Interrogation de données hétérogènes dans les systèmes NoSQL orientés graphes

Mohammed El Malki[2], Hamdi Ben Hamadou[1], Max Chevalier[1], André Péninou[2], Olivier Teste[2]

1. Université Toulouse 3 Paul Sabatier, IRIT (CNRS/UMR5505)
prenom.nom@irit.fr

2. Université Toulouse 2 Jean Jaurès, UT2C, IRIT (CNRS/UMR5505)
prenom.nom@irit.fr

RESUME. La flexibilité des systèmes NoSQL, qui consiste à ne plus garantir un schéma unique pour un ensemble de données, aboutit à des masses de données hétérogènes rendant leur interrogation plus complexe pour les utilisateurs, qui doivent connaître les différentes formes (c'est-à-dire les différents schémas) des données manipulées. Cet article se focalise sur cette problématique de l'interrogation des données hétérogènes dans les systèmes NoSQL orientés graphes. L'enjeu est de simplifier pour les utilisateurs l'interrogation de ces masses de données hétérogènes en rendant transparente leur hétérogénéité. L'article propose de construire un dictionnaire de similarité entre labels et attributs. A partir de ce dictionnaire la requête utilisateur peut être automatiquement réécrite pour intégrer la variabilité des données.

ABSTRACT. The NoSQL systems falls within the "schemaless" principle consisting in providing more than a single schema for a dataset, thus allowing a wide variety of representations. This flexibility leads to a large volume of heterogeneous data, wish makes their interrogation more complex for the users, who are compelled to know the different forms (i.e. the different schemas) of this data. This paper addresses this issue and focus on simplifying for users the heterogeneous data interrogation process in graph-oriented NoSQL systems. The paper proposes to build a similarities dictionary between labels and attributes. From this dictionary, the user query is automatically extended to integrate the variability of underlying data.

Mots-clés : NoSQL, similarité, flexibilité des schémas

KEYWORDS: NOSQL, NEO4J, SCHEMALESS, SIMILARITY

1. Introduction

En raison de leur capacité à gérer efficacement d'importantes masses de données, les systèmes de stockage « not-only-SQL » ou NoSQL, connaissent un important développement (Floratou et al., 2005) (Stonebraker, 2010). Parmi les différentes approches NoSQL, les systèmes orientés graphes permettent de modéliser les données sous la forme de graphes (Holzschuher and Peinl, 2013). Les données sont représentées sous la forme de nœuds, de relations et de propriétés (Roussy 2016), permettant ainsi de modéliser les différentes interactions entre les données. Ce type de représentation joue un rôle central dans de nombreux domaines tels que les réseaux sociaux, le Web sémantique, les sciences du vivant (interactions de protéines).

Les systèmes NoSQL, et donc les systèmes orientés graphes, caractérisés par le principe de « *schemaless* » (Cattel, 2010) ne garantissent plus un schéma unique pour un ensemble de données. Ainsi chaque nœud et chaque relation possède son propre ensemble de propriétés, permettant ainsi une grande variété de représentation (Chevalier et al., 2015). Cette flexibilité aboutit à des masses de données hétérogènes (schémas différents), rendant leur interrogation plus complexe pour les utilisateurs, qui doivent connaître les différentes formes (c'est-à-dire les différents schémas) des données manipulées. Cet article se focalise sur cette problématique de l'interrogation des données hétérogènes dans les systèmes NoSQL orientés graphes. L'enjeu est de simplifier pour les utilisateurs l'interrogation de ces masses de données hétérogènes en rendant transparente leur hétérogénéité. Cet article propose de construire un dictionnaire de similarité entre labels et attributs. A partir de ce dictionnaire la requête utilisateur peut être automatiquement réécrite pour intégrer la variabilité des données réelles.

Le reste du document est structuré comme suit. La section 2 illustre le problème, la section 3 traite l'état de l'art. Nous présentons notre solution d'interrogation des données hétérogènes, appelé *EasyGraphQuery*, dans la section 4. Les premiers résultats de l'évaluation expérimentale sont présentés dans le section 5.

2. Illustration du problème

2.1. Notations préliminaires

La modélisation des données dans les systèmes NoSQL orientés graphes consiste à considérer la base de données comme un graphe. La Figure 1 illustre un exemple simple de graphe $G = (V, E, \gamma)$ où $V = \{u_1,\ldots u_{13}\}$ représente les nœuds, $E = \{e_1,\ldots e_9\}$ représente les arêtes et $\gamma : E \rightarrow V \times V$ est une fonction déterminant les paires de nœuds reliés par les arêtes (appelés aussi relations).

Les différents nœuds peuvent être décrits sous une forme textuelle comme ci-dessous Figure 1. Chaque nœud comporte un ou plusieurs labels (ex. *Author : Poet*) et peut être caractérisé par des attributs (ex. *Firstname*). On constate que ce graphe comporte des éléments (nœuds et attributs) hétérogènes.

