Université Abou Bekr Belkaid جامعة أبي بكر بلقايد
تلمسان الجزائر

République Algérienne Démocratique et Populaire
Université Abou Bakr Belkaid– Tlemcen
Faculté des Sciences
Département d'Informatique

Mémoire de fin d'études

pour l'obtention du diplôme de Master en Informatique

Option: *Système d'Information et de Connaissances (S.I.C)*

Thème

Comparaison et Optimisation d'outils d'indexation et de recherche plein texte dans des sources RDF, basés sur le langage SPARQL

Réalisé par :

- **Mr. Taleb Tariq**
- **Mr. Settouti Ahmed Khalid Yassine**

Présenté le 01 Juillet 2012 devant la commission composé de MM.

Président : - *Benammar A.*

Encadreurs : - *Midouni S. D.* / - *Settouti S.L.*

Examinateurs : - *Khitri S.* / *Halfaoui A.*

Année universitaire : 2011-2012

Comparaison et optimisation d'outils d'indexation et de recherche plein texte dans des sources RDF, basés sur le langage SPARQL

Tariq TALEB, Ahmed Khaled Yassine SETTOUTI

4 juillet 2012

"Seul le travail peut nous consoler d'être nés"

Miguel de Unamino 1913

Ecrit en LaTeX 2_ε

Remerciement

Merci mon Dieu de nous avoir donné la volonté de travailler et le pouvoir de raisonner pour achever ce mémoire.

Un grand merci à nos parents de nous avoir beaucoup soutenu.

En témoignage de nos profonds sentiments de respect et de remerciements, nous tenons à exprimer nos sincères gratitudes à nos encadreurs Mr. MIDOUNI Djallal et Mr. SETTOUTI Lotfi pour leur soutien, leurs conseils et leur bienveillance durant l'élaboration de ce projet.

Nous remercions Mr. Benammar qui nous a fait l'honneur de présider le jury, Mme. Khitri d'avoir accepter d'évaluer ce travail et Mme. Halfaoui d'avoir accepter de le juger.

Nous tenons à adresser nos remerciements à tous les enseignants du département d'informatique et du département des Mathématiques de l'université de Tlemcen. Nous remercions toutes les personnes qui ont contribué de près ou de loin à l'accomplissement de ce travail.

Dedicaces (Taleb Tariq)

Je dédie ce mémoire

A

Mon très cher père et ma très chère mère

En témoignage de ma reconnaissance envers, les sacrifices et tous les efforts qu'ils ont fait pour mon éducation ainsi que ma formation

A

Mon cher frère et ma chère soeur

A

mes amis, mes collègues, et à toute la promotion de master 2 Informatique

Ainsi

Je dédie ce mémoire à tous les futurs chercheurs voulant continuer l'étude proposée dans ce domaine.

Dedicaces (Settouti Ahmed Khalid Yassine)

C'est avec une très grande joie que je dédie ce mémoire à tous les futurs chercheurs voulant continuer l'étude proposée dans ce domaine.

A ma grand-mère et à ma mère qui ont tant sacrifié pour mes études.

A mon oncle et mes tantes pour leurs conseils très utiles.

A nos amis, nos collègues, et à toute la promotion d'Informatique.

Résumé

Depuis l'apparition du web sémantique, les gens se sont pressés à l'utiliser. Du coup, des données RDF sont presque partout et en grand volume. En plus, les gens veulent aussi la possibilité de chercher dans ces graphes de la même façon que de chercher dans un document texte.

Notre objectif est d'appliquer une recherche plein texte dans une requête SPARQL. On a choisi Jena et Sesame vu qu'ils possèdent déjà un outil de recherche en texte intégral. On leur a ajouté une fonctionnalité semblable puis on l'a comparé avec l'outil existant. Les résultats étaient très satisfaisants car la nouvelle approche dans ces moteurs a démontré une rapidité dans la plupart des cas, en plus d'autres critères qui sont importants pour les utilisateurs.

Cette approche ouvre une porte sur d'autres études comparatives plus larges couvrant plus de plateformes que Jena et Sesame. Notre approche se déroule uniquement sur Lucene. Pour celà, la seule condition qu'un moteur doit vérifier est le support du langage SPARQL.

Mots-clés : SPARQL, recherche plein-texte, RDF, Jena, Sesame, Lucene, Fonction Filtre.

Glossaire

API Application Programming Interface.

CSV Comma-separated values.

HTTP HyperText Transfer Protocol.

IDF Inverted Document Frequency.

JSON JavaScript Object Notation.

OWL Ontology Web Language.

RDF Ressource description framework.

RDFS RDF Schema.

RDQL RDF Data Query Language.

SeRQL Sesame RDF query language.

SIREn Semantic Information Retrieval Engine.

SPARQL SPARQL Protocol and RDF Query Language.

STAFF Sparql - based full-Text search Add Filter Function.

SQL Structured Query Language.

TF Term Frequency.

URI Uniform Resource Identifier.

W3C World Wide Web Consortium.

XML Extensible Markup Language.

YARS Yet Another RDF Store.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Après l'explosion du web, les bases de données et les bases documentaires ne cessaient d'accroitre en termes de volume et de complexité ; ce qui ne représentait pas une tâche facile pour les interroger. Certaines de ces sources se présentent comme des données RDF, et beaucoup de normes que ce soit langages ou plateformes ont été mises en place pour aider à interroger d'une manière fiable ces sources de données. Ces normes fonctionnent avec le principe d'appariement par patterns, et l'introduction de la recherche plein texte dans ces derniers n'a pas été chose facile.

Jusqu'à l'instant les plateformes indexent d'abord les données et les interrogent ensuite. Beaucoup de moyens s'offrent dans ce domaine comme le serveur d'indexation full-text d'apache nommé *Lucene*. Pour choisir une norme par rapport à une autre, il faut réfléchir longuement parce que ceci ne dépend pas que d'un seul critère. Il n'y a pas que la rapidité ou la robustesse qui comptent mais aussi l'interopérabilité et l'expressivité des requêtes.

Il a eu beaucoup de travaux comparatifs ces dix dernières années entre les différentes plateformes mais qui ne se concentraient pas sur l'aspect plein texte de la recherche et qui se contentaient de comparer qu'avec des requêtes simples.

Notre travail consiste à prendre deux plateformes les plus utilisées, les comparer mais étendre ensuite cette comparaison pour répondre à la question suivante :

Jusqu'à quel point pouvons-nous aller afin de garantir une recherche plein texte rapide avec une plateforme d'interrogation de graphes RDF basée sur le langage SPARQL ?

Notre travail servira sûrement à connaitre à quel point pouvons nous pousser la rapidité à bout d'une plateforme d'interrogation de graphes RDF en se basant sur le langage SPARQL.

La suite de notre document se présente en trois chapitres :

- Le premier chapitre décrit les différents outils qu'on a utilisé de près ou de loin durant notre recherche, ainsi que l'ensemble des travaux réalisés dans ce domaine.

- Le deuxième chapitre démontre les différents critères des moteurs choisis, ainsi qu'une étude comparative entre eux.
- Le troisième chapitre présente la conception et l'implémentation de notre système et ses avantages.

I *Présentation d'outils et notions de base*

I.1 Introduction

Il existe partout dans le monde, différentes informations, sous différents formats, avec différents niveaux de structuration. Pour gérer chaque type d'information, il faut maitriser un outil spécifique. Pour gérer une base de données, il faut avoir un système de gestion des bases de données ainsi que la maitrise d'un des langages permettant la manipulation des tuples comme *SQL* . C'est le même principe pour les graphes *RDF* qui nécessitent un moteur de recherche (comme *Jena* et *Sesame*) et la maitrise d'un langage de manipulation de triplets comme *SPARQL.*

Dans ce chapitre, nous allons voir ce qu'on veut dire par web sémantique, par mesures de similarité et par recherche plein texte. Nous allons ensuite découvrir des outils comme *Lucene* et *Solr*. A la fin, nous allons parler du langage *SPARQL* et de la norme *RDF*.

I.2 Web Sémantique

Selon Tim Berners-Lee : *" Le web sémantique n'est pas un web distinct mais bien un prolongement du web que l'on connait, dans lequel, on attribue à l'information une signification clairement définie, ce qui permet aux ordinateurs et aux humains de travailler en plus étroite collaboration" [2]*

Et dans une autre définition : *"C'est un immense espace d'échanges de ressources entre machines permettant à des utilisateurs d'accéder à de grands volumes d'informations et à des services variés". [2]*

Le web actuel a été conçu à la base pour être lisible par les humains et les machines, mais compréhensible que par les humains. Le besoin était de transformer ce web *"syntaxique"* en un web *"compréhensible par les machines"*.

C'est de là qu'est née l'initiative du web sémantique : *"Un web qui parle aux machines"*[4].

I.3 Mesures de similarité

Une mesure ou une distance de similarité textuelle est une quantité représentant le taux de ressemblance ou le taux de différence entre deux chaines de caractères ou même deux documents texte. Certaines mesures ont besoin d'un modèle pour représenter les documents

afin de calculer la distance entre eux, mais d'autres n'ont pas besoin.

On peut citer parmi ces modèles, le modèle vectoriel qui admet plusieurs mesures de similarité comme le *PRODUIT CARTESIEN* calculé de telle sorte :

$$\sum_{i=1}^{n} X_i \times Y_i [8]$$

Et la mesure de *COSINUS* calculée comme :

$$\frac{\sum_{i=1}^{n} X_i \times Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2} \times \sqrt{\sum_{i=1}^{n} Y_i^2}} [8]$$

On peut citer une distance qui n'a pas besoin de modèles pour représenter les documents, et les considère en conséquence comme deux chaines de caractères brutes. C'est la distance de *Levenshtein* pour *Vladimir Levenshtein* qui a été défini en 1965. Elle consiste tout simplement à calculer le nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer afin de passer d'une chaine de caractères à l'autre [24].

I.4 Recherche plein texte

La recherche plein texte (appelée aussi recherche en texte intégral ou recherche de texte libre) est une technique de recherche dans un document électronique, ou une base de données textuelle. En examinant tous les mots de chaque document enregistré, elle essaye de trouver ceux qui correspondent aux besoins de l'utilisateur du moteur de recherche [13].

Les termes donnés par l'utilisateur représentent la requête à satisfaire, ce qui signifie que la recherche plein texte permet de sélectionner des documents répondant à cette requête. En option, elle peut trier les documents restitués selon cette dernière. Le plus souvent, on trie les documents selon une mesure de similarité par rapport à la requête [13] .

Les deux notions de requête et de similarité varient beaucoup selon le besoin applicatif, mais le plus simple des cas est de considérer la requête comme un ensemble de termes et la mesure de similarité comme la fréquence des termes dans chaque document.

En cette situation, la recherche consiste à générer un index qui lie entre les documents et les termes qui se citent dedans sous forme d'une table. Cette table affiche les documents dans une dimension et les termes dans une autre puis les lie avec la fréquence de chaque

terme dans chaque document. Pour rechercher, il suffit de consulter l'index au lieu de balayer toute la base documentaire.

I.5 Lucene

Lucene est un moteur de recherche full-text, libre, écrit en Java (à 100%) et permet d'indexer et de rechercher du texte. C'est un projet open source de la fondation Apache mis à disposition sous sa licence. Il est également disponible pour les langages Ruby, Perl, C++, PHP [5]. Il est rapide, modulable, stable, performant [17]. Ces principales fonctionnalités sont l'indexation de données et la recherche dedans[5].

Au début, les systèmes qu'ils soient réparties (pas tellement réparties vu que c'était avant l'explosion du web) ou centralisés, utilisaient le système de classification de Dewey qui était efficace et simple mais pas scalable. Ce qui causa un vrai problème lors de l'explosion du web ou plutôt l'explosion des bases documentaires dans le web[5][16].

Lucene est venu à ce moment là pour régler le problème, vu que sa méthode se résume à créer un index en un format spécifique pour accélérer la procédure de recherche. Lors de la recherche, on analyse l'index pour trouver des références à des documents dont des termes spécifiques donnés se répètent dedans[5].

La qualité d'une recherche se mesure par le positionnement et la pertinence des résultats mais ceci rentre dans le cadre de la recherche d'informations en général.

La rapidité est un facteur déterminant pour traiter une vaste quantité d'informations. De même pour le support des requêtes (simples et complexes) et le support des interrogations (par termes et par phrases). Ces caractéristiques sont aussi importantes qu'une syntaxe facile à prendre en main pour saisir des requêtes[5][16].

On peut utiliser deux types de parseurs lors de la formulation de la requête. Le premier s'appelle *DisMax* et qui fait référence à *Disjunction Max*, le deuxième est un parseur standard [23].

La différence entre eux est que l'un prend la requête comme une chaine de caractères contenant uniquement les termes à rechercher dans l'index ; mais le second contient une conjonction ou une disjonction de clauses sous forme de :

$$attribut : valeur$$

ou de :

$$attribut : valeur1\ TO\ valeur2$$

Le second format est réservé aux attributs de type ordonné comme les chiffres comprenant les entiers, les réels ... etc.

