Université de Montpellier

Rapport de projet NoSQL
Partie II - Évaluation et Analyse des Performances

El Houiti Chakib
Kezzoul Massili

4 janvier 2022

Introduction

Ce rapport fait suite à la première partie du projet qui concerne l'implémentation d'un moteur de requête *SPARQL* en étoiles en utilisant l'approche *hexastore*. Dans cette partie, nous allons analyser les performances de notre implémentation par rapport à d'autres implémentations. Notamment celle de Jena.

Dans un premier temps, nous allons préparer et analyser des bancs d'essais en utilisant *WatDiv*[1]. *WatDiv* est système développé afin de mesurer les performances d'un moteur de requête *SPARQL*. Il consiste en la génération de jeux de données ainsi que des jeux de requêtes.

Dans un second temps, nous allons définir et comparer plusieurs plans de tests afin d'en trouver un (ou plusieurs) qui donne des résultats correctes, significatifs et interprétables.

Enfin, viendra la partie concrète d'évaluation des performances. On exécutera les plans de tests précédemment réalisés. Suivant les résultats obtenus, nous les présenterons selon des représentations graphiques que nous allons analyser. Nous expliquerons aussi les raisons à la base de ces résultats.

1 Bancs d'essais

1.1 Préparation des bancs d'essais

La première partie est de générer les jeux de données sur lesquelles les tests vont être exécutés. Pour cela, on utilise le programme *WatDiv*.

1.1.1 Génération des données

La génération d'un *dataset* se fait en utilisant la commande suivante :

```
./watdiv -d <model-file> <scale-factor>
```

Listing 1 – Commande pour la création d'un dataset

- *model-file* est la template sur laquelle le programme se base pour créer le jeu de données ;
- Pour un $scaleFactor = 1$, on obtient environ $100K$ triplets. On peut donc augmenter le nombre de triplets en augmentant la valeur de $scaleFactor$.

Afin de répondre au besoin de l'évaluation, nous avons généré un *dataset* de $500K$ triplets ainsi qu'un autre de $1M$ de triplets.

1.1.2 Génération des requêtes

Afin de générer des requêtes, nous avons écrit un script *Shell* qui, à partir de quelques templates, génère pour chacune d'elle un fichier contenant 1000 requêtes.

```
for template in ${TEMPLATE_DIR}/*.sparql-template;
do
  template_name=${template##*/}
  template_name=${template_name%.sparql-template}
  querie_file=$RESULT_DIR"/"$template_name".queryset"

  if [ $overwrite -eq 0 ] && [ -f $querie_file ]; then
      echo -e "File '$querie_file' already exists.\nDo you want to overwrite ? (any key to continue / CTRL-C to exit)"
      read $x
      echo -e "Overwriting..."
```

1. Waterloo SPARQL Diversity Test Suite.

```
        overwrite=1
    fi

    $WATDIV/watdiv -q $WATDIV/model/wsdbm-data-model.txt ${template} $NB_QUERIES 1 >
    $querie_file
    i=$((i+1))
done
```

Listing 2 – "Extrait du script qui génére les requêtes

1.2 Analyse des bancs d'essais

L'analyse des bancs d'essais se fait principalement sur la *qualité*[2] des requêtes générer.

1.2.1 Première version

Une première version du banc a été réalisée en utilisant les templates fournies de base. Quelques statistiques ont été extraites de ces templates afin de visualiser la *qualité* des ses requêtes.

