Evaluation et optimisation d'un algorithme de traitement de requêtes agrégatives

Kamel Taouche

Université Claude Bernard Lyon 1
kamel.taouche@etu.univ-lyon1.fr

Abstract. La recherche agrégée a pour but de collecter des informations réparties dans plusieurs ressources afin de produire des réponses complètes sous formes d'objets complexes [15]. *AGASearch* [13] est un algorithme basé sur une méthode de matching de graphes permettant de réaliser des recherches agrégées sur une base de graphes. Malgré ses performances en terme de précision, *AGASearch* présente certaines limites notamment en terme de temps de réponse. Dans ce travail, nous réalisons une évaluation de performances pour *AGASearch*. Ensuite, nous introduisons des améliorations qui ont pour but de réduire le temps de réponse aux requêtes. Pour cela, nous avons utilisé un banc d'essai de référence et nous avons élaboré un jeu de requêtes. Nous avons ensuite amorcé la conception d'une approche permettant une relaxation sémantique de requêtes en s'appuyant sur des ontologies. Les expérimentations effectuées montrent l'apport des optimisations réalisées.

Abstract. Aggregated search aims at collecting information from several sources and building an answer by combining fragments of such information. *AGASearch* is an algorithm designed to compose graphs in order to perform aggregated search in the context of graph databases. From an operational point of view, *AGASearch* displays some limitations in term of query evaluation time. Our work is to conduct experiments in order to identify the bottleneck in the processing pipeline and to provide necessary modifications in order to improve the whole evaluation process. To do this, we made use of an established data set and we designed a set of appropriate queries. We also started the design of a semantic based approach to aggregate query optimization. The evaluation shows the effectiveness of our approach

Key words: Recherche agrégée, algorithme, requête agrégative, requête graphe.

1 Introduction

Avec l'émergence du Web, la gestion de données est devenue un important challenge pour les différents acteurs du Web [2]. En effet, ces données sont utilisées dans de nombreux domaines tels que la gestion des documents scientifiques [14], les réseaux sociaux [17], etc. En exploitant les richesses informationnelles offertes par ces données, plusieurs applications visant à offrir des services à l'utilisateur ont été développées. A titre d'exemple, on peut citer les moteurs de recherche tels que Google[1] et Yahoo[2].

Les techniques d'interrogation de données utilisées par la plupart des services existants retournent des résultats sous forme de liens vers des pages Web pertinentes pour chaque requête formulée par l'utilisateur. Cependant, ce type de recherche a déjà montré ses limites dans le cadre où l'information recherchée est répartie dans plusieurs sources indépendantes [12]. L'utilisateur, dans ce cas, doit alors consacrer beaucoup de temps pour examiner l'ensemble des documents retournés afin de récupérer les informations désirées.

Pour illustrer cette problématique, considérons l'exemple suivant: Imaginons une recherche sur le Web d'un acteur de cinéma ne possédant pas de site Web. Dans le cas idéal, le résultat de la recherche devra comporter des informations telle qu'une brève biographie de l'acteur, des photos, le nom de sa compagne, des informations sur ses films, des avis et commentaires de son fan club, etc. Cependant, les résultats de la requête peuvent ne pas être fournis dans une seule page, mais dans un ensemble de pages dispersées. Pour avoir une réponse à sa requête d'information, l'utilisateur doit parcourir différentes sources telles que des sites de l'actualité des célébrités, des sites spécialisés de cinéma, des moteurs de recherche d'images, etc. Cela pourrait être fastidieux. La recherche d'information agrégée est une alternative qui vise à répondre à ce type de requête. Son principe consiste à réunir toutes les informations pertinentes susceptibles de satisfaire le besoin de l'utilisateur dans un seul document. La plupart des moteurs de recherche offrent déjà ce type d'agrégation en exploitant la diversité de formats de données (image, vidéo, texte, etc.) et la nécessité de les intégrer dans la même interface. La construction d'un agrégat dans la Figure 1 semble être une meilleure solution pour synthétiser toutes les informations souhaités par

[1] http://www.google.com
[2] http://www.yahoo.com

l'utilisateur. Le résultat de la recherche contient une image, une biographie, des jaquettes de film, des informations de différents formats et de différentes provenances. Cependant, à notre connaissance, la cohérence sémantique du résultat obtenu par combinaison de fragments n'a fait l'objet d'aucune étude approfondie.

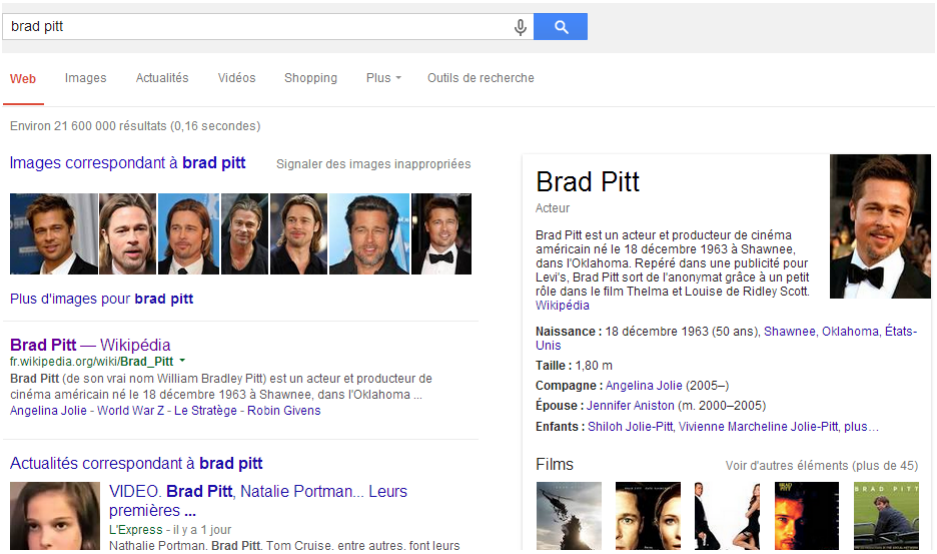


Fig. 1. Résultat du moteur de recherche Google de la requête (Brad pitt)

La recherche agrégée se définie comme *"un mécanisme permettant de chercher et assembler des informations à partir d'une variété de sources et de renvoyer le résultat dans une seule interface"* [15]. Le processus de recherche d'information agrégée est constitué de trois étapes [11] :

a) Répartition de la requête: cette étape est responsable de la sélection des sources de données pertinentes parmi l'ensemble des sources qui sont susceptibles de contenir l'information satisfaisant une partie de la requête.

b) Recherche de fragments d'information : c'est une étape qui sert à filtrer parmi la liste retenue les informations pertinentes qui peuvent être de formats divers (vidéo, image, etc.) comme par exemple une phrase dans un texte ou un paragraphe dans un livre et supprimer les informations qui sont redondantes ou qui ne sont pas utiles.

c) Agrégation des résultats : une fois l'ensemble des fragments identifié, il ne suffit pas de les placer aléatoirement dans un document mais une étape d'organisation est nécessaire. Cette étape consiste à les assembler pour construire une réponse satisfaisant un besoin d'information. Cet assemblage consiste à trouver des associations logiques entre ces fragments.