Ayant seulement ces deux types de requêtes, on ne pouvait pas exprimer le besoin de trouver des documents qui ne contenaient pas une valeur précise pour un attribut. Alors la syntaxe a été modifiée pour introduire un signe aux clauses dans le parseur standard qui sont + et − et qui veulent dire si l'attribut devrait posséder la valeur associée ou pas [23]. Comme :

$$+attribut : valeur$$

retourne les documents qui ont un champ nommé *attribut* et qui a comme valeur $valeur$.

$$-attribut : valeur$$

retourne les documents qui ont un champ nommé *attribut* et qui n'a pas comme valeur $valeur$ [23].

Pour plus de détails dans les requêtes, le caractères $*$ a été introduit. De ce fait :

$$-attribut : *$$

retourne tous les documents qui ne possède de champs nommés *attribut* [23].

Pour les recherches non géographiques, Lucene utilise une formule pour calculer le score donné à un document d lors de l'exécution de la requête q qui est sous la forme suivante :

$$\sum_{t \in q} tf(t \in d) \times idf(t) \times bst \times IN(t.field \in d)[1]$$

Alors que :

- tf est la fréquence du terme t dans le document d.
- idf est la fréquence du document inverse pour le terme t.
- bst est la valeur d'importance attribuée au champ lors de l'indexation.
- IN est la valeur normalisée pour le champ donnant ainsi le nombre de terme dedans.

I.6 Solr

Solr est un moteur de recherche (sous forme d'un serveur pour entreprises) libre basé sur la bibliothèque Java de Lucene, proposant des API XML et JSON par HTTP (parce que c'est le protocole utilisé pour la communication avec le serveur en utilisant par défaut le port 8983 fournissant même une interface d'administation)[23].
Il a été mis en place à cause du fait que les licences des plateformes de recherche étaient chères à ce moment là ; et du côté de l'OpenSource, il n'éxistait pas de solutions complètes[25].
La figure I.1 représente l'architecture de l'application Solr.

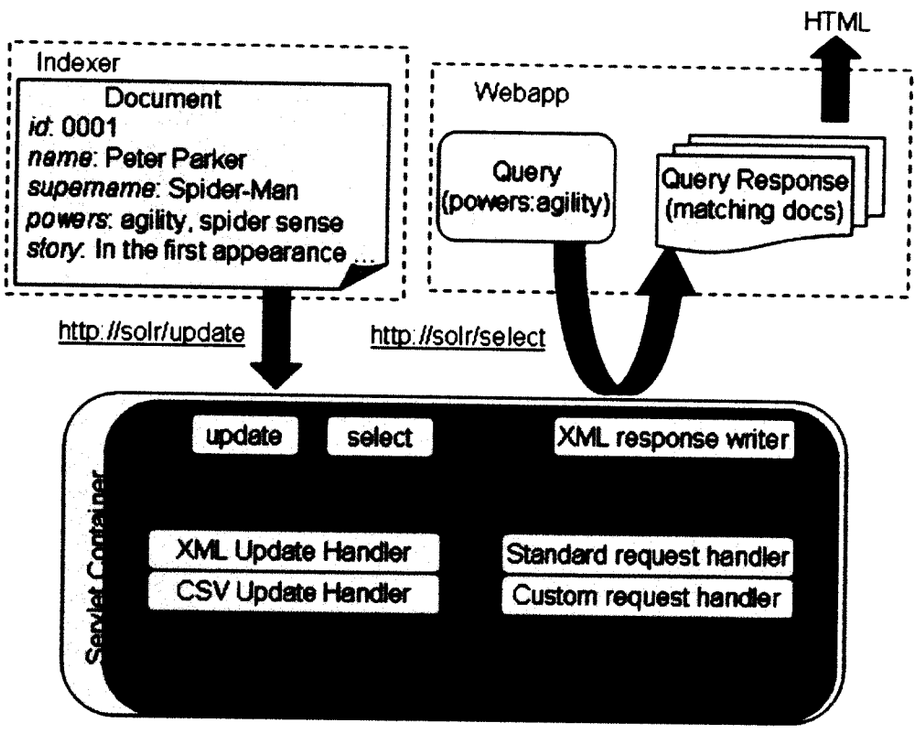

FIGURE I.1 – Architecture applicative de Solr [25]

Il y'a deux entrées dans Solr, l'une pour manipuler l'index et l'autre pour l'intérroger. La réponse à une requête donnée est écrite dans le langage XML grâce aux deux gestionnaires de requêtes standard et personnalisé. On peut indexer des documents dans le langage XML et CSV par défaut sans avoir à définir leurs gestionnaires de mises à jour. Si on veut indexer des documents dans d'autres langages, il suffit de définir leurs gestionnaires

de mises à jour. Toutes ces options sont déployables dans un conteneur servelet, afin de communiquer avec le client via le protocole HTTP, mais le coeur de l'application reste toujours Lucene.

I.7 RDF

Resource Description Framework (RDF) est une façon de formaliser les ressources web ainsi que leurs métadonnées, d'une manière à permettre le traitement automatique de telles descriptions. Développé par le W3C, RDF est le langage de base du web sémantique. L'une des syntaxes (ou sérialisations) de ce langage est RDF/XML. D'autres sérialisations de RDF sont apparues ensuite, cherchant à rendre la lecture plus compréhensible, c'est le cas par exemple de Notation3 (ou N3) [20].
Un document structuré en RDF est un ensemble de triplets RDF sous forme :

$$(sujet, predicat, objet)$$

- Le sujet représente la ressource à décrire ;
- Le prédicat représente un type de propriété applicable à cette ressource ;
- L'objet représente une donnée ou une autre ressource : c'est la valeur de la propriété.

I.8 SPARQL

RDF est un format de données de graphe orienté et étiqueté pour représenter des informations dans le Web. Cette spécification définit la syntaxe et la sémantique du langage d'interrogation SPARQL (*SPARQL Protocol and RDF Query Language*) pour RDF. SPARQL peut être utilisé pour exprimer des interrogations à travers diverses sources de données, que les données soient stockées nativement comme RDF ou vues comme du RDF via un logiciel médiateur (middleware). SPARQL est capable de rechercher des motifs de graphe (graph patterns) obligatoires et optionnels ainsi que leurs conjonctions et leurs disjonctions. SPARQL gère également le test extensible des valeurs et la contrainte des interrogations par un graphe RDF source. Les résultats des interrogations SPARQL peuvent être des ensembles de résultats ou des graphes RDF [26].
Un graphe RDF G peut être représenté de la façon suivante :

$$G = \{(s,p,o)|s \in (P \cup N) \wedge p \in P \wedge o \in (P \cup N \cup L)\}[22]$$

Tel que P est l'ensemble des prédicats, L l'ensemble des littéraux et N l'ensemble des ressources .

Exemple :

$G = \{$

$(http://www.example.org/resources\#person6913,$

$http://www.example.org/properties\#id, 6913),$

$(http://www.example.org/resources\#person6913,$

$http://www.example.org/properties\#nom, Taleb),$

$(http://www.example.org/resources\#person6913,$

$http://www.example.org/properties\#prenom, Yassine),$

$(http://www.example.org/resources\#person6913,$

$http://www.example.org/properties\#visite, A0),$

$(A0, http://www.example.org/properties\#date, 2008-1-8T13:48:18Z),$

$(A0, http://www.example.org/properties\#maladie, Malaria),$

$(A0, http://www.example.org/properties\#remarque, Etat\ stationnaire),$

...

}

La figure I.2 constitue une représentation graphique de l'ensemble des triplets G.

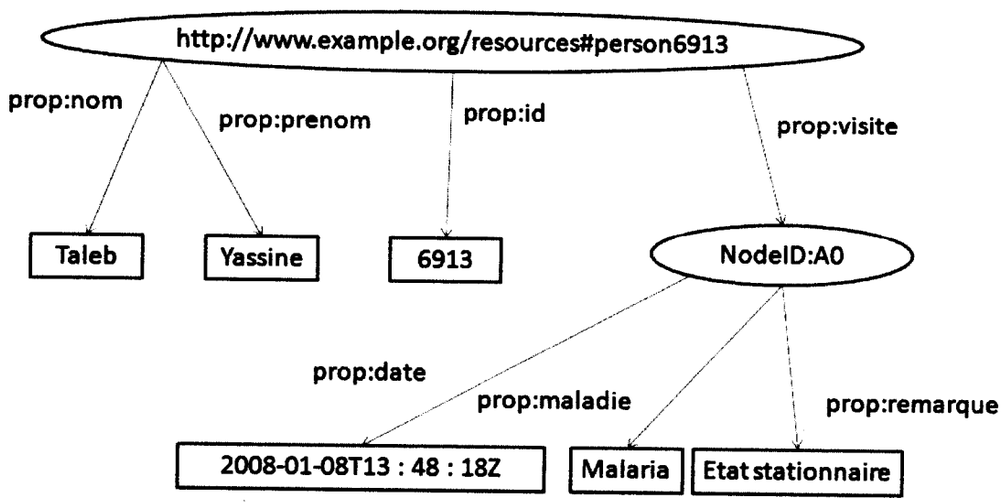

FIGURE I.2 – Représentation du graphe G

Un graphe est représenté analytiquement par une suite des triplets qu'il contient, on peut donc écrire simplement :

$$G = (P \cup N) \times P \times (P \cup N \cup L)$$

On note :

$$PN = P \cup N \ \ et \ \ T = P \cup N \cup L$$

Ce qui donne :

$$G = PN \times P \times T$$

SPARQL est un langage de requêtes pour l'interrogation des graphes RDF à base de "$graph - matching$". Pour le faire, il a besoin d'un pattern. L'élément pat est appelé pattern si et seulement si :

$$pat = (s,p,o) | s \in (PN \cup V) \wedge p \in (P \cup V) \wedge o \in (T \cup V)$$[21]

Tel que V est l'ensemble des variables qu'on peut utiliser pour l'appariement.

Comme :

$(?x, http://www.example.org/properties\#visite, ?y)$

$(?x, http://www.example.org/properties\#nom, Taleb)$

$(?x, http://www.example.org/properties\#prenom, Yassine)$

$(?x,\ ?y,\ ?z)$

Dans le dernier cas les variables sont : x, y et z. On note :

$$M = (PN \cup V) \times (P \cup V) \times (T \cup V)$$

En d'autres termes, on a[21] :

$\forall x, y \in M,$

$$(x\ AND\ y) \in M,$$

$$(x\ UNION\ y) \in M,$$

$$(x\ OPT\ y) \in M,$$

$$(x\ FILTER\ R) \in M$$

Tel que R est un "$Built - In\ Condition$", et il est construit à partir d'éléments de $(V \cup P \cup L)$, de constantes, d'opérateurs logiques tels que $\wedge, \vee, et\ \neg$, d'opérateurs d'inégalité $<, >, \leq\ et\ \geq$, de l'opérateur d'égalité $=$ et de quelques prédicats logiques comme "$bound, isBlank, isIRI$" [21].

On note B l'ensemble des "$Built - In\ Condition$".

Alors [21] :

$$(x, y \in V \wedge c \in P \cup L) \Rightarrow (bound(x) \in B \wedge (x = c) \in B \wedge (x = y) \in B)$$

$$x, y \in B \Rightarrow \neg(x), (x \vee y), (x \wedge y) \in B$$

$$\forall P \in M,\ \forall R \in B,\ (P\ FILTER\ R) \in M \Rightarrow var(R) \subseteq var(P)$$

Alors que $var(R)$ et $var(P)$ sont l'ensemble des variables mentionnées respectivement dans R et P.

Le principe du "$graph - matching$" revient au rôle d'une fonction de "$mapping$" qui peut être notée "f" et qui associe pour chaque variable d'un pattern un élément de T .

$$f : V \longrightarrow T[21]$$

Exemple :

Soit la fonction de mapping suivante :

$f(x) = http://www.example.org/resources\#person6913$

$f(y) = http://www.example.org/properties\#id$

$f(z) = 6913$

Chaque fonction a un domaine de définition et dans notre cas, on note l'ensemble relatif à "f" comme "$dom(f)$" et qui va contenir l'ensemble des variables ou des éléments de V qui ont une image avec la fonction "f" [21].

Dans ce cas le domaine de la fonction de notre exemple f sera : $dom(f) = \{x, y, z\}$, car ces variables sont les seules à avoir une image de cette fontion de mapping.

Deux fonctions de "mapping" f_1 et f_2 sont dites compatibles si et seulement si :

$$\forall x \in dom(f_1) \cap dom(f_2), f_1(x) = f_2(x)[21]$$

Pour notre cas, soit les deux fonctions de mapping f_1 et f_2 :

$f_1(x) = http://www.example.org/resources\#person6913$

$f_1(y) = http://www.example.org/properties\#id$

$f_1(z) = 6913$

Et

$f_2(x) = http://www.example.org/resources\#person6913$

Les deux fonctions ont x comme élément en commun dans leurs domaines respectifs, et ont la même image pour ce dernier.

