. Cette opération est gourmande en temps et explique les résultats obtenus.

2.3. Solution retenue

Suite à nos précédentes expérimentations, nous avons utilisé ces deux classes à notre avantage pour obtenir le programme le plus performant.

2.3.1. Dictionnaire (LinkedHashSet → HashMap)

Surpris de l'excellent résultat du HashSet pour la création de dictionnaire, nous avons décidé de nous orienter vers cette méthode.
Toutefois, nous avons opté pour l'utilisation d'un LinkedHashSet pour conserver le contenu dans l'ordre de lecture, et non par l'utilisation d'un hash code comme le fait HashSet.
Par la suite, nous le transformons en HashMap pour la partie concernant les index.

```
for (String values : dictionaryLinkedHashSet) {
      dictionaryHashMap.put(counter.incrementAndGet(),values.toString());
}
```

Le temps de calcul et l'espace en mémoire ne sont pas impactés par cette transformation.

Datasets	100K	500K	1M
Nombre de couples (clé-valeurs)	12 225	51 043	98 974
Taille en mémoire	671 Ko	3.06 Mo	6 Mo
Temps d'exécution	755 millisecondes	1 seconde 738 millisecondes	3 secondes 803 millisecondes

2.3.2. Index (HashMap inversé)

L'un des problèmes rencontrés avec le HashMap (Key, Value) est que la méthode get nous donne la valeur du HashMap. Mais pour créer nos index, nous souhaitons obtenir la clé depuis la valeur.

Pour cela, nous créons un nouveau dictionnaire ayant pour clé nos valeurs et comme valeurs nos anciennes clés.

Exemple :
{0=Tommie scalpel Mandarin's, 1=40711906, 2=naturalized}
devient
{Tommie scalpel Mandarin's=0, 40711906=1, naturalized=2, 98627796=3}

```
static HashMap<Integer,String> dictionaryHashMap = new HashMap<Integer,String>();
static HashMap<String, Integer> HashMapSPO = new HashMap<String, Integer>();

for(HashMap.Entry<Integer, String> entry : dictionaryHashMap.entrySet()){
      HashMapSPO.put(entry.getValue(), entry.getKey());
}
```

Ceci nous permet finalement d'utiliser la méthode get et obtenir les valeurs pour créer nos index.

Datasets	100K	500K	1M
Nombre de triplets	107 338	529 580	1 084 308
Taille en mémoire	6 * 2.06 Mo 12.36 Mo	6 * 11.1 Mo 66.6 Mo	6 * 23.1 Mo 138.6 Mo
Temps d'exécution	576 millisecondes	3 secondes 655 millisecondes	9 secondes 253 millisecondes

Nous pouvons nous rendre compte que le temps d'exécution est extrêmement satisfaisant au vu des résultats obtenus précédemment.
De plus, nous pouvons observer que la taille des fichiers n'évolue que très peu en fonction des méthodes utilisées.

Suite à un protocole rigoureux tout au long de nos expérimentations, nous pouvons proposer un dictionnaire et des index fonctionnels très peu coûteux en temps d'exécution et en espace de stockage au vu de la quantité de ressources proposées par les différents fichiers RDF.

3. Requêtes en étoiles

Tout d'abord, nous créons un string query pour y inscrire la requête que nous souhaitons effectuer. Ce système a été modifié pour pouvoir lire un dossier contenant de multiples requêtes.

```
String query = "SELECT ?v0 WHERE {?v0 <http://purl.org/dc/terms/Location>
<http://db.uwaterloo.ca/~galuc/wsdbm/City13> }";
```

Par la suite, nous récupérons les variables présentes dans le select afin de les garder en mémoire pour une potentielle utilisation lors du traitement qui sera expliquée par la suite.

```
List<String> variable = new ArrayList<String>();

for (ProjectionElem string : projection.getProjectionElemList().getElements()) {
                    variable.add(string.getSourceName().toString());
}
```

Pour chaque branche, nous vérifions où se situent les variables et les constantes. Étant donné que les variables n'apparaissent que dans les sujets et les objets nous avons donc 3 possibilités.
Soit seulement le sujet est une variable, soit l'objet et le troisième cas est le sujet et l'objet sont des variables.

Lorsque nous pouvons situer correctement les variables, nous procédons à des traitements différents en fonction de la configuration.

Variable : sujet ⇒ !(sp.getSubjectVar().hasValue())
Étant donné que le prédicat et l'objet sont des constantes, nous pouvons utiliser l'index OPS pour récupérer l'ensemble des valeurs "sujet" qui répondent à la requête de la branche.

Variable : objet
Ici, nous sommes dans le cas où l'objet est une variable. De ce fait, nous pouvons utiliser l'index SPO pour obtenir le résultat de la branche.

Variable : sujet et objet
Pour le troisième cas, seul le prédicat est une constante. Nous devons donc récupérer les valeurs situées dans un HashMap et non dans une ArrayList comme les traitements cités précédemment. Pour cela, nous récupérons les valeurs du HashMap où la clé est IndexPSO.get(predicateInd). Ensuite, en fonction de la variable demandée dans le select nous récupérons soit les clés qui correspondent au sujet, soit les valeurs qui correspondent à l'objet. Nous pouvons aussi récupérer les deux.

Il est notable de préciser que l'utilisation du HashMap nous permet en temps constant d'accéder aux valeurs.
Le résultat obtenu (un ou plusieurs éléments) est ainsi stocké dans un buffer.