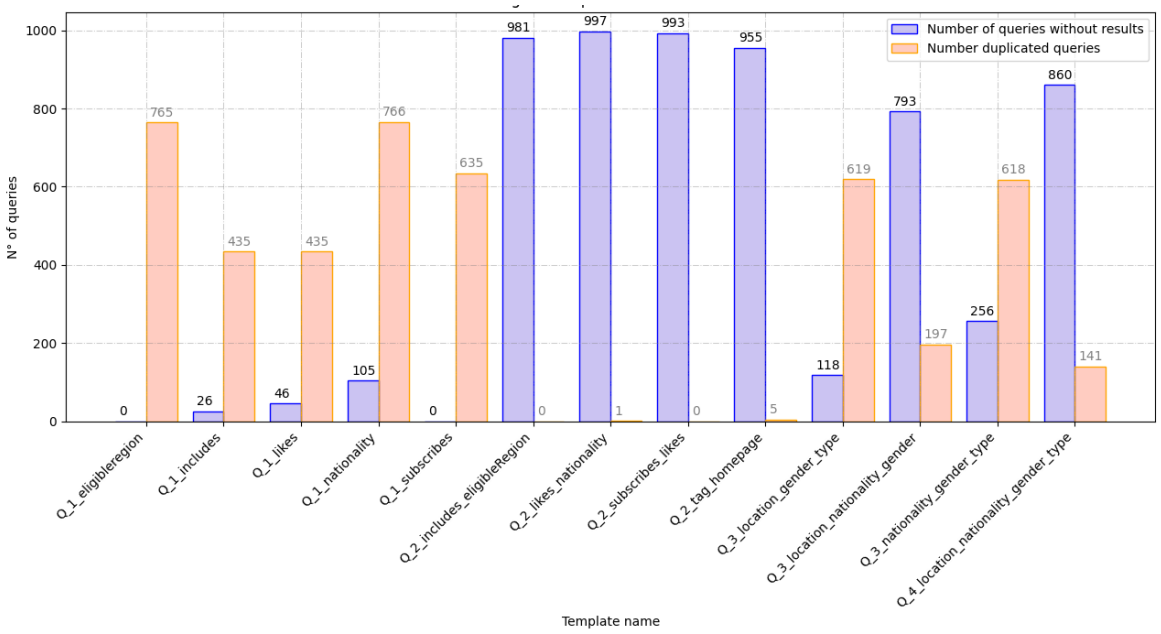

FIGURE 1 – Nombres de requêtes sans réponses et dupliquées

Sur la figure 1, on voit pour chaque template le nombre de requêtes sans réponses (en bleu) ainsi que le nombre de requêtes en doubles (en orange). Le nom des templates commence par Q_i, où i represente le nombre de *patterns* des requêtes associées.

On observe clairement sur l'histogramme que trois groupes de templates se forment.

1 pattern Ces templates ont un grand nombre de requêtes dupliquées et peu de requêtes sans résultats. Cela est probablement dû au fait que plus la requête est simple (peu de pattern) et plus elle a de résultats. Par contre, cela diminue le nombre de requêtes *différentes* qu'on peut générer pour cette template.

2. On définit ici la *qualité* d'une requête par la qualité des analyses qui découlerons de celles-ci

2 patterns On observe sur ce groupe que quasiment toutes ses requêtes sont uniques mais aussi qu'elles sont toutes sans résultats. Ce qui n'est pas souhaitable. Nous reviendrons plus tard sur ces templates afin de voir d'où viens ce problème.

3 patterns ou plus Ce groupe donne un résultat assez varié. On observe clairement ici une corrélation entre le nombre de doublons et le nombre de requêtes sans réponses. Plus l'un est élevé et plus l'autre est faible.

Ces résultats ne sont pas satisfaisants, car ils biaiseront fortement les résultats des performances. En effet, un nombre élevé de requêtes sans réponses affectera forcément le temps d'exécution de notre implémentation, car celle-ci s'arrête dès qu'un des patterns ne donne pas de résultat. La requête ne s'exécute pas entièrement. Cette valeur doit donc être *minimal*.

