EXGRAF: Exploration et Fragmentation de Graphes au Service du Traitement Scalable de Requêtes RDF

Abdallah Khelil[1,3], Amin Mesmoudi[2], Jorge Galicia[1], Ladjel Bellatreche [1]

[1]LIAS, ISAE-ENSMA
(abdallah.khelil, bellatreche, jorge.galicia)@ensma.fr,
[2]Université de Poitiers
amin.mesmoudi@univ-poitiers.fr
[3]Université Oran 1, Algérie

Résumé. Les facilités de représentation de données offertes par RDF ont largement contribué à son succès et sa standardisation. Il est le langage incontournable pour le Web, la Biologie, etc. En allégeant la notion de schéma, il offre une représentation flexible des données. Son adoption rapide par les fournisseurs des données a contribué à la multiplication des masses de données RDF nécessitant à la fois d'un traitement efficace et scalable. Pour satisfaire ces besoins, plusieurs systèmes ont été proposés que nous divisons en deux catégories principales : (1) les systèmes orientés-mémoire comme gStore et (2) les systèmes orientés-disque comme RDF-3X et Virtuoso. Les systèmes de la 1ère catégorie sont très gourmands en mémoire surtout lorsqu'il s'agit de traiter une masse de données RDF. Ceux qui appartiennent à la 2ème catégorie utilisent la table traditionnelle en changeant automatiquement structure logique d'une donnée RDF. En conséquence, ils sont moins performants pour des requêtes complexes. Dans cet article, nous proposons une nouvelle approche orientée-disque (appelée EXGRAF) de traitement de requêtes RDF. Elle est basée sur l'EXploration logique de graphe RDF et la Fragmentation physique de triplets RDF. Nos expérimentations en utilisant des jeux de données réelles et synthétiques montrent le bon compromis entre l'efficacité et le passage à l'échelle de EXGRAF.

1 Introduction

Les systèmes traditionnels de stockage et de gestion des données ont été largement impactés par les nouveaux fournisseurs de données dites "graphes" comme le Web, le Web des données, les réseaux sociaux, la biologie. Leur approche consistant à définir un *schéma rigide (une structure)* dédié au stockage des données est rapidement devenue contraignante (Zou et al., 2011). Pour palier ce problème et satisfaire la structure graphe des données, de nouvelles représentations de données ont émergé. Resource Description Framework (RDF) est l'un des efforts menés par le World Wide Web Consortium (W3C) pour relier les données du Web. Il utilise la notion de triplets qui consiste à représenter chaque information sous forme d'un triplet $< sujet$, *prédicat*,$objet >$. Cette structure offre une flexibilité dans la collecte de données. Son adoption rapide par les fournisseurs des données a contribué à la multiplication des

masses de données RDF nécessitant un traitement efficace et scalable. Pour motiver nos propos, prenons le cas de trois graphes de connaissances : *Freebase* [1] qui regroupe 2,5 milliards triplets (Bollacker et al., 2008), DBpedia[2] avec plus de 170 millions triplets (Lehmann et al., 2015). Le Cloud Open Data (LOD) relie plus de 10 000 jeux de données et regroupe plus de 150 milliards de triplets[3]. Le nombre de sources de données RDF a doublé durant la période (2015-2018).

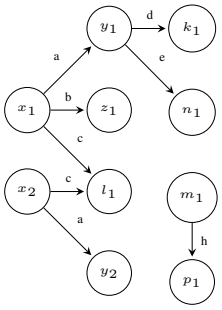
Le langage SPARQL a largement contribué à l'exploitation facile des données RDF. En même temps, il a poussé la communauté à proposer des techniques de traitement de requêtes efficaces. Cela est dû principalement à la non-adaptation directe des systèmes de gestion des données traditionnelles pour la gestion des triplets. Ces systèmes ne sont pas en mesure de garantir à la fois le passage à l'échelle et la performance de requêtes pour deux raisons principales : **(i)** l'absence d'un schéma explicite dans une base de données RDF et **(ii)** la présence d'un nombre élevé d'auto-jointures que pourrait engendrer une requête SPARQL.

L'explosion des sources de données RDF a poussé les chercheurs et les industriels à proposer des systèmes de traitement de données RDF que nous pouvons classer en deux catégories principales. Les systèmes orientés-disques et fortement inspirés des SGBDs traditionnels (Neumann et Weikum, 2008; Weiss et al., 2008). Dans ces systèmes, les triplets RDF sont stockés dans une grande table de trois attributs ($< sujet$, *prédicat*, $objet >$) comme dans la Figure 1. Le fait qu'il y ait une seule table, la majorité des requêtes SPARQL sont transformées en une requête SQL invoquant des auto-jointures coûteuses nécessitant n^2 opérations (pour une seule auto-jointure, où n représente la taille de table). Pour optimiser ces opérations, le recours aux index est indispensable. Certes, ces derniers améliorent les performances de certaines requêtes, mais ils introduisent des frais supplémentaires pour leur maintenance et leur stockage. D'autres techniques d'optimisation ont été proposées comme le regroupement de triplets par prédicat (tables de propriétés) (McBride, 2002) permettent d'améliorer les performances de certaines requêtes (sous forme d'étoile). Les systèmes de la deuxième catégorie comme *gStore* (Zou et al., 2011) sont orientés-mémoire. Le processus d'évaluation de requêtes est équivalent à la recherche de correspondances au sein d'un graphe. Ces systèmes conservent la représentation originale des données RDF, mais ils doivent gérer la mémoire pour établir ces correspondances, surtout pour des graphes à large échelle. En conséquence, ces systèmes ne passent pas souvent à l'échelle et ne garantissent pas une meilleure performance de leurs requêtes. Cette analyse nous motive à proposer une approche d'évaluation de requêtes SPARQL qui passent à l'échelle et satisfait la performance de requêtes.