Notez que deux fonctions avec deux domaines disjoints sont toujours compatibles, et une fonction qui a un domaine vide est compatible avec toute fonction de "*mapping*" [21]. Vu que dans les deux derniers cas, l'intersection des domaines des fonctions de mapping est un ensemble vide et ne comporte en conséquence aucun élément commun, de là on peut dire que la formule de la compatibilité est vérifiée.

On définie une suite d'operateurs sur les fonctions de "*mapping*", supposant que X et Y sont deux ensembles de fonctions de "mapping" définis ci dessous.

X :

x	y
$http://www.example.org/resources\#person6913$	6913
$http://www.example.org/resources\#person693$	693
$http://www.example.org/resources\#person613$	613
$http://www.example.org/resources\#person913$	913

Y :

x	z
$http://www.example.org/resources\#person6913$	*Taleb*
$http://www.example.org/resources\#person693$	*Settouti*
$http://www.example.org/resources\#person613$	*Saouli*
$http://www.example.org/resources\#person913$	*Hadj slimane*

On a les trois opérateurs donnés dans l'ordre suivant : jointure, union et différence.

$$X \bowtie Y = \{x \cup y \mid x \in X \wedge y \in Y \wedge x \text{ et } y \text{ sont compatibles}\} [22]$$

x	y	z
$http://www.example.org/resources\#person6913$	6913	$Taleb$
$http://www.example.org/resources\#person693$	693	$Settouti$
$http://www.example.org/resources\#person613$	613	$Saouli$
$http://www.example.org/resources\#person913$	913	$Hadjslimane$

Le résultat de la jointure de X et Y représente l'ensemble des patients cités dans X et Y, avec leurs identifiants et noms respectifs.

$$X \cup Y = \{x \mid x \in X \vee x \in Y\}[22]$$

x	y	z
$http://www.example.org/resources\#person6913$	6913	
$http://www.example.org/resources\#person693$	693	
$http://www.example.org/resources\#person613$	613	
$http://www.example.org/resources\#person913$	913	
$http://www.example.org/resources\#person6913$		$Taleb$
$http://www.example.org/resources\#person693$		$Settouti$
$http://www.example.org/resources\#person613$		$Saouli$
$http://www.example.org/resources\#person913$		$Hadjslimane$

Le résultat de l'union entre X et Y est tout simplement le contenu de X suivi du contenu de Y.

$$X \setminus Y = \{x \in X \mid \forall\, y \in Y\ ,\ x \textit{ et } y \textit{ ne sont pas compatibles}\}[22]$$

x	y

Le résultat de la différence entre X et Y est l'ensemble des patients cités dans X et non cités dans Y munis de leurs identifiants.Le tabeau affiché en dessus est vide car tous les patients cités dans X sont cités dans Y.

Apres avoir vu ces trois opérateurs, on peut en déduire la jointure extérieure gauche :

$$X \rtimes Y = (X \bowtie Y) \cup (X \setminus Y)[22]$$

x	y	z
$http://www.example.org/resources\#person6913$	6913	$Taleb$
$http://www.example.org/resources\#person693$	693	$Settouti$
$http://www.example.org/resources\#person613$	613	$Saouli$
$http://www.example.org/resources\#person913$	913	$Hadjslimane$

Le résultat de la jointure extérieure gauche est le même que la jointure introduite un peu auparavant, vu que la différence entre X et Y est un ensemble vide. Dans le cas contraire, on aurait eu des patients cités dans X et non cités dans Y, ayant un identifiant mais pas de nom. Quand on lance une requête dans SPARQL, il résulte généralement un tableau de données contenant les variables du pattern de la requête dans les colonnes et les différents appariements correspondants qu'il a trouvé dans chaque ligne. Ceci s'appelle bien le " semantics" du pattern de la requête lancée.

Mathématiquement, on le note $[x]_D$ sachant que x est le pattern et D est l'ensemble des triplets d'un graphe RDF et on le définie de la façon suivante :

$$[x]_D = \{f \mid dom(f) = var(x) \wedge f(x) \in D\}[21]$$

Par abus de notations, on veut dire par $f(x)$ l'ensemble des images des éléments $var(x)$ avec la fonction " f" De cette définition se découlent les propriétés suivantes [21] :

$$[x\ AND\ y]_D = [x]_D \bowtie [y]_D$$

$$[x\ OPT\ y]_D = [x]_D \rtimes [y]_D$$

$$[x\ UNION\ y]_D = [x]_D \cup [y]_D$$

$$[(x\ FILTER\ R)]_D = \{f \in [x]_D \mid f \models R\}$$

Comme si on avait le pattern p suivant :

$prop : http://www.example.org/properties\#$

$(?x\ prop : nom\ ?nom\ OPT\ ?x\ prop : prenom\ ?prenom)$

x	nom	$prenom$
$http://www.example.org/resources\#person6913$	$Taleb$	$Yassine$
$http://www.example.org/resources\#person693$	$Settouti$	$Ahmed$
$http://www.example.org/resources\#person613$	$Saouli$	$Yacine$
$http://www.example.org/resources\#person913$	$Hadj\ slimane$	$Omar$

On dit qu'une fonction de mapping "f" satisfait un Built-In Condition "R" et on le note $f \models R$ dans l'un des cas suivants [22] :

$$R = bound(?X)\ et\ ?X \in dom(f)$$

$$R = ?X = c\ et\ f(?X) = c$$

$$R = ?X = ?Y\ et\ f(?X) = f(?Y)$$

$$R = \neg(R')\ et\ R' \in B \wedge \neg(f \models R')$$

$$R = R' \wedge R''\ et\ f \models R' \wedge f \models R''$$

$$R = R' \vee R''\ et\ f \models R' \vee f \models R''$$

Comme :

La fonction de mapping "f" définit ci-dessous :

$f(x) = http://www.example.org/resources\#person6913$

$f(y) = 6913$

$f(z) =' Taleb'$

satisfait le built in condition $R = (bound(?z)) \wedge (?y = 6913)$. Car la variable z possède une image avec la fonction f et l'image de y avec cette dernière est égale à 6913.

I.9 Extension et passage à iSPARQL

Le grand défaut qu'avait SPARQL, était le fait qu'il ne connaissait pas la notion de mesures de similarité. Il faisait donc un appariement basé sur une égalité parfaite. Pour palier à ce problème, on devait passer à iSPARQL [7].

Un pattern virtuel V_t est un triplet sous la forme :

$$\{(?variable\ldots)\ namespace : nom\_fonction\ (Argument\ldots)\}[22]$$

Ou :

- *nom_fonction* est le nom local de la fonction à utiliser pour le pattern virtuel.
- *namespace* est l'espace de nommage hébergeant les fonctions pour les patterns virtuels.
- *?variable* est une variable qui va contenir le résultat de la fonction, ils peuvent être plusieurs, dans ce cas, il faudrait prévoir plusieurs variables.
- *Argument* est ce qui est passé à la fonction comme argument, il peut être constant comme il peut être variable. Il peut être seul comme ils peuvent être plusieurs.

I.10 Conclusion

Dans ce chapitre, nous avons introduit des notions de base concernant des outils comme Lucene et SPARQL, car Jena et Sesame leurs font appel. Ils utilisent le langage SPARQL pour construire des requêtes d'interrogation. Ils permettent des recherches de texte intégral en utilisant Lucene. Ils emploient des patterns virtuels pour mettre Lucene en relation avec les requêtes SPARQL. Ils calculent la distance entre les termes grâce aux distances de similarité textuelles.

N'importe qui voulant faire une étude comparative entre deux normes de restauration de triplets proposant une recherche plein texte, devrait connaitre des connaissances assez approfondies concernant les notions que nous venons de présenter dans ce chapitre. Dans ce qui suit, nous allons voir comment les outils présentés dans ce chapitre s'interopèrent pour réaliser la fonction de recherche plein texte dans des sources RDF.

II *Outils d'indexation et d'interrogation à base de SPARQL*

II.1 Introduction

Le monde est un sac de choses qui s'auto complètent dans chaque domaine. En informatique, il y'a toujours de nouveaux outils (logiciels, plateformes, langages) pour répondre aux besoins de leurs utilisateurs. Mais ceci génère un problème de choix lorsqu'on veut travailler dans un domaine précis avec un besoin précis sans savoir quel outil utiliser. Dans ce cas, il faudrait tester tous les outils possibles et en juger le meilleur, ce qui fait un travail supplémentaire.

Dans ce chapitre, nous allons voir les deux moteurs de recherche Jena et Sesame avec une défintion mathématique de leurs patterns virtuels. Ensuite, nous allons présenter des tableaux comparatifs entre eux. Pour finir, nous allons mettre le point sur quelques travaux similaires déjà faits dans ce domaine.

II.2 Jena

Jena est une palteforme de web sémantique en Java, elle assure la lecture et l'écriture de données *RDF* sous forme de modèles. Ce dernier peut être interrogé par *SPARQL* et mis à jour par SPARUL [27].

C'est la plateforme la plus utilisée concernant les sources *RDF*. Elle supporte *OWL* et possède une communauté active. Deux moteurs sont diponibles pour la restauration des données ; *TDB* (native triple store) et *SDB* (triple store on top of a relational database) [14].

Conçu pour batir des applications du web sémantique, elle offre des outils pour la lecture des sources *RDF* dans différents formats, pour l'inférence à base de règles et pour restaurer un grand nombre de triplets afin de les exploiter de manière optimale [14].

Jena a été construite pour la manipulation des graphes *RDF* en général. elle possède une partie pour l'interrogation avec *SPARQL* qui s'appelle *"ARQ"*. On lui a récemment ajouté les fonctionnalités de Lucène pour une possibilité d'indexation et de recherche "full-text". On donna à cette fonctionnalité un composant nommé *"LARQ"*[15].

La fonctionnalité la plus utilisée dans *"LARQ"* est l'instruction de *"textMatch"* qui utilise un espace de nommage généralement désigné par "pf" et référé par :

$$http://jena.hpl.hp.com/ARQ/property\#[15]$$

La syntaxe minimale de cette requête est :

$$?varaible\, pf : textMatch\, "Keyword"$$

Tel que *Keyword* est le mot qu'on veut rechercher, comme :

$$?x\, pf : textMatch\, "Fadi"$$

Mais la syntaxe complète est sous la forme de :

$$?variable\, pf : textMatch\, ("Keyword"\ precision\, limite)$$

Tel que *precision* est un réel entre 0 et 1 qui représente le score minimal d'appariement que doivent avoir les noeuds résultants, et *limite* est un entier qui représente le nombre de résultats à afficher. Comme :

$$?x\, pf : textMatch\, ('+text'\, 0.5\, 100)$$

Dans ce cas, on a limité le nombre de résultats à 100 et on a choisi seulement ceux qui correspondent au mot clé à plus de 50%.
Pour cette fonction, il existe trois modes (ou trois patterns) ; Le premier retourne les noeuds litéraux qui correspondent au mot clé *"Keyword"* ou qui s'approchent un peu de lui (plus que le θ). Le deuxième retourne les ressources de type prédicats qui ont une valeur qui colle parfaitement ou du moins plus que la précision vis à vis à *"Keyword"* et le dernier retourne les noeuds externes pas présents dans le graphe RDF.

II.2.1 Pattern 1

Supposant que D est un graphe *RDF*, $\theta \in [0, 1]$ et $n \in N$. Dans le modèle 1, on aura [15] :

$$[?variable\, pf : textMatch\, 'Keyword'\, \theta\, n]_D = \{t_1, t_2, \ldots, t_n\}$$

Tel que : $\forall i$,

$$t_i \in L$$

$$SIM(value(t_i),'Keyword') \geq SIM(value(t_{i+1}),'Keyword')$$

$$SIM(value(t_i),'Keyword') \geq \theta$$

Ou, *value* retourne la chaine de caractères inscrite dans le litéral (ou tout simplement sa valeur), *SIM* retourne la distance de similarité entre la valeur du litéral t_i et le mot clé voulu recherché *"Keyword"*.
En d'autres termes, la fonction *"textMatch"* retourne dans ce cas les *n* premiers litéraux correspondants à la recherche *"Keyword"* et ayant un score superieur à θ.

II.2.2 Pattern 2

Supposant que D est un graphe *RDF*, $\theta \in [0,1]$ et $n \in N$. Dans le modèle 2, on aura [15] :

$$[?variable\, pf : textMatch\, 'Keyword'\, \theta\, n]_D = \{t_1, t_2, \ldots, t_n\}$$

Tel que : $\forall i$,

$$t_i \in P$$

$$SIM(value(t_i),'Keyword') \geq SIM(value(t_{i+1}),'Keyword')$$

$$SIM(value(t_i),'Keyword') \geq \theta$$

Ou, *value* retourne l'adresse attribuée au prédicat (ou tout simplement son URI), *SIM* retourne la distance de similarité entre l'URI du prédicat t_i et le mot clé voulu recherché *"Keyword"*.
En d'autres termes, la fonction *"textMatch"* retourne dans ce cas les *n* premiers prédicats correspondants à la recherche *"Keyword"* et ayant un score superieur à θ.

