Université de Montpellier

FACULTÉ DES SCIENCES DE MONTPELLIER

M2 IASD

Un moteur d'évaluation de requêtes en étoile

Rédigé par :

Sofian ETTAHIRI 21901227
Khadim FALL 22200674

Encadrants :

Guillame PERUTION-KHILI
Federico ULLIANA

8 décembre 2023

Table des matières

1 Introduction **3**

1.1 But du projet . . . 3

1.2 RDF . . . 3

1.3 Création d'une structure de données . . . 3

1.3.1 Création d'un dictionnaire associatif . . . 3

1.3.2 Mise en place de l'approche hexastore . . . 3

2 Création d'un dictionnaire et d'un index **4**

2.1 Dictionnaire . . . 4

2.2 Index . . . 4

3 Accès aux données **6**

3.1 Évaluation d'une requête en étoile . . . 6

3.2 Arguments . . . 7

3.3 Temps . . . 8

1 Introduction

Le projet est disponible à l'adresse suivante sur GitHub : https://github.com/khadim007/moteur_-valuation_de_requ-tes_en_-toile.

1.1 But du projet

Ce rapport expose les différentes étapes impliquées dans la réalisation d'un projet axé sur l'implémentation de l'approche hexastore pour l'interrogation des données RDF, conformément aux consignes énoncées. L'objectif central de cette initiative est de développer les procédures permettant l'évaluation des requêtes en étoile, exprimées en syntaxe SPARQL, en utilisant l'approche hexastore.

1.2 RDF

Le RDF (Resource Description Framework) est un modèle standard pour représenter les données sur le web. Il se base sur une structure triplet sujet-prédicat-objet qui décrit les informations sous forme de graphes orientés. Par exemple, l'expression "<Bob, knows, Bob>" peut être représentée par le triplet RDF "<1, 2, 1>", avec une correspondance (1, Bob), (2, knows) permettant d'associer un identifiant unique à chaque ressource de la base RDF.

1.3 Création d'une structure de données

La réalisation du projet se décompose en deux phases principales.

1.3.1 Création d'un dictionnaire associatif

Dans une première phase, il est nécessaire de créer un dictionnaire qui associe un identifiant numérique à chaque ressource de la base RDF. Cette démarche vise à obtenir un stockage plus compact des données en remplaçant chaque triplet par des valeurs numériques. Par exemple, le triplet <Bob, knows, Bob> pourrait être représenté par <1, 2, 1>, avec une correspondance (1, Bob), (2, knows). Cela permet de minimiser l'espace nécessaire pour stocker les informations tout en préservant leur signification.

1.3.2 Mise en place de l'approche hexastore

Dans une seconde phase, il convient de mettre en œuvre un index adapté au système de persistance choisi afin de permettre une évaluation efficace des requêtes. Dans le cadre de ce projet, l'approche hexastore est sélectionnée pour l'indexation des données RDF. L'approche hexastore est une méthode d'indexation spécifique qui utilise six index pour stocker les triplets RDF de manière à faciliter la recherche et la récupération des informations. Cette approche optimisée offre une structure bien définie pour une recherche rapide des données dans le contexte des requêtes SPARQL.

2 Création d'un dictionnaire et d'un index

2.1 Dictionnaire

Il est maintenant temps de nous interresser à la création d'un dictionnaire. Pour rappel les fichier RDF sont représenté sous forme de triplets (s, p, o) :

1. s subject : representant la ressource à décrire.
2. p predicat : representant un type de propriété qui peut être appliqué à la ressource.
3. o object : representant une donnée qui est la valeur de la propriété.

Nous avons donc représenter ces informations sous formes de dictionnaire en attribuant à chaque valeurs de nos triplet une clé tout en respectant l'unicté, c'est à dire que chaque ressource possède une unique clé. Dans le cas du fichier *100K.nt* nous trouvons 12225 clés.
Pour illustrer voici l'exemple de l'application de notre dictionnaire sur le fichier *sample_data.np* :

```
-----------------Dictionnaire-----------------
Clé : 0, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User0
Clé : 1, Valeur : http://schema.org/birthDate
Clé : 2, Valeur : "1988-09-24"
Clé : 3, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/userId
Clé : 4, Valeur : "9764726"
Clé : 5, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User1
Clé : 6, Valeur : "2536508"
Clé : 7, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User2
Clé : 8, Valeur : "5196173"
Clé : 9, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User3
Clé : 10, Valeur : "1995-12-23"
Clé : 11, Valeur : "2019349"
Clé : 12, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User4
Clé : 13, Valeur : "1982-07-28"
Clé : 14, Valeur : "8378922"
Clé : 15, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User5
Clé : 16, Valeur : "9279708"
Clé : 17, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User6
Clé : 18, Valeur : "4432131"
Clé : 19, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User7
Clé : 20, Valeur : "5345433"
Clé : 21, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User8
Clé : 22, Valeur : "2351480"
Clé : 23, Valeur : http://db.uwaterloo.ca/~galuc/wsdbm/User9
Clé : 24, Valeur : "8256018"
```