Le nombre de doublons aussi doit être de préférence proche de zéro. Dans le cas contraire une requête qui apparaît trop souvent influencera plus que les autres la moyenne du temps d'exécution. Si par exemple, cette requête s'exécute plus rapidement alors la moyenne se verra sous-estimé. Par contre, ce facteur reste moins important que le nombre de requêtes sans réponses. De plus, le nombre de doublons, i.e le nombre de requêtes qui se répète au moins une fois, n'est pas un nombre représentatif du problème des doublons. En effet, si par exemple, on a N doublons et que c'est une seule requête qui se répète N fois. Alors ce cas sera beaucoup plus problématique que le cas où on a $N/2$ requêtes qui se répètent 2 fois chacune. Donc le nombre maximum de répétition d'une requête est un indicateur à privilégié au nombre de doublons.

1.2.2 Vers une amélioration du banc d'essai

Les requêtes à un seul pattern donnent un résultat satisfaisant. On a donc décidé de les laisser tel quel. Par contre, les requêtes avec deux patterns ne le sont pas du tout. Pour remédier à cela nous avons changé les templates associées. On obtient à la fin la figure suivante :

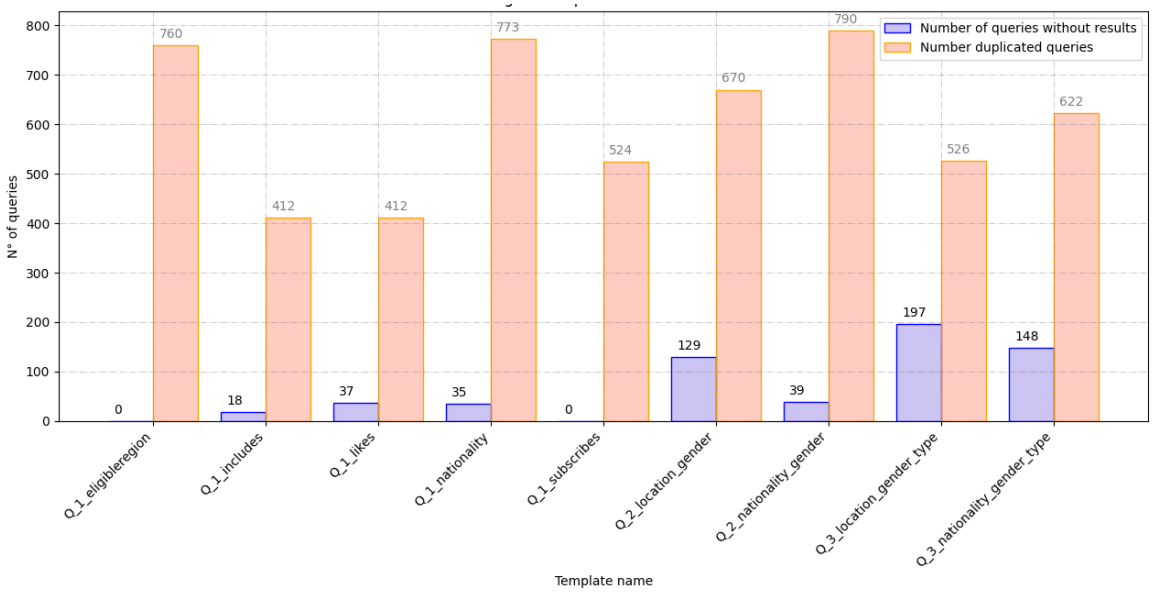

FIGURE 2 – Nombres de requêtes sans réponses et dupliquées - amélioré

On voit ici que le nombre de requêtes sans réponses est assez faible. Par contre, le nombre de doublons

n'a pas changé. Ceci est dû à l'implémentation de *WatDiv* qui ne permet pas [3] la génération de requêtes sans doublons.

Pour remédier à cela, nous avons calculé, comme précisés dans la section précédente, le nombre maximum de répétition d'une requête (Qu'on va noter M dans ce qui suit). Ce qui nous donne le graphe ci-dessous.