Plusieurs travaux [22][15][1] ont été proposés dans le contexte de la recherche agrégée. Parmi les approches existantes, on peut citer *AGASearch* [13] qui est l'objet de notre travail. Cette approche est une solution basée sur la collecte de fragments d'informations candidats, à partir d'une source de données. Ces fragments qui représentent des résultats intermédiaires sont soumis à une opération de validation en vue d'identifier les données les plus pertinentes par rapport à la requête. L'assemblage de ces données (agrégation) constitue le résultat final de la requête. En effet, *AGASearch* est assez efficace quand il est utilisé dans le cadre des bases de données de taille moyenne. De ce fait, l'algorithme en question doit être testé sur des bases de données de taille plus importante pour vérifier le passage à l'échelle.

Notre travail consiste, dans un premier temps, à élaborer et à mener une compagne d'évaluation afin d'analyser les goulots d'étranglement de l'approche *AGASearch* quand il s'agit de garantir le passage à l'échelle en termes de volume de données. Suite à ce travail, nous avons proposé une optimisation permettant d'élaguer, à temps, des résultats intermédiaires qui ne contribuent pas aux réponses finales. Et enfin, nous avons exploré une direction permettant la relaxation des requêtes en s'appuyant sur une ontologie. Cette approche nous permettra, en absence des réponses exactes, de retourner les résultats les plus proches (sémantiquement) de la requête.

Le reste du rapport est organisé comme suit: Dans la Section 2 nous présentons le cadre de la recherche agrégée. Dans la Sections 3 nous détaillons l'algorithme *AGASearch*, après avoir introduit quelques pré-requis. Dans la Section 4 nous procédons à un audit pour identifier les verrous liés au passage à l'échelle d'AGASearch. Nous proposons dans la Section 5 des améliorations qui nous permettront d'optimiser les temps d'exécution des requêtes. Nous rapportons les résultats issus de nos expérimentations dans la Section 6. La Section 7 conclut ce rapport en mentionnant certaines perspectives de ce travail.

2 Etat de l'art

Dans cette section, nous présentons deux états de l'art. Le premier concerne les différentes approches de la recherche agrégée et le deuxième présente les approches de reformulation de requêtes en vue de les relaxer. Ces deux domaines concernent les travaux que nous avons réalisés dans le cadre de ce stage, à savoir, l'optimisation et la relaxation des requêtes agrégatives.

2.1 Recherche agrégée

La recherche agrégée vise à chercher et à assembler des fragments d'information. Ces fragments peuvent avoir des formats (texte, image, vidéo, etc.) et des granularités (document, mot, paragraphe, etc.) différents. Le problème qui rend cette recherche agrégée complexe est principalement lié à la manière de construire le résultat final et les multiples façons de combiner les fragments pour répondre à une requête.

Les travaux consacrés à ce type de recherche peuvent être classés en deux catégories [15]: La première catégorie est appelée **agrégation de source**. Elle consiste à interroger plusieurs sources de données puis à fusionner les résultats récupérés pour les présenter à l'utilisateur sous forme de liste ordonnée selon la pertinence par rapport à un besoin d'information, c'est le principe de fonctionnement des méta-recherche [21][20]. Ce sont des moteurs de recherche interrogeant d'autres moteurs de recherche (dogpile[3], metacrawler[4], etc.) puis ils fusionnent les résultats renvoyés par ces derniers pour les présenter à l'utilisateur. Ces "méta-moteurs" combinent les avantages de plusieurs moteurs de recherche. Les premiers méta-moteurs ont été crées pour faire face à l'impossibilité d'indexer tout le Web au moyen d'un seul moteur de recherche. Le résultat d'une requête peut être obtenu par la combinaison de résultats émanant de plusieurs moteurs. Dans cette catégorie, on peut également citer la recherche agrégée interverticale [3][16] qui agrège des résultats retournés par des sources. Ces moteurs de recherche verticaux indexent un type spécifique de document (image, vidéo, texte, etc.) ou des thèmes particuliers (news, voyage, commerce, etc.). Ce sont des moteurs de recherche dédiés, bien que, relevant du moteur de recherche général et non pas de plusieurs moteurs indépendants comme dans le cas de la méta-recherche. On peut aussi citer la recherche fédérative ou l'utilisateur soumet

[3] http://www.dogpile.com
[4] http://www.zoo.com/

une requête et sélectionne un nombre de sources souvent autonomes (intranet d'entreprise, catalogue de bibliothèque, etc.). Cette recherche doit identifier la ressource la plus pertinente c'est-à-dire celle qui possède le plus grand nombre de documents pertinents dans le cas où l'utilisateur ne la définit pas explicitement. Le système basé sur ce type de recherche retourne les résultats souvent fusionnés dans une liste ordonnée ou séparés (groupés par ressource d'origine).

La deuxième catégorie, appelée **agrégation de contenus**, ne se contente pas d'afficher des résultats en réponse à une requête sous forme de liste sans relation entre celles-ci, mais vise à identifier les relations nécessaires entre les éléments pour construire cette agrégation. Cette dernière sera présentée sous forme de document fictif qui n'existe pas auparavant dans l'une des sources, ce document contenant des clusters où chaque cluster représente un résumé de documents Web retournés.

L'agrégation du contenu a été utilisée dans beaucoup de domaines de recherche: La recherche agrégée relationnelle qui s'intéresse à l'agrégation de résultat pour des requêtes de type classe. Etant donnés des instances et des attributs de la classe requête, ce travail [10] cherche à trier les attributs en fonction des instances les plus représentatives, afin de présenter les résultats dans un tableau.

L'agrégation de résultat dans les documents semi-structuré [16]. La génération de langage naturel [18] est parmi les approches qui ont abordé cet aspect d'agrégation dans la mesure où elle vise à produire des documents ou des brochures. Le résumé automatique [8] peut également être vu comme une forme d'agrégation, produisant un résultat construit à partir de plusieurs sources. Le résumé permet aux utilisateurs d'avoir une idée sur le contenu de plusieurs documents en extrayant des phrases et les reliant les unes avec les autres.

2.2 Approximation des requêtes

De nouvelles possibilités de relaxation sont apparues avec l'existence d'une ontologie au sein de données RDF. Nous présentons dans cette sous-section deux approches sur l'approximation des requêtes. La première consiste à approximer ou relaxer les requêtes sémantiques, tandis que la deuxième combine la recherche sémantique et la recherche par mots-clés.

Approches de relaxation de requêtes : Des travaux ont été réalisés sur la recherche sémantique de l'information [7][9]. Cette dernière consiste à approximer les requêtes utilisateur en se basant sur une ontologie [7], *CORESE* est un moteur de recherche sémantique, qui permet la consultation des données RDF(S) et libre à l'utilisateur d'étendre sa requête sémantiquement. En introduisant des clauses spécifiques à sa requête, l'utilisateur choisit le type d'approximation qu'il souhaite. Ces approximations ont pour finalité de repérer des chemins de relations entre deux instances. Pour exploiter des concepts proches de ceux interrogés par l'utilisateur, celui-ci peut recourir à ces approximations de différentes manières dans une même requête. Les résultats sont sélectionnés en considérant leur analogie sémantique avec la requête d'origine. La similarité sémantique entre deux concepts est l'aboutissement du calcul de la distance de similarité entre l'ensemble des paires de concepts de l'ontologie. Cette distance signifie que deux concepts sont similaires sémantiquement s'ils sont éloignés de la racine, mais ont un ancêtre proche. Les réponses peuvent aussi s'appliquer à des instances d'autres concepts ou de relations, quand ceux-ci sont liés aux concepts et relations de la requête. CORESE offre aux utilisateurs la possibilité de trouver un chemin de relations entre deux ressources RDF à longueur inférieure ou égale à un seuil spécifier dans la requête.
L'objectif premier de ces approximations est de parvenir à donner suite aux requêtes dans le cas d'indisponibilité des instances de concepts ou de relations recherchées. En l'absence d'instances de concepts, le système exploite des instances de concepts proches sémantiquement, ou permet à l'utilisateur de reformuler sa requête. L'approche développée par [9] transforme la recherche d'instances d'une relation sémantique, formulée dans une requête SPARQL, en une recherche d'instances des classes correspondant au domaine de cette relation. Ce procédé de relaxation tend à résoudre le problème de l'incomplétude des bases de connaissances, en éliminant la contrainte de relation des instances, tout en maintenant la cohérence sémantique par l'utilisation de jointures dans la requête.

