

Persistence de données RDF : mini-moteur

Guillaume VELAY – Badr HERRESS

14 novembre 2016

Table des matières

Introduction	2
Implémentation	2
Parsage triplets RDF	2
Dictionnaire	2
Graphe de données	2
Statistiques sur les données	2
Arbre des préfixes des relations	3
Arbre de voisinage FP-TREE	3
Parsage d'une requête étoile et pré-traitements	3
Graphe de la requête étoile et mesure de la sélectivité des branches	3
Résolution	4
Optimisations	4
Difficultés	5
Utilisation du programme	5
Benchmarks	5
Outils utilisés	6
Les métriques	6
Vérification et correction de notre système	6
Technique d'évaluation	7
Experiment factoriel	7
Représentation graphique des résultats	9
Conclusion	10

Introduction

Nous avons implémenté un mini-moteur qui renvoie les résultats correspondants à une requête étoile sur des données RDF. Ce système traite les données et les requêtes en mémoire vive, il est implémenté en Java.

Nous nous sommes donné le défi de résoudre ce problème avec l'approche des graphes.

Le projet est disponible vers le lien github suivant <https://github.com/proGuix/HMIN313>.

Implémentation

Nous avons suivi toutes les étapes du cours concernant les graphes mais nous l'avons adapté aux requêtes étoiles. Nous avons aussi incorporé nos propres optimisations et indexes pour rendre le système le plus efficace possible.

Parsage triplets RDF

Au commencement du programme, on vous demande de choisir quel fichier de triplets rdf choisir. La classe FileRDFParser va se charger d'enregistrer chaque uri et littéral différents dans un dictionnaire.

Dictionnaire

Une fois le parsage du fichier des triplets rdf fait, on récupère le dictionnaire sous la forme d'un objet Dictionary de FileRDFParser. Cet objet contient le graphe des données GraphData. A chaque fois qu'un nouveau triplet est parsé, on enregistre les uri, littéraux et également on construit le graphe des données. Le dictionnaire utilise une HashMap avec pour clé un entier et comme valeur une chaîne de caractères.

Graphe de données

Cet objet est représenté par un GraphData. Quand on enregistre les triplets sous forme de graphe, on fusionne les relations qui possède le même objet, nous avons donc des relations possédant plusieurs prédicats, les prédicats de chaque relation sont triés selon l'ordre du dictionnaire car dans les données nous traitons seulement des entiers car plus simple de manipulation et de comparaison. Le fait d'avoir des relations qui possèdent des prédicats triés améliore la rapidité de la recherche d'un voisin selon un ensemble de prédicats.

Statistiques sur les données

Dans le graphe des données, nous calculons pour chaque prédicats différents son nombre d'occurrence que nous enregistrons dans une HashMap. C'est utile quand on veut restreindre la recherche d'une donnée possédant un ensemble de prédicat. La recherche va s'effectuer d'abord sur les prédicats les moins occursents.

Arbre des préfixes des relations

A partir du graphe des données, nous avons construit un objet `PrefixTreeData` qui trie dans un arbre les préfixes correspondant aux relations dans le graphe des données. Cette objet est une idée qui nous est venue à l'esprit pour récupérer l'ensemble des noeuds qui possèdent un ensemble de prédicats donné. La recherche est alors plus rapide que si nous l'avions fait directement dans le graphe des données en parcourant les sommets du graphe ou les relations.

Arbre de voisinage FP-TREE

L'arbre de voisinage est représenté par l'objet `GraphNeighborBis`. Il s'agit d'un FP-TREE. C'est un arbre de préfixe sur les relations sortantes d'un noeud donné, il enregistre tous les voisins accessible à partir d'un ensemble de prédicats donné. Ce arbre possède également des `linkedList`, il s'agit pour chaque prédicat de tracer un chemin reliant le même prédicat. La recherche d'un voisin selon un ensemble de prédicats p_1, \dots, p_n va donc constituer une comparaison sur chaque prédicats de chaque `linkedList` correspondant aux prédicats p_1, \dots, p_n . Une optimisation consiste à trier l'arbre de voisinage en largeur et en profondeur selon l'ordre des prédicats dans le dictionnaire, cela à pour but de créer naturellement des `linkedList` ordonnées.