Dans cet article, nous proposons nouvelle approche, appelée EXGRAF qui exploite les avantages de chaque catégorie de systèmes ci-dessus discutés. Plus concrètement, EXGRAF est une approche orientée-disque et qui fait appel à la fragmentation et l'indexation pour réduire le nombre d'entrées/sorties (E/S) et à l'exploration du graphe pour conserver la structure logique d'une donnée RDF. Pour l'évaluation de requêtes SPARQL, EXGRAF utilise une stratégie fortement inspirée du modèle *Volcano* Graefe (1994). Ce dernier offre un environnement riche pour concevoir des systèmes de bases de données, les heuristiques pour optimiser et exécuter en parallèles des requêtes et allouer efficacement la mémoire.

Ce papier est organisé comme suit : nous discutons dans la section 2 les travaux connexes. Ensuite, la section 3 détaille et illustre les différents processus de notre approche EXGRAF.

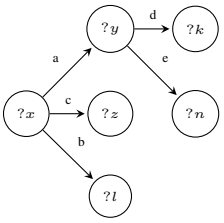
1. `http://www.freebase.com`, 2. `http://wiki.DBpedia.org`, 3. `http://lodstats.aksw.org`



(a) Graphe logique

S	P	O
x_1	a	y_1
x_1	b	z_1
x_1	c	l_1
x_2	c	l_1
x_2	a	x_1
y_1	d	k_1
y_1	e	n_1
m_1	h	p_1

(b) Table de triplet



(c) Requête

FIG. 1: Exemples de représentation RDF

Nous présentons notre étude expérimentale dans la section 4. Enfin, la section 5 conclut ce papier et donne un aperçu de nos travaux futurs.

2 État de l'art

Les SGBDs ont largement contribué au succès de la technologie de bases de données transactionnelles (e.g. les applications bancaires). Ils doivent fournir le même succès aux nouvelles applications autour du Web, la Biologie et l'Astronomie. Dans ces dernières, le schéma de leurs bases de données n'est pas toujours connu à l'avance et change souvent.

Plusieurs travaux ont identifié les limites des SGBD traditionnels qui sont liées à la performance de requêtes et la capacité de gérer des données dynamiques. Les données dans ces nouveaux domaines ont deux caractéristiques principales : (i) elles sont souvent représentées sous forme de graphe et (ii) les opérateurs d'évaluation de requêtes sont basés principalement sur l'exploration des graphes à l'aide des *arcs entrants* et *sortants*.

2.1 Approches relationnelles

Les premières approches utilisées pour traiter les données RDF proposent de stocker et d'interroger des triplets à l'aide d'un SGBD traditionnel (Cyganiak, 2005). Cela nécessite un stockage des triplets dans une seule table relationnelle et la traduction de requêtes SPARQL en SQL. Chaque triplet SPARQL génère une auto-jointure SQL, ce qui implique un nombre important d'auto-jointures difficile à optimiser.

Approches indexées intensives La deuxième catégorie d'approches (*i.e.* RDF-3x (Neumann et Weikum, 2008), Hexastore (Weiss et al., 2008)) s'appuie sur l'indexation excessive de données. En effet, les chaînes de caractères liées aux triplets sont remplacées par des identifiants. Ces identifiants sont ensuite séparément stockés et indexés en utilisant des ordres différents (SPO, OPS, ...). Chaque ordre est stocké en tant qu'arbre de type B+ groupé. Dans ce type d'outils, les motifs de triplets (Stocker et al., 2008) peuvent être évalués par une requête de plage dont la jointure est très efficace grâce à l'utilisation de la jointure par fusion sur des données indexées. Les inconvénients de ce type d'approches sont liés principalement à : 1) l'utilisation de l'espace supplémentaire à cause de la réplication excessive de données, et 2) le coût exorbitant qui peut être engendré par l'analyse complète des structures utilisés (e.g. SPO).

Table de propriétés Cette stratégie a été proposée dans le cadre de Jena (McBride, 2002) et de DB2-RDF (Briggs, 2012). Ce type d'approches permet d'éviter certaines jointures pour les requêtes de type *Sujet-Sujet*. Cependant, il pose certains problèmes lors du traitement d'autres types de jointures. De plus, il ne prend pas en charge le stockage de sujets ayant des valeurs multiples pour la même propriété.

Tables binaires Une autre approche, basée sur les tables binaires (Abadi et al., 2007), a été proposée afin de combler les inconvénients des approches basées sur les tables de propriétés. Cette approche prend en charge les propriétés à valeurs multiples. Seules les données pertinentes sont stockées, *i.e.*, la gestion des valeurs NULL n'est pas nécessaire. De bonnes performances sont observées pour les jointures de type *Sujet-Sujet*, contrairement aux jointures de type Objet-Sujet.

2.2 Approches basées sur des graphes

Dans le contexte d'approches basées sur la recherche de correspondances de graphes comme dans le système gStore (Zou et al., 2011) qui utilise une liste adjacente pour stocker les propriétés. Il s'agit d'une table de propriétés qui utilise une liste pour ignorer les propriétés avec des valeurs *NULL*. Un index basé sur *S-Tree* (Deppisch, 1986) est utilisé pour accélérer les opérations de recherche de correspondances. Malheureusement, ces approches ne contrôlent pas la quantité de mémoire utilisée, ce qui induit un dépassement de capacité lors du traitement de certaines requêtes à la différence des approches basées sur la jointure.