Approches hybrides : Elles combinent la recherche mots-clés et la recherche sémantique pour remédier à l'incomplétude des annotations sémantiques [4]. La requête utilisateur comprend deux parties: l'une sémantique (requête SPARQL) et l'autre mots-clés. En cas de non disponibilité de réponses à l'une des parties de la requête, les réponses correspondent à la partie mots-clés seule ou à la partie sémantique uniquement. Dans l'approche avancée par [6], l'utilisateur

formule des requêtes sémantiques et les réponses affichées sont retenues suivant leur proximité sémantique avec la requête et leur degré de similarité lexicale avec les mots-clés de celle-ci. L'extraction des mots-clés de la requête se fait à travers les labels des concepts, propriétés et instances contenus dans la requête. Une opération de similarité globale est définie par combinaison des valeurs de similarité lexicale et sémantique. L'utilisateur peut également définir un coefficient qu'il associe à chacune desdites similarités en vue d'arrêter avec précision la similarité globale.

3 Pré-requis

Dans cette section, nous allons présenter quelques notions de base afin de faciliter la compréhension de la problématique abordée dans ce travail.

3.1 Web sémantique

Le Web sémantique, est une initiative menée par le W3C[5] visant à rendre les contenus sur le Web manipulables par des machines. Pour ce faire, plusieurs langages de description de données ont été conçus. A titre d'exemple, on peut citer RDF pour la description des données, RDFS ou OWL pour la définition d'ontologies. Grâce au pouvoir expressif de ces langages, des inférences peuvent être effectuées sur les différents contenus. Ainsi, des nouvelles connaissances peuvent être extraites par déduction.

Modèle RDF RDF (Resource Description Framework) est un modèle de représentation et de structuration de données développé par W3C. Un document encodé en RDF constitue un ensemble de triplets, où chaque triplet est une association <sujet, prédicat, objet>

- Le sujet représente la ressource à décrire.
- Le prédicat représente un type de propriété décrivant la ressource.
- L'objet représente une donnée ou une autre ressource. En effet, il s'agit de la valeur de la propriété.

[5] http://www.w3.org/

Le sujet et le prédicat sont nécessairement identifiés par une URI (Uniform Ressource identifier). L'objet peut ne pas l'être car il peut prendre la valeur d'un littéral. Les littéraux sont des valeurs de types primitifs, tels que les nombres, les chaînes de caractères, les dates, etc. Un document RDF peut-être considéré comme un multi-graphe labellisé décrivant les ressources et les relations qui les lient, les sujets et les objets constituent les nœuds, reliés par des arc dont leur label sont des prédicats. En pratique, les ressources sont représentées dans des ellipses et les littéraux par des rectangles. Un document RDF peut être considéré comme un graphe étiqueté, décrivant des ressources et des relations (Figure 2)

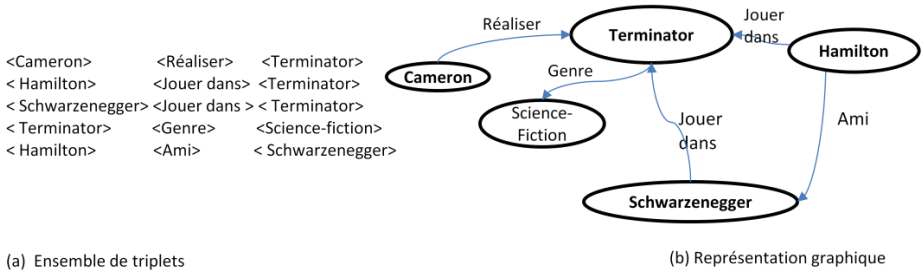


Fig. 2. Exemple RDF Dataset

Triplestores et langage SPARQL Un triplestore est un environnement conçu pour le stockage et l'interrogation de données RDF. Plusieurs triplestores existent dans la littérature. A titre d'exemple, on peut citer Sesame, Virtuso et Jena. Cette dernière est une plateforme reconnue qui fournit une API Java pour manipuler des données RDF. Pour interroger les triplets RDF d'une base, SPARQL (Sparql Protocol and RDF Query Language) [19] est le langage recommandé par le W3C. Les données d'un triplestore peuvent être considérées comme un seul document RDF. Une requête SPARQL est pour le RDF comme est le SQL pour les bases de données relationnelles. Le nom d'une variable commence par un point d'interrogation (?). La clause SELECT précise pour quelles variables on va extraire les valeurs qui vont la substituer dans le résultat. La clause WHERE compose la conjonction de conditions, sous la forme d'un ensemble de triplets.

3.2 Présentation d'*AGASearch*

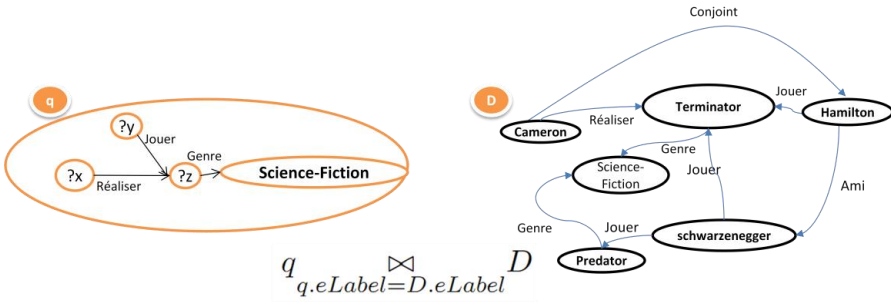
Dans cette section, nous avons repris les définitions et la terminologie utilisées dans l'approche d'*AGASearch* [13]. Dans ce contexte, les graphes étiquetés, les nœuds et les arêtes représentent des entités et des relations. Les attributs associés à des entités et à des relations sont appelées étiquettes. Plus formellement, un graphe étiqueté g est défini comme un 6-uplet $(V, E, L_v, L_e, F_v, F_e)$ où V est l'ensemble de nœuds; $E \in V \times V$ est l'ensemble des arcs qui relie deux nœuds. L_v est l'ensemble des étiquettes des nœuds; L_e est l'ensemble des étiquettes d'arcs ; F_v est une fonction $V \rightarrow L_v$ qui attribue des étiquettes aux nœuds (fonction d'étiquetage) et F_e est une fonction $E \rightarrow L_e$ qui attribue des étiquettes aux arcs. L' ensemble de nœuds et l'ensemble d'arcs d'un graphe g sont désignées par $V(g)$ et $E(g)$, respectivement. Les graphes étiquetés sont généralement classifiés, selon l'orientation ou pas de leurs arcs, en deux classes principales: les graphes étiquetés orientés comme les documents XML et RDF et les graphes étiquetés non orientés comme les réseaux sociaux.