L'objet qui contient tous les arbres de voisinages de chaque noeud du graphe des données s'appelle `NeighIndexBis`, il les contient tous dans une `HashMap`.

Parsage d'une requête étoile et pré-traitements

Une fois que le graphe des données est chargé, avec l'arbre des préfixes et les arbres de voisinages, le programme va demander un document de requêtes à charger.

L'objet qui parse une requête étoile est `QueryParser`. Les pré-traitements effectués sont semblables à celui de la création du graphe de données : des branches de la requête peuvent être fusionnées si ces branches ont le même objet et les prédicats des branches sont triés selon l'ordre du dictionnaire. Lorsque nous créons le graphe de la requête étoile nous établissons également un ordre sur la longueur des branches de la plus longue à la plus petite. Cela va nous servir pour obtenir l'ordre final de traitement des branches dans la résolution.

Graphe de la requête étoile et mesure de la sélectivité des branches

L'objet correspondant au graphe de la requête étoile est `QueryGraph`.

Une fois que le parsage de la requête est effectué ainsi que l'ordre initial des du traitement des branches, nous allons définir l'ordre final en jouant sur le nombre d'occurrences d'un prédicat dans les données. Nous avons crée une heuristique qui nous a fourni un ordre que nous avons conjecturé comme correct par rapport à un ordre total sur la sélectivité des prédicats.

Voici un exemple, nous avons dans l'ordre de la plus petite à la plus grande occurrence ces prédicats présents dans la requête étoile : p_3, p_1, p_2 donc nous sélectionnons en premier les branches qui contiennent p_3 sachant que les branches sont aussi triées par longueur de la plus grande à la plus petite, puis les branches restantes contenant p_1 et enfin p_2 .

L'algorithme qui rendrait un ordre total prendrai en premier les branches qui contiennent p_3 puis les trierai suivant le nombre d'occurrence de p_1 et p_2 mais ce tri est possible que si on touche aux données car il faudrait regarder d'abord dans l'arbre des préfixes des relations, puis pour

chacun des sujets trouvés il faudrait regarder dans leur arbre de voisinage et compter pour p_1 et p_2 les occurrences d'objets.

Cependant si on avait plus de statistiques sur la distribution des données, on pourrait améliorer l'ordre qu'on obtient sans toucher aux données.

Finalement, sachant que le traitement des requêtes et l'ordonnement des branches peut drastiquement impacter sur le temps de résolution, nous aurions aimé étudier plus en profondeur d'autres heuristiques pour avoir le meilleur compromis entre le traitement d'une requête et le temps de résolution.

Résolution

La résolution se fait dans l'objet `Resolution`. Une fois que tous nos éléments cités précédemment sont créés, nous pouvons commencer l'algorithme. Celui ci prend la première branche de la requête étoile selon l'ordre précédemment cité et cherche tous les noeuds qui possèdent les mêmes prédicats que cette branche dans l'arbre des préfixes. Nous obtenons donc l'ensemble des noeuds initiaux sur lesquels nous allons effectuer la suite de l'algorithme. Toutes les solutions à la requête sont incluses à l'intérieur de cet ensemble. Ensuite, pour chacun de ces noeuds on vérifie l'existence de toutes les branches de la requête (prédicats et objets) à ce noeud grâce à l'arbre de voisinage. Si et seulement si toutes les branches correspondent à ce noeud alors il est solution.

La grande difficulté dans cet algorithme est de restreindre au maximum l'ensemble initial de recherche, plus il est réduit et moins il est coûteux en temps.