II.2.3 Pattern 3

Le troisième modèle est spécial, il concerne les resources externes du graphe, comme les fichiers *PDF* et *HTML*. La recherche dans ces fichiers est entièrement la reponsabilité de Lucene. Supposant que D est un graphe *RDF*, $\theta \in [0,1]$ et $n \in N$. Dans le modèle 3, on aura [15] :

$$[?variable\, pf : textMatch\, 'Keyword'\, \theta\, n]_D = \{t_1, t_2, \ldots, t_n\}$$

Tel que : $\forall i$,

$$t_i \in DOC$$

$$SIM(value(t_i),' Keyword') \geq SIM(value(t_{i+1}),' Keyword')$$

$$SIM(value(t_i),' Keyword') \geq \theta$$

Ou, *value* retourne la chaine de caractères inscrite dans le document externe (ou tout simplement son texte), *SIM* retourne la distance de similarité entre le texte du document t_i et le mot clé voulu recherché *"Keyword"*. Et DOC est l'ensemble des documents ou ressources externes de façon générale associés au graphe interrogé.
En d'autres termes, la fonction *"textMatch"* retourne dans ce cas les *n* premiers documents ou ressources externes correspondants à la recherche *"Keyword"* et ayant un score superieur à θ.

II.3 Sesame

Sesame est une plateforme open source plus orientée pour l'interrogation et l'analyse des graphes RDF. Créé et maintenu par la société Aduna, et à l'origine une partie du projet "On-To-Knowledge", c'est un projet de restauration de triplets pour les applications du web sémantique [18].
Sesame ne supporte pas que SPARQL comme langage mais bien d'autres, et offre une interface d'administration modulaire pour insérer de nouvelles fonctionnalités. Bien qu'il supporte les recherches plein texte, il supporte aussi les recherches géospatiales [19].
Pour faire une recherche en texte intégral, Sesame utilise son composant LuceneSail ayant la syntaxe d'une requete comme suit :

$PREFIX\ search :< http : //www.openrdf.org/contrib/lucenesail\# >$
$SELECT\ ?x\ ?score\ ?snippet\ WHERE\{$
$?x\ search : matches\ ?match.$
$?match\ search : query\ "person";$
$search : property\ rdfs : label;$
$search : score\ ?score;$
$search : snippet\ ?snippet.\}$[12]

- "search" est un simple espace de nommage (obligatoire parce que sans lui, on ne sait pas quelle fonction utiliser)
- "matches" est la fonction qui fait l'appariement, elle retourne les littéraux correspondants à la requête (obligatoire puisqu'il représente le déclencheur de l'opération)
- "query" porte comme paramètre la chaine de caractères à rechercher (l'équivalent de 'Keyword' dans LARQ et il est obligatoire)
- "property" sélectionne le type de litéraux à rechercher dedans
  - elle n'est pas obligatoire et vise la recherche sur tous les litéraux par défaut.
- "score" retourne le score de l'appariement lors de l'execution de la requête
  - elle n'est pas obligatoire.
  - elle doit toujours avoir un paramètre variable.
- "snippet" comprend des fragments de textes contenu dans le résultat.
  - elle n'est pas obligatoire.
  - elle doit toujours avoir un paramètre variable.

On peut en plus utiliser :

- Les métacaractères dans l'argument de la fonction $query$, comme ? veut dire n'importe quel caractère, et $*$ n'importe quelle chaine de caractères.
- Le caractère $\sim$ pour montrer que la recherche se fait d'une manière approximative ou $floue$ avec une distance de similarité de $Levinshtine$. Si on ne met pas de barre limite pour la distance minimale, elle sera de 0.5 [12].

Supposant que D est un graphe RDF sur lequel on va exécuter une requête avec Sesame. Le mot clé de la requête est θ et le résultat retourné dans la variable $match$ est défini comme suit :

$$x \in N | (x, p, o) \in D \wedge type(o) = type \wedge SIM(o, \theta) \geqslant d$$

Tel que :

- $type(o)$ retoune le type de l'objet o.
- $SIM(o, \theta)$ retourne la mesure de similarité entre l'objet o et la requête θ.
- $type$ est initialement égal à $rdf : literal$ mais peut être manipulé pour des recherches plus précises.
- d est initialement égal à 0.5 comme mesure moyenne entre un objet et une requête mais on peut le manipuler pour avoir des recherches plus larges ou plus restreintes.

II.4 Comparaison entre Jena et Sesame

Dans le monde des développeurs d'applications basées sur le web sémantique, il y a 2 grandes écoles en ce qui concerne les triplestores : les adeptes de Jena, et ceux de Sesame.

Sesame propose des fonctionnalités proches de Jena, moins complètes, mais sans doute un peu plus faciles à intégrer ; la philosophie de son API est différente, et, s'il s'agit de faire simplement de la manipulation de données RDF (sans inférence compliquée, sans gestion d'ontologie OWL), le choix entre les deux est essentiellement une question de goût ! Son interface web d'administration vous facilitera par contre énormément le travail s'il s'agit de configurer rapidement un triplestore et de charger des données RDF dedans.

On donne ci-dessous un tableau comparatif des fonctionnalités entre Jena et Sesame ; il faut noter que, là où Jena propose d'emblée un certain nombre de modules avec beaucoup de fonctionnalités, pour Sesame ces fonctionnalités peuvent être présentes dans l'écosystème autour de Sesame : OWLIM ou Ali Baba [11].

La comparaison va comprendre d'abord des critères généraux concernant des utilisateurs non expérimentés. Puis, des critères d'entrées et de sorties, des critères concernant la gestion du RDF et d'OWL. Ensuite, des critères concernant le langage utilisé (SPARQL). Aprés ça, on va voir des critères concernant le schéma et le comportement (inférence). Pour finir, on a mis des critères concernant les interfaces d'administration et d'autres caractéristiques diverses.

II.4.1 Général

Jena est apparu avant Sesame, c'est une licence Apache et un standard de HP alors que Sesame est une licence BSD-Style et un standard d'Aduna. Ils sont Open Source, supportent des interrogations basées sur les graphes, ils utilisent Lucene comme norme d'indexation et il existe des mises a jour régulières pour les deux API.

On peut dire que Jena est plus robuste et interopérable que Sesame de plus il utilise un serveur (Joseki) et que pour chaque triplet il indexe son prédicat ainsi que son objet alors que Sesame n'indexe que les litéraux, comme nous le montre le tableau II.1 ci-dessous.

Critères	Sesame	Jena
Existe depuis	2004	2000
Licence	BSD-style http ://www.opensource.org /licenses/BSD-3-Claus	Apache http ://www.apache.org /licenses/LICENSE-2.0
Support des interrogation basés sur les graphes	Oui	Oui
Objets indexé	Litéraux	Paires de pédicats et litéraux
Java framework	Oui	Oui
Building semantic web applications	Oui	Oui
Norme d'indexation	Lucene	Lucene
Open source	Oui	Oui
RDF Repositories	Oui	Non
storage layer support	PostgreSQL, MySQL, Microsoft SQL Server and Oracle	HSQLDB, MySQL, PostgreSQL, Derby, Oracle, Microsoft SQL Server
Autres langages	Ruby,Pearl, PHP5 ou Delphi	
HTTP	Oui	Oui
Interopérabilité	-	+
Robustesse	-	+
Server		Joseki
Help	+	-
External reasoners		DIG
Standard	Aduna	HP
Mis à jour régulièrement	Oui	Oui

TABLE II.1 – Comparatif général des critères

II.4.2 Entrées/Sorties

Sesame possède un analyseur intégré et le sérialiseur pour le TriG et les syntaxes TriX y compris les extensions syntaxiques TriX, Jena tolère les sorties en RDF/XML-ABBREV Jena et Sesame possèdent une API pour la lecture, de traitement et de l'écriture RDF des données dans les formats XML, N-triples et Turtle, comme nous montre le tableau II.2 ci-dessous.

Critères	Sesame	Jena
Entrées/Sorties en TriG, TriX	Oui	Non
Entrées Sorties aux formats RDF/XML, N3, Turtle, N-Triples	Oui	Oui
Sortie en RDF/XML-ABBREV	Non	Oui

TABLE II.2 – Comparatif des critères Entrées/Sorties [6]

II.4.3 SPARQL

Jena et Sesame supportent les requêtes SPARQL dans la majorité de ses versions comme il existe des utilitaires en ligne de commande pour exécuter une requête SPARQL.
Par contre, la puissance de Sesame s'avère dans le support d'autres langages de manipulation de triplets tels que : RDQL, RQL et SeRQL, comme nous montre le tableau II.4 ci-dessous.

Critères	Sesame	Jena
Support de SPARQL	Partially	Fully
Requête en SPARQL 1.0	Oui	Oui
Mise à jour en SPARQL	Oui	Oui
Autres langages supportés	RDQL, RQL et SeRQL	Non
Création de requêtes SPARQL programmatiquement	Oui (package org.openrdf.query.parser.sparql.ast)	Oui (module ARQ)
Serveur SPARQL	Oui (de base à travers le serveur Sesame)	Oui (module Fuseki)
Utilitaire en ligne de commande pour executer une requête SPARQL	Oui, via la commande "console"	Oui
Indexation plein texte des données du graphe RDF par Lucene	Non, mais possible en utilisant AliBaba ou l'extension " LuceneSail" : http ://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail	Oui, module LARQ

TABLE II.3 – Comparatif SPARQL[11]

II.4.4 Gestion du RDF

Jena comme Sesame permettent la gestion en mémoire des RDF, le stockage des RDF dans une base relationnelle comme dans des fichiers binaires ainsi le support des graphes nommé et les transactions (commit,rollback).

De plus Jena permet le paramétrage des graphes RDF via un fichier de configuration grâce à son module *assembler*, comme nous montre le tableau II.3 ci-dessous.

Critères	Sesame	Jena
Gestion du RDF en mémoire	Oui	Oui
Stockage RDF dans une base relationnelle	Oui	Oui (module SDB)
Stockage RDF dans des fichiers binaires	Oui (native RDF repository)	Oui (module TDB)
Support des graphes nommés	Oui	Oui
Support des transactions (commit, rollback)	Oui	Oui
Paramétrage de graphes RDF via un fichier de configuration	Non	Oui (module assembler)

TABLE II.4 – Comparatif du gestion RDF[11]

II.4.5 OWL

Jena possède un API de manipulation OWL et RDFS, comme nous montre le tableau II.5 ci-dessous.

Critères	Sesame	Jena
API de manipulation OWL et RDFS : manipulation des classes, propriétés, domain, ranges, restrictions, etc.	non	Oui, natif

TABLE II.5 – Comparatif d'OWL[11]

II.4.6 Inférence

Sesame admet l'inférence RDFS native dans son module mais pour les autres inférences telles que OWL-lite native, OWL DL et les inférences à base de règle il utilise OWLIM par contre Jena admet toutes les inférences dans son module, comme nous montre le tableau II.6 ci-dessous.

Critères	Sesame	Jena
Inférence RDFS native	Oui	Oui
Inférence OWL-Lite native	Non, mais possible en utilisant OWLIM	Oui
Inférence OWL-DL	Non, mais possible en utilisant OWLIM)	Non, mais possible en connectant le moteur Pellet
Inférence à base de règles	Non, mais possible en utilisant OWLIM	Oui

TABLE II.6 – Comparatif d'Inférence[11]

II.4.7 Schéma

Jena est plus rapide que Sesame mais ne détecte pas les prédicats multi-valués et n'admet pas de multiples requêtes par mot clé de plus Sesame est plus précis et expressive que Jena, comme nous montre le tableau II.7 ci-dessous.

Critères	Sesame	Jena
Rapidité	-	+
Expressivité	+	-
Précision	+	-
Détection des prédicats multi-valués	Oui	Non
Multiple field-based keyword query	Oui	Non

TABLE II.7 – Comparatif shéma

II.4.8 Administration

Sesame possède une interface utilisateur (sesame-workbench) en se connectant au localhost du port 8080/sesame-workbench mais avant il faut ajouter les deux fichiers War (sesame.war et workbench.war) au serveur, comme nous montre le tableau II.8 ci-dessous.

Critères	Sesame	Jena
Interface utilisateur d'administration : créer un stockage RDF, ajouter des données, naviguer, etc.	Oui (sesame-workbench)	Non

TABLE II.8 – Comparatif d'Administration

II.4.9 Divers

Jena possède un mécanisme de listeners pour monitorer les événements sur le RDF comme il possède un utilitaire de vérification de données RDF avec son module eyeball enfin, pour la génération de fichiers de constantes Java à partir d'une ontologie Sesame utilise AliBaba et Jena utilise son module schemagen, comme nous montre le tableau II.9 ci-dessous.