- u_1:Author{firstname:'Charles',lastname:'Baudelaire',birth_date:1821,date_of_death:1867}
- u_2:Writer{firstname:'Paul',lastname:'Verlaine',birth:1844}
- u_3:Author:Poet{firstname:'Guillaume',lastname:'Apollinaire',birth:1880,death:1918}
- u_4:author{firstname:'Arthur',lastname:'Rimbaud',birth_date:1854,death_of_death:1891}
- u_5:Book{number:1,title:'Les fleurs du mal',year:1857}
- u_6:Book{number:2,title:'Le Spleen de Paris',year:1869}
- u_7:book{number:3,title:'Les Paradis artificiels',year:1860}
- u_8:Work{number:4,title:'Poèmes saturniens',year:1866}
- u_9:Work{number:5,title:'Fêtes galantes'}
- u_{10}:Publication{number:5,title:'Alcools'}
- u_{11}:Book{number:6,title:'Une saison en enfer',year:1873}
- u_{12}:Work{number:7,title:'Les illuminations',year:1886}
- u_{13}:Publication{number:8,title:'Le Bateau ivre',year:1920}

Sur le graphe de la Figure 1, on peut remarquer que pour une sémantique probablement équivalente le nom d'une relation peut varier (soit `To_Write`, soit `To_Compose`) ; comme les nœuds du graphe, les arêtes (labels relations) et leurs attributs peuvent être hétérogènes.

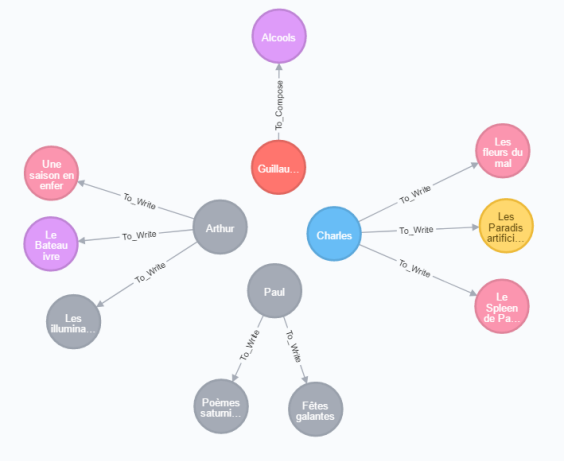


Figure 1 : Exemple de graphe.

2.2. Facettes de l'hétérogénéité

L'hétérogénéité peut être considérée selon différentes facettes (Shvaiko et Euzenat, 2005) suivant les éléments de structures qui constituent un graphe (attributs) mais aussi labels des nœuds ou des relations, ainsi que les extrémités reliées). La première facette, l'hétérogénéité structurelle, désigne le problème qu'une donnée peut être représentée par des éléments de structure variables. La seconde facette, l'hétérogénéité syntaxique, désigne le problème qu'un élément de structure peut être désigné de manière variable ; *par exemple, les attributs 'birth_date' et 'birth' dans*

les nœuds désignent toutes les deux une date de naissance d'un auteur. La troisième facette, l'hétérogénéité sémantique, désigne le problème que deux éléments différents peuvent correspondre à une même donnée, ou inversement qu'un élément peut correspondre à des données variables ; *par exemple, les relations 'To_Write' et 'To_Compose' ont le même sens.*

Dans cet article nous étudions ces différentes facettes de l'hétérogénéité. Néanmoins, nous ne traitons pas l'hétérogénéité d'entités (Getoor et Machanavajjhala, 2012) en considérant que pour chaque entité conceptuelle, lui correspond un composant élémentaire du graphe, c'est-à-dire, un nœud ou une relation ; nous ne considérons pas la possibilité qu'une entité corresponde à un sous-graphe.

2.3. Problématique de l'interrogation de graphes NoSQL hétérogènes

Nous utilisons le système de stockage *Neo4j* pour illustrer notre étude de cas. Ce système propose le langage propriétaire *Cypher* (Holzschuher and Peinl, 2013) permettant d'exprimer des requêtes manipulant la base de données orientée graphe. Nous utiliserons ce langage pour illustrer nos propos. Nous limitons notre étude aux opérateurs de projection et de sélection (ou restriction).

Exemple. Considérons, toujours en utilisant le graphe de la Figure 1, une requête exemple pour chacun des opérateurs.

Projection. « `Obtenir le nom, le prénom et l'intitulé des œuvres littéraires des auteurs` »

```
match (n:Author)-[]-(m)
return n.firstname, n.lastname, m.title
```

On obtient alors le résultat ci-dessous ne faisant apparaitre qu'une partie des auteurs et des ouvrages réalisés par les auteurs. Ce résultat incomplet est dû à l'hétérogénéité du graphe. Ainsi l'auteur `Paul Verlaine` n'apparaît pas car le nœud n'est pas de type (on parle de label) `Author` mais `Writer`.

firstname	lastname	title
Charles	Baudelaire	Les Paradis artificiels
Charles	Baudelaire	Le Spleen de Paris
Charles	Baudelaire	Les fleurs du mal
Guillaume	Apollinaire	Alcools

Projection et Sélection. « `Obtenir le titres des œuvres littéraires de l'auteur Baudelaire` »

```
match (n:Author)-[]-(m:Book)
where lastname = 'Baudelaire' return m.title
```

On obtient encore un résultat incomplet, du point de vue de l'utilisateur, encore dû à l'hétérogénéité des données présentent dans le graphe. Le problème dans ce cas est dû à l'hétérogénéité des labels associés aux nœuds qui sont étiquetés soit `Book` soit `book`. Ce dernier n'est pas reconnu.

title
Les fleurs du mal
Le Spleen de Paris

On constate donc qu'une utilisation classique de Cypher dans un contexte de graphe hétérogène peut conduire l'utilisateur à bâtir des analyses et des décisions sur des données incomplètes.

Nous proposons dans cet article une approche permettant à un utilisateur d'exprimer simplement une requête à partir des attributs, sans avoir à tenir compte des différences structurelles, syntaxiques et sémantiques, tout en conservant les structures originelles des graphes. La requête permet d'obtenir un résultat « complet », de manière transparente par rapport à l'hétérogénéité des données (sans avoir à connaître et à manipuler avec exhaustivité les différentes facettes de l'hétérogénéité présente).