Dans notre travail, nous adoptons un stockage sous forme de graphe. Comme nous avons des requêtes avec des prédicats constants, nous ne stockons que les arcs sortants et entrants en tant que SPO et OPS. Nous proposons aussi de fragmenter SPO et OPS, chaque fragment stocke un ensemble de sujets (dans le cas de SPO) ou d'objets (dans le cas de OPS) ayant les mêmes arcs (prédicats). Une technique d'élagage basée sur l'extraction des fragments est proposée afin de considérer que les données pertinentes. Chaque fragment est ensuite stocké séparément sous forme d'arbre B+ groupé. Nous proposons également un mécanisme d'évaluation permettant d'exploiter les différents types de structure de données. Enfin, nos opérateurs d'évaluation prennent en compte la quantité de mémoire disponible ce qui permet d'éviter tout débordement.

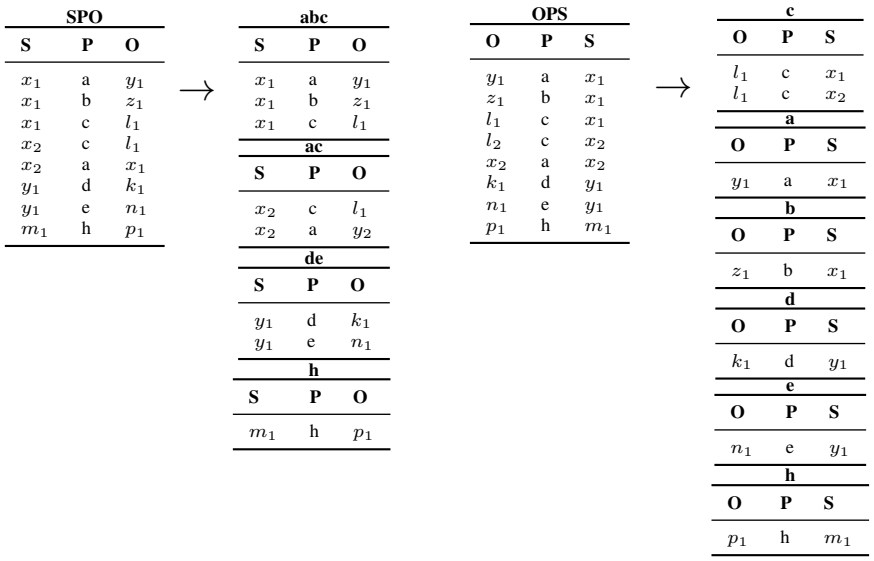
3 Notre approche : EXGRAF

Dans cette section, nous commençons par présenter un cadre formel permettant d'évaluer les requêtes SPJ (Select-Project-Join). En raison d'un manque de place, nous nous concentrons dans cet article que sur les requêtes impliquant des prédicats constants. Les autres requêtes de type Group by et Order By sont largement discutées dans notre rapport de recherche [2]. Ensuite, nous présentons notre technique de stockage de graphes. Finalement, nous discutons les techniques d'évaluation de requêtes utilisées par EXGRAF.

3.1 Stockage des graphes

Les techniques de stockage de graphes existantes proposent de matérialiser les arcs sortants, les arcs entrants ou les deux. En effet, le choix du type d'arcs à utiliser lors de l'évaluation des requêtes pourrait impacter fortement les performances.

2. https://www.lias-lab.fr/publications/32823/RapportderechercheKHELIL.pdf



(a) Fragmentation SPO (b) Fragmentation OPS

FIG. 2: Fragmentation de données

Dans cet article, nous avons opté pour les deux structures SPO et OPS. Un exemple de stockage d'un graphe sous forme SPO et OPS est donné dans les Figures 2a et 2b respectivement. Les triplets, représentés comme SPO et OPS, et sont stockés sous forme d'arbres B+ groupés. Cela permet une extraction de triplets avec un nombre d'accès disque de l'ordre de $log(n)$. Une analyse complète de ces structures n'est pas nécessaire pour les requêtes sélectives. Le problème majeur des SPO et OPS est qu'ils stockent des informations hétérogènes. On peut trouver par exemple dans le même graphe, des informations portant à la fois sur le sport et les transactions boursières.

Il est évident que dès qu'une requête est reçue par le système, elle va manipuler qu'un sous-ensemble de données qui reflète le même thème. Ce thème est généralement défini par un ensemble de prédicats. Nous proposons alors de diviser, en fragments, SPO et OPS par rapport aux prédicats qui couvrent les sujets dans le cas de SPO (ou objet dans le cas de OPS). Cela correspond à l'idée des ensembles de caractéristiques (Neumann et Moerkotte, 2011). Toutefois, les ensembles de caractéristiques sont utilisés uniquement pour collecter des statistiques sur les données. Il s'agit en effet de structures logiques.

Définition 1 ***(Fragment)*** *Un fragment est une partie de SPO (ou OPS) qui vérifie un ensemble de prédicats.*

Dans notre travail, les données sont physiquement stockées sous forme de fragments [3]. Dans le cas d'une requête (non sélective) où nous devons effectuer une analyse complète, seuls les fragments pertinents (qui satisfont la requête) seront analysés.

3. Dans la suite de cet article, nous utilisons le mot fragment au lieu d'ensembles de caractéristiques afin de désigner les fragments physiques de SPO ou OPS.

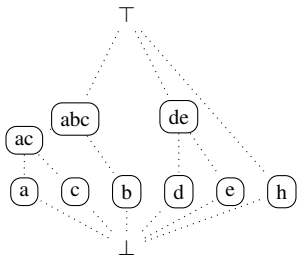


FIG. 3: Exemple : treillis SPO

Supposons que nous avons la requête suivante :

```
SELECT ?x ?y ?z WHERE { ?x a ?y AND ?x c ?z }
```

Nous avons deux fragments avec deux étiquettes $\{a, c\}$ et $\{a, b, c\}$. Il est clair que pour répondre à la requête, nous devons analyser les deux fragments. Pour trouver efficacement des fragments pertinents, nous indexons les étiquettes (l'ensemble de prédicat) des fragments en tant que des treillis (Aït-Kaci et al., 1989). On aura alors un treillis pour SPO et un autre pour OPS. La Figure 3 montre un exemple de treillis pour SPO.