De ce fait, nous effectuons ce traitement pour chaque branche. Toutefois, après chaque nouvelle branche rencontrées dans la requête, nous effectuons une jointure sur les résultats obtenus pour récupérer les valeurs répondant aux deux branches.

Un système de parser a été mis en place afin d'effectuer un ensemble de requêtes contenues dans un dossier.
Nous écrivons les requêtes et leurs différents résultats dans un fichier texte, ce processus pourra être activé ou non par l'utilisateur lors de l'utilisation du fichier JAR.

```
File directory = new File(args[0]);
File[] contents = directory.listFiles();
List<String> querys = new ArrayList<String>();
for (File f: contents) {
      querys.addAll(App.parseFile(f.getAbsolutePath()));
}
```

4. Moteur de requêtes

4.1. Méthode M2A

Pour cette partie, nous avons mis en place la méthode M2A qui peut prendre en compte n'importe quel ensemble de patrons de triplets sans s'occuper d'optimisations.
La difficulté a été de prendre en compte les différents cas de triplets se présentant. En effet, il est possible de rencontrer des patterns avec une seule ou deux variables, ces mêmes variables peuvent être présentes ou non dans le HashMap final qui contiendra les résultats de la requête pour chaque valeur. Dans le programme, ce HashMap est midResult; de plus, nous avons midResultBuffer qui est le HashMap permettant de récupérer les valeurs pour chaque triplet rencontré.

Une fois un triplet traité, il est important de connaître la présence de ces variables dans midResult.
Si le pattern ne possède qu'une variable, nous ajoutons cette dernière dans midResult si elle ne s'y trouve pas, sinon nous effectuons une jointure entre les valeurs contenues dans midResult et celles contenues dans midResultBuffer.

```
for(int i=0;i<midResult.get(variable).size(); i++) {
	if(!(midResultBuffer.contains(midResult.get(variable).get(i)))) {
		for(ArrayList<Integer> valueParVar : midResult.values()) {
			valueParVar.remove(i);
		}
		if(i>=0) {
			i--;
		}
	}
}
return midResult;
```

Dans le cas où le triplet contient deux variables, nous devons utiliser une nouvelle méthode pour effectuer la jointure entre les deux HashMap. En effet, il faut garder les valeurs en communs ainsi que répercuter les lignes supprimées sur les autres variables contenues dans midResult et en cas d'ajout d'informations, il est nécessaire de modifier midResult pour ajouter des lignes.

La signature de la méthode se présente de cette manière :

Type de retour : `HashMap<String,ArrayList<Integer>>`

Paramètres :

`HashMap<String,ArrayList<Integer>> midResult` : Le HashMap midResult qui contient le résultat des triplets précédemment traités.
`String variable` : La variable qui va permettre de relier midResult et midResultBuffer.
`ArrayList<Integer> midResultBuffer` : Le HashMap qui contient les valeurs obtenues lors du triplet actuel.
`String variableToAdd` : La variable qui va devoir être ajoutée dans la table (celle qui ne permet pas de joindre).
`ArrayList<Integer> tableToAdd` : Les valeurs de la variable à ajouter dans la table.

Nous avons donc trois cas qui se présentent à nous :

- Le premier est qu'aucune variables ne sont présentes dans midResult, on ajoute tout simplement leurs valeurs dans le HashMap.

- Le deuxième est qu'une seule des deux variables se situe dans midResult. Nous devons donc lier les deux HashMap avec "variable" et ajouter "variableToAdd" dans midResult ainsi que ses valeurs "tableToAdd".

- Le troisième est que les deux variables sont présentes dans midResult, de ce fait nous sélectionnons soit le sujet, soit l'objet pour effectuer le même traitement que lors du deuxième cas.

4.2. Création JAR et utilisation arguments

Pour mener à bien ce projet, il est nécessaire de pouvoir exécuter le programme depuis un fichier JAR et ainsi éviter l'utilisation d'un environnement de développement comme Eclipse.
Le fichier JAR contient donc le programme ainsi que toutes les librairies permettant le fonctionnement de ce dernier.

Nous avons des arguments permettant d'indiquer le chemin du dossier de requêtes, le chemin vers le fichier RDF contenant le jeu de données et le chemin vers le fichier de sortie où sont stockés les résultats des requêtes.

Nous avons aussi mis en place l'argument -star_queries qui permet de forcer l'utilisation de la méthode M1. Lors de la présence de cet argument, nous vérifions pour chaque requête présente dans le fichier que l'utilisation de la méthode en étoile est possible.
Dans le cas contraire, nous utilisons la méthode M2A.

L'argument -verbose permet d'afficher différentes informations telles que le temps de création des dictionnaires et index, le temps de création des requêtes. Ces informations sont affichées dans la console utilisée pour exécuter le fichier jar.

L'argument -output permet de créer un csv qui détaille plusieurs informations. Nous avons les différents fichiers utilisés, les différents temps obtenus ainsi que le nombre d'index créés.

L'argument -jena permet de considérer jena comme un oracle et de vérifier la validité des résultats obtenus par notre système. Un fichier csv permettra d'obtenir les résultats de chaque requête obtenus par jena et notre système. Un booléen permettra de savoir si les résultats sont équivalents ou non.