Optimisations

En ce qui concerne le FP-TREE (arbre de voisinage), nous l'avions initialement implémenté naïvement comme dans le cours. Notre première réflexion sur une optimisation concernait les `LinkedList` (lignes rouges), nous pensions que si nous n'avions pas des listes mais des graphes qui reliraient les prédicats entre eux par un lien d'inclusion (liste de prédicats inclus dans une autre) nous gagnerions du temps de calcul dans la recherche d'un voisin. Mais après une discussion avec notre professeur, celui-ci nous a dissuadé de faire de la sorte car les listes de prédicats des branches n'étaient pas forcément contiguës. Mais nous aurions pu quand même effectuer cette optimisation en changeant le lien d'inclusion par le lien de sous ensemble d'une liste de prédicats qui contient un sous ensemble d'une autre liste de prédicats, mais la complexité pour la création de ceci devenait exponentielle sur le nombre de branches.

Au bout de la discussion, notre professeur nous a donné une optimisation pour la recherche d'un voisin dans le FP-TREE, il s'agissait d'implémenter l'arbre de voisinage non pas avec une structure arborescente mais en utilisant des tableaux qui stockeraient pour chaque noeud de l'arbre son `begin/end`. La grande propriété de cette notation est qu'il est possible de savoir si deux noeuds n_1 et n_2 sont soit l'un ancêtre de l'autre, soit l'un dans une branche de gauche ou de droite par rapport à l'autre par des comparaisons numériques des `begin/end`. Si $begin_2 > begin_1$ et $end_2 < end_1$ alors n_1 est ancêtre de n_2 . Si $begin_2 > begin_1$ et $end_2 > end_1$ alors n_1 est dans une branche à gauche de celle de n_2 . Si $begin_2 < begin_1$ et $end_2 < end_1$ alors n_1 est dans une branche à droite de celle de n_2 . L'optimisation permet donc d'éviter des vérifications en parcourant les branches de l'arbre.

Nous avons aussi pensé à mixer notre algorithme avec une méthode par dichotomie et selon la distribution des données, cela auraient peut être été encore plus rapide.

Nous avons une autre optimisation, celle d'implémenter la solution avec le langage C++ pour ce qui est de la gestion de la mémoire, malheureusement nous n'avons pas eu assez de temps.

Difficultés

La principale difficulté que nous avons rencontré se base sur le temps d'étude. Si nous avions eu plus de temps nous aurions par exemple choisit d'utiliser plutôt des HashMap que des ArrayList dans certains cas. Ou par exemple l'étude d'autres heuristiques qui nous aurait permis de gagner du temps de résolution en se basant sur la distribution des données.

Nous avons aussi pris du temps pour comprendre d'abord la solution par l'approche des graphes.

Le code actuel n'est pas factorisé, vu le nombre d'objet que nous devons implémenter. Nous n'avons pas pris le temps d'enlever ce qui ne sert à rien car nous avons ré-implémenté des objets par manque d'efficacité.

Utilisation du programme

Notre application s'appelle **app.jar**. Si vous voulez accéder au lien github de notre projet, les informations pour compiler et exécuter le programme y sont inscrites dans le README en anglais. Sinon ci-joint au rapport vous trouverez le projet déjà prêt à l'emploi.

La commande pour exécuter le programme est **java -jar app.jar** et laissez vous guider. Les fichiers de triplets RDF se trouve dans le dossier **test/dataset**, le programme vous donne le choix de prendre entre 100K.rdf et 500K.rdf. Ensuite vous devez attendre que le programme charge les données, il vous donne à chaque chaque instant le nombre de triplets qu'il a déjà chargés.

Une fois que les données sont créées, vous devez choisir le fichier de requêtes à exécuter, les fichiers se trouvent dans **test/queries**, il y a en a douze, chacun de ces fichiers possèdent cent requêtes.