3. Etat de l'art

Dans cette section, même si ce travail peut être assimilée à une forme d'alignement nous abordons uniquement les deux approches principales d'interrogation utilisées et appliquées d'une manière générale dans les systèmes NoSQL.

L'approche d'homogénéisation dite *pivot* consiste à modifier la structure lors du stockage et à interroger les données dans un schéma pivot homogène. Par exemple, (Tahara et al., 2014) proposent le système *Sinew* qui aplatit les données et les charge dans un SGBD relationnel (tables). (Beyer et al., 2011) propose un nouveau langage de script pour interroger simultanément des données stockées dans des magasins différents et les requêtes sont découpées pour être distribuées en se basant sur le paradigme *Mapreduce* (Thusoo et al., 2009). Au-delà des coûts d'évaluation des requêtes, l'utilisateur doit connaitre les structures ou les métadonnées de l'ensemble des structures pour les interroger correctement. Cette approche d'homogénéisation a pour avantage de faciliter l'interrogation pour l'utilisateur qui manipule ainsi un schéma unique mais nécessite des pré-traitements pouvant s'avérer coûteux et difficilement compatibles avec des environnements dynamiques.

La seconde approche considérée dans la littérature consiste à inférer les différents schémas pour permettre leur interrogation. Dans ce contexte, des travaux s'attaquent à la problématique d'intégration et la découverte de nouveaux schémas (Wang et al., 2015) et (Herrero et al., 2016). Les travaux de (Herrero et al., 2016) proposent d'extraire séparément tous les schémas présents afin d'aider l'utilisateur à connaitre tous les schémas et tous les attributs présents (Wang et al., 2015), (Hamdi et al., 2018). Dans (Wang et al., 2015) les auteurs proposent d'homogénéiser tous les schémas dans un même « schéma universel » (skeleton) afin d'aider l'utilisateur dans la découverte d'attributs ou de sous-schémas. Comme l'approche précédente, l'intégration de nouveaux schémas nécessite de revoir tous les autres schémas déjà stockés ce qui va à l'encontre de la technologie NoSQL dont la vélocité est une des caractéristiques principales ; De plus l'hétérogénéité doit toujours être gérée par l'utilisateur lors des requêtes.

4. Le système EasyGraphQuery

Le principe que nous retenons est de gérer la variabilité (ou au moins une partie de celle-ci) automatiquement. On suppose que l'utilsateur pose une requête en ne connaissant qu'un schéma de données ; par exemple *Author, Birthdate, To_write*. La requête est ensuite réécrite et étendue pour intégrer la variabilité des données réelles. Pour cela, un dictionnaire permet d'associer à chaque élément du schéma (labels et attributs) une liste d'éléments « équivalents » que nous calculons par des mesures de similarité.

Le calcul de la similarité est effectué entre les éléments du graphe susceptibles d'être hétérogènes. Nous prenons en compte dans cet article les labels et les attributs des nœuds et des arêtes, pris séparément. Les facettes de l'hétérogénéité prises en compte sont l'hétérogénéité structurelle, l'hétérogénéité syntaxique et l'hétérogénéité sémantique. Ainsi, deux matrices sont construites afin de déterminer les similarités entre éléments du graphe : la matrice de similarité syntaxique est basée sur la mesure *Leivenshtein* (**Erreur ! Source du renvoi introuvable.**a) tandis que la matrice de similarité sémantique se base sur la mesure Lin (**Erreur ! Source du renvoi introuvable.**b). Nous ne détaillons pas également les pré-traitements appliqués lors de multi-termes comme pour `To_Write` ou `birth_date` lors des calculs des matrices. On peut envisager d'étendre l'approche avec d'autres mesures de similarités, et d'améliorer le processus par une combinaison pondérée de ces diverses mesures (Shvaiko et Euzenat, 2005) (Megdiche, et al., 2016).

Exemple. Considérons le graphe de la Figure 1. Le label `Author` du nœud u_1 (première ligne des matrices) est comparé avec les différents labels des nœuds du graphe. Pour déterminer les labels similaires ∇_{Author} nous utilisons $\forall j \in [1,N]$, $max\ (Leivenshtein\ (u_1, u_j), Lin(u_1, u_j)) \geq 0.8$[1] ; ∇_{Author} = { `Author`, `author`, `Writer` }.

De manière analogue, le label `To_Write` de l'arête est comparé avec les autres labels des arêtes. On obtient alors dans notre cas ∇_{To_Write} = { `To_Write`, `To_Compose` }.

Le processus de calcul des similarités est également appliqué sur les attributs des nœuds et des arêtes. En appliquant la même combinaison des mesures de similarité *Leivenshtein* et *Lin* contenues dans les matrices, nous construisons le dictionnaire des données des attributs ; $\Delta_{Author.birth\_date}$ = { `Author.birth_date`, `Writer.birth`, `Author.birth`, `author.birth` }. Par ailleurs, ces deux matrices de similarité servent à construire le dictionnaire de similarité, constitué d'une clé correspondant à un

[1] Le choix du seuil est arbitraire, la fonction du calcul du seuil ne fait pas l'objet de cet article.

attribut donné et de sa valeur correspondant à la liste des attributs similaires (syntaxique et sémantiques).

4.1. Modélisation des données et du dictionnaire

Dans ce qui suit nous formalisons les différentes définitions nécessaires à la modélisation des données et du dictionnaire.