Exemple d'utilisation :

java -jar rdfengine.jar requetes 2Mdataset.rdfxml.rdf results.queryset -output -verbose -jena

requetes : dossier de requêtes
2Mdataset.rdf : fichier du jeu de données
results.queryset : fichier pour les résultats

5. Analyse des bancs d'essai

5.1. Types de benchmark utilisés

Lors de notre développement, nous avons réalisé des micro benchmarks qui nous ont été extrêmement utiles.
En effet, il nous semblait important de vérifier le travail réalisé au fur et à mesure de l'avancement. Par exemple, il ne nous semblait pas envisageable de développer un dictionnaire et des index qui prendraient plusieurs minutes à être calculés. Nous voulions, que le programme soit fonctionnel, mais aussi performant, un travail conséquent a été nécessaire sur le choix et le développement des méthodes à utiliser.

L'ensemble de ces micro benchmarks nous a permis de vérifier la validité des index, pour cela, nous avons comparé les résultats obtenus au jeu de données. En effet, nous vérifions que les triplets stockés étaient bien existants.

Concernant la performance, nous avons comparé les temps de calcul avec nos camarades. Cette entraide nous a permis de nous situer et de savoir s'il était possible d'améliorer et d'optimiser notre travail.

De plus, nous avons utilisé l'application Jena pour comparer notre système au moteur SPARQL de cette dernière. Nous savons que Jena et SPARQL sont en développement depuis de nombreuses années, nous devons le considérer comme l'oracle et il peut être compliqué d'atteindre ses performances sur des requêtes et des jeux de données complexes. Toutefois, ceci nous permet également de nous situer face à une application réelle et de nous rendre compte du travail effectué.

5.2. Tests possibles

5.2.1. Micro

Un micro benchmark permet de tester un morceau de code autonome et spécifique, par exemple une boucle, l'appel d'une méthode, mais aussi la création d'un array.
Pour ce dernier, nous pouvons tester l'efficacité d'une boucle. En effet dans certains cas une boucle ***for*** peut se montrer inutile et la boucle ***while*** se montrera plus efficace en termes de temps.
Par exemple, vérifions qu'une valeur est présente dans une Array de longueur n, une boucle ***while*** s'arrêtera lorsque la valeur souhaitée est rencontrée ou au pire des cas après n vérifications.
La boucle ***for*** devra quant à elle effectuer les n vérifications même si la valeur se trouve à l'indice 0.

5.2.2. Standard

Un standard benchmark est rédigé en collaboration dans le but de tester un système dans sa globalité.
Par exemple, un test possible est de calculer le temps d'exécution et la validité de notre système pour un jeu de données et de requêtes face à un ou plusieurs systèmes concurrents.

5.2.3. Réel

Comme application réelle, nous pouvons nommer le moteur de recherche de Google. En effet, ce dernier regroupe une immensité d'informations qui peuvent être utilisées comme données.

5.3. WatDiv

WatDiv est un benchmark développé par l'université de Waterloo.
Ce dernier permet de mesurer les performances d'un système RDF à l'aide de requêtes SPARQL.

Ce benchmark permet de générer des jeux de données ainsi que des requêtes.
Il se trouve être paramétrable par les utilisateurs, en effet, il est possible de préciser plusieurs fonctionnalités telles que la probabilité qu'une entité de type X soit associée à une entité de type Y.

WatDiv propose quatre catégories de requêtes :

- linéaires,
- en étoiles,
- en flocons et
- complexes.

Comme expliqué dans le préambule, WatDiv va nous permettre de générer différents jeux de données et de requêtes.

6. Évaluation et analyse des performances

6.1. Plan des tests à réaliser

Plusieurs tests peuvent être effectués pour évaluer les performances de notre prototype. Le travail devra être effectué sur une même machine, car le but est de comparer nos prototypes et non nos machines.

Tout d'abord, nous pouvons nous concentrer sur la création du dictionnaire et des index en termes de temps de calcul.
Toutefois, le résultat devra être pris en compte avec le temps d'exécution des requêtes pour y voir un réel intérêt. Un dictionnaire et des index plus longs à créer ne signifient pas que les requêtes seront plus ou moins courtes à exécuter.

Par la suite, nous pouvons effectuer un benchmark sur l'exécution de requêtes en étoiles. Nous pourrons comparer les temps d'exécution sur divers points tels que l'exécution globale du programme et le temps d'exécution moyen par requête.
Ces requêtes seront traitées par les différentes méthodes proposées par nos camarades et les nôtres. Les valeurs obtenues seront comparées avec Jena pour situer la performance des modèles.
De plus, nous effectuerons des tests avec un nombre de patterns différents pour visualiser des potentielles différences entre les prototypes et Jena.

Ensuite, nous pourrons nous concentrer sur le traitement de requêtes considérant n'importe quel ensemble de patterns. Ceci nous permettra de tester les méthodes M2A sur une autre catégorie de requêtes.

Les expérimentations devront comparer les prototypes sur plusieurs points.
Les principaux sont la validité des résultats obtenus et les temps de calcul. Jena étant considéré comme un oracle, nous pourrons connaître le pourcentage de requêtes validées.
Nous pourrons voir si les programmes se démarquent sur les temps de calcul et/ou la validité des réponses.

Toutefois, un des grands intérêts de ce benchmark sera de comparer notre prototype à Jena et celui de nos camarades.
Peu importe les résultats, nous pourrons analyser le programme de nos camarades pour visualiser les différents points et comprendre ce qui a joué en notre faveur ou défaveur. Chaque groupe aura sa manière de penser et de développer l'ensemble des prérequis pour mener à bien ce projet.