Le programme vous donne une trace des résultats en console, dès que toutes les requêtes d'un fichier sont exécutées, le programme vous redemande si vous voulez exécuter un autre fichier de requêtes.

Quand les requêtes d'un fichier sont exécutées, le programme écrit les résultats dans le dossier **test/queries_result_and_time**. Par exemple pour le fichier de requêtes **Q_1_includes.queryset** le programme renvoie les fichiers **Q_1_includes.queryset_result.csv** et **Q_1_includes.queryset_time.csv**. Pour chacun de ces deux fichiers, chaque ligne numéro *N* correspond au résultat ou au temps (en ms) de la requête numéro *N*.

Benchmarks

Durant ce projet et afin de réaliser nos tests nous avons utilisé un benchmark standard, qui est WatDiv.

Test possible pour chaque type de benchmark.

WatDiv est un benchmark fait pour mesurer les performances d'un gestionnaire de données rdf. Il est composé de deux générateurs un pour les données et le second pour les requêtes sur ce jeu de données. WatDiv permet aussi aux utilisateur de créer leurs propres jeux de données en paramétrant les entités et les associations.

Pour nos tests nous avons utilisé le benchmark pré-configuré du Moodle : 100K et 500K de triplets pour les données et 12 fichiers de 100 requêtes.

Outils utilisés

Les tests ont été réalisés sur deux ordinateurs différents :

- MAC OS X 10.11.6, 1.6 GHZ Intel Core i5, 8 Go RAM, 121 GO SSD.
- Linux Ubuntu 14.04 LTS, Intel Core i5, 4 Go RAM, 750 Go.

On a aussi utilisé l'outil WatDiv afin de générer des données et des requêtes pour le test de notre programme.

Les métriques

Pour ces tests on va se baser sur deux métriques :

- Le temps de chargement des données par le programmes.
- Le temps d'exécution et de réponses pour les requêtes effectuées par le programme.

On va considérer les facteurs suivants :

CPU	1.6 GHZ Intel Core i5	1.6 GHZ Intel Core i5
RAM	8 Go	4 Go
La taille des données	100 000 triplets	500 000 triplets
OS	MAC OS X	Linux Ubuntu 14.04

Nous pensons que la taille des données sera la facteur le plus important suivie de la RAM et la CPU et enfin l'OS. Et nous pouvons les répartir en facteurs principaux (RAM et Data size) et secondaires (CPU et OS)

Vérification et correction de notre système

Afin de nous assurer du bon fonctionnement de notre système, nous avons effectué les mêmes requêtes sur le même jeux de données mais cette fois-ci en utilisant Jena. Dans le dossier **com-pletude/**, vous trouverez une classe Complétude qui compare les résultats de deux fichiers : jena_results.csv et app_results.

La procédure est la suivante :

- on entre les résultats d'une requete de jena dans un dictionnaire.
- pour chaque resultat d'une requete de notre application, on regarde si le resultat a déjà été traité comme doublon.
- s'il n'existe pas déjà comme doublon alors on vérifie si le résultat existe dans le dictionnaire.
- s'il existe dans le dictionnaire, on met la valeur dans un ensemble ensExist, s'il ne contient pas on entre la valeur dans ensNExist.

On a ainsi l'ensemble ensExist des résultats qui sont vérifiés, l'ensemble ensNExist des résultats non vérifiés et enfin l'ensemble ensDbls des résultats doublons.

Nous avons pu vérifié que notre système est à 100% complet sans doublons sur les 1200 requêtes pour les dataset 100K et 500K.

Technique d'évaluation

Nous pensons que les mesures cold, warm seront plus intéressants à utiliser dans la métrique qui concerne le temps d'exécution d'une requête, en effet lors de cette opération le système va chercher l'information dans la mémoire de l'ordinateur.