Critères	Sesame	Jena
Mécanisme de listeners pour monitorer les événements sur le RDF	Non	Oui
Génération de fichiers de constantes Java à partir d'une ontologie	Non, mais possible en utilisant Ali-Baba	Oui (module schemagen)
Utilitaire de vérification de données RDF	Non	Oui (module eyeball)

TABLE II.9 – Comparatif divers[11]

II.5 Travaux similaires

Jena et Sesame sont deux plateformes Java pour construire des applications du web sémantique. Elles peuvent aussi indexer et intérroger des sources RDF et OWL respectivement grâce à LARQ et LuceneSail. Ces outils utilisent tout les deux Lucene pour l'indexation, et SPARQL pour l'interrogation.

Des travaux ont été proposés dans ce contexte. Parmis eux :

- *Enrico Minack et al.* dans leur projet intitulé *"Benchmarking Fulltext Search Performance of RDF Stores"* ou ils ont comparé entre Jena, Sesame, Yars et Virtuoso. Dans cette étude, ils ont choisi, une suite de requêtes d'abord. Ils les ont déroulé une par

une sur chacune des plateformes choisies, et ont noté les temps d'évaluation. Ce projet avait comme résultat que c'est Sesame et Jena qui étaient les plus efficaces, parce que les autres retournaient des erreurs comme "Out Of Memory" et prenaient un temps plus grand lors des évaluations des requêtes complexes. Ce projet avait de bons résultats vis à vis de nos choix de plateformes, mais ne se concentrait que sur l'aspect général des inférences, et ne faisait pas tellement allusion aux outils de recherche de texte libre [9].

- *Martin Filliau* avec son rapport intitulé *"Semantic Mashup to Query Localised Data"* dont lequel il comparait entre les différents composants de Jena, Sesame, 4store, Talis et SiREN. Mais sa recherche visait un côté théorique et n'entrait pas dans un cadre pratique (implémentation et tests) parce que la comparaison n'était pas son but principal [10].

- *Christian Bizer et Andreas Schultz* dans leur projet intitulé *Berlin SPARQL Benchmark* dans lequel ils ont comparé entre Sesame, Jena TDB, Jena SDB, Virtuoso TS, Virtuoso RV, D2R Server, mySQL et Virtuoso SQL selon le temps de chargement de triplets. Le problème est qu'ils n'ont pas parlé des autres caractéristiques de ces moteurs comme l'indexation et l'interrogation en texte libre [3].
- "Thomas Francart" qui a comparé entre Jena et Sesame mais selon des critères généraux comme les tableaux comparatifs de la section précédante [11].

On remarque que ces travaux sont interessants mais ne se concentrait pas sur l'aspect technique et pratique des outils de recherche en texte intégral.

II.6 Conclusion

Si deux moteurs de gestion du RDF sont tous les deux aussi utilisés, ça veut dire que les deux ont leurs avantages et leurs inconvenients par rapport aux autres (et aussi l'un par rapport à l'autre). Si on connait ces avantages et ces inconvenients, on saura quand utiliser l'un et quand utiliser l'autre.

Ce qui pourra nous faire gagner le temps de la réflexion avant l'utilisation d'une platforme. Nous avons alors opté à faire évoluer les deux leaders en leurs ajoutant des fonctions de filtre qui auront comme but une recherche de texte libre.

Ensuite, nous allons comparer nos deux approches avec les outils d'origine de Jena et Sesame

III *STAFF (Sparql-based full-Text search Add Filter Function)*

III.1 Introduction

Il existe beaucoup d'API permettant l'interrogation de sources RDF, certaines proposent une recherche pleine texte et d'autres non. Parmi ces API, on peut citer Virtuoso, YARS, SiReN ... etc. Mais les plus utilisés sont Jena et Sesame, et qui sont aussi considérés comme les meilleurs dans la plupart des domaines. Notre travail consiste à établir une nouvelle approche visant la recherche pleine texte, et ensuite comparer enfin cette dernière avec l'approche d'origine de l'API.

Dans ce chapitre, nous allons présenter brièvement STAFF acronyme de Sparql - based full-Text Add Filter Function. Ensuite, comment a-t-on conçu ce dernier. Après, un exemple de son implémentation. Et à la fin une évaluation avec les outils d'origine (Jena et Sesame).

III.2 Présentation du système

Les deux moteurs utilisent un pattern virtuel pour la recherche en texte intégral. Quand on a plusieurs patterns (qu'ils soient virtuels ou pas), certaines approches penchent vers le calcul des interprétations possibles de chacun puis faire une intersection entre les deux ensembles de fonction de mapping. Cette approche est très proche du raisonnement humain et très facile à implémenter mais peut prendre beaucoup de temps et d'espace lors de l'exécution. De plus, les approches d'origine choisissent généralement de laisser à l'utilisateur le choix de retourner le score d'appariement, alors qu'il en n'a pas besoin. Le traitement de la restitution du score ne sera qu'une tâche en plus dans le travail.

Notre approche est simple, au lieu de mettre un pattern virtuel qui calcule et retourne des résultats et leurs scores à partir d'un index, on utilise une fonction de filtre personnalisée qui va nous dire si les nœuds correspondent à une requête donnée ou pas. Pour cela, elle retournera un littéral booléen dont la valeur vraie signifie que le nœud se trouve parmi les résultats de la requête et fausse pour son absence dedans.

La fonction personnalisée a besoin de quatre arguments :

- Un élément qu'il soit ressource concrète, prédicat ou littéral, dont on va voir s'il correspond à la requête voulu exécuter.
- La requête elle-même qui va être exécutée.

- Un réel représentant le score que les résultats de la requête doivent avoir.
- Un entier représentant le nombre de premiers documents à prendre en compte parmi les résultats retournés.

On va implémenter cette approche dans les deux restituteurs de triplets les plus utilisés dans le monde et comparer chacun des cas, la fonction personnalisée avec le pattern virtuel d'origine en termes de temps d'exécution, et d'autres performances.

III.3 Conception du système

Avant de comparer les API, on va voir comment indexer le contenu des documents RDF. On a choisi de procéder le plus simplement possible, donc en collectant tout les triplets des sources voulues indexer. Ensuite capturer les ressources, littéraux et prédicats de ces derniers. Pour enfin indexer chacun comme un document à part dans notre index.
L'avantage qu'on aura en procédant ainsi est que les nœuds dupliqués dans les graphes ne seront transformés qu'en un seul document dans l'index, ce qui réduit la taille de l'index et réduit le temps de l'exécution de la requête.
En d'autres parts, les prédicats et les ressources concrètes sont considérés de la même manière vu que chacun représente une URI pour Lucene. On les a donc traité de la même sorte, en leur attribuant un seul attribut qui est "uri"et qui contient l'URI de la ressource ou du prédicat en question. De même, pour les littéraux, on associe un seul attribut nommé "text" pour contenir l'étiquette contenu dans le littéral.
STAFF a pour but de comparer entre les quatre implémentations deux par deux (ceux de Jena et ceux de Sesame). Il est donc intéressant voir même obligatoire de pouvoir se comporter avec les quatre implémentations d'une manière uniforme malgré l'hétérogénéité de leurs approches et des nôtres.
On a donc mis une classe abstraite représentant un API quelconque, quatre classes concrètes qui l'héritent (quatre selon le nombre d'implémentations qu'on a) mais indirectement vu que deux classes abstraites héritent d'AbstractAPI et qui représentent un API quelconque de Jena et de Sesame. Le figure III.1 montre un diagramme de classes représentant la hiérarchie des API sans les détails concernant leurs attributs et leurs méthodes.

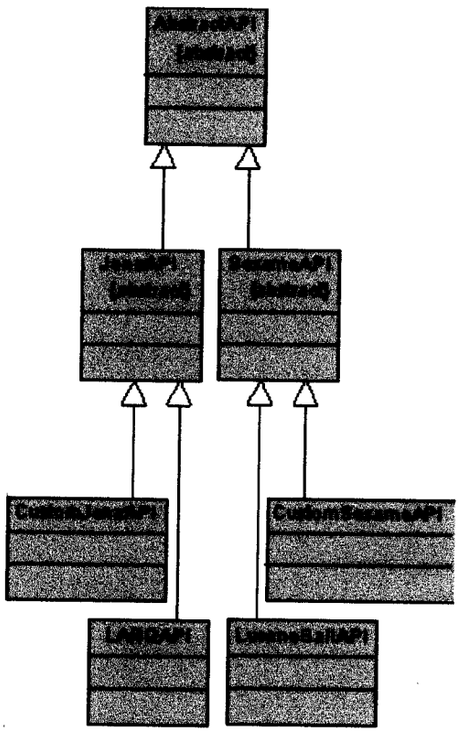

FIGURE III.1 – Vue globale sur les APIs

Les méthodes abstraites que contient la classe AbstractAPI sont evaluate et afficherResultats, c'est pour cette raison qu'elles se répètent au moins une fois dans la hiérarchie. La figure III.2 représente la classe AbstractAPI avec ses attributs et ses méthodes.

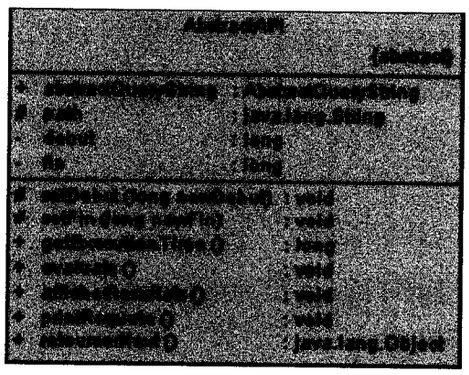

FIGURE III.2 – Classe AbstactAPI

– On va parler de l'attribut abstractQueryString un peu plus tard dans ce mémoire, mais on peut tout de même dire qu'il représente la requête à exécuter par l'une de nos implémentations.

- L'attribut path représente le chemin absolu du fichier qu'on va interroger.
- L'attribut debut représente le temps de début de l'évaluation.
- L'attribut fin désigne la fin de cette évaluation.

A part les getters et les setters des divers attributs de la classe, on remarquera qu'il ne reste que les méthodes :

- printRequete pour afficher la requête que va exécuter l'objet.
- evaluate qui va évaluer la requête abstractQueryString.
- afficherResultats qui va tout simplement numéroter et afficher les résultats de l'évaluation.
- retournerNext renvoi le résultat suivant parmis l'ensemble des interprétations restituées par l'une des implémentations.

La classe AbstractQueryString représente une requête qu'on va exécuter, elle contient donc deux attributs de type chaine de caractères représentant respectivement les entêtes de la requête et les conditions que doivent vérifier ses résultats. On a appelé ces attributs entetes et core (de la classe String).

Parmi ses méthodes, il existe :

- Une méthode pour ajouter un espace de nommage.
- Une méthode pour retourner la requête de façon complète sous forme de chaine de caractères.
- Une méthode pour construire ou débuter la requête, car les requêtes commencent toutes de la même façon quelque soit l'implémentation.
- Une méthode pour ajouter une requête (un pattern).

La seule méthode abstraite est celle qui ajoute des filtres ce qui fait de la classe Abstract-Querystring une classe réellement abstraite comme son nom l'indique.

Les classes qui étendent AbstractAPI sont au nombre de quatre, LARQAPI et Lucene-SailAPI sont désignés respectivement pour l'implémentation d'une recherche plein texte dans Jena et dans Sesame. Les deux autres sont CustomJenaAPI et CustomSesameAPI qui représentent respectivement nos approches dans les normes Jena et Sesame. Ces quatre classes concrètes n'héritent pas directement de la classe abstraite parce qu'elles ont le même comportement deux à deux. Alors on a mis deux classes abstraites qui héritent directement de la classe AbstractAPI et qui encapsulent le comportement commun entre l'approche personnalisé et l'approche d'origine de chaque implémentation. La figure III.3 représente la même hiérarchie de classes montrée dans la figure III.1 avec les attributs et

les méthodes.