L'opérateur de satisfaction de treillis permet de trouver les fragments pertinents en fonction d'un ensemble de prédicats donné comme paramètre. Chaque nœud du treillis référence d'ailleurs un ensemble de prédicats et une liste de fragments satisfaisant ces prédicats.Dans la section suivante, nous proposons une technique de compression de ces fragments

3.1.1 Compression

Comme expliqué précédemment, les triplets RDF sont organisés en plusieurs fragments. Chaque fragment est stocké en tant qu'arbre B+ groupé. Pour mieux explorer le graphe de données, nos méthodes d'évaluation doivent localiser les fragments hébergeant les arcs entrants et sortants du sujet et de l'objet de chaque triplet. Cela nous a poussé à étendre la représentation d'un triplet comme suit : $< node_1$,sg,predicate,$node_2$,sg_{in},$sg_{out} >$
où : $node_1$, sg, $node_2$, sg_{in} et sg_{out} représentent respectivement le sujet en cas d'un triplet SPO ou l'objet en cas d'un triplet OPS, l'ID du fragment stockant les arcs sortants dans le cas où le $node_1$ est un sujet ou des arcs entrants dans le cas d'un objet, l'objet en cas de triplet SPO ou le sujet en cas de triplet OPS, l'ID du fragment stockant les arcs entrants du $node_2$ et l'ID de fragment stockant les arcs sortants du $node_2$.
$node_1$ et $node_2$ sont représentés avec 64 bits (8 octets), tandis que sg, sg_{in} et sg_{out} sont représentés avec 32 bits (4 octets). Nous utilisons un seul bit pour représenter le changement du prédicat. En effet, comme les données sont triées dans l'ordre sujet, prédicat et objet et que pour toutes les nœuds du fragment, nous avons le même ensemble de prédicats, nous proposons d'utiliser un seul bit pour indiquer le changement du prédicat.

Notre approche de compression des données est inspirée de celle utilisée dans RDF-3X (Neumann et Weikum, 2008). Le nombre d'octets utilisés pour représenter ces informations varie en fonction du nombre d'octets utilisés pour encoder chaque élément du triplet. Le nombre d'états pour chaque élément stocké est le suivant :

Indicateur	Prédicat	Node$_1$	Fragment	Node$_2$	Fragment (in)	Fragment (out)
17 bits	1 bit	0-8 octets	0-4 octets	0-8 octets	0-4 octets	0-4 octets

FIG. 4: Illustration de la compression

— $node_1$: nous avons 9 états : 0 pour indiquer que le $node_1$ ne change pas par rapport au triplet précédent. i $\in [1..8]$ pour indiquer le numéro que nous avons utilisé pour encoder le nouveau $node_1$. Nous avons donc besoin de 4 bits pour représenter les 9 états.
— sg, sg_{in} et sg_{out} : nous avons besoin de 5 états. 0 pour indiquer que cette information ne change pas ou NULL. i $\in [1..4]$ pour indiquer le numéro que nous avons utilisé pour encoder le nouveau identifiant de fragment. Nous avons donc besoin de 3 bits pour représenter les 5 états. Pour $node_2$, nous utilisons i $\in [1..8]$ pour indiquer le numéro que nous avons utilisé pour encoder le nouveau $node_2$. Nous avons donc besoin de 4 bits pour représenter les 8 états.
— $predicat$: il ne faut qu'un bit pour indiquer le changement. Nous n'avons pas besoin de stocker le prédicat.

Comme illustré dans la Figure 4, il faut au total 18 bits. Ces bits représentent l'indicateur du nombre d'octets utilisés pour encoder chaque triplet. Il est à noter que seules les pages feuilles stockant les triplets sont compressées.

3.1.2 Accès arbre B+

Dans cette section, nous discutons notre stratégie de recherche en masse d'arcs entrants (ou sortants) d'un nœud à l'aide d'un arbre B+. Nos opérateurs d'évaluation utilisent la structure de l'arbre B+ pour trouver les arcs pertinents. L'algorithme naïf consiste à effectuer une recherche sur l'arbre B + pour chaque nœud dont nous avons besoin. Si nous avons n nœuds, nous avons besoin de $n * log(n)$ pour trouver toutes les arcs nécessaires.

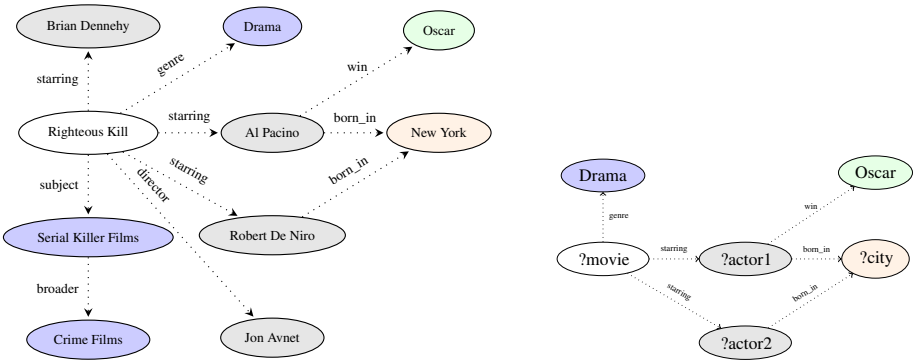
Nous proposons une autre stratégie. En effet, nous commençons d'abord par trouver le premier nœud. Lors de cette opération, nous mémorisons les limites des pages intermédiaires que nous avons utilisées. Dès que nous passons au nœud suivant, nous comparons avec les limites précédemment mémorisées. Nous déclenchons une recherche partielle si nous violons les limites mémorisées. L'entrée de notre algorithme est un vecteur de nœuds et la sortie est un ensemble d'arcs entrants (ou sortants) liés à ces nœuds.