Afin de réaliser nos mesures, nous avons décidé d'opérer de la manière :

- Nous exécutons notre système, puis celui du groupe avec qui nous devons comparer et enfin celui de Jena sur deux jeu de données de tailles différentes (100.000 et 500.000 triplets).
- Nous exécutons ensuite les 1200 requêtes sur chacun d'eux.
- Nous obtenons alors 6 fichiers csv représentant chacun le temps d'exécutions des 1200 requêtes pour un système et une taille de données
- Cette opération sera réalisée 5 fois, une au démarrage de l'ordinateur («cold») et quatre autres après au moins une exécution («warm»)
- Nous obtenons ainsi 30 fichiers avec chacun 1200 résultats. Nous parcourrons chacun des fichiers afin de réaliser une moyenne arithmétique du temps d'exécution d'une requête.
- Nous obtenons donc 10 résultats pour chacun des systèmes, représentés dans les tableaux suivants :

Les résultats pour 100.000 triplets.

Système	Temps 1 "Cold"	Temps 2 "Warm"	Temps 3 "Warm"	Temps 4 "Warm"	Temps 5 "Warm"
Notre système	1.5016ms	1.4775ms	1.3883ms	1.3558ms	1.3866ms
Jena	1.0892ms	0.9892ms	1.1584ms	1.0261ms	0.97ms
Système de Florient Dieu	0.6388ms	0.5578ms	0.5476ms	0.4967ms	0.5459ms

Les résultats pour 500.000 triplets.

Système	Temps 1 "Cold"	Temps 2 "Warm"	Temps 3 "Warm"	Temps 4 "Warm"	Temps 5 "Warm"
Notre système	5.9758ms	5.9366ms	5.8858ms	5.8691ms	5.8875ms
Jena	1.8723ms	1.65ms	1.4592ms	1.7138ms	1.5723ms
Système de Florient Dieu	1.7689ms	1.7381ms	1.6690ms	1.6072ms	1.6061ms

Experiment factoriel

Nous allons commencer par réaliser un expriment factoriel 2^2 , en utilisant le facteur de la mémoire et celui de la taille de données. Le temps affiché représente le temps d'exécution des 1200 requêtes.

	4 GB mémoire	8 GB mémoire
100.000 triplets	1569ms	1669ms
500.000 triplets	7922ms	7076ms

Considérons la variable x_a associée à la mémoire ($x_a = -1$ pour 4GB et $x_a = 1$ pour 8GB) et la variable x_b associée à la taille de la donnée ($x_b = -1$ pour 100.000 triplets et $x_b = 1$ pour 500.000 triplets).

Le modèle de la non régression linéaire :

$$y = q_0 + q_a \cdot x_a + q_b \cdot x_b + q_{ab} \cdot x_a \cdot x_b$$

Dans le cas du (low,low) ce qui revient à dire une mémoire de 4GB et une taille de données de 100.000 triplets, on a :

$$1569 = q_0 - q_a - q_b + q_{ab}$$

Dans le cas du (high,low) ce qui revient à dire une mémoire de 8GB et une taille de données de 100.000 triplets, on a :

$$1669 = q_0 + q_a - q_b - q_{ab}$$

Dans le cas du (low,high) ce qui revient à dire une mémoire de 4GB et une taille de données de 500.000 triplets, on a :

$$7922 = q_0 - q_a + q_b - q_{ab}$$

Dans le cas du (high,high) ce qui revient à dire une mémoire de 8GB et une taille de données de 500.000 triplets, on a :

$$7076 = q_0 + q_a + q_b + q_{ab}$$

Obtient donc :