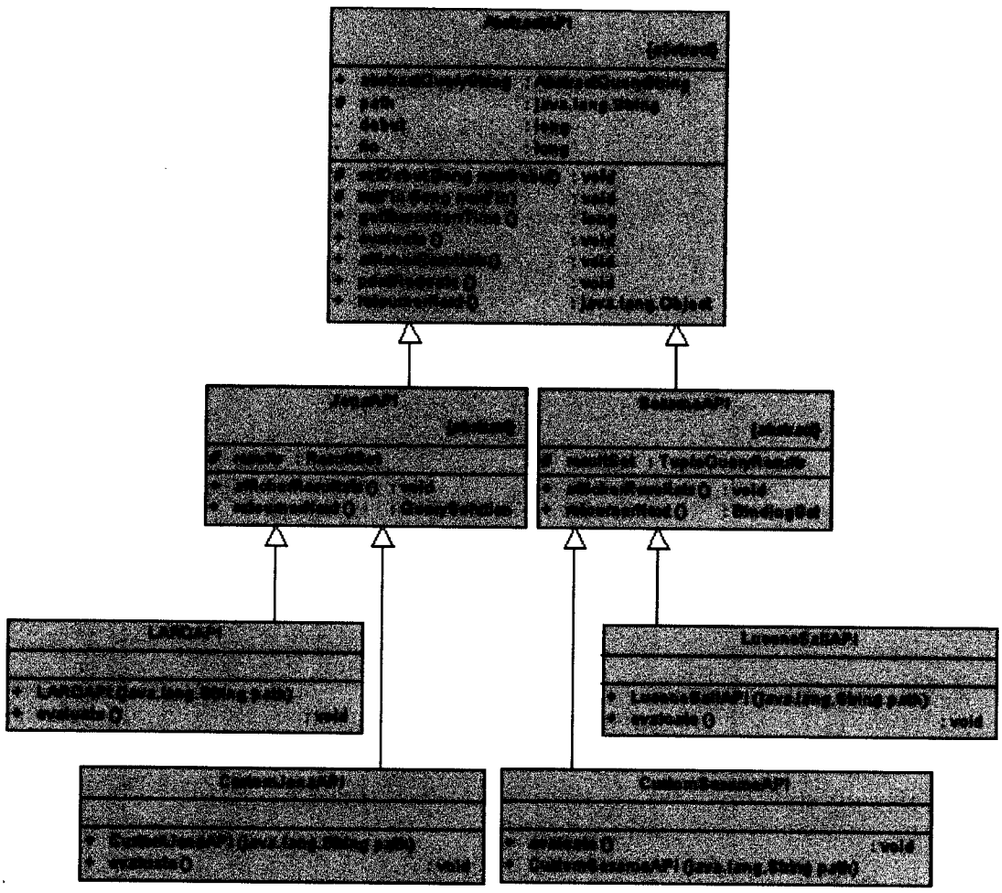

FIGURE III.3 – Vue sur les classes

Procédant de haut en bas. On remarque d'abord que c'est une hiérarchie et que les seules classes concrètes sont les feuilles de l'arbre. Ce qui veut dire que si l'on veut instancier un objet de l'une de ces classes, ça sera sûrement l'une des sous classes LARQAPI, LuceneSailAPI, CustomJenaAPI et CustomSesameAPI, donc l'une des implémentations directes d'une recherche plein texte que ça soit d'origine ou personnalisée, sur Jena ou sur Sesame.

Pour garantir une certaine modificabilité de notre architecture, on a veillé à ce que le code se répète au minimum. On a donc réuni dans la classe mère tout ce qui est nécessaire pour le déroulement de n'importe quelle requête dans n'importe quel API choisi (ce qu'on vient de dire un peu avant).

Vu que les implémentations dans Jena et celles dans Sesame diffèrent dans le comportement et les ressources nécessaires pour exécuter une requête, il est donc préférable de

faire hériter de la classe AbstractAPI au moins deux classes qui sont dans notre cas, JenaAPI et SesameAPI et qui représentent respectivement tout ce qui est nécessaire pour exécuter une requête dans Jena et dans Sesame. Après avoir choisi l'API, il faut choisir si l'approche soit d'origine ou personnalisée, on a donc préféré mettre les deux classes abstraites.

Ces deux classes contiennent chacune les résultats retournés par l'évaluation d'une requête SPARQL, et des méthodes permettant de les manipuler comme celles qui les affichent et celles qui retournent le suivant.

On arrive enfin à la base de notre pyramide qui présente les classes concrètes. Puisqu'elles diffèrent dans l'évaluation, on a préféré donner à chacune une méthode pour évaluer. Chacune d'elles contient son propre constructeur à qui on associe le chemin absolu de la ressource RDF voulu indexer et interroger. La figure III.4 représente tout le coeur de STAFF mais sans les détails concernant les attributs et les classes.

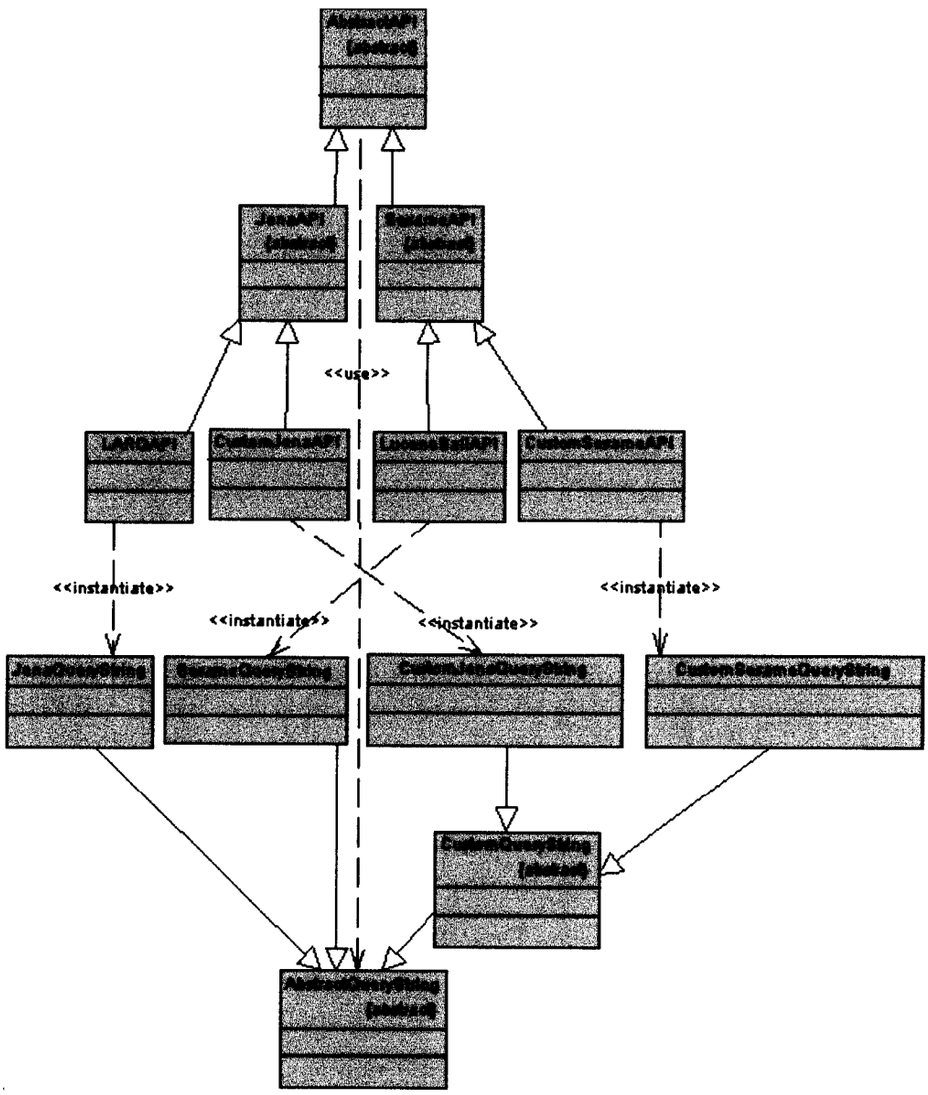

FIGURE III.4 – Schéma global

La classe AbstractAPI comprend un attribut de la classe AbstractQueryString. En conséquence, elle utilise cette classe. Les autres classes concrètes qui héritent d'AbstractAPI, ne font qu'instancier les classes concrètes qui héritent de la requête abstraite.

III.4 Implémentation du système

III.4.1 Environnement de dévelopement

Nous avons développé notre application sur une machine Intel (R) Core (TM) i7, muni d'une fréquence d'horloge de 2.20 GHz et d'une mémoire cache de 6 MO. Cet ordinateur a une mémoire vive de 4 GO et un seul système d'exploitation Windows 7 (x64 bits). STAFF a été implémenté en langage Java vu que c'est un langage purement orienté objet ; ce qui nous a aidé à mettre en oeuvre les deux hiérarchies de classe avec les différentes relations entre elles.

On a utilisé plusieurs bibliothèques telles que ARQ-2.8.8 pour LARQ et openrdf-sesame-2.6.5 pour LuceneSail et Lucene-3.6 pour implémenter le serveur d'indexation Lucene.

III.4.2 Présentation du prototype

La figure III.5 représente l'interface de la fenêtre principale au lancement du prototype.

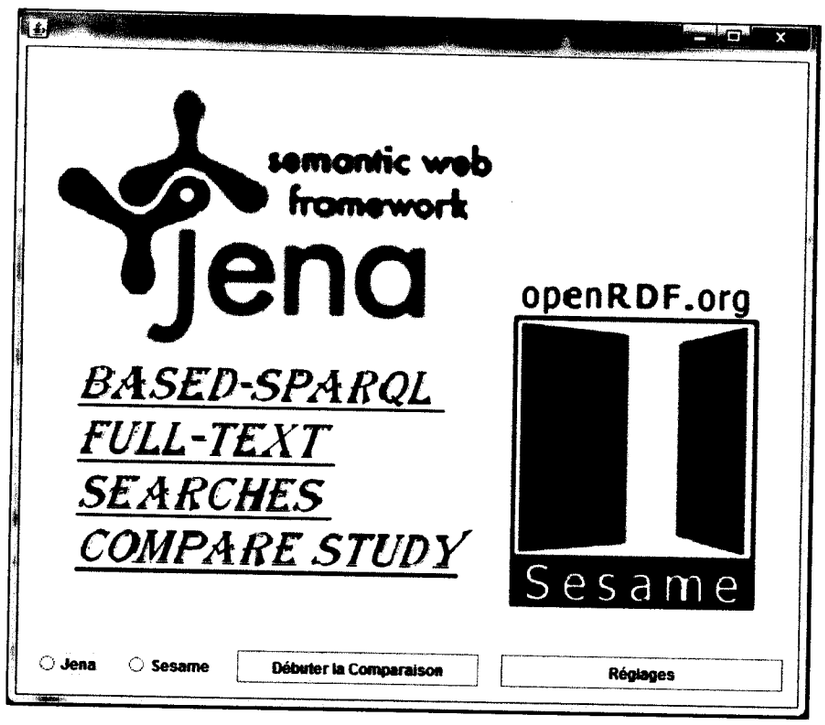

FIGURE III.5 – Fenêtre principale

La figure III.6 montre la fenêtre responsable des réglages de la requête à exécuter. Elle apparait en cliquant sur le bouton "Réglages".

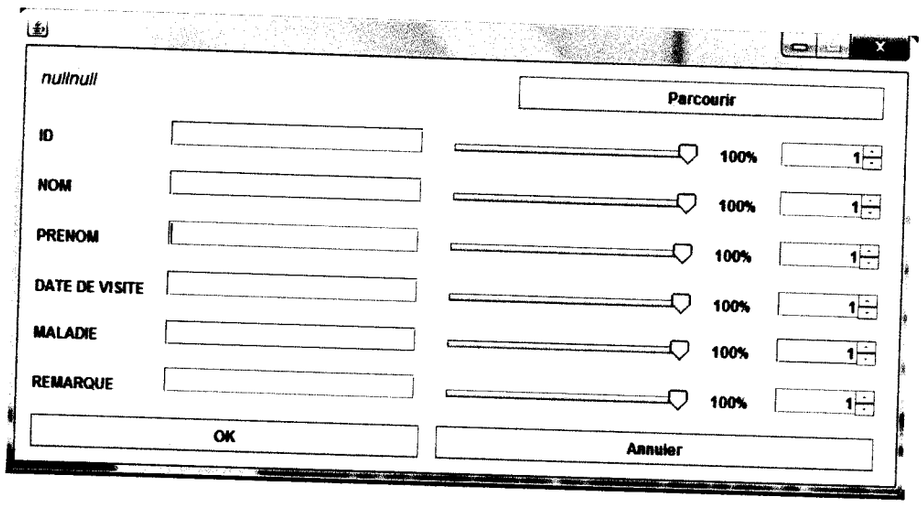

FIGURE III.6 – Fenêtre des réglages

Supposant qu'on veuille chercher les patients dont le nom de famille commence par la lettre 's' comme "Settouti" et "Saouli" et dont le prénom commence par la suite de caractères "ya" comme "Yassine" et "Yasser". La figure III.7 montre comment régler cette recherche.

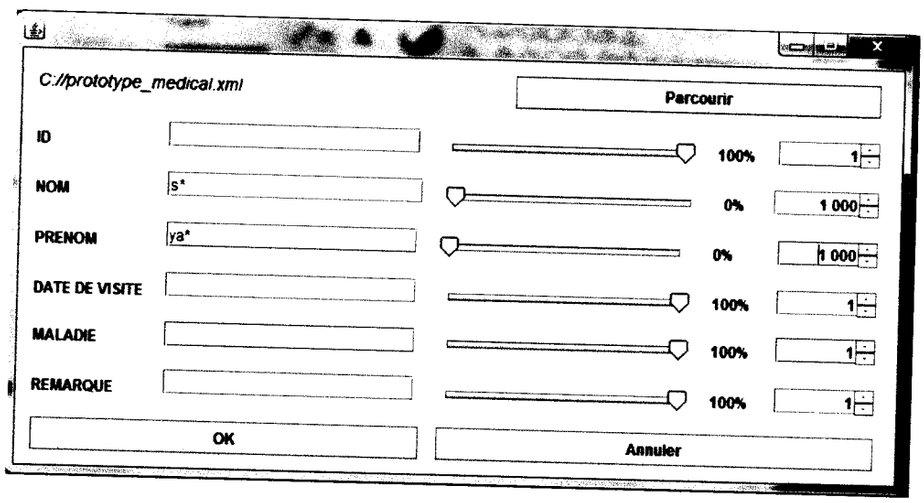

FIGURE III.7 – Réglages faits

Une fois le formulaire rempli, on peut choisir l'API et enclencher le test. Il suffit de cliquer sur le bouton "Débuter la comparaison" de la fenêtre principale. La figure III.8 montre le premier résultat de la recherche.