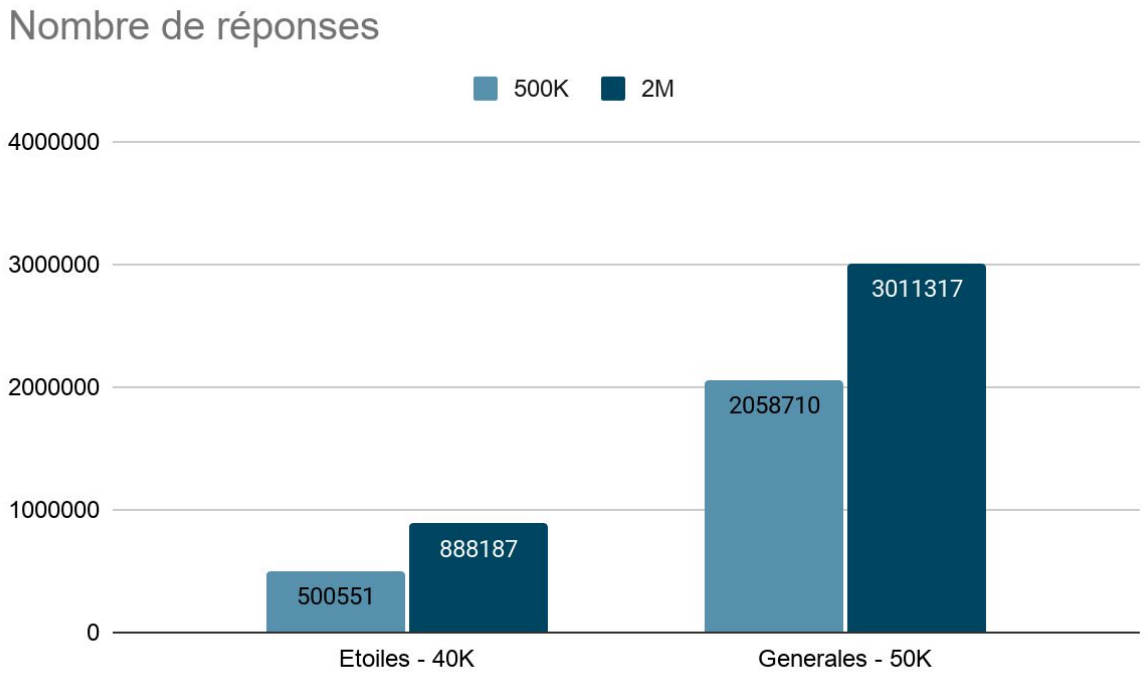
Ces données nous permettront par la suite de prendre du recul afin de visualiser le comportement de notre programme et comprendre où ce dernier est satisfaisant et où il peut être amélioré.

Pour mener à bien nos expérimentations, nous allons utiliser WatDiv pour générer les données et les requêtes. Nous utiliserons également le jeu de données présent sur Moodle. Différentes catégories de requêtes seront utilisées pour mener à bien les expérimentations.
Chaque programme aura les mêmes fichiers en entrée et sera effectué sur la même machine pour obtenir des résultats cohérents.
Jena sera défini comme l'oracle pour comparer nos systèmes et ainsi connaître le pourcentage de requêtes valides.
Ensuite, nous utiliserons les fichiers csv et/ou les consoles des différents programmes pour acquérir les informations nécessaires telles que les résultats et les temps d'exécution.

6.2. Préparation des bancs d'essais

Pour la préparation des bancs d'essai, nous avons créé deux jeux de données de 500 000 et 2 millions de triplets, de plus, nous avons généré 130 000 requêtes en étoiles et 50 000 générales à l'aide des différents patterns proposés.

Toutefois, nous nous sommes rapidement rendus compte qu'énormément de requêtes en étoile avaient zéro réponse. Pour remédier à cela, nous avons retiré ou modifié les templates qui présentaient ce problème.
Nous obtenons donc 40 K requêtes en étoile et 50 K générales.



Par la suite nous avons récupéré le nombre de requêtes qui ont zéro réponse.

Nombre de réponses vides	Requêtes en étoiles	Requêtes générales
Dataset 500K	9710	10014
Dataset 2M	3272	4507

Il est important de noter que sur les 50 000 requêtes générales, 10 000 sont en étoiles.

6.3. Hardware et Software

Pour réaliser nos tests, nous avons utilisé une machine Windows composée :

- d'un i5-9600 K cadencé à 3.70GHz.
- de 16Go de ram en DDR4 3000MHz et
- d'un SSD de 256 Go avec Lecture à 550 Mo/s - Écriture à 500 Mo/s

Le programme a été exécuté depuis un fichier jar pour éviter l'utilisation de l'IDE eclipse et minimiser les temps de calculs externes au programme.

Ceci nous semble adapté à l'analyse des performances du système, car nous avons pu bénéficier d'une machine puissante capable de gérer d'importantes quantité de données et de les traiter en peu de temps.

6.4. Métriques, Facteurs et Niveaux

Métriques

En termes de métriques pour évaluer notre moteur de requêtes RDF, nous pouvons lister

- le temps de réponse pour une requête;
- la qualité des réponses qui sera mesurée avec JENA considéré comme l'oracle où toutes ses réponses sont vraies, on parlera de correction et de complétude,
- le débit, combien de requêtes peuvent être traitées sur un temps donné
- les ressources utilisées par le système (taille de stockage et de mémoire utilisée)

Facteurs/Niveaux

Par la suite, on peut également noter plusieurs facteurs qui entrent en jeu lors de l'évaluation du système.