Définition 1. Un graphe G est défini par (V, E, γ)
– $V = \{u_1,\ldots, u_N\}$ est l'ensemble des nœuds du graphe ;
– $E = \{e_1,\ldots, e_M\}$ est l'ensemble des arêtes du graphe ;
– $\gamma : E \rightarrow V$ x V est la fonction qui associe chaque arête aux sommets reliés.

On note $\mathcal{L} = \{l_1, \ldots, l_L\}$ un ensemble de termes désignant les différents labels de nœuds et de relations possibles.

Définition 2. Un nœud u_i est défini par (L_i, S_i)
– $L_i \subseteq \mathcal{L}$ est l'ensemble des labels caractérisant le nœud ;
– $S_i = \{a_{i,1},\ldots,a_{i,n_i}\}$ est le schéma du nœud, constitué par un ensemble d'attributs.

On note $S_V = \bigcup_{i=1}^{i=N} \left(\bigcup_{l_k \in L_i} \bigcup_{a_{i,j} \in S_i} l_k.a_{i,j} \right)$ le schéma des nœuds du graphe.

Définition 3. Une arête e_i est définie par $(l_i, S_i, u_{i,1}, u_{i,2})$
– $l_i \in \mathcal{L}$ est le label caractérisant l'arête ;
– $S_i = \{a_{i,1},\ldots,a_{i,n_i}\}$ est le schéma l'arête constitué par un ensemble d'attributs ;
– $u_{i,1}$ et $u_{i,2}$ sont les nœuds origine et cible reliés par l'arête ; $\gamma(e_i) = \{(u_{i,1}, u_{i,2})\}$.

On note $S_E = \bigcup_{i=1}^{i=M} \left(\bigcup_{a_{i,j} \in S_i} l_i.a_{i,j} \right)$ le schéma des arêtes du graphe. On note alors $S_G = S_N \cup S_M$ le schéma des attributs du graphe.

On remarque que $\mathcal{L} = \left(\bigcup_{i=1}^{i=N} L_i \right) \cup \left(\bigcup_{i=1}^{i=M} l_i \right)$.

Exemple. Considérons le graphe de la Figure 1.

– $V = \{ u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}, u_{11}, u_{12}, u_{13} \}$;
– $E = \{ e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9 \}$;
– $\gamma = \{ (e_1,(u_1, u_5)), (e_2,(u_1, u_6)), (e_3,(u_1, u_7)), (e_4,(u_2, u_8)), (e_5,(u_2, u_9)), (e_6,(u_3, u_{10})), (e_7,(u_4, u_{11})), (e_8,(u_4, u_{12})), (e_9,(u_4, u_{13}))\}$.

Dans la Figure 1, on trouve le nœud u_1 = ({*Author*}, {*firstname*, *lastname*, *birth_date*, *date_of_death*}) et le nœud u_7 = ({*Book*}, {*number*, *title*, *year*}). De même, on trouve l'arête e_3 = (*Book*, {}, u_1, u_7).

Afin de prendre en compte les différentes facettes de l'hétérogénéité du graphe (structurelle, syntaxique et sémantique), nous introduisons un dictionnaire de données

permettant de déterminer pour chaque élément du graphe (label de nœud ou d'arête, attribut de nœud ou d'arête) les éléments similaires.

Définition 4. Le dictionnaire des données $dict_{label}$ est défini par

$$dict_{label} = \{ (l_i, \nabla_i) \}$$

– $l_i \in \mathcal{L}_G$ est un label du graphe ;

– $\nabla_i = \bigcup_{\forall l_j \in \mathcal{L}_G | sim(l_i,l_j) \geq \delta} l_j \subseteq \mathcal{L}_G$ est l'ensemble des labels similaires. La fonction de similarité, notée *sim*, calcule un score normalisé compris entre [0..1] exprimant le taux de ressemblance (plus l_i et l_j sont similaires, plus le score est proche de 1). $\delta \in [0..1]$ est le seuil à partir duquel les labels l_i et l_j sont considérés comme similaires.

Définition 5. Le dictionnaire des données $dict_{attribut}$ est défini par

$$dict_{attribut} = \{ (l_i.a_{i,k}, \Delta_{i,k}) \}$$

– $l_i.a_{i,k} \in S_G$ est un attribut du graphe ;

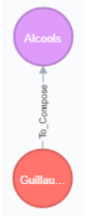
– $\Delta_{i,k} = \bigcup_{\forall l_j.a_{j,l} \in S_G | sim(a_{i,k},a_{j,l}) \geq \delta} l_j.a_{j,l} \subseteq S_G$ est l'ensemble des attributs similaires.

Remarque. Dans cet article nous ne prenons pas en compte l'hétérogénéité structurelle au niveau des labels mais uniquement entre les attributs ; cela signifie qu'un attribut peut être situé à diverses positions dans le graphe, repérées par les labels qui préfixent la propriété.

4.2. Noyau algébrique d'opérateurs

L'interrogation repose sur un ensemble d'opérateurs élémentaires formant un noyau minimum fermé. On note G_{in} le graphe interrogé et G_{out} le graphe résultat. Les graphes sont représentés sous une forme tabulaire.

Exemple. Considérons le sous graphe ci-dessous issu de la Figure 1 et sa forme tabulaire. Les labels et les références des arêtes aux nœuds reliés ne sont pas exprimés dans la forme tabulaire.



Graph Table
u3 : {firstname : 'Guillaume', lastname : 'Apollinaire', birth : 1880, death : 1918 }
u10 : {number : 5, title : 'Alcools' }
e6 : {year : 1913 }

Figure 2 : Représentation tabulaire des graphes.