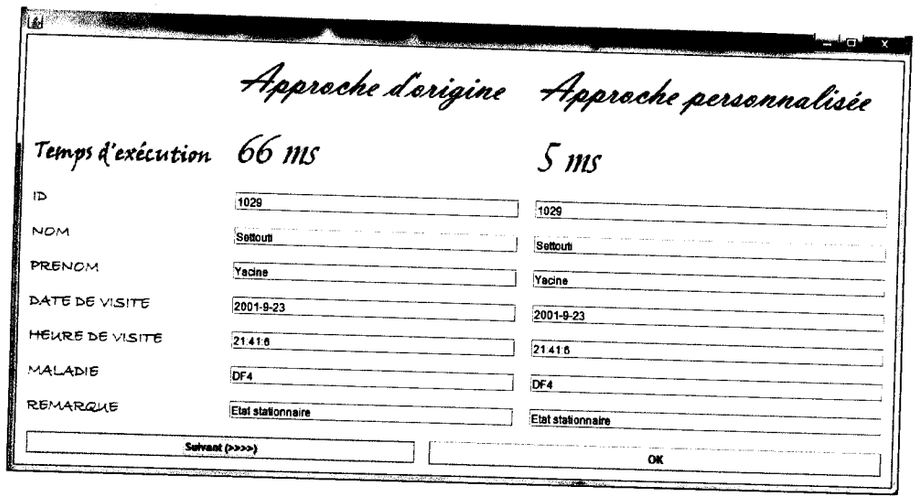

FIGURE III.8 – Premier résultat pour Jena

Le patient à gauche a été trouvé par l'approche d'origine de Jena et celui à droite par l'approche personnalisée. On peut balayer les résultats avec le bouton "Suivant (>>>>)". La figure III.9 montre l'un des résultats suivants de la recherche lancée.

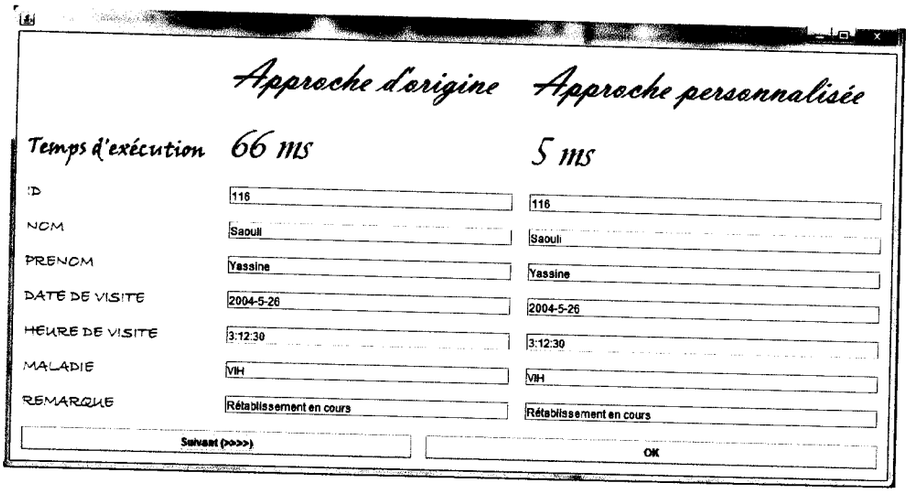

FIGURE III.9 – Résultat suivant pour Jena

On remarque que tous les patients restitués par notre moteur de recherche ont un nom de famille commençant par la lettre 's' et un prénom par la suite "ya".

III.5 Evaluation du système

Les tests désignés dans cette section ont été exécutés sur un ordinateur portable de marque DELL possédant 3Go de mémoire vive avec un processeur Core2Duo de 2.10 GHZ de fréquence d'horloge et de 4Mo de mémoire cache. Tout ça, sous un service pack 3 de Windows.

Puisqu'on a réussi à faire une recherche plein texte dans les deux moteurs Jena et Sesame sans faire appel à leurs propres outils, il fallait vérifier que notre approche était suffisament performante que celle d'origine dans chacun des deux moteurs. On a donc fait des tests sur chacun des critères de génie logiciel qui nous semblaient avantageux.

Les tests ont été exécutés sur un fichier comportant 70000 ressources représentant des patients. Chaque ressource possède un identifiant, un nom, un prénom et une autre ressource représentant sa dernière visite. Chaque visite possède une date, une maladie et la remarque du médecin.

Ce schéma a été inspiré de celui du projet "PROTO URGENCE", validé le 28 Mars 2010 par les sociétés arisem, SWORD, antidot et MONDECA. Et qui avait comme de restaurer l'essentiel des données à partir de données RDF médicales en plein texte à un moment d'urgence.

III.5.1 Sesame VS STAFF

Précision et Simplicité

Dans l'approche d'origine de Sesame, on ne fait pas de recherche plein texte selon un littéral, mais selon une ressource. Les résultats de ce type de recherche dans Sesame sont des ressources directement liées à un littéral répondant à la requête désignée. C'est assez bien et assez simple quand on ne sait pas quel littéral choisir mais moins précis quand on veut rechercher dans un seul littéral précis et pas dans les autres.

En d'autres termes, si on choisit la simplicité dans Sesame, on sacrifiera la précision et vise versa. Dans STAFF, on a la possibilité d'avoir les deux critères en même temps.

Procédons par un exemple pour éclaircir les choses :

Si on voudrait chercher les patients nommés " taleb yassine". Dans Sesame, on aura deux choix, soit :

$PREFIX\ search :< http : //www.openrdf.org/contrib/lucenesail\# >$

$PREFIX\ prop :< http : //www.example.org/properties\# >$

$SELECT *$

$WHERE\{$

$?x\ prop : nom\ ?nom.$

$?x\ prop : prenom\ ?prenom.$

$?x\ search :\ matches\ ?match.$

$?match\ search : query\ "taleb\ yassine";$

$search : score\ ?score.$

$FILTER\ (?score > 0.0).\ \}$

Ou bien :

$PREFIX\ search :< http : //www.openrdf.org/contrib/lucenesail\# >$

$PREFIX\ prop :< http : //www.example.org/properties\# >$

$SELECT *$

$WHERE\{$

$?x\ prop : nom\ ?nom.$

$?y\ prop : prenom\ ?prenom.$

$?x\ search : matches\ ?match1.$

$?match1\ search : query\ "taleb";$

$Search : score\ ?score1.$

$FILTER\ (?score1 > 0.0).$

$?y\ search : matches\ ?match2.$

$?match2\ search : query\ "yassine";$

$Search : score\ ?score2.$

$FILTER\ (?score2 > 0.0).$

$FILTER\ (?x =?y)\ \}$

Dans le premier cas, Sesame va retourner d'abord les patients nommés "Taleb Yassine" puis les patients ayant un nom de famille "Taleb" et enfin les patient ayant le prénom de "Yassine". C'est bien, vu que la requête est simple par rapport à la deuxième mais qui ne répond pas exactement ou précisément au besoin demandé. Dans la deuxième approche, le besoin est satisfait vu Sesame nous retourne que les patients nommés "Taleb Yassine" mais la requête n'est pas aussi simple que la première.

L'équivalent de ces requêtes dans STAFF sur le moteur Sesame est :

$PREFIX\ ct :< http : //example.org/customtools/ >$

$PREFIX\ prop :< http : //www.example.org/properties\# >$

$SELECT *$

$WHERE\{$

$?x\ prop : nom\ ?nom.$

$FILTER\ (ct : repondantRequete(?nom,' taleb', 0.0, 1)).$

$?x\ prop : prenom\ ?prenom.$

$FILTER\ (ct : repondantRequete(?prenom,' yassine', 0.0, 1)).\ \}$

On a combiné la précision et la simplicité. Pour une précision optimale, on a quatre lignes de code dans la rubrique WHERE au lieu de six ou onze pour LuceneSail.

On peut aller plus loin avec la précision en la mettant à 0.0 mais c'est aussi possible avec LuceneSail. Mais Sesame ne peut pas limiter le nombre de résultats à prendre en compte, et c'est possible dans STAFF. C'est un point de moins pour Sesame au niveau de la précision.

Rapidité et Taux de bugs

Pour tester la rapidité des deux approches dans Sesame, on va exécuter quatre requêtes :

- Les patients ayant comme nom de familles "Taleb ".
- Les patients ayant comme nom de familles " Taleb " et comme prénom " Yassine".
- Les patients ayant comme nom de familles " Taleb ", comme prénom " Yassine " et se sont rétablis de leurs maladies au cours de la dernière visite.
- Les patients ayant comme nom de familles " Taleb ", comme prénom " Yassine " qui se sont rétablis de la " Malaria " au cours de leur dernière visite.

La figure III.10 montre l'histogramme résultant de la comparaison.

On remarque que STAFF n'a pas dépassé une milliseconde lors de son temps d'exécution. Tandis que LuceneSail a fait des centaines de millisecondes. Il faut savoir que ce qui est représenté dans l'histogramme est un cumule des temps d'exécution à cinq reprises. Cela veut dire que chaque requête a été déroulée cinq fois. On a calculé la somme de ces temps après. L'excès temporel des autres requêtes par rapport à la troisième est dû au fait que

l'approche d'origine bug trop par rapport à l'approche personnalisée.
Pour répondre à la question de l'introduction générale, on peut dire qu'on peut arriver à un temps d'exécution égal à 0 millisecondes en implémentant l'approche STAFF dans Sesame.

Légèreté

Le principe de la légèreté se voit quand on essaie de comparer les deux approches dans un même programme, LuceneSail plante en disant

$$"java.lang.OutOfMemoryError : Javaheapspace"$$

alors que STAFF ne se plante jamais.

Complétude de l'index

LuceneSail n'indexe que les littéraux, alors que STAFF indexe les prédicats et les ressources aussi. Ceci aura comme avantage l'élargissement du champ des recherches en texte intégral.

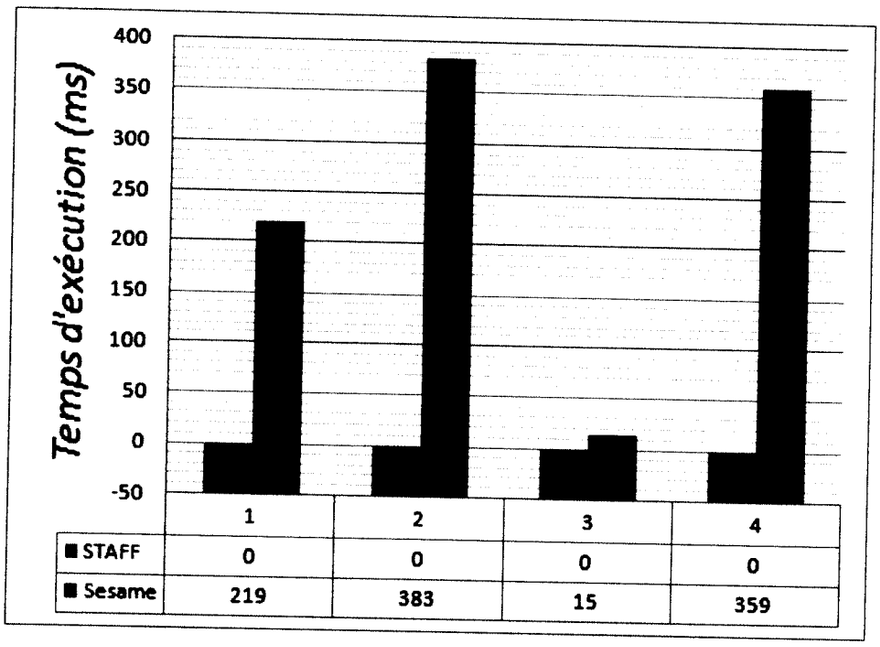

FIGURE III.10 – Sesame VS STAFF

III.5.2 Jena VS STAFF

Scalabilité

On a remarqué que STAFF gagne en temps d'exécution par rapport à Jena quand on emploie des requêtes complexes ou peu complexes. Pour prouver ceci de manière pratique, on a d'abord comparé selon une requête large puis spécifique :

- L'ensemble des patients ayant :
  - un identifiant commençant par « 73 ».
  - un nom de famille « Taleb ».
  - un prénom commence par « ya ».
  - a fait sa dernière visite en 2009.
    - pour la maladie de Malaria.
    - rétabli à la fin de visite ou en phase de rétablissement.
- L'ensemble des patients ayant :
  - un identifiant commence par « 73 ».
  - nommés Taleb Yassine.
  - ayant la dernière visite
    - en 2009.
    - sur la maladie de Malaria.
    - fini par un rétablissement ou une phase de rétablissement.