Formalisation: ***Définition 1*** (base de données de graphes) Une base de données de graphe D est une collection de graphes de données g_i où $D = (g_1, g_2, ...g_n)$.
Définition 2 (ensemble des candidats) Un ensemble de candidat C est un ensemble de données de graphes de D qui contiennent toutes les caractéristiques figurant dans la requête q.
Définition 3 (ensemble des réponses) Un ensemble de réponses A est une collection de graphes de données qui sont isomorphes à une requête graphe q.
Définition 4 (ensemble de graphes non isomorphes) Un ensemble de graphes non isomorphe N est un ensemble de graphes de données à partir de C qui ne sont pas isomorphes à q.
Définition 5 (problème d'agrégation des graphes) étant donné q une requête et un ensemble non isomorphes $N = (g_1, g_2....g_m)$, le problème de l'agrégation de graphes requête est de trouver un sous-ensembles différent $S = (g_1, g_2....g_k)$ parmi N (tel que $k \leq m$) pour lequel l'assemblage de fragments (sous-graphes) $P_{g1}, P_{g2}.....P_{gk}$ de graphe $g_1, g_2....g_k$ respectivement , conduit à q , tel que $q = (P_{g1} \bowtie P_{g2} \bowtie ... \bowtie P_{gk})$.

AGASearch est une approche qui aborde le problème d'assemblage des frag-

ments à partir de plusieurs graphes, pour construire un résultat à la requête de l'utilisateur par une agrégation quand il n'y a pas un seul graphe satisfaisant toutes les contraintes de la requête. le défi était de trouver comment comparer d'une manière efficace entre deux graphes et comment réduire l'espace de recherche par conséquent le nombre de paire à comparer. La recherche agrégative donne un résultat exacte en combinant plusieurs graphes à condition de trouver la meilleure façon de construire cette agrégation et déterminer la participation de graphe dans celle ci. L'idée est que l'utilisateur n'a pas à spécifier une opération de jointure entre les fragments. La façon dont les fragments peuvent être combinés est un processus de découverte et doit reposer sur un algorithme spécifique. Constatant le manque d'une solution capable de combiner ces fragments pour répondre à la requête d'où la proposition de l'algorithme *AGASearch* qui se base sur un schéma relationnel pour tirer profit de la robustesse et l'efficacité de ce dernier. Il stocke les graphes sous forme d'une table relationnelle, et chaque arc est transformé en tuple avec les champs qui le distingue d'une manière unique ***(edgeID, GraphId, eLabel, svLabel, dvlabel)***. puis vient l'étape de la recherche des arêtes communes qui est le cœur de cette approche car les arêtes sont les éléments qui portent les informations les plus importantes dans le graphe et aussi pour s'assurer que toutes les arêtes (prédicats) de la requête q se trouvent dans D. Cette fonction découvre toutes les arêtes se trouvant à la fois dans la requête et dans la base de graphe. La table *(Commonedge)* est le résultat de cette opération (Figure 3). Cette table contient toutes les arêtes communes *elabel* (Prédicat) entre q et D.

Une fois cette table *Commonedge* construite vient l'étape de décomposition de la requête graphe qui consiste pour une requête donnée q à la décomposer en deux partie: une partie constante qui est l'ensemble d'arêtes reliant deux sommets étiquetés et une partie anonyme constituée d'arêtes reliant des variables. On commence par valider la partie constante si elle n'existe pas dans la table des arêtes communes (*Commonedge*) nous mettons rapidement fin au processus sinon on exécute l'algorithme pour la partie anonyme. Pour identifier les correspondances entre la requête anonyme q et l'agrégation de graphe D, l'algorithme commence par la première variable dans la requête, celle qui a le plus d'arêtes qui lui sont connectées. Nous extrayons tout les candidats possible dans la *Commonedge* ayant les mêmes arêtes que celle de la variable. Après avoir éliminer ceux qui ne sont pas valide par le biais de la vérification qui consiste: pour chaque candidat et ses nœuds adjacents étiquetés dans la requête de chercher



graphIDQ	graphID	edgeIDQ	edgeID	eLabel	svLabel	dvLabel
1	1	1	3	Réaliser	Cameron	Terminator
1	1	2	2	Jouer	Hamilton	Terminator
1	1	3	1	Genre	Terminator	Science-Fiction
1	1	2	2	Jouer	Schwarzenegger	Terminator
1	1	1	2	Jouer	Schwarzenegger	Predator
1	1	3	1	Genre	Predator	Science-Fiction

Table Commonedges

Fig. 3. Example de création du *Commonedge*

les arêtes qui les relient et de vérifier les triplets produits s'ils existent dans la *Commonedge*. Une fois la liste des candidats épurée, nous générons une nouvelle requête à partir de q pour chaque candidat en assignant sa valeur à la variable et on répète d'une manière récursive le processus pour la variable suivante jusqu'à ce que l'intégralité des nœuds de la requête soient étiquetés. La requête résultante est considérée comme une correspondance qui sera ajoutée à la réponse finale. La raison pour laquelle le traitement commence par la première variable la plus connectée est d'améliorer l'efficacité de la recherche et mettre fin à l'algorithme le plutôt possible dans le cas ou il n y a pas de réponse et aussi plus on a d'arêtes qui pointe vers un nœud plus on réduit le nombre de candidats. Sachant que pour chaque candidat il faut générer une nouvelle requête, donc moins est le nombre de candidats, moins est le nombre de requêtes générées.

Algorithme *AGASearch* :

1. En C , trouver un ensemble de sommets S (candidats de l'étiquette) qui ont des arêtes communes avec v
2. Si ne trouve aucun sommet ($S = \phi$) , terminer l'algorithme.
3. Retirer du S les candidats invalides (étape de vérification).

4. Si tous les candidats sont éliminés ($S = \phi$; après vérification). Alors terminer l'algorithme.
5. Pour chaque étiquette V_L en S

(a) générer une nouvelle requête q_v à partie de q en assignant V_L à v (ie v devient un sommet constant)
(b) aller a la prochaine variable dans AV (v = prochain sommet en AV)
(c) Si tous les sommets sont étiquetés ($v = \phi$) , une correspondance de la requête est trouvée. nous ajoutons ce mappage au résultat final. Sinon, un appel récursif est effectué en utilisant q_v et le nouveau sommet v comme entrée.
Nous clarifions la phase de vérification de l'étape 3. Dans q nous avons un ensemble de constantes $K_v = k_1, k_2, ...k_m$ qui sont voisins de v. Ensuite, nous cherchons les arcs reliant deux sommets $e_i \in v \times k_i$ dans un ensemble d'arêtes communes C. S'il existe au moins un e_i (c.-à-d e_i, v, k_i) qui ne se trouve pas dans C, nous concluons que ce candidat n'est pas valide et il doit être retiré de S.

4 Audit d'*AGASearch*

Dans cette section, nous analysons les performances d'*AGASearch* pour repérer les verrous et le goulot d'étranglement dans le but d'apporter des améliorations et d'optimiser ses performances. Le processus d'exécution de cet algorithme peut être schématisé selon la Figure 4.