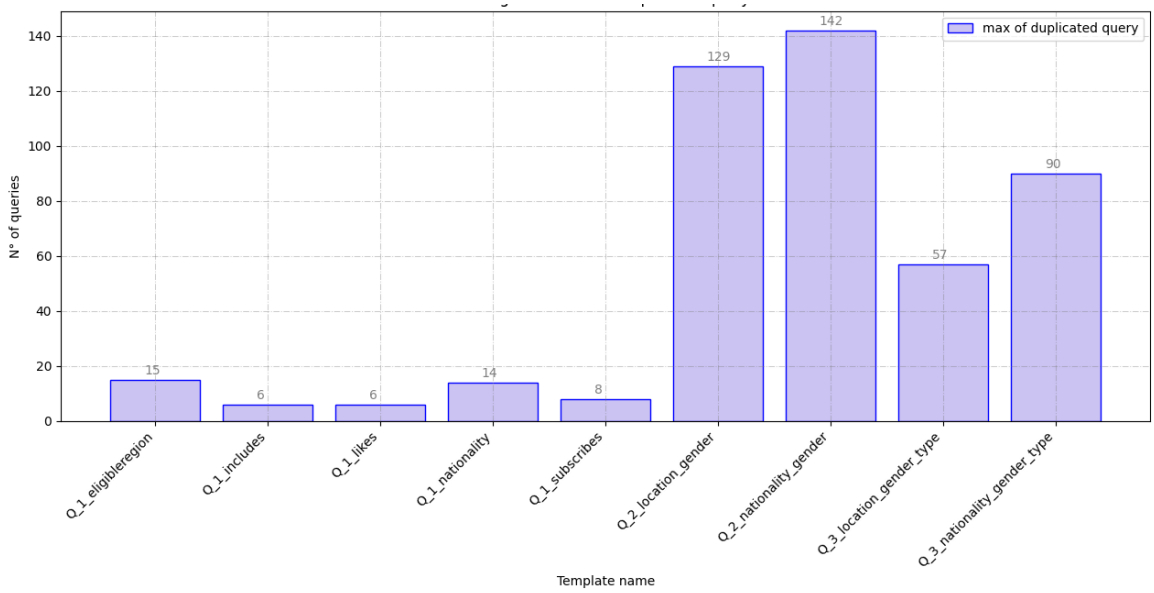

FIGURE 3 – Nombres maximum de répétition d'une requête par template

On observe ici que, pour les requêtes à un seul pattern, M n'excède pas 15. Ce qui est un bon résultat. Mais, pour ce qui est des requêtes à deux pattern et plus, on a un M qui tourne autour des 100. Donc une requête qui se répète 100 fois influencera forcement la partie d'évaluation des performances.

En changeant les templates on n'a pas réussi à minimiser les deux valeurs de façon satisfaisante. Le meilleur résultat qu'on a pu avoir est le suivant :

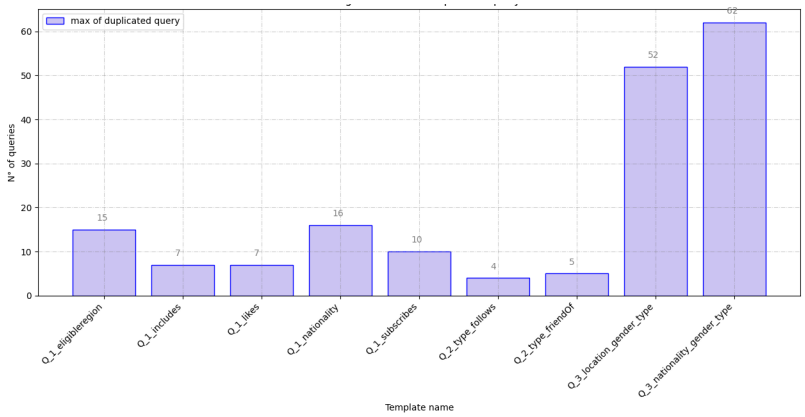

FIGURE 4 – Nombres maximum de répétition d'une requête par template - amelioré

3. En tout cas, pas à notre connaissance

On a réussi a trouvé des templates à deux patterns avec un résultat satisfaisant, mais pour ce qui est des requêtes à 3 patterns, il est beaucoup plus compliqué d'en trouver. Si une template donne peu de duplications alors elle donnera forcément plus de requêtes sans réponses. On a donc décidé de privilégier cette dernière.