- CPU qui aura une cadence située entre 1.5GHz et 5GHz,
- la taille de la mémoire en RAM pouvant se situer entre 512MB, 1GB, 2GB, 4GB, 8GB, 16GB et 32GB,
- les disques durs (HDD et SSD) qui vont avoir des temps de lecture et d'écriture de l'ordre des Mo/s voire Go/s pour les plus performants, mais aussi réalisé des milliers de tr/min pour les HDD,
- le workload utilisé peut être considéré comme un facteur entrant en jeu car il peut proposer des milliers voir des centaines de milliers de requêtes,
- de même pour le dataset utilisé qui peut proposer des millions de patterns,
- pour terminer la version du système, qui peut être Linux, Mac et Windows, de plus, eux-mêmes peuvent avoir plusieurs versions (Windows 7, 8 et 10).

Selon nous, les facteurs principaux sont le CPU pour sa charge de calculs, la mémoire RAM et les disques pour leur temps d'accès aux données.

D'un autre côté, les facteurs secondaires sont la version du système qui va impacter de manière logicielle sur les performances.
Et également les jeux de requêtes et de données utilisés qui vont impacter sur la quantité de données à traiter, qui sont donc indépendantes du matériel utilisé. Nous pouvons tout à fait réaliser des millions de requêtes en quelques minutes sur un bon ordinateur avec un programme bien optimisé et calculer un millier de requêtes à l'aide d'un vieux système et d'un programme peu optimisé.

6.5. Évaluation des performances - Représentation des résultats

Concernant la préférence des mesures cold et warm, en suivant les définitions données dans le cours et en effectuant quelques recherches, nous en sommes arrivés à ces conclusions. De plus, il vous sera présenté la manière dont nous allons réaliser ces mesures en pratique à l'aide des différents outils et fichiers à notre disposition.

À propos du temps de réponse, il peut être intéressant d'effectuer une mesure cold pour obtenir le pire des cas, mais aussi de réaliser une mesure warm pour obtenir le véritable temps de réponse d'une requête.
Notre programme permet de connaître les différents temps d'exécution tels que celui de l'ensemble des requêtes à l'aide de timestamp. Nous pourrons réaliser la moyenne sur le temps obtenu pour connaître le temps moyen de traitement d'une requête.

Ensuite, pour la qualité des réponses, nous pouvons effectuer des mesures cold et warm pour vérifier s'il peut exister une différence de résultat selon ces dernières.
Toutefois, concernant cette métrique, nous nous demandons si ces mesures ont réellement un impact.

Pour poursuivre sur le nombre de requêtes traitées sur un temps donnée, il nous paraît important de réaliser une mesure warm, car une fois le système chauffé et en utilisation, nous devons connaître les capacités de notre système.
Étant donné que nous connaissons le nombre de requêtes traitées et le temps de traitement de ces dernières, nous pouvons obtenir une approximation sur le nombre de ces dernières dans un temps donné. Ceci est toujours possible à l'aide des différents timestamp utilisés dans notre programme.

Pour terminer, il est important de réaliser des mesures cold et warm sur les capacités de stockage et de mémoire pour comparer ces données en fonction de la mesure demandée. Une fois le dictionnaire et les index créés ainsi que les requêtes stockées en mémoire, le disque ne sera plus utilisé excepté lors de la création et de l'écriture des fichiers résultats. Il est important de connaître le plus haut pic d'utilisation des disques et de la mémoire.
L'obtention de la mémoire utilisée peut s'obtenir en récupérant la taille des fichiers et des objets calculés, mais également en utilisant le gestionnaire des tâches pour connaître l'utilisation de la mémoire de notre JVM.

Nous pouvons vérifier la complétude de notre système en vérifiant qu'il réponde bien à l'ensemble des questions à l'aide des fichiers résultats créés.
Concernant la correction, nous pouvons vérifier la comparaison avec Jena pour déterminer si les requêtes sont évaluées correctement.

De plus, nous avons analysé le système concurrent et nous pouvons en conclure que le système est correct et complet.

Suite à l'expérience 2^2 faisant varier la taille des données et de la mémoire, nous pouvons en conclure que plus la taille des données augmente, plus les temps de calcul augmentent.
Si nous n'allouons pas assez de mémoire au programme, ce dernier lèvera une exception et entraînera l'arrêt du programme.
Une fois la quantité de mémoire atteinte pour garder les données en RAM, il n'est plus nécessaire d'en allouer plus.

Pour ce test 2^2, nous avons obtenu ces valeurs et allons pouvoir calculer l'importance des facteurs via le modèle de régression.