3.2 Évaluation de requêtes

Dans cette section nous commençons par présenter les structures que nous allons utiliser pour décrire formellement notre approche. Ensuite, nous définissons les opérations faites par le moteur d'exécution des requêtes de notre système.

Définition 2 ***(Graphe de données)*** *Un graphe de données est représenté par $G = \langle V_c, L_V, E, L_E \rangle$ où V_c est un ensemble de sommets correspondant à tous les sujets et les objets d'un graphe de données, L_V est une collection d'étiquettes de sommets, E est une collection d'arcs qui relient les sujets et les objets correspondants, et L_E est une collection d'étiquettes d'arcs. Étant donné un arc $e \in E$, son étiquette d'arc est sa propriété.*

La Figure 5a montre un exemple d'un graphe de données. Ce graphe contient dix triplets et décrit les informations relatives au film "Righteous Kill".



(a) Graphe de données (b) Requête

FIG. 5: Exemple d'un graphe de données et d'une requête

Définition 3 ***(Graphe de requête)*** *Un graphe de requête est désigné par Q = $\langle V, L_V, E, L_E \rangle$, où $V = V_p \cup V_c$ est un ensemble de sommets correspondant à tous les sujets et objets d'une requête, où V_p est une collection de sommets de paramètres, et V_c est tel que défini dans la définition précédente. En tant que convention de dénomination, nous distinguons les variables des éléments de V_c par le biais d'un symbole de point d'interrogation (e.g., ? Nom, ? X), L_V est une collection d'étiquettes de sommet. Pour un sommet $v \in V_p$, son étiquette de sommet est $\emptyset$[4]. Un sommet $v \in V_c$, E et L_E sont déjà discutés dans la Définition 2.*

Définition 4 ***(Étoile de données)*** *Nous appelons étoile de données l'ensemble des arcs liées à un nœud donné dans le graphe de données. Si les arcs sont sortants, nous appelons cette étoile de données "étoile de données sortante". Nous utilisons le symbole : $SD_f(x)$, où $x \in V_c$, pour désigner l'étoile de données sortante obtenue à partir de x. Dans ce cas, x est appelé la tête du étoile de données. Si les arcs sont entrants, nous appelons cette étoile de données "étoile de données entrante". Nous utilisons le symbole : $SD_b(x)$. Pour simplifier, nous utilisons $\overrightarrow{x}$ (respectivement $\overleftarrow{x}$) pour désigner les étoiles sortant (respectivement entrant) obtenues à partir du noeud x. Nous appliquons le même principe pour distinguer les étoiles dans une requête. Nous utilisons étoile de requête sortante ($SQ_f(x)$) et étoile de requête entrante ($SQ_b(x)$), où $x \in V_c \cup V_p$, pour désigner les étoiles obtenues à partir des triplets sortants et des triplets entrants respectivement.*

Les étoiles de données sortantes étendent la définition de tuple dans les bases de données relationnelles, nous ne pouvons pas représenter plus d'une valeur pour un attribut. En effet, dans le cas d'un attribut comme clé primaire, nous l'utilisons comme tête du étoile de données, sinon, nous utilisons l'identifiant (ID) de l'enregistrement.

Proposition 1 *Étant donné un ensemble de prédicats lié à un nœud et un ensemble de prédicats dans une étoile de données SQ, un nœud du treillis S satisfait ($\models$) l'étoile de données SQ si les prédicats (SQ) $\subseteq$ prédicats (S).*

4. $\emptyset$ est utilisé pour désigner un élément vide

SQ	
$SQ_f(?movie)$	{(?movie,genre,drama),(?movie,starring, ?actor1),(?movie,starring, ?actor2)}
$SQ_f(?actor1)$	{(?actor1,win,Oscar),(?actor1,born_in, ?city)}
$SQ_f(?actor2)$	{(?actor2,born_in, ?city)}
$SQ_b(Oscar)$	{(?actor1,win,Oscar)}
$SQ_b(?city)$	{(?actor1,born_in, ?city),(?actor2,born_in, ?city)}
$SQ_b(?actor1)$	{(?movie,starring, ?actor1)}
$SQ_b(?actor2)$	{(?movie,starring, ?actor2)}
$SQ_b(Drama)$	{(?movie,genre,Drama)}

TAB. 1: Exemple : Étoile de requête

À partir de notre exemple de requêtes, nous pouvons extraire les étoiles de données affichées dans le tableau 1.

Nous expliquons maintenant comment évaluer une requête en évaluant des étoiles de requêtes.

Définition 5 ***(Correspondance)*** *Pérez et al. (2006)) Une correspondance est une fonction partielle $\mu : V_p \rightarrow V_c$ d'un sous-ensemble de variables V_p aux nœuds constants V_c. Le domaine de correspondances μ, écrit DOM(μ), est défini comme le sous-ensemble de V_p pour lequel μ est défini. Par M nous dénotons l'univers de tous les correspondances.*

Nous définissons ensuite la principale notion de compatibilité entre les correspondances. De façon informelle, deux correspondances sont compatibles s'ils ne contiennent pas de liaisons de variables contradictoires, *i.e.* si les variables partagées sont liées toujours à la même valeur dans les deux correspondances.

Définition 6 ***(Compatibilité des correspondances)*** *Étant donné deux correspondances $\mu 1$, $\mu 2$,nous dirons que $\mu 1$ est compatible avec $\mu 2$ si $\mu 1$(?x) = $\mu 2$(?x) pour tout ?x $\in$ dom($\mu 1$) $\cap$ dom($\mu 2$). Nous écrivons $\mu 1 \sim \mu 2$ si $\mu 1$ et $\mu 2$ sont compatibles, et $\mu 1 \not\sim \mu 2$ sinon.*

Nous notons $vars$ la fonction qui retourne des variables de l'élément passé comme paramètre. Par exemple, nous notons $vars(t)$toutes les variables du modèle triplet t, tandis que $vars(SQ_f(x))$ dénote toutes les variables de l'étoile de requête $SQ_f(x)$.