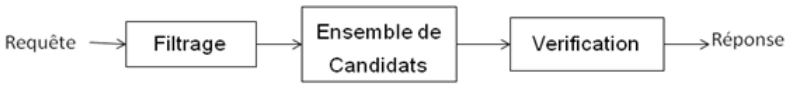


Fig. 4. Work flow du processus

4.1 Benchmark

Après avoir étudier certains benchmarks (LUBM[6],SP2[7]...) pour le RDF classés par le W3C, on a retenu le benchmark de Berlin (BSBM) [5] car il convient à notre besoin qui consiste à évaluer la performance de l'approches. Le choix de

[6] http://swat.cse.lehigh.edu/projects/lubm/
[7] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B

ce benchmark BSBM est motivé aussi par le fait que son générateur de données nous offre la possibilité de contrôler le volume de données en spécifiant un paramètre en entrée qui est le nombre de produits sachant que la réalisation du BSBM s'inspire des cas d'e-commerce où un ensemble de produits est offert par différents fournisseurs, sur lesquels différents consommateurs ont posté des avis. Chaque produit contient un nombre varié de caractéristiques. Le benchmark de Berlin n'a pas été conçu pour des besoins de raisonnement complexe, mais juste pour mesurer la performance des requêtes par rapport à de grandes quantités de données RDF. Ici, notre objectif est d'évaluer le gain de temps possible apporté par nos éventuelles améliorations d'*AGASearch* ce qui nous permettra de les valider. Le jeu de requêtes proposé par ce benchmark contient des clauses (fonction regex, OPTIONAL) qui ne sont pas supportées actuellement par *AGASearch*, ce qui nous à mener à concevoir notre propre jeu de requêtes adapté à notre expérimentation.
Les jeux de données utilisés durant nos expérimentations ont été obtenus grâce au générateur de données fourni, configuré pour générer des fichiers au format CSV, grâce a son paramètre (nombre de produits), on contrôle la taille du jeu de données. Les requêtes utilisées sont des requêtes interrogatives où nous avons fait varier le nombre de variables et de constantes de 1 à 4 ce qui nous a donné un jeu de 16 requêtes ($Q_{i,j}$ avec i :nombre de variables et j: nombre de constantes), Cette expérimentation a été menée sur une machine avec 8 Go de RAM et processeur cadencé à $3.2Ghz$.

4.2 Evaluation des performances

Après un audit de l'algorithme et de son implémentation, nous avons considéré les mesures suivantes: (temps de réponse total, taille de la *Commonedge*, nombre de requêtes de vérification, temps de vérification et nombre de vérifications négatives, temps de la création de la *Commonedge*). L'ensemble des résultat est porté dans la Figure 5.

4.3 Interprétation des résultats

Les résultats de l'expérimentation ont montré que le temps de réponse et la taille de résultat est étroitement liée au nombre de constantes, tel qu'il est illustré dans la Figure 6. En effet, plus le nombre de constantes est petit, plus la recherche agrégative est ardue. Cela dû à l'augmentation importante du nombre de candidats en absence de filtre. Par ailleurs, plus on a de variables dans la requête plus

Requête	Taille CommonEdge	Temps réponse	Temps vérif.	Rapport temps vérif.	# Vérif.	#Verif non valide	% Verif non valide
Q1,1	100	2,42	0,10	4%	100	0	0%
Q1,2	111	1,78	0,18	10%	200	89	44%
Q1,3	211	2,08	0,26	12%	300	89	29%
Q1,4	217	2,16	0,35	16%	400	183	45%
Q2,1	10 879	388,71	292,38	75%	12 879	12 579	97%
Q2,2	10 880	481,64	428,01	88%	21 380	21 277	99%
Q2,3	10 886	792,38	728,95	91%	32 059	31 950	99%
Q2,4	11 086	626,54	573,29	91%	32 170	31 951	99%
Q3,1	10 979	398,54	291,71	73%	13 079	12 579	96%
Q3,2	10 985	549,11	485,96	88%	21 502	21 372	99%
Q3,3	11 085	808,67	739,78	91%	32 181	31 951	99%
Q3,4	11 086	1 065,16	1 000,27	93%	42 740	42 529	99%
Q4,1	13 290	5 974,79	5 075,40	84%	162 879	157 757	96%
Q4,2	13 291	698,15	628,96	90%	22 880	22 745	99%
Q4,3	13 292	705,66	640,59	90%	22 856	22 739	99%
Q4,4	13 293	995,51	929,03	93%	44 238	44 101	99%

Fig. 5. Tableau: des mesures de l'audit

le temps de réponse augmente, car pour chaque variable, l'algorithme récupère tous ses candidats dont le nombre varie en fonction du domaine de cette variable. La phase de vérification et la génération de requêtes devient alors coûteuse. La vérification de toutes les combinaisons possibles pour ne retenir finalement que celles qui existent dans la table *Commonedge*, nécessitera un parcours de tous les candidats d'une variable pour les comparer avec l'ensemble des candidats des autres variables voisines (produit cartésien).

De cela, on remarque que le temps d'exécution est étroitement lié au nombre de variables dans la requête et le nombre de candidats pour chaque variables. Prenant le cas de la $Q_{4,1}$, qui est la requête la plus coûteuse du fait qu'elle contient le plus de variables et le moins de constantes, ajouté à cela, le fait que l'une de ses variable $?PF$ possède un nombre important de candidats. Les candidats ne seront validés qu'après une vérification. Cependant cette phase de vérification est très coûteuse en matière de temps d'exécution. En effet, elle occupe presque 70% du temps de réponse (Figure 7), car pour chaque candidat d'une variable, l'algorithme envoi une requête SQL à la base de données pour vérifier si le triplet (candidat, prédicat, constante adjacente) existe dans la table *Commonedge*. Il

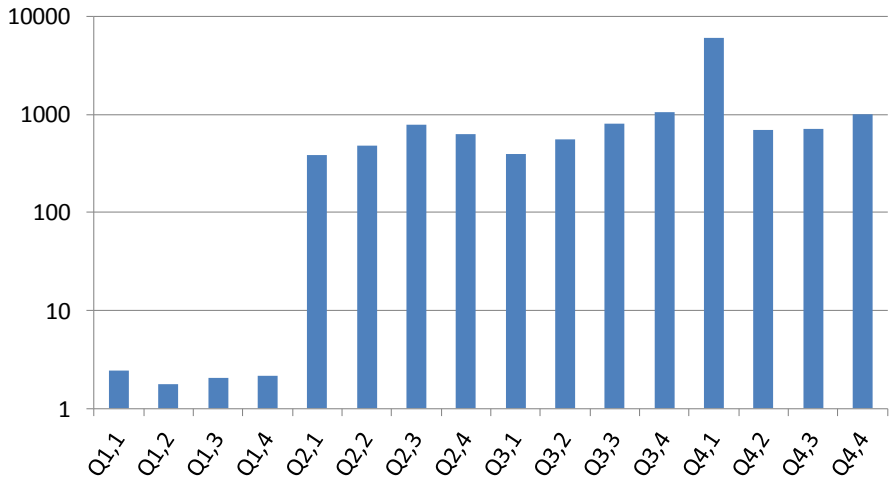


Fig. 6. Impact du nombre de constantes et variables sur le temps de réponse

est donc clair que le nombre d'accès à la base dépendra du nombre de candidats pour une variable donnée et influencera considérablement sur le temps de traitement surtout dans le cas où la taille de la table interrogée est importante.