	Mémoire 4Go	Mémoire 8Go
500K triplets (65Mo)	10144ms	9882ms
2M triplets (265Mo)	164000ms	156000ms

xA = -1 si on utilise 4Go de mémoire et +1 si 8Go sont utilisés
xB = -1 if 65Mo de données sinon +1 si 265Mo utilisés

Nous avons donc ce modèle de régression non linéaire.
$y = q0 + qA\ xA + qB\ xB + qAB\ xA\ xB$

(low,low) mémoire=4Go, données = 65 Mo, temps = 10144 (ms)
10144 = q0 – qA – qb+ qAB

(high,low) mémoire=8Go, données = 65 Mo, temps = 9882 (ms)
9882 = q0 + qA – qb– qAB

(low,high) mémoire=4Go, données = 265 Mo, temps = 164000(ms)
164000 = q0 – qA + qb– qAB

(high,high) mémoire=8Go, données = 265 Mo, temps = 156000(ms)
156000 = q0 + qA + qb+ qAB

10144 = q0 – qA – qb+ qAB
9882 = q0 + qA – qb– qAB
156000 = q0 + qA + qb+ qAB
164000 = q0 – qA + qb– qAB

y = 85000 - 2065xA + 74993xB - 1934xAxB

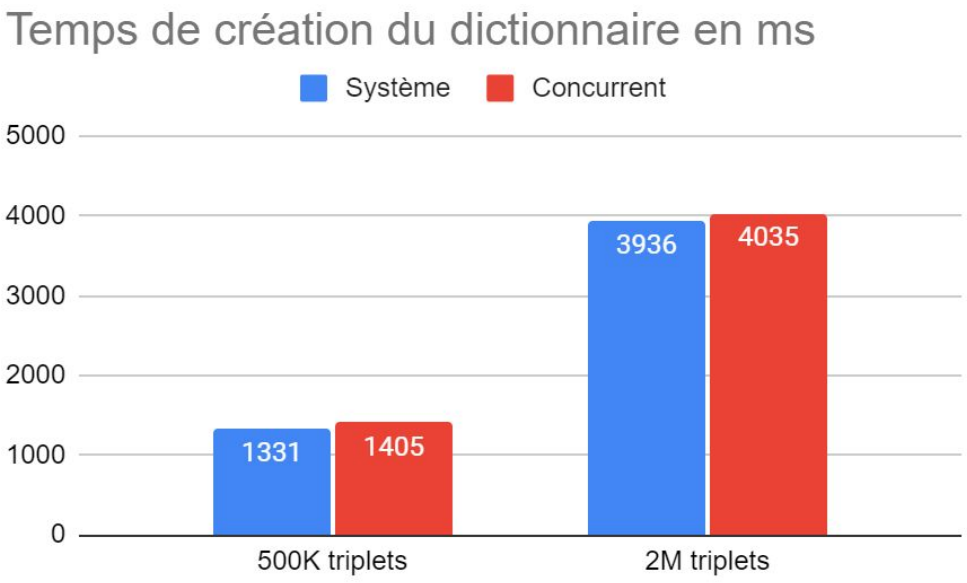
Nous pouvons interpréter cette fonction de cette manière :
– le temps de réponse moyen est de 85 000 ms,
– l'ajout de mémoire améliore les performances de 2065 ms,
– plus de données peuvent réduire les performances de 75 993 ms,
– il n'y a pas d'interaction entre la mémoire et les données.

Nous avons donc réalisé nos tests afin de comparer notre système à Jena ainsi qu'au système concurrent.
Ces derniers ont été réalisés sur 40 K requêtes en étoile et 50 K générales.
Nous avons comparé notre système au système concurrent sur les temps de création des dictionnaires et des index.
Le test final comprenant Jena a été réalisé sur le temps d'exécution des différentes requêtes.

De plus, nous nous sommes assurés que les mesures étaient réalisées de façon comparable entre notre système et celui choisi comme concurrent. Pour cela, nous avons vérifié le code source pour déterminer si les mesures étaient effectuées au même moment et sur les mêmes parties de code.

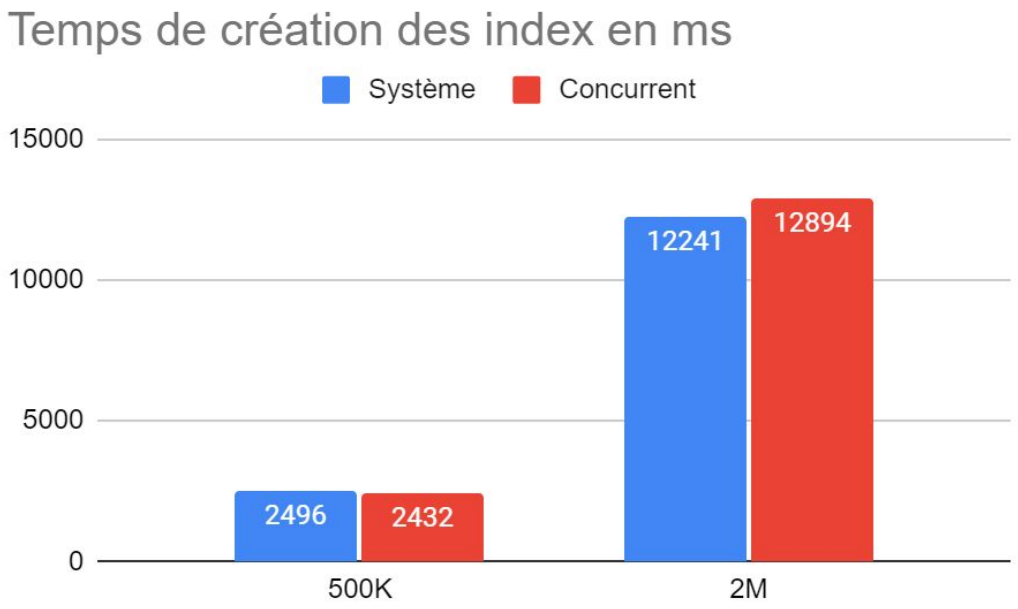
Vous pouvez ainsi observer les différents tests réalisés et analyser les résultats obtenus sous forme d'histogrammes. Chaque mesure a été effectuée 10 fois pour obtenir des résultats cohérents et représentatifs du système testé.