Ces opérateurs élémentaires permettent d'exprimer des opérations de projection et de sélection (restriction).

Définition 6. La projection permet de réduire le graphe aux éléments de structure spécifiés dans le patron structurel (« pattern ») et la liste d'attributs projetés.

$$_{Pattern}\pi_{Attribute}(G_{in}) = G_{out}$$

– *Pattern* est un chemin de la forme l_0–l_1–…–l_p où chaque $l_i \in \mathcal{L}_G$ désigne la classe d'un nœud ou d'une arête.
– *Attribute* est un ensemble (possiblement vide) d'attributs $l_i.a_{i,k}$ où $l_i \in Pattern$ et $a_{i,k} \in S_i$.

Définition 7. L'opérateur de sélection permet de restreindre les éléments de structures aux seuls éléments satisfaisant un prédicat de sélection ; on note :

$$_{Pattern}\sigma_{Predicate}(G_{in}) = G_{out}$$

– *Pattern* est un chemin de la forme l_1–l_2–…–l_p où chaque $l_i \in \mathcal{L}_G$ désigne la classe d'un nœud ou d'une arête.
– *Predicate* est un prédicat (ou condition) de sélection. Un prédicat simple est une expression $l_i.a_{i,k}\ \omega_k\ v_k$ avec $a_{i,k} \subseteq S_i$ est un attribut, $\omega_k \in \{=;>;<;\neq;\geq;\leq\}$ est un opérateur de comparaison, et v_i une valeur. Les prédicats peuvent se combiner avec les opérandes $\Omega = \{\vee, \wedge, \neg\}$ formant un prédicat complexe.

Les prédicats de sélection complexe combinant plusieurs prédicats sont représentés sous sa forme normale conjonctive : $Predicate = \wedge_x(\vee_y p_{x,y})$.

Exemple. Considérons les requêtes de la section 2.3. Nous pouvons exprimer ces requêtes en représentation algébrique (interne) comme suit :

Projection. « `Obtenir le nom, le prénom et l'intitulé des œuvres littéraires des auteurs` »

$$_{\texttt{Author- -1}}\pi_{\texttt{Author.firstname,Author.lastname,1.title}}$$

Le résultat obtenu est donné dans le tableau ci-dessous. Lorsque les attributs sont projetés, les identifiants des nœuds (u_i) et des arêtes (e_i) sont perdus ; ceci rompt le principe de fermeture du noyau algébrique, ne permettant donc pas de combiner ce résultat avec une nouvelle opération.

Table 1 : Résultat de l'opération de projection.

G_{out}
{firstname : 'Charles', lastname : 'Baudelaire', title : 'Les Paradis artificiels' }
{firstname : 'Charles', lastname : 'Baudelaire', title : 'Les fleurs du mal' }
{firstname : 'Charles', lastname : 'Baudelaire', title : 'Le Spleen de Paris' }
{firstname : 'Guillaume', lastname : 'Apollinaire', title : 'Alcools' }

Projection et Sélection. « Obtenir le titres des œuvres littéraires de l'auteur Baudelaire »

$$\text{Author- -1}\pi_{\text{Author.firstname,Author.lastname,1.title}}(\text{Author- -1}\sigma_{\text{Author.lastname='Baudelaire'}})$$

Le résultat obtenu est donné dans le tableau suivant.

Table 2 : Résultat de la composition d'opérations de sélection et de projection.

G_{out}
{firstname : 'Charles', lastname : 'Baudelaire', title : 'Le Spleen de Paris' }
{firstname : 'Charles', lastname : 'Baudelaire', title : 'Les fleurs du mal' }

L'utilisation de cette représentation interne des opérations d'interrogation sur les graphes ne prend pas en charge l'hétérogénéité des éléments présents dans le graphe. Les résultats de ces requêtes restent donc incomplets au regard des informations présentent dans le graphe. Nous présentons dans la suite le processus de réécriture de ces requêtes internes permettant d'obtenir un résultat complet, de manière transparente pour l'utilisateur, et dynamique (sans transformation des données).

4.3. Algorithme de réécriture des requêtes

Notre approche consiste à faciliter l'interrogation pour les utilisateurs, par reformulation automatique des requêtes. Ce processus exploite le dictionnaire et indirectement les matrices de similarité des données afin de reformuler la requête en déterminant les éléments (nœuds, arêtes et attributs) similaires. L'algorithme suivant décrit ce processus de récriture automatique de la requête utilisateur.

La fonction *exists*(*A, L*) permet de vérifier l'existence dans L, du pattern constitué à partir des labels issus de A. L'opération d'union, notée $\cup$, permet d'unifier deux graphes ; $G_1 \cup G_2 = G_{out} \mid V_{out} = V_1 \cup V_2$; $E_{out} = E_1 \cup E_2$; $\gamma_{out} : E_{out} \rightarrow V_{out} \times V_{out} \mid e_{out} \in \gamma_1 \vee e_{out} \in \gamma_2$.