Nous écrivons μ(t) pour indiquer le motif triplet obtenu lors du remplacement de toutes les variables ?x $\in vars(t)$ dans t par μ(?x).

Dans ce qui suit, nous utilisons $[\![x]\!]_G$ pour désigner le processus pour trouver les correspondances pertinents de x par rapport à G.

Définition 7 ***(Évaluation de triplet)*** *Nous appelons $[\![t]\!]_G$ le processus permettant de trouver la correspondances liées à un motif de triplet t par rapport à un graphe G. $[\![t]\!]_G$ est formellement défini comme suit : $[\![t]\!]_G := \{\mu | dom(\mu) = vars(t)$ et $\mu(t) \in G\}$*

De façon informelle, nous essayons de trouver des correspondances de telle sorte que lorsque nous remplacerons les nœuds variables par des constantes correspondantes, le triplet obtenu se trouve dans le graphe de données.

Dans ce qui suit, nous utilisons une évaluation de triplet pour définir une évaluation d'une étoile de requête.

Définition 8 ***(Évaluation de l'étoile de requête)*** *De façon informelle, l'évaluation d'une étoile de requête permet de trouver des correspondances pour les variables dans l'étoile de requête par rapport au graphe de données. Pour chaque triplet t dans l'étoile, nous essayons de trouver l'ensemble des correspondances satisfaisant t. Nous construisons ensuite les correspondances de l'étoile de requête en joignant des correspondances liées aux triplets de l'étoile de requête.*

Formellement, l'évaluation d'une requête $SQ(x)$ par rapport au graphe de données G est définie comme suit :

$[\![SQ(x)]\!]_G := \{[\![t_1]\!]_G \bowtie [\![t_2]\!]_G \bowtie ... \bowtie [\![t_n]\!]_G | n = card(SQ(x)\}$

où :

$[\![t_i]\!]_G \bowtie [\![t_j]\!]_G = \{\mu_l \cup \mu_r | \mu_l \in [\![t_i]\!]_G$ *et* $\mu_r \in [\![t_j]\!]_G$, $\mu_l \sim \mu_r$ *et* $\mu_l(t_i) \neq \mu_r(t_j)\}$

A partir de cette définition, nous définissons l'évaluation de l'opérateur de jointure de deux étoiles de requête. En effet, la jointure sera utilisé comme opérateur de base de l'évaluation de la requête.

Définition 9 ***(Jointure des étoiles)*** *De façon informelle, la jointure de deux étoiles de requête permet d'assembler les correspondances obtenues en évaluant deux étoiles de requête. Formellement, l'évaluation d'une jointure entre deux étoiles de requête est définie comme suit :*

$[\![SQ_i]\!]_G \bowtie [\![SQ_j]\!]_G = \{\mu_l \cup \mu_r | \mu_l \in [\![SQ_i]\!]_G, \mu_r \in [\![SQ_j]\!]_G$ *et* $\mu_l \sim \mu_r\}$

Nous prenons une correspondance obtenu à partir de l'étoile de requête de gauche et une autre de l'étoile de requête de droite, nous vérifions si les deux correspondances sont compatibles, l'union de ces correspondances est une correspondance valide pour la jointure des deux étoiles de requête.

Avant de présenter l'évaluation des requêtes à l'aide des étoiles de requête, nous allons définir le concept de couverture. En effet, nous utilisons ce concept pour garantir qu'un ensemble donné des étoiles de requête permette une évaluation correcte de la requête.

Définition 10 ***(Couverture d'étoile)*** *Nous notons $Cover_q(SQ)$ l'ensemble des triplets de requête partagés avec l'étoile de requête.*

$Cover_q(SQ) = \{t | t \in triplets(q) \cup triplets(SQ)\}$

A partir des définitions précédentes, nous pouvons définir l'évaluation des requêtes à l'aide d'un ensemble d'étoile de requête, comme suit :

Proposition 2 *Étant donné un ensemble d'étoiles $\{SQ_1, SQ_2,...,SQ_n\}$ qui couvrent la requête, i.e., $Cover_q(SQ_1) \cup Cover_q(SQ_2) \cup ... \cup Cover_q(SQ_n) = triplets(q)$, l'évaluation de q en utilisant l'ensemble des étoiles de requête est définie comme suit :*

$[\![q]\!]_G = [\![SQ_1]\!]_G \bowtie [\![SQ_2]\!]_G \bowtie ... \bowtie [\![SQ_n]\!]_G$

par rapport aux fragments, nous pouvons définir l'évaluation de la requête comme suit :

$[\![q]\!]_G = \bigcup_{sg \models SQ_1} [\![SQ_1]\!]_{sg} \bowtie \bigcup_{sg \models SQ_2} [\![SQ_2]\!]_{sg} \bowtie ... \bowtie \bigcup_{sg \models SQ_n} [\![SQ_n]\!]_{sg}$

Il est à noter que les résultats de l'évaluation d'un triplet de l'étoile de requête d'une requête est un ensemble de correspondance. Pour le reste de cet article, ω dénote cet ensemble. Un ensemble de correspondance, qui représente tous les résultats obtenus en évaluant un triplet,

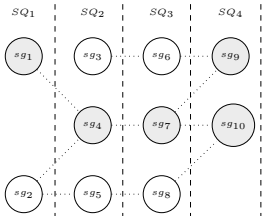


FIG. 6: Plan d'exécution

une étoile de requête ou une requête, est noté Ω (*i.e.*, $\Omega = \{\omega\}$) Étant donné une requête et les étoiles obtenues à partir des arcs entrants sortants, nous pouvons facilement montrer que l'ensemble des étoiles de requête permettant d'évaluer la requête n'est pas unique.