Premier test : Création des dictionnaires



Ce test a pour but de comparer les différents temps de création des dictionnaires sur des jeux de données de 500 K et 2 millions de triplets.
Nous pouvons observer que nous obtenons quasiment les mêmes temps avec un très léger avantage sur notre système.

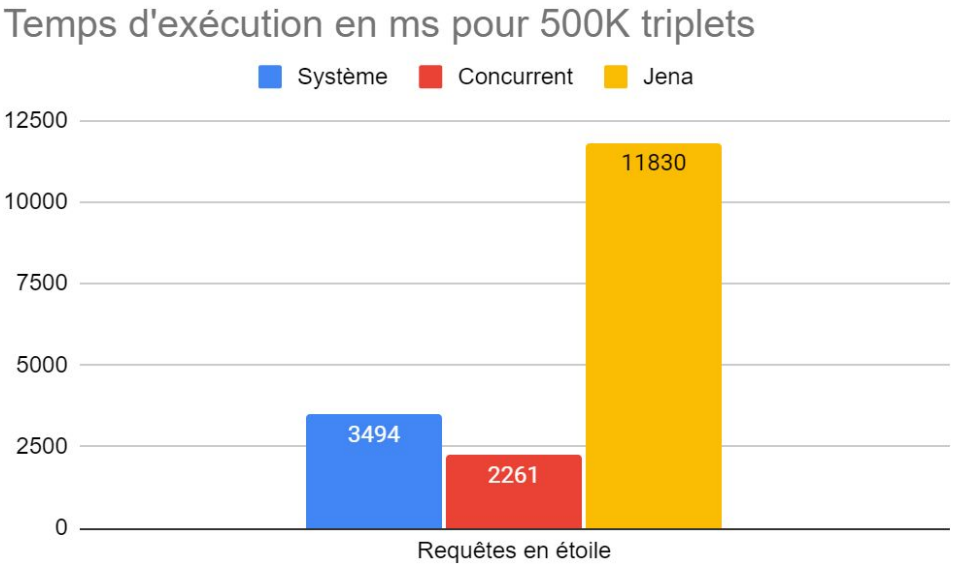
Deuxième test : Création des index



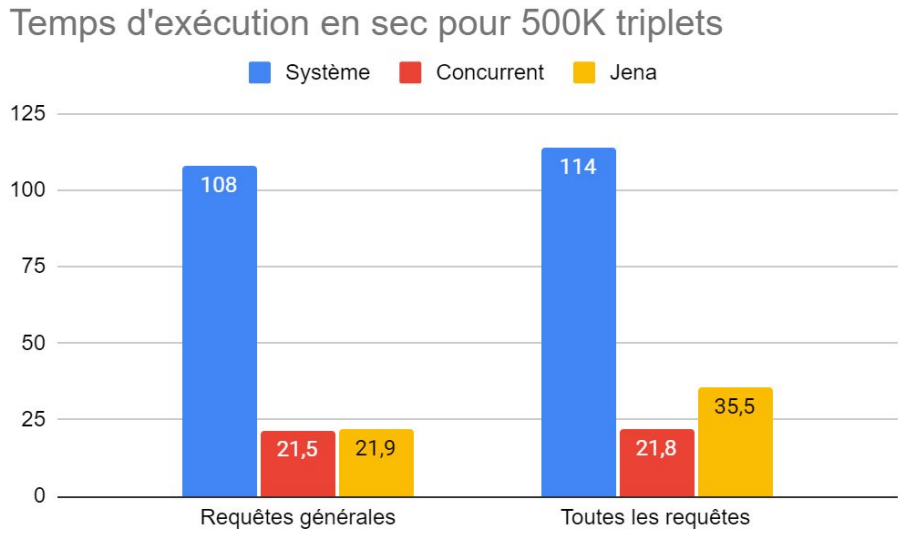
Également pour ce test, nous obtenons les mêmes temps d'exécution pour la création des index.
Ces résultats s'expliquent du fait que nous utilisons les mêmes structures pour créer et gérer le dictionnaire et les index.
De plus concernant les index, nos deux groupes ont créé les 6 index présentés dans le cours soient ops, osp, spo, sop, pos et pso.

Troisième test : Exécution des requêtes

Nous avons tout d'abord réalisé nos tests sur un dataset contenant 500 K triplets. Il y est présenté les résultats obtenus sur les différentes requêtes.