Algorithme : Extension automatique de la requête utilisateur

entrée : Q
sortie : Q_{ext}

begin
  $Q_{ext} \leftarrow id$
  foreach $q_x \in Q$ **do**
    switch q_x **do**
      case $_{Pattern}\pi_{Attribute}$ **do** *// projection*
        $L_{ext} \leftarrow \prod_{i=1}^{p} \nabla_i$
        $A_{ext} \leftarrow \prod_{\forall a_{i,k} \in Attribute} \Delta_{i,k}$
        $q_{ext} \leftarrow id$
        foreach $L \in L_{ext}$ **do**
          foreach $A \in A_{ext}$ **do**

```
                    if exists(A,L) then q_ext ← q_ext ∪ _L π_A
                    end
                end
            end
            Q_ext ← Q_ext o ( q_ext )
        end
        case _Pattern σ_Predicate do                    // sélection
            L_ext ← ∏_{i=1}^{p} ∇_i
            P_ext ← ∧_x ( ∨_y ( ∨_{a_{l,k}∈Δ_{x,y}} l_x.a_{l,k} ϖ_{l,k} v_{l,k} ) )
            q_ext ← id
            foreach L∈L_ext do
                foreach P∈P_ext do
                    if exists(P,L) then q_ext ← q_ext ∪ _L σ_P
                    end
                end
            end
            Q_ext ← Q_ext o ( q_ext )
        end
    end
end
end.
```

Exemple. Considérons la requête $_{\texttt{Author-}\ -1}\pi_{\texttt{Author.firstname,Author.lastname,1.title}}(_{\texttt{Author-}\ -1}\sigma_{\texttt{Author.firstname='Charles'}\wedge\texttt{Author.lastname='Baudelaire'}})$.

L'opérateur de projection est réécrit en fonction des différents labels similaires du pattern, ∇_{Author} = { `Author`, `author`, `Writer` } et ∇_{Book} = { `Book`, `book`, `Publication` }, et des différents attributs similaires, $\Delta_{Author.firstname}$ = { `Author.firstname`, `Writer.firstname`, `author.firstname` } et $\Delta_{Author.lastname}$ = { `Author.lastname`, `Writer.lastname`, `author.lastname` }.

L'algorithme construit les ensembles suivants à partir desquels l'opérateur est réécrit.

L_{ext} = { `Author`, `author`, `Writer` } x {} x { `Book`, `book`, `Publication` }
= { {`Author,,Book`}, {`Author,,book`}, {`Author,,Publication`}, {`Writer,,Book`}, {`Writer,,book`}, {`Writer,,Publication`}, {`author,,Book`}, {`author,,book`}, {`author,,Publication`} }

A_{ext} = { `Author.firstname`, `Writer.firstname`, `author.firstname` } x { `Author.lastname`, `Writer.lastname`, `author.lastname` } x { `Book.title`, `book.title`, `Publication.title` }
= { {`Author.firstname,Author.lastname,Book.title`}, …, {`author.firstname,author.lastname,Publication.title`} }

La projection ainsi réécrite est de la forme suivante :

$_{\texttt{Author- -Book}}\pi_{\texttt{Author.firstname,Author.lastname,Book.title}} \cup$

$_{\texttt{Writer- -Book}}\pi_{\texttt{Writer.firstname,Writer.lastname,Book.title}} \cup$

$_{\texttt{author- -Book}}\pi_{\texttt{author.firstname,author.lastname,Book.title}} \cup$

$_{\texttt{Author- -book}}\pi_{\texttt{Author.firstname,Author.lastname,Book.book}} \cup$

$_{\texttt{Writer- -book}}\pi_{\texttt{Writer.firstname,Writer.lastname,Book.book}} \cup$

$$\text{author- -book}\pi_{\text{author.firstname,author.lastname,Book.book}} \cup$$

$$\text{Author- -Publication}\pi_{\text{Author.firstname,Author.lastname,Book.Publication}} \cup$$

$$\text{Writer- -Publication}\pi_{\text{Writer.firstname,Writer.lastname,Book.Publication}} \cup$$

$$\text{author- -Publication}\pi_{\text{author.firstname,author.lastname,Book.Publication}}$$

L'opérateur de sélection est réécrit en fonction des différents labels similaires du pattern de sélection, ∇_{Author} = { Author, author, Writer }, et des différents attributs similaires du prédicat, $\Delta_{Author.lastname}$ = { Author.firstname, Writer.firstname, author.firstname } et $\Delta_{Author.lastname}$ = { Author.lastname, Writer.lastname, author.lastname }.

L'algorithme construit les ensembles suivants à partir desquels l'opérateur est réécrit.

L_{ext} = { Author, author, Writer } x {} x { Book, book, Publication }
= { {Author,,Book}, {Author,,book}, {Author,,Publication},
{Writer,,Book}, {Writer,,book}, {Writer,,Publication},
{author,,Book}, {author,,book}, {author,,Publication} }

On représente les prédicats normalisés avec des ensembles :

Autheur.firstname='Charles' ∧ Autheur.lastname='Baudelaire'
≡ { { Author.firstname='Charles' }, { Author.lastname='Baudelaire' } }

Ainsi :

P_{ext} = { { Author.firstname='Charles', Writer.firstname='Charles', author.firstname='Charles' }, { Author.lastname='Baudelaire', Writer.lastname='Baudelaire', author.lastname='Baudelaire' } }

