Persistance de données RDF : mini-moteur

Guillaume VELAY – Badr HERRESS

7 novembre 2016

Table des matières

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 chaine 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 occurrents.

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 GraphNeighboorBis. 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 $p_1, ..., p_n$ va donc constituer une comparaison sur chaque prédicats de chaque linkedList correspondant aux prédicats $p_1, ..., p_n$. 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 linkedLink 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'ordonnancement 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 avions aussi pensez à mixer notre algorithme avec une méthode par dichotomie et selon la distribution des données, cela auraient peut être été encore plus rapide.

Nous avions 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 factoriser, vue le nombre d'objet que nous devions 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 **testsuite/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 **testsuite/queries**, il y a en a neuf, 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 **testsuite/queries_result_and_time**. Par exemple pour le fichier de requêtes Q_1_includes.queryset le programme renvoient 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.

Tests

Le dossier **testsuite/queries_result_and_time** contient déjà les tests effectués sur tous les fichiers de requêtes qui ont été interrogé sur le fichier 500K.rdf. Observer par exemple les fichiers Q_1_includes.queryset_results.csv et Q_1_includes.queryset_time.csv.

Si vous lancez le programme est que vous relancez l'exécution sur le fichier de requêtes correspondant dans **testsuite/queries** alors ces deux fichiers seront remplacés.

Nous conseillons d'utiliser le document 100K.rdf parce que sur un de nos ordinateurs personnels le temps de chargement de ce fichier est de quatre à cinq minutes, alors que pour le chargement du 500K.rdf il nous a fallu compter 2 heures.