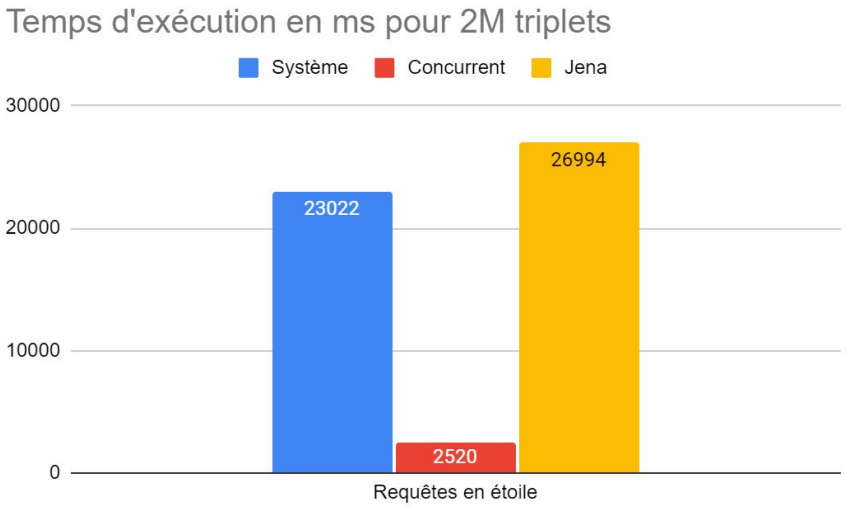


Nous pouvons observer que Jena est le programme demandant le plus de temps d'exécution pour les requêtes en étoile sur 500 K triplets. Ceci s'explique par le fait que Jena réalise un calcul sur l'ordre d'évaluation du workload et prend donc plus de temps sur la mise en place d'optimisations. Le système concurrent se montre un peu meilleur sur ce test. La tendance sera à vérifier sur les autres tests réalisés.

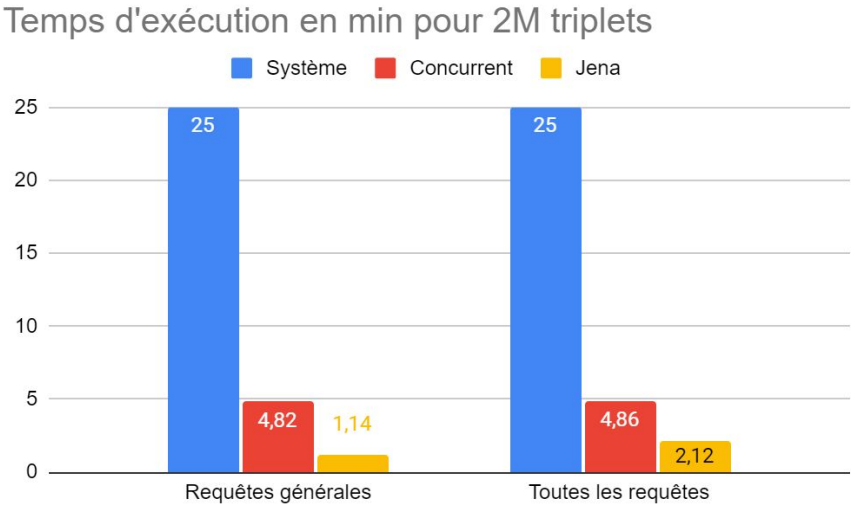


Concernant l'exécution des requêtes générales et de la totalité des requêtes, notre système éprouve des difficultés face aux deux systèmes concurrents. Nous pensons que ces différences viennent de notre système sur la mise à jour des anciennes valeurs avec les nouvelles, car une requête peut retrouver une variable dans plusieurs triplets.
Nous devons réaliser un traitement lourd qui dégrade les temps d'exécution. Nous verrons si cette tendance se retrouve avec les 2 millions de triplets.

Le système concurrent se démarque encore une fois, et arrive à battre Jena sur l'ensemble des requêtes. Toutefois, sur les requêtes générales, Jena atteint les temps du système concurrent. Il faudra analyser les résultats sur 2 millions de triplets pour voir si l'optimisation de Jena lui permet de faire mieux que ce dernier.



Concernant le traitement des requêtes avec 2 millions de triplets, le système concurrent obtient des résultats extrêmement satisfaisants. Jena perd toujours énormément de temps sur l'optimisation. Quant à notre programme, le comportement de ce dernier se dégrade et se situe désormais au niveau de Jena sans avoir à réaliser d'optimisations.



Pour le dernier test Jena arrive enfin en tête, son optimisation a payé sur des requêtes extrêmement lourdes comportant de nombreuses variables et données. Le système concurrent propose toujours d'excellents résultats de l'ordre de quelques minutes.
Notre programme montre de nouveau qu'il souffre d'un problème et atteint 25 minutes de temps d'exécution. La tendance à penser que le problème vient de la méthode qui lie les anciennes valeurs avec les nouvelles se confirme. Toutefois, le problème peut également survenir lors de la lecture des valeurs des variables.

7. Conclusion

Pour conclure, sur le plan technique, nous sommes satisfaits des tests réalisés, ils nous ont permis de nous rendre compte que notre programme avait un problème d'optimisation sur le traitement des requêtes, mais proposait d'excellentes méthodes pour la création des dictionnaires et des index. Toutefois, malgré ce défaut, nous avons conçu un système capable de répondre correctement à l'ensemble des requêtes proposées.

Ce projet nous a fait découvrir énormément de choses, car après avoir travaillé sur Jena, il était intéressant de comprendre et de développer un tel moteur de requêtes.
Il nous a également demandé d'être extrêmement assidus. Le travail devait être fait sérieusement, car une étape mal réalisée pouvait jouer sur les performances du projet. Il a fallu passer de nombreuses heures pour obtenir des résultats satisfaisants lors des diverses étapes de création.

La mise en place de réunions pour chaque groupe a été vraiment bénéfique, car elles nous permettaient de présenter le travail réalisé et d'obtenir des critiques constructives pour améliorer notre travail. Nous pouvions également obtenir des éclaircissements sur les zones de flou qui pouvaient apparaître.

La partie concernant les tests pour comparer notre système à celui de nos camarades et à Jena était l'une des plus intéressantes car elle nous a permis de voir le système pensé par d'autres élèves et d'analyser les choix qu'ils ont fait pour développer certaines structures.

Nous espérons que ce projet satisfait les attentes attendues pour ce module.