La sélection devient alors :

$$\text{Author- -Book}\sigma_{\text{Author.firstname='Charles'}\wedge\text{Author.lastname='Baudelaire'}} \cup$$

$$\text{Writer- -Book}\sigma_{\text{Writer.firstname='Charles'}\wedge\text{Writer.lastname='Baudelaire'}} \cup$$

$$\text{author- -Book}\sigma_{\text{author.firstname='Charles'}\wedge\text{author.lastname='Baudelaire'}} \cup$$

$$\text{Author- -book}\sigma_{\text{Author.firstname='Charles'}\wedge\text{Author.lastname='Baudelaire'}} \cup$$

$$\text{Writer- -book}\sigma_{\text{Writer.firstname='Charles'}\wedge\text{Writer.lastname='Baudelaire'}} \cup$$

$$\text{author- -book}\sigma_{\text{author.firstname='Charles'}\wedge\text{author.lastname='Baudelaire'}} \cup$$

$$\text{Author- -Publication}\sigma_{\text{Author.firstname='Charles'}\wedge\text{Author.lastname='Baudelaire'}} \cup$$

$$\text{Writer- -Publication}\sigma_{\text{Writer.firstname='Charles'}\wedge\text{Writer.lastname='Baudelaire'}} \cup$$

$$\text{author- -Publication}\sigma_{\text{author.firstname='Charles'}\wedge\text{author.lastname='Baudelaire'}}$$

4.4. L'architecture de EasyGraphQuery

L'ensemble des mécanismes est mis en œuvre dans le système *EasyGraphQuery* dont l'architecture est donnée Figure 3, et **Erreur ! Source du renvoi introuvable.**comprenant les composantes suivantes :

– ***Dictionary*** : (cf. Section 4.1).

- ***SimilartyMatrix*** : il s'agit des matrices de similarité syntaxique et sémantique stockées sous forme d'un fichier JSON, hébergé dans le répertoire de Neo4J.
- ***Query Rewriter engine*** : prend en entrée la requête adressée par l'utilisateur, extrait les attributs similaires depuis le dictionnaire et reformule la requête.
- ***Synchronisation Engine***. Permet d'actualiser le dictionnaire à chaque requête d'insertion. Pour ce faire, on utilise deux fichiers pour sauvegarder la date des dernières mises à jours ; le premier *DMJ_Dictionary* enregistre, pour chaque attribut, la date de modification faite au niveau du dictionnaire, sous la forme « *attribut : date de modification* » tandis que le second *DMJ_Matrix* enregistre pour chaque attribut la date de modification au niveau des matrices de similarité. Avant de toute actualisation du dictionnaire on compare si la date de *DMJ_Matrix* est plus récente.
- ***Data structure extractor* :** extrait les noms des labels et des attributs depuis le fichier des logs pour éventuellement les insérer dans les matrices.

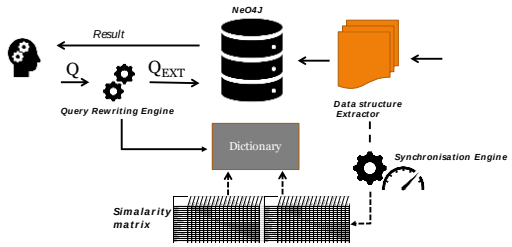


Figure 3 : L'architecture d'*EasyGraphQuery*.

La création du dictionnaire est faite de manière automatique au moment de l'insertion des données puis il est actualisé à chaque nouvelle insertion ou une actualisation des données. Pour des raisons de performance, la mise à jour est effectuée en continu et en arrière-plan.

5. Expérimentations

Jeu de données. Pour valider notre approche, nous considérons des données d'ontologies en raison de leur forte hétérogénéité structurelle. Nous avons utilisé la collection *Conference Track* mise à disposition par OAEI 2017[2]. Nous avons généré des instances synthétiques : 16 ontologies décrivant chacune une organisation d'une conférence. Le but étant d'évaluer le coût d'interrogation, les temps de génération et de chargement ne sont pas évalués.

Environnement de tests. Nous employons un cluster composé d'une machine (i5-4 cœurs, 8Go de RAM, 2To de disque, 1Gb/s de réseau) sur lequel nous avons installé une instance de Neo4J – version 3.2.

[2] http://oaei.ontologymatching.org/2017/

Le jeu de requêtes. Nous avons défini un jeu de 10 requêtes ; 3 requêtes de sélection, 3 requêtes de projection et 4 requêtes pour évaluer la combinaison sélection-projection. Pour la projection, le nombre d'attributs projeté est choisi entre 1 et '*'.

5.1. Construction du dictionnaire et de la matrice de similarité

Dans cette expérience nous étudions le temps nécessaire pour la création et l'actualisation du dictionnaire de similarité. Le Tableau 1 montre le temps de maintenance du dictionnaire au-fur-à-mesure que les ontologies sont insérées. Le résultat est nettement influencé par le nombre d'éléments (labels, relations, attributs) déjà présents dans le graphe. En effet, le fichier des logs est régulièrement analysé par notre parser mais il n'est pas nettoyé à chaque passage.

Tableau 1 Temps de maintenance du dictionnaire (en secondes) en fonction du nombre d'ontologies

Nombre d'ontologies	2	4	6	8
Tems de création /mise à jour (en secondes)	1.3s	4.2s	13.5s	18.7s
Taille du dictionnaire (KB)	2.7KB	2.9KB	3.4KB	3.5KB
Taille du fichier de logs analysé (parsé) en KB	1333KB	14534KB	17602KB	21265KB

5.2 Evaluation du module de réécriture de requêtes

Dans cette expérience, nous étudions le coût additionnel de notre solution, c'est à dire une interrogation avec une reformulation de la requête via notre algorithme de similarité, par rapport au coût de la requête sans reformulation, dite requête initiale. Nous comparons aussi le coût de la requête reformulée par rapport au cumul des coûts des sous-requêtes i.e. celles provenant de la décomposition des requêtes reformulées.