$$1569 = q_0 - q_a - q_b + q_{ab}$$

$$1669 = q_0 + q_a - q_b - q_{ab}$$

$$7922 = q_0 - q_a + q_b - q_{ab}$$

$$7076 = q_0 + q_a + q_b + q_{ab}$$

Alors

$$y = 4559 - 86.5x_a + 2940x_b - 336,5x_{ab}$$

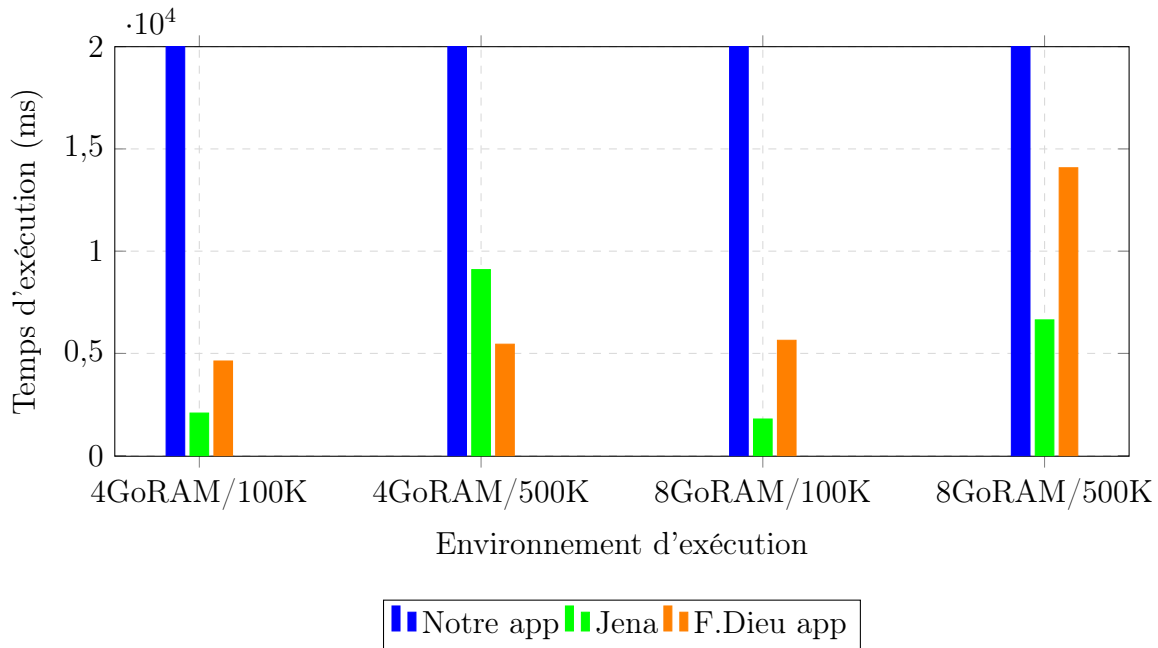
Cette équation nous permet de déduire certaines propriétés de notre système :

- On remarque que l'effet de la variation de la mémoire sur le temps d'exécution des requêtes est quasiment nulle et peut être considéré comme négligeable.
- Par contre la variation de la taille du jeu de données peut être considéré comme le facteur principal dans la variation du temps d'exécution.
- l'interaction entre les deux facteurs est aussi négligeable que la variation effectuée par la mémoire

Ensuite nous avons voulu faire un experiment factoriel avec 2^{4-1} , avec 4 qui représente les facteurs cités dans la section «Les métriques» (CPU, RAM, Data size, OS) et 1 le facteurs parmi ceux précédemment cité dont on considérera qu'un seul niveau et qui se trouve être la CPU. Pour réaliser cet experiment nous avons besoin d'un ordinateur sous Mac avec 4Go de RAM et un autre sous Ubuntu avec 8 GO de RAM afin d'avoir les résultats nécessaire. Malheureusement nous n'avons à disposition aucun des deux matériels cité précédemment .