2 Hardware et Software

Les tests se feront dans une machine avec les caractéristiques suivantes :

Hardware

CPU Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 3072 KB cache × 4

Mémoire vive 7,6 Gio

Disque 1 To Toshiba @ 5400 RPM

Software

Système d'exploitation Pop!_OS 21.04, 64 bits

Java OpenJDK 11.0.11 2021-04-20

Python Python 3.9.5

Cette configuration n'est pas optimale. En effet, la machine utilisée est un ordinateur portable, ce qui n'est pas le serveur de base de données le plus approprié et le plus représentatif. De plus, à cause de la limitation au niveau de la mémoire vive, on ne peut pas charger en mémoire un *dataset* de plus de $2M$ de tuples.

3 Plans de tests

Avant de définir les plans de tests, nous devons tout d'abord définir les métriques permettant d'évaluer les performances du moteur de requêtes.

3.1 Métriques

Le moteur de requêtes peut-être évalué sur plusieurs critères : la qualité, l'efficacité et la performance.

Qualité On définis la qualité d'un système par le pourcentage des réponses correctes données par ce dernier. Pour cela, on compare nos résultats avec ceux d'un système de confiance. Dans notre cas, ça sera *Jena*.

- **La correction** est le pourcentage de réponses de notre système qui sont aussi des réponses de *Jena*
- **Complétude** est le pourcentage de réponses de *Jena* qui sont aussi des réponses de notre système.

Temps de réponse C'est le temps d'évaluation d'une requête ($ms/query$)

Débit Le nombre de requêtes évaluées en un certain temps ($query/ms$)

Ressources utilisées Le nombre de CPU et de mémoire utilisés par le système (Mo ou temps de CPU).

Scalabilité — Dégradation du temps de réponse avec plus de données.
— Réduction du temps de réponse avec plus de ressources.

3.2 Facteurs

Un facteur est une variable ayant des alternatives qui peuvent affecter la mesure des performances du système. Dans notre cas, les facteurs qu'on peut prendre en compte sont les suivants (classés selon leur importance) :

— Taille de la mémoire
— Taille des données
— Workload
— Type CPU

Chaqu'un de ces facteurs admet des niveaux (des valeurs possibles) qui peuvent affecter grandement la mesure des performances. Dans l'idéal, on voudra tester toutes les valeurs possibles (*Full Factorial Design*). Mais cela est généralement impossible, car le nombre d'expérimentation à réaliser est exponentielle. Une autre manière de procéder est de faire varier un seul facteur à la fois. Par contre, cette méthode ignorent l'interaction entre les facteurs qui peuvent parfois être importante. Il nous reste le *Fractional Design* qui consiste à choisir un sous-ensemble des facteurs et de réaliser dessus un *Full Factorial Design*. Dans la partie d'évaluation, nous allons réaliser un 2^2 *Factorial Design*. On choisira 2 facteurs avec 2 niveaux chacun. En effet, cette méthode est simple à analyser avec un modèle de régression non-linéaire. Cette expérimentation portera sur la taille de la mémoire et des données, car ce sont les deux principaux facteurs.

3.3 Protocoles

Dans cette partie, nous allons définir le protocole de test. Celui-ci se découpe en deux groupes : le test de la qualité des réponses et la mesure des performances.