Tableau 2 Comparaison du temps d'éxécution (en secondes) des requêtes reformulées et des requêtes initiales (sans reformulation)

		Requête reformulée	Requête initiale	Cumul de requêtes résultantes
Projection	Q1.1	316	222	316
	Q1.2	0.160	0.008	0,166
	Q1.3	0.027	0.0013	0.027
Sélection	Q2.1	4.05	2.98	4.8
	Q2.2	0.77	0.77	0.85
	Q2.3	2.34	1.73	3.73
Combinaison (projection & sélection)	Q3.1	0.2734	0.2082	0.4062
	Q3.2	0.0055	0.0059	0.0073
	Q3.3	0.434	0.0431	0.9342
	Q3.4	0.324	0.324	0.659

Le Tableau 2 rapporte le temps d'exécution des requêtes reformulées, les requêtes initiales et le cumul des temps d'exécution des sous-requêtes. Une première

comparaison porte sur les temps d'exécution des requêtes réécrites et le cumul de temps d'exécution des sous requêtes. Nous pouvons observer que notre solution est, au pire, égale au cumul des sous requêtes, et elle peut aller jusqu'à 2 fois plus vite (cas des requêtes de combinaison par exemple dans le cas de notre jeu de données). En revanche, elle est au mieux égale au temps d'exécution d'une requête initiale.

Pour mieux interpréter ces résultats nous avons tracé l'exécution de nos requêtes. où nous pouvons remarquer par exemple que pour la requête Q1.2 reformulée (où notre algorithme fait appel à l'opérateur '*Union'*), Neo4J lance l'exécution des deux '*Match'* en parallèle ; et l'union des deux résultats n'est consolidé qu'après la fin de l'exécution du dernier '*Match'* (celui comportant le plus grand nombre de lignes). Plus précisent dans cette requête de projection Q2.1, deux types de labels sont évalués : le premier correspond au label de la requête initiale et qui traite 35054 lignes ; le second correspond au label ajouté par notre algorithme de réécriture et qui traite 10000 lignes. Le nombre de lignes explique les résultats du Tableau 2 et montre pourquoi notre solution est au au pire égale au temps d'exécution de la sous requêtes la plus lente et au mieux elle est égale à la requête initiale.

6. Conclusion

Dans cet article nous avons défini une approche qui repose sur la construction de dictionnaires de similarité des données permettant une réécriture des requêtes utilisateurs sans transformer les données stockées. Cette approche calcule, pour chaque attribut l'ensemble des attributs similaires (hétérogénéités syntaxique, sémantique et structurelle) pour réécrire de manière transparente les requêtes des utilisateurs. Les requêtes réécrites permettent de compléter les requêtes initiales et retourner l'ensemble complet des données. L'originalité de notre approche est de prendre en compte la variabilité de données « à la place de l'utilisateur ». Cette variabilité n'est plus résolue lors de l'écriture de chaque requête mais lors de la construction du dictionnaire qui reste un élément à approfondir. Nous avons abordé ce point au travers de matrices de similarité dans cet article et d'autres modalités de calcul restent à étudier. L'apport majeur est que la même requête utilisateur évaluée à des moments différents sera évaluée en fonction de l'état courant du dictionnaire : si des nouvelles données hétérogènes ont été ajoutées, cette variabilité sera automatiquement prise en compte dans l'évaluation de la requête.

En perspectives, nous souhaitons élargir le noyau algébrique d'opérateurs supportés dans notre approche ; en intégrant les opérations d'agrégation. Nous souhaitons également étudier l'impact des matrices avec un volume plus important.

Bibliographie

Beyer, K. S., V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, et E. J. Shekita . Jaql : A scripting language for large scale semi structured data analysis. VLDB (2011).

Floratou, A., N. Teletia, D. J. DeWitt, J. M. Patel, et D. Zhang (2012). Can the elephants handle the nosql onslaught ? VLDB 5(12), 1712–1723.

Florian Holzschuher and René Peinl. 2013. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4J. (EDBT '13), 195-204.

Getoor, L., Machanavajjhala, A. (2012). Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, Vol.5(12), 2018-2019.

H. Ben Hamadou, F. Ghozzi, A. Péninou, O. Teste. Interrogation de données structurellement hétérogènes dans les bases de données orientées documents (EGC 2018).

M. Chevalier, M. El Malki, A. Kopliku, R. Tournier, Olivier Teste. How Can We Implement a Multidimensional Data Warehouse Using NoSQL? Springer, p. 108-130, Vol. 241, (LNBIP), 2015.

R. Cattell. 2011. Scalable SQL and NoSQL data stores. SIGMOD Rec. 39, 4 (May 2011), 12-27.

I. Megdiche, O. Teste, C. Trojahn dos Santos, An Extensible Linear Approach for Holistic Ontology Matching, (ISWC'16), p.393-410.

S. Melnik, H. Garcia-Molina and E. Rahm, "Similarity flooding: a versatile graph matching algorithm and its application to schema matching, ICDE (2002), p. 117-128.

Shvaiko, P., Euzenat, J. (2005). A survey of schema-based matching approaches. *Journal on Data Semantics IV*, Stefano Spaccapietra (Ed.) Springer, 146-171.

Stonebraker, M. (2012). New opportunities for new sql. Communications of the ACM 5(11), 10–11.

Tahara, D., T. Diamond, et D. J. Abadi (2014). Sinew : a sql system for multi-structured data. In 2014 SIGMOD, pp. 815–826. ACM.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, et R. Mur- thy (2009). Hive : a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment 2(2), 1626–1629.

Wang, L., S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, et C. Wangz (2015). Schema management for document stores. Proceedings of the VLDB Endowment 8(9), 922–933.