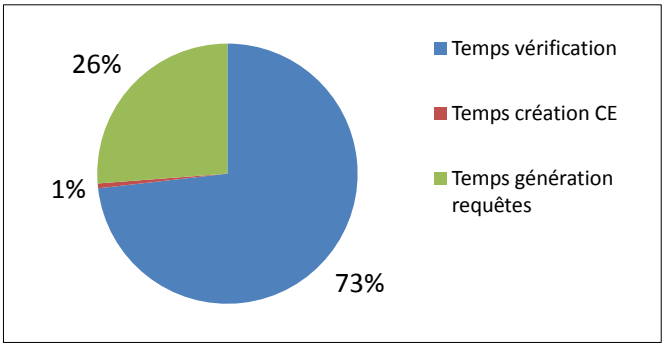


Fig. 7. Coût de chaque phase du processus

Après analyse il s'est avéré que la majorité des requêtes de vérification retournaient des réponses négatives en raison du fait que la majorité des candidats (90%) ne sont pas valides (voir Figure 8).
La construction de la *Commonedge* est le résultat d'une union de requêtes de

jointure entre q et G de façon à ce que toutes les combinaisons possibles des champs de jointure soient utilisés, ce qui fait qu'elle contient un nombre beaucoup plus élevé de tuples que ceux réellement nécessaires.

$$\sigma_{s=?}(Q \underset{\text{p,o}}{\bowtie} G) \bigcup \sigma_{o=?}(Q \underset{\text{p,s}}{\bowtie} G) \bigcup (Q \underset{\text{s,p,o}}{\bowtie} G) \bigcup \sigma_{o=? \wedge s=?}(Q \bowtie G) \qquad (1)$$

Nous avons ainsi porté notre réflexion sur la manière de réduire le nombre de requêtes SQL générées nécessaires pour la vérification de la validité des candidats, à défaut de ne pas pouvoir filtrer les résultat lors de la création de la *Commonedge*.

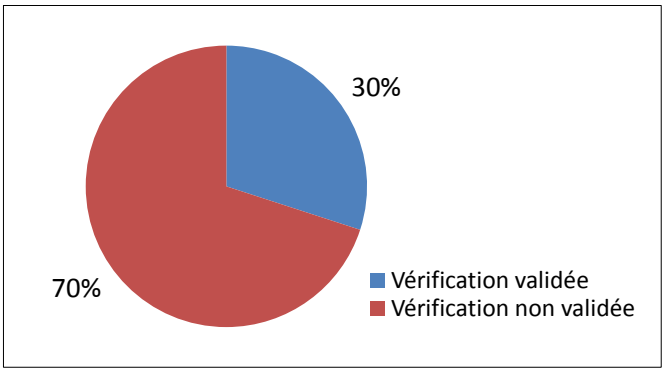


Fig. 8. Rapport du résultat de la vérification

5 Approche proposée

L'évaluation de cet algorithme a révélé que les temps d'exécution sont très importants, ce qui limite particulièrement son utilisation. La nature naïve de cet algorithme qui consiste à récupérer des candidats pour chaque variable de la requête. Ce qui est particulièrement coûteux dans le cas d'un jeu de données de plusieurs millions de triplets. En plus s'il possède un domaine de définition de variables large (toutes les valeurs qu'on peut assigner aux variables). Les expérimentations effectuées ont montré que l'étape de vérification occupait une partie très considérable du temps d'exécution de la requête, car pour chaque candidat d'une variable, l'algorithme envoie une requête SQL à la base de données pour

vérifier si le triplet qui est composé du candidat de la variable est lié à une constante existe dans la *Commonedge*.

Pour remédier à ce problème, nous avons en premier lieu appliqué un filtre (ajout d'une contrainte) pour réduire le nombre des candidats lors de l'extraction. Cela revient à ramener les candidats ayant des arêtes communes avec la variable cible, et qui possèdent des nœuds adjacents inclus dans l'ensemble des candidats déjà validés pour la variable précédente dans la requête q. Cette opération réduit le nombre de candidats et par conséquent le nombre des requêtes SQL nécessaires à la vérification de ces candidats. Cependant, cette solution nécessite toujours la phase de vérification pour la validation des candidats. En effet , bien que le nombre des candidats est réduit, le coût de vérification reste important. Le fait que ce dernier soit encore considérable pour certaines requêtes nous a mené à essayer d'optimiser encore cet algorithme. Pour cela, en deuxième lieu, nous avons consolidé cette solution en éliminant toute la phase de vérification dans la mesure où les candidats ramenés était nécessairement valides pour une variable donnée.

Par ailleurs, pour améliorer la qualité des résultats, nous avons recouru à la technique de relaxation des requêtes [9] en utilisant une ontologie. Cette technique permet à la demande de l'utilisateur d'élargir le résultat de la requête notamment dans le cas d'indisponibilité d'une réponse.

Ces optimisations apportées ont donné lieu à un nouveau algorithme qu'on a baptisé *AGASearch+*.

5.1 Présentation de l'algorithme optimisé *AGASearch+*:

Notre couvre deux aspects, le premier a pour but de réduire le nombre de candidats qui seront par la suite en majorité invalidés après la phase de vérification, et le deuxième aspect tente d'étendre la requête sémantiquement pour élargir le champs de recherche.

5.2 Aspect agrégatif

Comme constaté lors des expérimentations que les temps d'exécutions était considérable, cela dû au nombre élevé des candidats (résultats intermédiaires) que lors de la phase de vérification 70% étaient éliminés car ils ne s'avérés pas valides,

cela nous à mener à réfléchir sur la manière de réduire le nombre de candidats et par conséquent le temps nécessaire pour les vérifier.

Lors de l'extraction de candidats à partir de la *Commonedge*, nous avons décidé d'inclure dans la requête SQL responsable de cette extraction une contrainte qui consiste à prendre en compte les constantes adjacentes de la variable cible. Ces constantes peuvent êtres des littéraux de la requête ou des instanciations d'une variable par ses candidats lors de la phase de génération de requêtes (phase4), les constantes jouant le rôle de filtre, éliminent une grande partie de candidats pour ne garder que ceux qui matchent avec les nœuds voisins. Cela nous mènera à dire que le fait de ne récupérer que des candidats valides rend la phase de vérification inutile.

Tout comme dans la version initiale, l'algorithme commence à partir de la première variable v en AV. Nous extrayons, par la suite, les candidats possibles (c'est à dire l'ensemble de S) à partir de C ayant les mêmes arêtes avec v. Cette fois ci on inclue les sommets voisins contenant des constantes. Ces dernières serviront de filtre à la requête d'extraction des candidats. Dans le cas où on a plusieurs constantes voisines à la variable cible (voir Figure 9) on utilise l'opérateur d'**intersection**. Cela permettra de faire en sorte que la requête ne retourne que les candidats vérifiant l'ensemble des sous-requête. Plus explicitement, on ramène que les candidats qui ont un lien avec la constante dans la *Commonedge*, cette dernière servira de filtre pour la requête d'extraction et contribuera à la réduction des réponses partielles, *AGASearch* résulte un produit cartésien pour effectuer toutes les vérifications de validité des candidats. Considérons le triple pattern ($<?x >< e ><?y >$) où x, y sont des variables x a n candidats et y a m candidats, on obtiendra $n \times m$ requêtes de vérification dans *AGASearch*. Cependant dans l'approche optimisé, pour sélectionner la variable à partir de laquelle on commence, on doit d'abord récupérer le nombre de chaque candidats pour ces deux variables et on commence par celle qui en possède le moins. Par ailleur plus est petit le nombre de candidats, moins on a de requêtes à générer (duplication) et moins on aura de requêtes pour extraire les candidats de la variable adjacente.

