# Atelier Web des Données

Organisateurs
Patricia Serrano Alvarado (LS2N, Université de Nantes),
Olivier Curé (LIGM, Université Paris-Est Marne la Vallée)

# PRÉFACE

Après une première édition en 2019 de l'Atelier Web des Données (AWD), nous proposons une seconde édition en 2020, à l'occasion de la conférence EGC'20 qui fêtera ses 20 ans à Bruxelles. Cet atelier vise à stimuler les discussions entre universitaires et industriels sur les défis du développement du Web des Données.

Depuis 2007, l'intérêt sur le Web des Données connaît une croissance importante et régulière. Les principes du Web des Données sont en train d'être adoptés par un nombre grandissant de fournisseurs de données. De plus, les technologies du Web des Données sont fondamentales pour la construction des graphes de connaissances, domaine en pleine croissance. En témoigne l'adoption dans des domaines aussi variés que les sciences de la vie, publications scientifiques, média, géographie, réseaux sociaux, approches industrielles, etc.

Afin de poursuivre le développement et l'adoption du Web des Données, des solutions facilitant l'intégration automatique, la décentralisation, la consommation, la capacité à déduire de nouvelles données/connaissances et la gestion du contrôle d'usage des jeux de données (contrôle d'accès, licences, législation, etc.) sont nécessaires.

Les contributions de cette seconde édition sont 2 articles longs et 1 démonstration logicielle. Le premier article propose une approche pour construire de manière interactive, des requêtes SPARQL analytiques. L'approche propose une interface de construction guidée de requêtes qui cache complétement le language SPARQL derriére une verbalization en langue naturelle. Le deuxième article propose une méthode pour anonymiser l'information représentée en graphes RDF. Cette méthode est une extension une de l'approche d'anatomisation pour des graphes de connaissances. La démonstration logicielle présente un outil de génération de mappings RDF pour des jeux de données structurés. L'outil fonctionne de manière semi-automatique en ne posant que des questions à l'utilisateur à propos de son jeu de données. L'objectif est l'automatisation de la partie du processus d'intégration au Web des Données, qui nécessite que l'utilisateur soit familiarisé avec RDFS, OWL et les langages de mapping RDF.

**Membres du comité de lecture**

Bernd Amann, LIP6, Université Sorbonne
Khalid Belhajjame, LAMSADE, Université Paris-Dauphine
Jérome David, Inria Rhône-Alpes, Université Grenoble Alpes
Emmanuel Desmontils, LS2N, Université de Nantes
Alban Gaignard, Institut du Thorax
François Goasdoué, IRISA, Université de Rennes 1
Luis-Daniel Ibañez, Université de Southampton
Clément Jonquet, LIRMM, Université de Montpellier
Myriam Lamolle, LIASD, Université Paris 8
Maxime Lefrançois, Laboratoire Hubert Curien, Ècole des Mines de Saint-Ètienne
Gabriela Montoya, Université Aalbork
Catherine Roussey, TSCF, Irstea
Fatiha Saïs, LRI, Université Paris
Hala Skaf-Molli, LS2N, Université de Nantes
Antoine Zimmermann, Institut Henri Fayol, Ècole des Mines de Saint-Ètienne

## TABLE DES MATIÈRES

# Construction guidée de requêtes analytiques sur des graphes RDF

Sébastien Ferré*

*IRISA, Campus de Beaulieu, 35042 Rennes cedex, France,
ferre@irisa.fr,
http://people.irisa.fr/Sebastien.Ferre/

**Résumé.** Alors que de plus en plus de données sont disponibles sous forme de graphes RDF, la possibilité de faire des requêtes analytiques devient un enjeu important du Web des données. Les approches existantes nécessitent la modélisation de cubes de données au-dessus des graphes RDF. Nous proposons une approche qui permet de répondre à des requêtes analytiques directement sur des graphes RDF non modifiés, en exploitant les possibilités de calcul de SPARQL 1.1 (expressions, agrégations). Nous nous appuyons sur le patron de conception `N<A>F` pour concevoir une interface de construction guidée de requêtes qui est intuitive en cachant complètement SPARQL derrière une verbalization en langue naturelle, et qui est réactive en donnant des résultats intermédiaires et des suggestions à chaque pas. Nos évaluations montrent que notre approche couvre une large classe de cas d'utilisations et passe à l'échelle de grands graphes.