La figure III.11 montre l'histogramme de la comparaison.

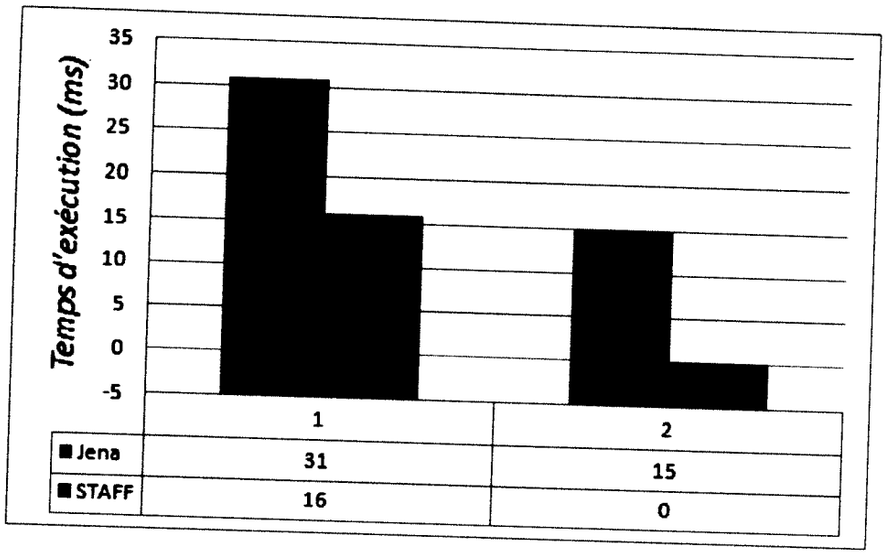

FIGURE III.11 – Jena VS STAFF

Ceci s'explique au fait que Jena n'indexe pas à notre façon les fichiers RDF interrogés. Nous nous contentons d'écraser un ancien document si le nouveau est le même, alors que Jena ne le fait pas. En plus, Jena indexe les paires de prédicats et objets, ce qui donne à l'index une redondance et une taille importante, et qui ne facilite pas la tache de la recherche d'information.

Pour répondre à la question de l'introduction générale, on peut dire qu'on peut arriver à un temps d'exécution égal à 15 millisecondes en implémentant l'approche STAFF dans Jena. On parle biensur de cas dont STAFF a dépassé LARQ en termes de temps d'exécution.

III.5.3 TrilpleStores VS STAFF

Dans cette partie, nous allons parler de l'efficacité de notre approche par rapport à n'importe quel restaurateur de triplets.

Dépendance de la syntaxe

La syntaxe utilisée pour appeler la fonction filtre, n'a pas une dépendance envers le moteur utilisé. On peut écrire la même fonction dans un autre moteur, il l'acceptera comme telle

vu qu'elle n'est dépendante que du langage SPARQL. Quand on dit que la syntaxe ne change pas, on veut parler des arguments, leurs types, et leur ordre.

Intervention minimale de la plateforme

Lors de la recherche plein texte, le moteur ne fait que l'appel de la fonction permettant de tester si l'élément est vérifié ou pas. Tout se passe avec Lucene, le moteur ne fait que l'appel.

L'indépendance par rapport à l'existence d'outils de recherche plein texte

Le moteur n'a pas besoin d'avoir un outil permettant une recherche plein texte pour pouvoir exécuter notre approche. Il suffit qu'il supporte le langage SPARQL pour le faire.

III.5.4 Tableau récapitulatif

Le caractère + veut dire que l'approcche personnalisée est avantagée par rapport à celle d'origine, le − veut dire le contraire et le = veut dire qu'elles sont égales pour ce critère. Le tableau III.1 propose un résumé de la comparaison sous une forme adéquate.

Critères	Sesame	Jena
Précision	+	=
Simplicité	+	=
Rapidité	+	+
Légerté	+	=
Taux de bugs	+	=
Scalabilité	+	+
Complétude de l'index	+	+
Légerté de l'index	=	=
Dépendance de la syntaxe par rapport au moteur	+	+
Intervention du moteur (minimale)	+	+
Indépendance aux outils de recherche plein texte	+	+

TABLE III.1 – Récapitulatif général

III.6 Conclusion

Dans ce chapitre, nous avons d'abord présenté brièvement notre travail. Ensuite, nous avons vu comment a-t-on modéliser STAFF. Après, nous avons déroulé un exemple d'ap-

plication de notre système. Enfin, nous avons présenté un tableau récapitulatif pour concrétiser au mieux l'étude comparative.

Conclusion générale et perspectives

Notre travail a consisté dans le fait de comparer entre deux moteurs d'interrogation de sources RDF. Puis à comparer entre la recherche plein texte du moteur avec une nouvelle implémenté dans ce dernier.

Notre travail a ses avantages, comme le fait de connaitre :

- Les points forts des plateformes.
- Les points faibles des moteurs.
- La différence entre les deux outils.
- Les avantages de l'un par rapport à l'autre.
- Les inconvénients de l'un par rapport à l'autre.
- Les domaines d'utilisation de chacun.

Mais comme tout travail, il a ses limites comme :

- Se limiter aux deux moteurs Jena et Sesame mais qui sont considérés comme les plus utilisés.
- Se limiter aux moteurs incorporant le langage SPARQL mais considéré comme le langage le plus utilisé.
- Se limiter aux moteurs utilisant Lucene mais considéré comme le serveur d'indexation plein texte le plus fiable.
- Se limiter à quelques critères de comparaison mais nécessaires

On peut palier à ses limites en :

- Etendant la comparaison envers des outils moins utilisés comme Mulgara et AllegroGraph.
- Passer même à d'autres langages de manipulations de triplets SeRQL, RDQL... etc.

- Passer à des moteurs utilisant d'autres principes d'indexation plein texte, voir même ceux qui ne fournissent pas d'outils de recherche plein texte.
- Essayer de toucher un large ensemble de critères de génie logiciel lors de la comparaison entre l'approche personnalisée et l'approche d'origine pour chaque moteur.

On peut aussi comparer entre les approches personnalisées de chaque moteur, mais ça ne servira pas à grand-chose vu que l'intervention du moteur dans les recherches plein texte est minimale.

Bibliographie

[1] Geoffrey Andogah and Gosse Bouma. Geoclef. Technical report, Computational Linguistics Group, Centre for Language and Cognition Groningen (CLCG), University of Groningen, Groningen, The Netherlands, 2007.

[2] H. James B.-L. Tim and L. Ora. The semantic web. In *Scientific American*, May 2001.

[3] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. Technical report, Freie Universitat Berlin, Web-based Systems Group, Garystr. 21, 14195 Berlin, Germany, 2010.

[4] A. Boukhadra. La composition dynamique des services web sémantiques à base d'alignement des ontologies owl-s. Mémoire de magister, Ecole national Supérieur d'Informatique, Alger, 2011.

[5] P.E et al. Bourne. The distribution and query systems of the rcsb protein data bank. *Nucleic Acids Research*, 32(90001) :0–end, 2004.

[6] Radim Cebis. Web information systems. Technical report, Semantic Web, 2007.

[7] Abraham Bernstein Christoph Kiefer and Markus Stocker. The fundamentals of isparql : A virtual triple approach for similarity-based semantic web tasks. Technical report, Department of Informatics, University of Zurich, Switzerland, 2008.

[8] G. Costantini. Produit scalaire et produit cartésien dans l'espace euclidien.

[9] Wolf Siberski Enrico Minack and Wolfgang Nejdl. Benchmarking fulltext search performance of rdf stores. Technical report, L3S Research Center Leibniz Universitat Hannover 30167 Hannover, Germany, 2010.

[10] Martin Filliau. Semantic mashup to query localised data. Technical report, Oxford Brookes University School of Technology, 2011.

[11] Thomas Francart. Rdf : Sesame, jena, comparaison des fonctionnalitées, Mai 2012.

[12] http ://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail. Lucenesail nepomuk, 2008.

[13] http ://docs.postgresql.fr/9.1/textsearch.html. Recherche plein texte. PostgreSQLFr, 2011.

[14] http ://jena.apache.org/. Apache jena, 2011/2012.

[15] http ://jena.sourceforge.net/ARQ/lucene arq. Larq - free text indexing for sparql, 2011.

[16] http ://lucene.apache.org/. Welcome to apache lucene, 2011.

[17] http ://lucene.apache.org/core/. Apache lucene core, 2011.

[18] http ://openrdf.org/doc/sesame/users/ch01.html. What's sesame.

[19] http ://www.openrdf.org/. Sesame (home), Juin 2012.

[20] http ://www.w3c.org/RDF/. Resource description framework (rdf), Février 2004.

[21] Marcelo Arenas Jorge Pérez and Claudio Gutierrez. Semantics and complexity of sparql. Technical report, Universidad de Chile, 2007.

[22] Christoph Kiefer. *Non-Deductive Reasoning for the Semantic Web and Software Analysis*. PhD thesis, University of Zurich, Octobre 2008.

[23] Rafal Kuc. *Apache Solr 3.1 Cookbook*. PACKT, 2011.

[24] Thierry Lecroq. Distance entre mots. Technical report, Université de Rouen, France, 2008.

[25] Aurélien Pontacq. Présentation d'apache solr. Technical report, Apache Solr, Juin 2009.

[26] Eric Prud'hommeaux. Langage d'interrogation sparql pour rdf, Janvier 2008.

[27] Konstantinos Tzonas. The jena rdf framework.

Résumé

Depuis l'apparition du web sémantique, les gens se sont pressés à l'utiliser. Du coup, des données RDF sont presque partout et en grand volume. En plus, les gens veulent aussi la possibilité de chercher dans ces graphes de la même façon que de chercher dans un document texte.
Notre objectif est d'appliquer une recherche plein texte dans une requête SPARQL. On a choisi Jena et Sesame vu qu'ils possèdent déjà un outil de recherche en texte intégral. On leur a ajouté une fonctionnalité semblable puis on l'a comparé avec l'outil existant. Les résultats étaient très satisfaisants car la nouvelle approche dans ces moteurs a démontré une rapidité dans la pluparts des cas, en plus d'autres critères qui sont importants pour les utilisateurs.
Cette approche ouvre une porte sur d'autres études comparatives plus larges couvrant plus de plateformes que Jena et Sesame. Notre approche se déroule uniquement sur Lucene.
Pour cela, la seule condition qu'un moteur doit vérifier est le support du langage SPARQL.

Mots-clés : SPARQL, recherche plein-texte, RDF, Jena, Sesame, Lucene, fonction filtre.

Abstract

Since semantic web appearing, people seem be interested by this technology, by the fact, there are much RDF sources scattered in the world. On top of that, people want the possibility to search in these graphs like searching in a text document too. Our objective is to apply a SPARQL-Based full-text request search. To do it, we have chosen Jena and Sesame which contain already this feature, then add to them another to compare it with the first one. Results have been very satisfactory, the fact that the new approach have demonstrated speed compared with original full-text tool of the two engines, in addition to other important qualities for Jena's and Sesame's users. This work open doors to other larger studies covering others SPARQL-Based engines, because full-text search takes place especially on Lucene. The only thing that triple stores engines must have to accept our approach is to support SPARQL language.

Keywords : SPARQL, full-text search, RDF, Jena, Sesame, Lucene, filter function.

ملخص

منذ ظهور شبكة الويب الدلالي، وتحث الناس على استخدامها، ظهرت بيانات RDF في كل مكان تقريبا، وبحجم كبير. وإضافة إلى ذلك، الناس تريد أيضا القدرة على البحث على الرسوم البيانية بكيفية البحث في نص. هدفنا هو تطبيق بحث النص الكامل بالاستعلام SPARQL . للقيام بذلك، اخترنا Jena وSesame والذين يمتلكون بالفعل هذه الأصول، ولكن أضفنا آخر ومن ثم مقارنتها مع القائمة. وكانت النتائج مرضية للغاية نظرا إلى أن هذا النهج الجديد في هذه المحركات قد أثبتت سرعة في معظم الحالات، إضافة إلى معايير أخرى التي تعتبر مهمة لمستخدمي هذه المحركات. هذا النهج يفتح بابا أمام دراسات أخرى و مقارنات أكبر تغطي منصات أخرى بدلا من Jena وSesame رأينا أن نهجنا تتم بشكل كامل على Lucene والشرط الوحيد هو أن المحرك يقبل SPARQL

الدالة الكلمات : فلتر وظيفة, RDF, SPARQL, النص الكامل بحث, Lucene, Jena, Sesame,