Nous utilisons le mot "Plan" pour faire référence à un ensemble d'étoiles de requête permettant d'évaluer une requête donnée. Formellement, un plan est défini comme suit :

Proposition 3 *Un plan est une fonction d'ordre sur un ensemble d'étoile de requête permettant d'évaluer des requêtes. Nous notons par* $p = [SQ_1, SQ_2, ..., SQ_n]$ *le plan formé en exécutant* SQ_1*, puis* SQ_2*,..., jusqu'à* SQ_n

Le choix des étoiles de requête et l'ordre d'évaluation permettent d'optimiser le processus d'évaluation des requêtes.

Variable	Nœud associé
?movie	Righteous Kill
?actor1	Al Pacino
?actor2	Robert De Niro

L'évaluation de $SQ_f(?movie)$ $\longrightarrow$

Variable	Nœud associé
?actor1	Al Pacino
?city	New York

L'évaluation de $SQ_f(?actor1)$ $\longrightarrow$

Variable	Nœud associé
?city	New York
?actor1	Al Pacino
?actor2	Robert De Niro

L'évaluation de $SQ_f(?city)$

?movie	?actor1	?actor2	?city
Righteous Kill	Al Pacino	Robert De Niro	New York

Résultats finaux

FIG. 7: Shéma d'exécution de plan

Illustrons tous ces concepts en utilisant l'exemple précédent. Nous commençons par examiner le graphe de données dans la Figure 5a et la requête dans la Figure 5b. Comme expliqué précédemment, nous traitons la requête en extrayant les étoiles sortantes et entrantes. Ces étoiles sont montrées dans table 1. Considérons qu'une seule façon (*i.e.*, un plan) pour évaluer la requête. En effet, nous considérons le plan suivant : $[SQ_f(?movie), SQ_f(?actor1), SQ_b(?city)]$

En rejoignant les correspondances obtenues à partir des différentes étoiles de requête du plan, nous obtenons le résultat final, comme indiqué dans figure 7. Comme montré dans la figure 6, chaque plan est lié à un ensemble de fragments. Nous proposons de faire un élagage en temps réel. En effet, un fragment intermédiaire est évalué seulement si ce fragment est référencé par un fragment qui a été évalué par une autre requête d'étoile.

TAB. 2: Jeux de données expérimentaux

	Watdiv			LUBM			
Taille (Millions)	10	100	200	10	20	40	100
Taille (Go)	1,43	14,5	28,5	1,62	3,22	6,69	16,2
# fragments SPO	13 002	39 855	27 464	11	11	11	11
# fragments OPS	641	1 088	1 520	13	13	13	13

4 Évaluation expérimentale

Nous avons évalué les performances de notre approche en utilisant deux bancs d'essai RDF connus (Watdiv et LUBM). Nous avons tout d'abord évalué la capacité des systèmes à charger des données supérieures à la mémoire principale disponible. Ensuite, nous évaluons les performances liées au traitement des requêtes.

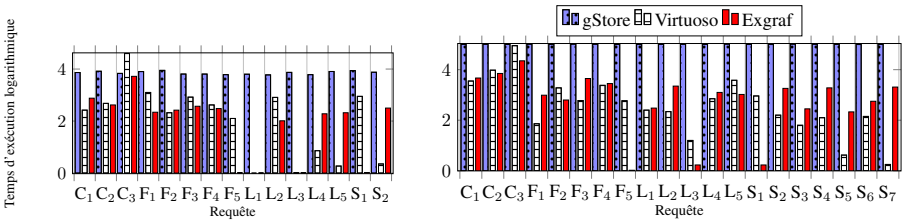
4.1 Configuration expérimentale

Matériel : Nous avons effectué toutes les expériences sur une machine dotée d'un processeur Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz CPU, 1TB Disque dur et 32Go de mémoire. Ubuntu server 16.04 LTS est utilisé comme système d'exploitation.
Logiciel : Les principaux composants de notre approche nommé EXGRAF (*i.e.*,modules de fragmentation, d'allocation et d'indexation) sont implémentés en Java. L'extraction des étoiles de données est codé avec C ++.
Systèmes comparés : Notre système a été comparé avec deux approches exploitant deux différents paradigmes d'exécution. Le premier gStore [5] qui est un système d'exécution basé sur la recherche de correspondances de graphes et puis le Système Virtuoso [6] qui est basé sur les techniques classiques d'évaluation.
Jeux de données : Nous évaluons et comparons les performances des systèmes à l'aide des benchmarks Watdiv et LUBM. Nous comparons le temps d'exécution pour répondre aux requêtes avec les différentes configurations (Linéaire, étoile, flocon de neige et complexe). La liste des requêtes ne figure pas ici pour des raisons d'espace mais elle se trouve dans notre rapport technique [7]. Nous comparons également la capacité des systèmes à gérer des jeux de données de différentes tailles. Nous avons généré des jeux de données avec 10, 100 et 200 millions de triplets pour Watdiv et des ensembles de données de 10, 20, 40 et 100 millions de triplets pour LUBM. La taille de chaque jeu de données est détaillée dans le tableau 2.