FIGURE 1 – Dictionnaire de *sample_data.nt*

2.2 Index

L'approche Hexastore est une technique spécifique pour stocker et interroger des données RDF. Une fois les dictionnaires crées , nous pouvons donc passé à la mise en place de notre index respectant l'approche hexastore. Cette approche consiste à créer un index pour toute les 6 permutations possible :

1. (s, p, o) *subject, predicat, object*
2. (s, o, p) *subject, object, predicat*
3. (p, s, o) *predicat, subject, object*

4. (p, o, s) *predicat, object, subject*
5. (o, p, s) *object, predicat, subject*
6. (o, s, p) *object, subject, predicat*

Cette structure de données permet d'optimiser les requêtes RDF en offrant un accès rapide à différentes combinaisons d'informations. L'Hexastore améliore les performances de certaines opérations de recherche en réduisant le nombre d'opérations de jointure nécessaires lors de l'évaluation de requêtes SPARQL.
Voici un exemple de notre index :

```
------------------spo-----------------
Clé1: 0 - Clé2: 1 - Valeurs: [2] | Clé2: 3 - Valeurs: [4] |
Clé1: 17 - Clé2: 3 - Valeurs: [18] |
Clé1: 19 - Clé2: 3 - Valeurs: [20] |
Clé1: 5 - Clé2: 3 - Valeurs: [6] |
Clé1: 21 - Clé2: 3 - Valeurs: [22] |
Clé1: 7 - Clé2: 3 - Valeurs: [8] |
Clé1: 23 - Clé2: 3 - Valeurs: [24] |
Clé1: 9 - Clé2: 1 - Valeurs: [10] | Clé2: 3 - Valeurs: [11] |
Clé1: 12 - Clé2: 1 - Valeurs: [13] | Clé2: 3 - Valeurs: [14] |
Clé1: 15 - Clé2: 3 - Valeurs: [16] |
```

FIGURE 2 – Index (s, p, o) sur le fichier *sample_data.nt*

Pour mieux comprendre notre structure voici un schéma explicatif de la première ligne :

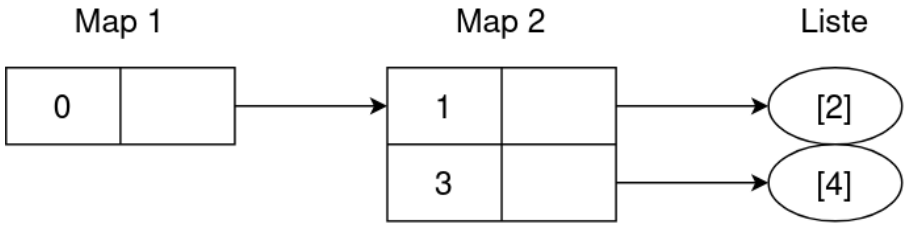

FIGURE 3 – Schéma explicatif

Pour mettre en place nos index hexastore, nous avons pris la décision de créer une structure de données sous forme de *HashMap*. Les clés de cette structure représentent la première ressource de nos triplets RDF. Chaque valeur de cette *HashMap* est elle-même une *HashMap*, où les clés correspondent à la deuxième ressource des triplets. Ce schéma implique que toutes les clés de la deuxième HashMap partagent comme première ressource celle de la clé du premier HashMap. De plus, les valeurs de cette deuxième HashMap sont des listes contenants les dernières valeurs des triplets.

3 Accès aux données

Maintenant que la mise en place de nos structure de données faite, nous pavons passer à l'accès aux données via des requêtes SPARQL. Pour mieux comprendre comment nous avons effectuer ces traitements, il faut comprendre ce que sont les patrons qui composent ces requetes. Dans SPARQL, les patrons ou patterns font référence aux clauses de requête utilisés pour définir les conditions que les données doivent respecter afin d'être sélectionnées ou filtrées. Ces patrons sont essentiels pour spécifier les motifs de recherche dans les données RDF.

Les patrons SPARQL sont principalement utilisés dans les clauses WHERE d'une requête SPARQL. Voici un exemple de requête :

```
SELECT ?v0 WHERE {
        ?v0 <http://schema.org/eligibleRegion> <http://db.uwaterloo.ca/~galuc/wsdbm/Country137> . }
```

FIGURE 4 – Première requete SPARQL du fichier *STAR_ALL_workload.queryset*

3.1 Évaluation d'une requête en étoile

Pour évaluer notre ensemble de requêtes, que nous récupérons à l'aide des méthodes *Main.parseData()* et *Main.parseQueries()* depuis un fichier, notre système va se baser sur les patrons que nous venons d'expliquer. Nous allons donc maintenir une liste de valeur possible pour la variable rechercher en analysant notre requête pattern par pattern. Voici comment cette analyse :