Exemple: Considérons la requête suivante (?offre :concerne ?produit), si la variable ?offre possède 1000 candidats et ?produit possède 100 candidats. Le fait

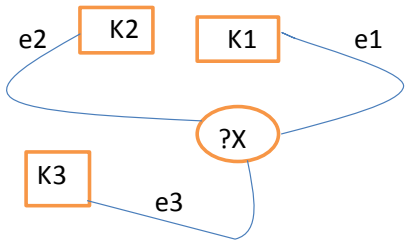


Fig. 9. Exemple de requête en étoile entourée de constantes

de commencer par la variable ?produit, nous génère 100 requêtes à exécuter pour étiqueter la variable ?offre. Par contre en commençant par la variable ?offre, 1000 requêtes seront nécessaires pour étiqueter la variable ?produit. De même pour la duplication des requêtes, quand on commence à partir de la variable possédant le moins de candidats, on générera moins de requêtes. Dans cet exemple on voit bien que le choix de la variable d'initialisation de l'algorithme impacte sur les performances et le temps de réponse.

Dans la nouvelle approche l'étape de vérification n'est plus nécessaire compte tenu du fait que, au moment de l'extraction, on ramène que des candidats déjà valides, sachant que c'est l'étape la plus couteuse du processus. Pour chaque candidat dans S, nous générons une nouvelle requête q_v en assignant ce candidat à v, puis nous passons à la prochaine variable dans AV en poursuivant de manière récursive le processus d'identification de sommets toute en appliquant le filtre qui réduit le nombre de candidats d'une manière très significative. Ce filtre permet de ne ramener que les candidats appropriés. Lorsque la requête est totalement étiquetée, elle est considérée comme une réponse et sera ajoutée au résultat final.

5.3 Aspect sémantique (approximatif)

Après avoir optimisé l'*AGASearch* et diminué le temps d'exécution de requête, On veut approximer maintenant ces requêtes utilisateur en s'appuyant sur une ontologie. Cette approche permet l'interrogation de données en offrant une technique d'approximation à la demande de l'utilisateur, lorsque l'exactitude des résultats n'est pas exigée. Cette approche fonctionne en insérant des clauses spécifiques à la requête en pointant le motif de la requête qu'on désire relaxer. Ces approximations consistent à rechercher des synonymes pour une constante en utilisant des concepts proches de ceux interrogés par l'utilisateur. Dans la

table synonyme on attribue un score pour chaque mot. Ce score définit la similarité avec les deux concepts et qui servira de trier les réponses par la suite.
L'objectif principal de ces approximations est de permettre de répondre aux requêtes même en cas d'absence d'instances recherchées. Le système retournera alors des instances proches sémantiquement, tout en permettant à l'utilisateur de formuler sa requête.
Cette méthode a pour but d'améliorer, en terme de pertinence, les résultats retournés lors de l'interrogation de la base. Un premier prototype de l'approche a été mis en place et a montré des bons résultats.

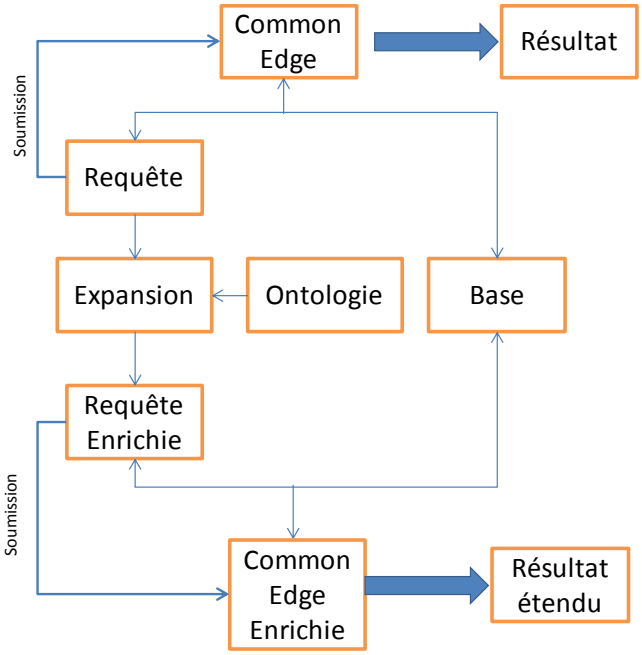


Fig. 10. Approche d'approximation

Une première requête non relaxée est soumise à l'algorithme afin d'extraire des données. Aucun résultat n'est retourné, une table Synonyme est ajoutée qui fait office d'ontologie. Cette table contient un mot et ses synonymes ainsi qu'un attribut *Score* contenant la valeur de similarité sémantique entre le mot et ses synonymes. La valeur de cet attribut servira par la suite à trier les résultats. La

même requête, relaxée cette fois, par l'insertion d'une clause spécifique devant le motif qu'on désire relaxer (à la demande de l'utilisateur), permettra, à l'aide de règles d'expansion de requêtes basées sur cette ontologie (table de synonymes), d'obtenir un résultat, malgré l'inexistence de résultat direct correspondant syntaxiquement à la requête d'origine. Cette technique permet ainsi d'obtenir plus de résultats reliés par des relations sémantiques correspondant davantage aux attentes de l'utilisateur.

Nous avons aussi utilisé une autre approche cette, fois-ci, elle s'appuie sur une ontologie (productType qui est une hiérarchisation de type produit) des produits qui ont le même type; si par exemple une requête de la forme (?offre concerne Produit1) on va s'étendre sur tous les produits qui ont le même type que le produit1 et on a ajouté un parametre qui accompagne la clause spécifique insérée dans la requête pour lui spécifier le niveau d'hiérarchie à atteindre sachant qu'on a une arborescence du genre (ProduitType2 subClassof ProduitType1).

6 Expérimentations

Nous avons mené une expérimentation pour évaluer nos contributions et propos. Pour cela nous avons utilisé le jeu de données et la charge de requêtes définit dans la Section 4.1.

Cette expérimentation consiste à comparer les deux algorithmes (AGA et AGA+) sur une taille fixe de jeu de données et en appliquant la charge de requêtes obtenue en faisant varier le nombre de variables et de constantes de 1 à 4. L'ensemble des résultats est porté dans la Figure 11 et visualisé graphiquement dans la Figure 12.

Le temps de réponse obtenu en appliquant cette méthode a beaucoup diminué par rapport aux expériences précédentes, à l'exception des requêtes avec une seule variable ($Q_{1,2}, Q_{1,3}, Q_{1,4}$). Cela est dû au coût de l'operateur d'intersection, le fait que, la variable est entourée de constantes, donnera lieu à une extraction de ses candidats par le biais de l'intersection des résultats des sous-requêtes.