4.2 Évaluation du pré-traitement

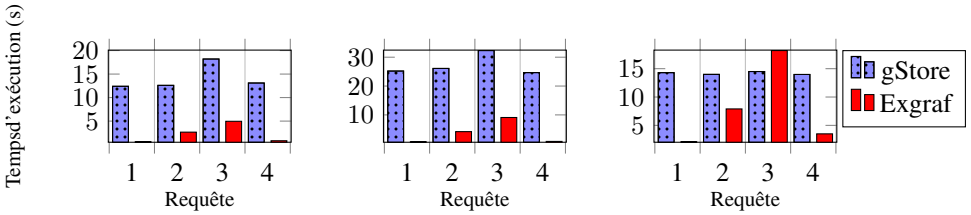
Nous avons d'abord testé la capacité des systèmes à pré-traiter et à charger les jeux de données RDF bruts (au format N-triplets). EXGRAF et Virtuoso ont pu charger tous les jeux de données du framework Watdiv. Au contraire, gStore n'a pas pu charger le jeu de données de 200 millions de triplets. Ceci est principalement dû au fait que gStore effectue le pré-traitement dans la mémoire principale et il est incapable de charger des graphes RDF qui n'y rentrent pas. Virtuoso charge le jeu de données dans une base de données relationnelle, et EXGRAF crée

5. https://github.com/pkumod/gStore
6. https://github.com/openlink/virtuoso-opensource
7. https://www.lias-lab.fr/publications/32595/khelil_rdf_processing_report.pdf



(a) 10 Millions (b) 100 Millions

FIG. 8: Performances des requêtes pour Watdiv



(a) 10 Millions (b) 20 Millions (c) 40 Millions

FIG. 9: Performances des requêtes pour LUBM

des fichiers de fragments (SPO et OPS). Le nombre de fragments SPO et OPS sur chaque jeu de données est indiqué dans le tableau 2. Le nombre de fragments SPO et OPS ne croît pas de manière exponentielle et leur nombre reste raisonnable par rapport à la taille des données.

4.3 Performances des requêtes

Watdiv : La performance des requêtes linéaires (L), en étoile (S), en flocon de neige (F) et complexes (C) est illustrée dans la Figure 8. Nous utilisons l'échelle logarithmique pour désigner les graphes puisque les performances de EXGRAF sont en moyenne 300 fois supérieures à celles de gStore. Les résultats pour 10 millions de triplets de Watdiv sont présentés dans la Figure 8a. Même si EXGRAF obtient des performances très similaires à celles de Virtuoso pour les requêtes en étoile et linéaires, notre modèle d'exécution permet de garantir le passage à l'échelle pour des ensembles de données et des requêtes plus complexes. Cela est montré avec les requêtes complexes et de flocons de neige dans lesquels notre système est en moyenne 1,6 fois plus rapide. On a le même comportement sur les mêmes requêtes exécutées sur un ensemble de données de 100 millions. EXGRAF est capable de répondre aux requêtes beaucoup plus complexes qui ne sont guère transformées en SQL avec des performances raisonnables.
LUBM : De même de ce qui a été fait pour Watdiv, nous avons évalué la capacité des systèmes à charger différentes tailles de jeux de données. gStore n'a pas pu charger le jeu de données de 100 millions de triplets, car il est supérieur à la mémoire principale disponible. Les performances de la requête pour les jeux de données de 10, 20 et 40 millions de triplets sont illustrées dans la Figure 9. Dans tous les cas, notre système surpasse les performances de gStore, on est 10 fois plus rapide que gStore.

5 Conclusion

Dans cet article, nous avons proposé une nouvelle approche (EXGRAF) orientée-disque pour évaluer efficacement des requêtes exécutées sur des données RDF volumineuses. L'intérêt de EXGRAF réside du fait qu'elle fait appel à l'exploration et la fragmentation de graphes. L'exploration permet de conserver la structure logique de la donnée RDF. EXGRAF s'inspire du système Volcano pour assurer une bonne gestion de la mémoire. Nos résultats sont encourageants et montrent que EXGRAF surpasse le système gStore considéré comme le plus performant en matière de traitdes données RDF représentées naïvement par des graphes.

Actuellement, nous menons des expériences intensives pour évaluer la robustesse de notre approche et nous travaillons sur la parallélisation de EXGRAF, en exploitant les partitions générées par le processus de fragmentation.

Références

Abadi, D. J., A. Marcus, S. R. Madden, et K. Hollenbach (2007). Scalable semantic web data management using vertical partitioning. In *VLDB*, pp. 411–422.

Aït-Kaci, H., R. Boyer, P. Lincoln, et R. Nasr (1989). Efficient implementation of lattice operations. *ACM TOPLAS 11*(1), 115–146.

Bollacker, K., C. Evans, P. Paritosh, T. Sturge, et J. Taylor (2008). Freebase : a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pp. 1247–1250.

Briggs, M. (2012). Db2 nosql graph store what, why & overview.

Cyganiak, R. (2005). A relational algebra for sparql. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, 35.

Deppisch, U. (1986). S-tree : a dynamic balanced signature index for office retrieval. In *ACM SIGIR*, pp. 77–87.

Graefe, G. (1994). Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng. 6*(1), 120–135.

Lehmann, J., R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al. (2015). Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web 6*(2), 167–195.

McBride, B. (2002). Jena : A semantic web toolkit. *IEEE Internet computing* (6), 55–59.

Neumann, T. et G. Moerkotte (2011). Characteristic sets : Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, pp. 984–994.

Neumann, T. et G. Weikum (2008). Rdf-3x : a risc-style engine for rdf. *PVLDB 1*(1), 647–659.

Pérez, J., M. Arenas, et C. Gutierrez (2006). Semantics and complexity of sparql. In *International semantic web conference*, Volume 4273, pp. 30–43. Springer.

Stocker, M., A. Seaborne, A. Bernstein, C. Kiefer, et D. Reynolds (2008). Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, pp. 595–604.

Weiss, C., P. Karras, et A. Bernstein (2008). Hexastore : sextuple indexing for semantic web data management. *PVLDB 1*(1), 1008–1019.

Zou, L., J. Mo, L. Chen, M. T. Özsu, et D. Zhao (2011). gstore : answering sparql queries via subgraph matching. *PVLDB 4*(8), 482–493.