Représentation graphique des résultats

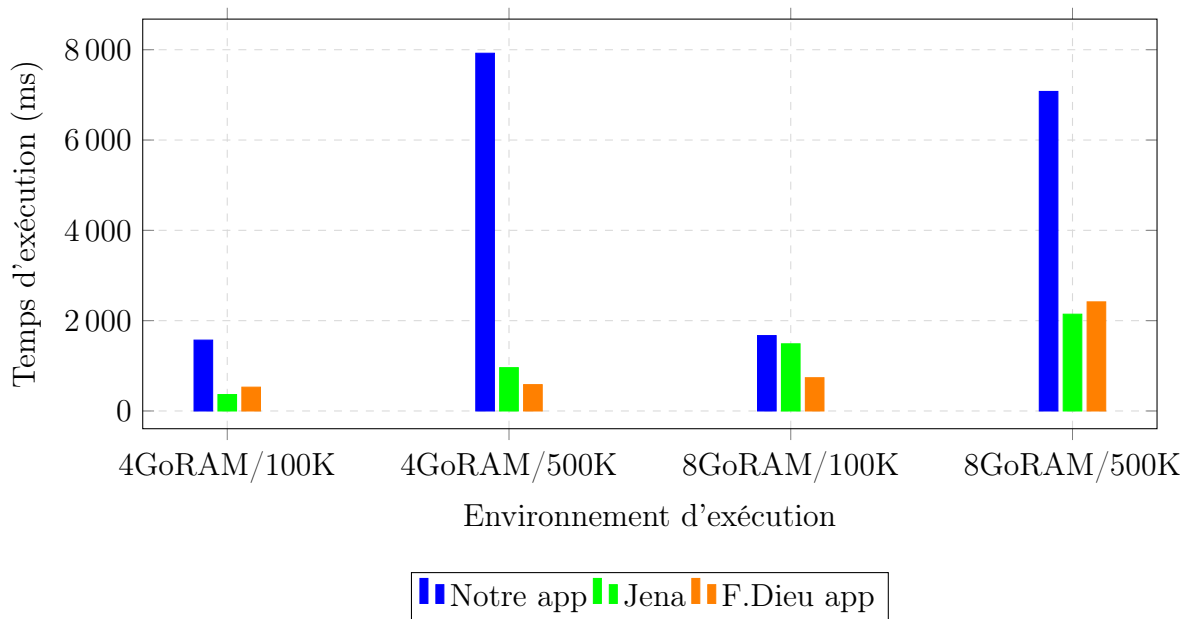
Temps d'import des données de notre système, de celui de jena et de celui de Florient Dieu selon 4 types de configuration.



Moyenne arithmétique de 3 temps sur 5

Ce premier graphique correspond au temps d'import des données dans le système de notre application, de celle de Florient Dieu et enfin celle de Jena. Nous remarquons clairement un temps beaucoup plus important pour notre application, qui dans tout les cas de figure, représentés ici, le temps d'import dépasse largement ceux des autres applications.

Temps d'exécution des 1200 requetes de notre système, de celui de jena et de celui de Florient Dieu selon 4 types de configuration.



Moyenne arithmétique de 3 temps sur 5

Le deuxième graphique correspond aux temps d'executions des requêtes de notre application, de celle de Florient Dieu et enfin celle de Jena. Nous remarquons que la taille de données a beaucoup d'importances dans les résultats. En effet les requêtes sur 100.000 triplets prennent moins de temps que sur les 500.000.

Pour les 100.000 triplets nous remarquons aussi que le temps d'exécution des requêtes prend un peu plus de temps pour notre application comparés aux autres, mais reste quand même assez proche. Contrairement à quand nous les exécutons sur 500.000 triplets, où là on a une différence plus flagrante.

Conclusion

Nous avons eu comme objectif dès le départ, d'essayer d'optimiser les approches graphes vu en cours afin de réaliser un mini-moteur rdf le plus performant possible. Nous avons eu une idée d'optimisation, que nous pensions performante. Nous l'avons implémenté pour ce projet, malheureusement les résultats obtenues étaient en dessous de nos prévisions avec une performance en dessous des applications avec lesquelles nous l'avons comparé.

La question à se poser maintenant est pourquoi cette optimisation, que nous pensions efficace, se retrouve être moins intéressante que les autres proposées.