3.3.1 Qualité

Afin de mesurer la qualité des réponses, nous allons définir un script shell qui exécute notre système ainsi que celui de *Jena*. Les résultats seront stockés dans un dossier nommé *test-qualité* dans le répertoire *output*. Ensuite, un script python sera appelé sur les deux résultats afin de les comparer. Dans un premier temps, nous allons calculer le pourcentage des réponses identiques. Si celui-ci n'est pas de 100%, on procédera, dans un second temps, au calcul de la *complétude* et de la *correction* suivant le même protocole.

3.3.2 Mesure des performances

La mesure des performances sera un peu plus complexe que la qualité. Notre objectif est d'obtenir un graphe comparant notre système à *Jena*. On calculera le temps de réponse en faisant varier la taille des données. On pourra y voir le temps de réponse ainsi que la scalabilité des deux systèmes.

Pour obtenir ces mesures, voici le protocole utilisé. À l'aide d'un script shell, on exécutera les deux systèmes sur plusieurs *dataset*. Les résultats seront stockés dans le répertoire *output/test-performances*. Ensuite, toujours à l'aide du script shell, un script python sera utilisé sur ces résultats (Les fichiers *stats.csv*), afin de récupérer les temps d'exécution et d'afficher les graphes correspondants.

4 Évaluation des performances

Dans cette partie, nous allons entrer dans le vif du sujet, i.e. l'évaluation des performances. Nous allons notamment parler de la qualité des réponses, le temps d'évaluation ainsi que la scalabilité de notre implémentation.

4.1 Qualité des réponses

Après avoir exécuté notre protocole de contrôle de qualité, nous avons obtenu un taux de 100% de réponses identiques entre notre implémentation et celle de *Jena*. Cela veut donc dire que notre implémentation donne exactement les mêmes réponses que *Jena*. Il est donc inutile de vérifier la *complétude* et la *correction*.

À noter que ces tests de qualité ont été réaliser sur deux datasets de 1 et 2 millions de tuples. Cette expérimentation est d'ailleurs reproductible en utilisant le script *test-qualite.sh* qui se trouve dans le répertoire *src*.

4.2 Temps d'évaluation et scalabilité

En réalisant le deuxième protocole décrit plus haut, on a obtenu le graphe suivant :