1. Dans un premier temps, il faut récupérer les clés correspondante aux prédicats et objets présentes dans le patron tout en vérifiant leurs existences, dans le cas contraire l'algorithme s'arrête . On va par la suite utiliser l'index (o, p, s) car c'est le sujet qui nous intéresse et que nous avons déjà les objets et prédicats dans nos patrons.
2. Nous sommes désormais en mesure de rechercher dans notre index une combinaison correspondant à notre motif. Si aucune combinaison n'est trouvée, l'algorithme s'arrête également. Pour réaliser cette recherche de motif, nous cherchons la clé de l'objet dans (o, p, s), ce qui nous dirige vers une *HashMap*. Les clés de cette *HashMap* correspondent aux prédicats, et nous recherchons celle qui correspond à la clé du prédicat du motif. La valeur de cette dernière *HashMap* représente la liste des valeurs possibles pour les sujets.
3. Une fois que nous avons des combinaisons correspondantes nous avons deux possibilités :
   (a) Premier cas : nous sommes en train d'analyser le premier pattern, nous pouvons donc ajouter les valeurs possibles des différentes combinaison pour notre variables dans notre liste
   (b) Second cas : nous sommes en train d'analyser un pattern qui n'est pas le premier de notre requêtes et il faut donc faire une intersection entre la liste des résultats déjà obtenue et l'ensemble des résultat qu'on vient d'obtenir.

À chaque évaluation de requête, nous ajoutons les valeurs obtenues dans une *Map* où la clé sera un entier correspondant au numéro de la requête. Ces resultats obtenues sont exporté dans un fichier dédiés situé dans le répertoire *data*.

Voici un exemple de résultat :

```
Pour la requete 1:
Clé :9674, Valeur :http://db.uwaterloo.ca/~galuc/wsdbm/Offer642
Clé :8881, Valeur :http://db.uwaterloo.ca/~galuc/wsdbm/Offer311
Clé :8571, Valeur :http://db.uwaterloo.ca/~galuc/wsdbm/Offer195
```

FIGURE 5 – Réponse de la requête de la figure 4

Une fois que nous avons obtenues nos résultats de notre moteur d'évaluation, il faut les vérifer. Pour cela nous allons utiliser **Jena**, qui est un framework nous permettons entre-autre d'interroger nos données RDF. On va donc lui passer les fichier RDF sur lesquels nous avons effectuer notre traitement ainsi que les requêtes a executer. Après verification on peut conclure que les résultats obtenus sont correctes pusiqu'ils correspondent à ceux de Jena.

3.2 Arguments

Une méthode efficace pour garantir la facilité et la reproductibilité des tests d'un programme consiste à le rendre exécutable via une ligne de commande spécifique. Pour notre application, la configuration de cette fonctionnalité est cruciale pour assurer une évaluation cohérente et pratique. À cette fin, les options suivantes sont intégrées pour permettre une exécution aisée du programme :

- **queries** : Cette option requiert comme valeur le chemin vers le fichier contenant les requêtes.
- **data** : Cette option requiert comme valeur le chemin vers le fichier contenant les données RDF.
- **Output** :Cette option requiert comme valeur le chemin vers le fichier où seront stockés les résultats de nos requêtes. Par défaut, nous les stockerons dans *resultat_moi.txt*.
- **Jena** : Si cette option est présente, nous allons dans un premier temps, à l'aide de Jena, exécuter les requêtes et les stocker pour ensuite les comparé à ceux obtenus par notre moteur d'évaluation. Les résulats de Jena seront également stocké par défaut dans le fichier *resultat_jena.txt*
- **warm "X"** : Cette option requiert comme valeur pour X un certain pourcentage représentant la portion des requêtes à prendre, par exemple si nous prenons 50 pour X, nous aurons alors le résultat des la moitié des requêtes. Par défaut, nous sommes à 100%.
- **shuffle** : Si cette option est présente, nous traiterons les requêtes de manière aléatoire.
- **export_query_results** : Enfin, si cette option est présente nous stockerons également le résultat de nos requêtes dans le fichier csv *query_results_moi.csv*. De plus si l'option Jena est activer le resultat des requêtes de Jena sera éagelement stocker dans le fichier csv *query_results_jena.csv*.

3.3 Temps

Un aspect essentiel de l'évaluation des performances d'un système de traitement de requêtes RDF réside dans la mesure précise et la visualisation des temps d'exécution des requêtes individuelles, ainsi que du temps total nécessaire pour évaluer l'ensemble de nos requêtes. Pour garantir une évaluation rigoureuse et une compréhension approfondie des performances du système, il est impératif de visualiser ces temps dans le terminal et de les exporter dans un fichier CSV que nous allons enrichir à chaque exécution, comme spécifié par l'option de sortie (output).

L'objectif principal est de consigner ces données de manière structurée pour permettre une analyse exhaustive. Chaque ligne du fichier CSV doit contenir des informations cruciales, telles que le nom du fichier de données, le répertoire des requêtes, le nombre de triplets RDF, le nombre de requêtes, et divers temps de lecture, de création et d'évaluation associés à l'exécution du workload.

Ces informations sont vitales pour établir des comparaisons, analyser les performances du système dans des contextes variés, et identifier les zones potentielles d'optimisation. Chaque paramètre de temps, s'il n'est pas disponible, doit être clairement indiqué comme "NON_DISPONIBLE", assurant ainsi la transparence et la clarté dans l'analyse des résultats.