En effet, la nouvelle approche consiste ainsi à extraire les candidats dont le nœud adjacent est égal à la constante voisine de la variable cible (instanciation en cours ou un littéral) dans la requête. Ainsi, s'il y a plus d'une constante adjacente à la variable, on construit une intersection de l'ensemble de sous-requêtes

		AGA				AGA+			
		#Variables				#Variables			
		1	2	3	4	1	2	3	4
#Constantes	1	2,42	388,71	398,54	5 974,79	2,37	56,39	214,17	241,55
	2	1,78	481,64	549,11	698,15	4,66	2,60	14,76	6,37
	3	2,08	792,38	808,67	705,66	9,73	3,70	19,14	7,95
	4	2,16	626,54	1 065,16	995,51	15,01	5,47	5,50	8,01

Fig. 11. Comparaison du temps de réponse entre les deux approches.

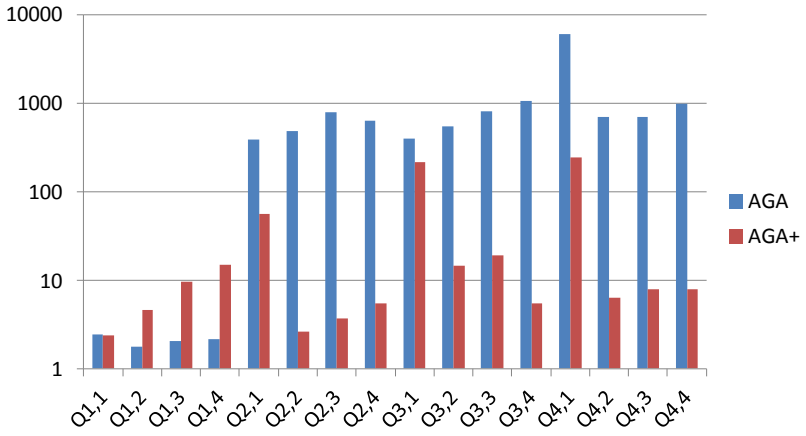


Fig. 12. Evaluation des performances d'AGASearch+

pour ne retenir que les candidats renvoyés par la totalité des sous-requêtes.
En augmentant le nombre de constantes de la requête on fait augmenter automatiquement le nombre de prédicats ce qui va nécessairement mener à une augmentation de la taille du *Commonedge*. Cette opération impliquera un nombre plus important de candidats pour la variable. Ajouté à cela, le fait que les constantes ajoutées ne sont pas filtrantes, le nombre de candidat ne sera pas moins élevé par rapport à la requête qui contenait pourtant moins de constantes. Ce qui engendre une augmentation du temps d'exécution, contrairement à ce qui était attendu. il s'est avéré aussi que la requête avec l'operateur ensembliste ($\cap$) est relativement couteuse, cela dû à la composition du résultat de chaque requête membre de l'opérateur ensembliste. Bien que le nombre de candidats est toutefois réduit, on constate que le temps de réponse global est élevé, surtout pour certaines re-

quêtes sous forme d'étoile (variable entourée de plusieurs constantes) reste élevé.

7 Conclusion et perspectives

Dans le cadre de ce stage, nous nous sommes intéressés à la recherche agrégée, dédiée aux bases de données de types graphes. Plus particulièrement, nous cherchons à optimiser l'algorithme *AGASearch* pour la recherche agrégée. Nous avons procédé à une évaluation de performances. Cela nous a permis d'identifier les opérations coûteuses utilisées dans le processus qui implémente *AGASearch*.
Les expérimentations que nous avons menées nous ont permis d'identifier les opérations coûteuses du processus d'évaluation de requêtes. Nous avons alors simplifier le processus en éliminant les étapes qui réalisent des calculs redondants ou sans effet sur le résultat final. Nous avons également travaillé sur les requêtes en ajoutant des ressources externes permettant d'élargir le champ de recherche. Ce travail n'est qu'à ses débuts et devrait se poursuivre dans un contexte plus général de conception d'un cadre qui allie agrégation, enrichissement sémantique et relaxation de requêtes.

References

1. *SIGIR '09: Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 2009. ACM. 606091.
2. S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart. *Web Data Management*. Cambridge University Press, 2012. Open access of the full text on the Web.
3. J. Arguello, F. Diaz, J. Callan, and J.-F. Crespo. Sources of evidence for vertical selection. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 315–322, New York, NY, USA, 2009. ACM.
4. R. Bhagdev, S. Chapman, F. Ciravegna, V. Lanfranchi, and D. Petrelli. Hybrid search: Effectively combining keywords and semantic searches. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 554–568, Berlin, Heidelberg, 2008. Springer-Verlag.
5. C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

6. P. Castells, M. Fernandez, and D. Vallet. An adaptation of the vector-space model for ontology-based information retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):261–272, 2007.
7. O. Corby, R. Dieng-Kuntz, F. Gandon, and C. Faron-Zucker. Searching the semantic web: approximate query processing based on ontologies. *Intelligent Systems, IEEE*, 21(1):20–27, Jan 2006.
8. H. T. Dang. Overview of duc 2006 (draft).
9. C. Hurtado, A. Poulovassilis, and P. Wood. A relaxed approach to rdf querying. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *The Semantic Web - ISWC 2006*, volume 4273 of *Lecture Notes in Computer Science*, pages 314–328. Springer Berlin Heidelberg, 2006.
10. A. Kopliku, M. Boughanem, and K. P. Sauvagnat. Towards a framework for attribute retrieval. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 515–524, New York, NY, USA, 2011. ACM.
11. A. Kopliku, K. Pinel-Sauvagnat, and M. Boughanem. Aggregated search: A new information retrieval paradigm. *ACM Comput. Surv.*, 46(3):41:1–41:31, Jan. 2014.
12. M. Lalmas. Aggregated search. 33:109–123, 2011.
13. T.-H. Le, H. Elghazel, and M.-S. Hacid. A relational-based approach for aggregated search in graph databases. In S.-g. Lee, Z. Peng, X. Zhou, Y.-S. Moon, R. Unland, and J. Yoo, editors, *Database Systems for Advanced Applications*, volume 7238 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin Heidelberg, 2012.
14. M. S. Mayernik. Metadata tensions: A case study of library principles vs. everyday scientific data practices. In *Proceedings of the 73rd ASIS&T Annual Meeting on Navigating Streams in an Information Ecosystem - Volume 47*, ASIS&T '10, pages 116:1–116:2, Silver Springs, MD, USA, 2010. American Society for Information Science.
15. V. Murdock and M. Lalmas. Workshop on aggregated search. *SIGIR Forum*, 42(2):80–83, Nov. 2008.
16. N. Naffakhi, M. Boughanem, and R. Faiz. Recherche d'information agrégée dans des documents xml basée sur les réseaux bayésiens. In *EGC*, pages 369–380, 2012.
17. A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 173–187, Washington, DC, USA, 2009. IEEE Computer Society.
18. C. Paris, S. Wan, and P. Thomas. Focused and aggregated search: a perspective from natural language generation. *Information Retrieval*, 13(5):434–459, 2010.
19. E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF, W3C recommendation. Technical report, World Wide Web Consortium, January 2008. http://www.w3.org/TR/rdf-sparql-query/.

20. E. Selberg and O. Etzioni. Multi-service search and comparison using the metacrawler. In *In Proceedings of the 4th International World Wide Web Conference*, pages 195–208, 1995.
21. K. Srinivas, P. V. S. Srinivas, and A. Govardhan. A survey on the performance evaluation of various meta search engines.
22. S. Sushmita, H. Joho, and M. Lalmas. A task-based evaluation of an aggregated search interface. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, SPIRE '09, pages 322–333, Berlin, Heidelberg, 2009. Springer-Verlag.