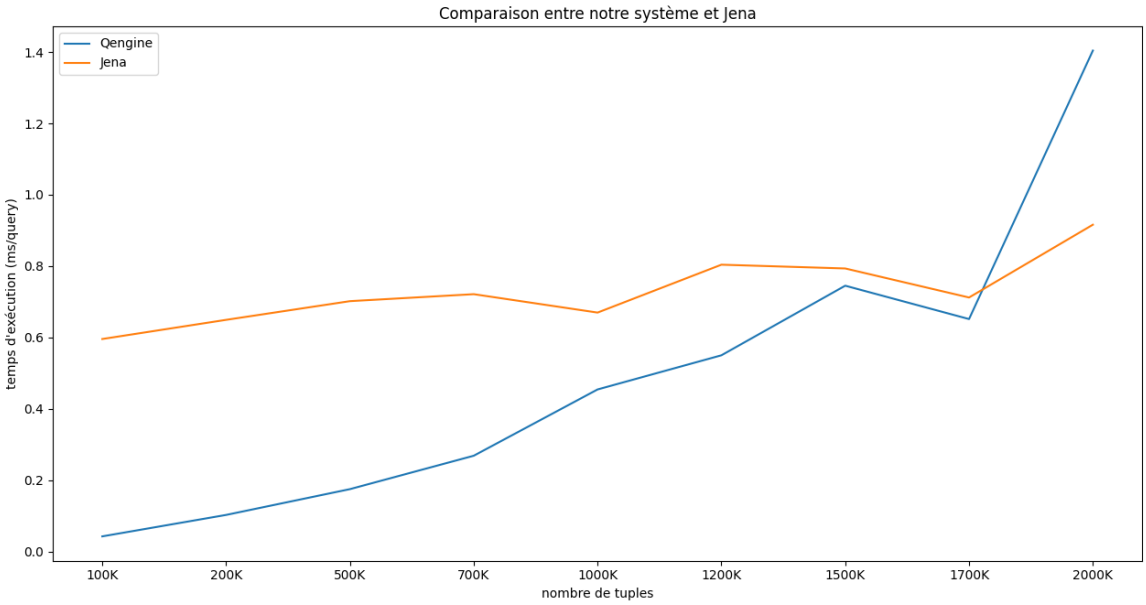

FIGURE 5 – Comparaison des deux systèmes

On remarque bien sur le graphe, que le temps d'exécution de notre système se dégrade, autrement dit le temps d'exécution augmente en variant le nombre de tuples entre $100K$ et $2000K$.

On remarque aussi que notre système est plus rapide que *Jena* quand il y a un petit nombre de tuples, ça peut s'expliquer du fait que notre système est conçu pour les requêtes en étoiles seulement, par contre *Jena* est fait pour tout type de requête *SPARQL*. On voit aussi que *Jena* est plus performant que notre système quand le nombre de tuples est de $2000K$. Le temps d'exécution de *Jena* est presque constant en variant le nombre de tuples, ce qui est tout le contraire de notre système.

4.3 Expérience sur la taille des données et de la mémoire

Nous avons, de plus, réaliser une expérience en faisant varier 2 facteurs avec 2 valeurs chacun. La taille des données (200K et 2M de tuples) ainsi que la mémoire (2Go et 4Go). Ensuite, on a calculé l'importance de ces facteurs via un modèle de régression non-linéaire suivant la formule suivante :

$$y = q_0 + q_A x_A + q_B x_B + q_{AB} x_{AB}$$

A étant le facteur de taille des données et B la mémoire. Nous avons obtenu les résultats suivants :

$$\frac{4 \times q_A^2}{4 \times (q_A^2 + q_B^2 + q_{AB}^2)} = 95\%$$

$$\frac{4 \times q_B^2}{4 \times (q_A^2 + q_B^2 + q_{AB}^2)} = 2\%$$

Ce qui indique que la taille des données est un facteur beaucoup plus influant sur le temps d'exécution. Par contre, la taille de la mémoire importe peu tant qu'elle suffit pour charger toute la base de données. Ce qui est un résultat assez logique. L'interaction entre les deux est quant à elle négligeable.

Ces résultats sont, comme les autres, reproductibles en utilisant le script *test-facteurs.sh*.

5 Conclusion

5.1 *Warm* et *Cold*

Petite précision concernant les mesures *Warm* et *Cold*. Les mesures *Warm* sont préférables pour le temps d'exécution des requêtes, car elles reflètent la réalité des DBMS. Par contre, pour les tâches réaliser qu'une seule fois, par exemple le chargement des données, les mesures *Cold* sont à préférer.

5.2 Apport des évaluations

Ces évaluations ont montré plusieurs points qui n'étaient pas évidant à première vue. On pense notamment, à la non scalabilité de notre système. En effet, les tests ont montré qu'à partir d'environ $2000k$ tuples, le système *Jena* deviens beaucoup plus performant que le nôtre. De plus, on pu démontré la correction et la complétude de notre système.

Aussi, grâce à l'expérience 2^2, on a bien vu l'importance de certains facteurs par rapport à d'autres.

5.3 Perspectives d'amélioration

Notre système traite les requêtes en étoiles seulement, on aurait voulu traiter plus de type de requête pour que notre système soit entièrement comparable à *Jena*.

On aurait pu supprimer les doublants pour peut-être avoir une meilleure évaluation, mais on a pensé qu'on aura un petit workload, donc ce qui était possible de faire, c'est de chercher encore plus loin en ce qui concerne le modèle et les templates, pour avoir un workload sans doublants et sans réponses vides.