## 1 Introduction

La différence entre recherche (dans des données) et analyse de données peut être illustrée par la différence entre *Quels films ont été réalisés par Tim Burton ?* et *Combien de films sont produits chaque année et dans chaque pays ?*. L'analyse de données a été beaucoup étudiée dans les bases de données relationnelles avec les entrepôts de données et OLAP (Chaudhuri et Dayal, 1997) mais n'en est qu'au commencement dans le Web des données (Kämpgen et Harth, 2011; Hoefler et al., 2013; Colazzo et al., 2014; Höffner et al., 2016; Sherkhonov et al., 2017). La plupart de ces travaux adaptent l'approche OLAP aux graphes RDF. Typiquement, des administrateurs de données doivent d'abord extraire des cubes de données à partir de graphes RDF en spécifiant pour chaque cube ce que sont les observations, les dimensions et les mesures (Cyganiak et al., 2013). Les utilisateurs finaux peuvent alors utiliser les interfaces classiques de type OLAP pour visualiser les cubes et leur appliquer des transformations. Plus récemment, l'approche par questions-réponses (*Question Answering*) (Lopez et al., 2011) a été appliquée aux requêtes analytiques sur des cubes de données RDF (Unger et al., 2016; Höffner et al., 2016; Atzori et al., 2019) ; ainsi que l'approche de construction guidée de requêtes, par exemple en s'appuyant sur des patrons de requêtes pré-définis (Kovacic et al., 2018). Le principal inconvénient de l'usage de cubes de données est que les utilisateurs finaux n'ont plus d'accès direct au graphe original, et ne peuvent explorer que les cubes qui ont été définis

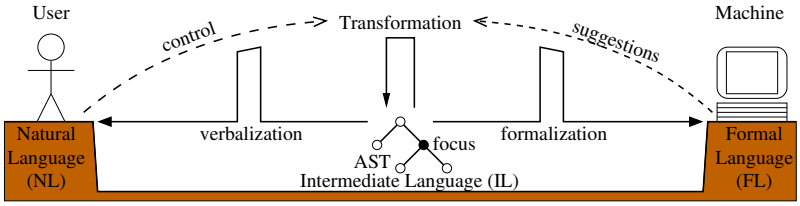



FIG. 1 – *Principe du patron de conception* N<A>F

par les administrateurs. Cet inconvénient est mitigé dans Colazzo et al. (2014) par un Schéma Analytique à partir duquel les utilisateurs peuvent dériver eux-mêmes de nombreux cubes. Cependant, il y a encore besoin d'un administrateur pour définir ce schéma.

Une conséquence du manque d'accès direct est une perte d'expressivité comparé à l'utilisation de SPARQL 1.1 (Kaminski et al., 2016). En effet, chaque vue OLAP peut être exprimée par une requête SPARQL où chaque ligne de résultat définit une cellule du cube avec ses coordonnées, alors que chaque cube de données ne permet qu'un ensemble limité de questions. Chaque nouvelle question peut nécessiter la définition d'un nouveau cube de données, ce qui nécessite généralement un administrateur. La contrepartie de l'expressivité de SPARQL est qu'il est bien plus difficile pour un utilisateur final d'écrire des requêtes SPARQL que d'intéragit via une interface OLAP. Un autre inconvénient de SPARQL est que les moteurs de requêtes ne sont pas optimisés pour les requêtes analytiques comme les moteurs OLAP. Cependant, ils sont déjà utilisables en pratique et leur optimisation dépasse le cadre de ce papier.

Dans ce papier, nous proposons une approche qui exploite les fonctionnalités de SPARQL 1.1 (expressions, agrégations) directement sur les graphes RDF non modifiés pour répondre à des requêtes analytiques. Nous nous appuyons sur le patron de conception N<A>F (Ferré, 2016) pour concevoir une interface de construction guidée de requêtes qui soit à la fois *intuitive* et *réactive*. Elle est intuitive en cachant complètement SPARQL derrière une verbalisation en langue naturelle des requêtes construites. Elle est aussi réactive en donnant des résultats intermédiaires et des suggestions de raffinement de la requête, à chaque pas d'interaction. Ainsi, notre interface utilisateur partage certains traits avec l'approche questions-réponses (verbalisation en langue naturelle) et avec OLAP (visualisation interactive).

Le papier est organisé comme suit. La section 2 présente succinctement le patron de conception N<A>F, qui sert de cadre formel à ce travail. La section 3 formalise la construction de requêtes analytiques dans le cadre N<A>F. La section 4 présente une évaluation sur le challenge QALD-6 et la section 5 présente une application aux données de Persée.

Le matériel en ligne associé à ce papier comprend une application web [1], les permaliens d'environ 70 requêtes analytiques et des captures vidéos (suivre `Learn > Examples` dans l'application web).

## 2 Le patron de conception N<A>F

La raison d'être du patron de conception N<A>F (Ferré, 2016) est de faire le pont entre les langues naturelles (NL) et les langages formels (FL, ici SPARQL). La figure 1 résume

1. http://www.irisa.fr/LIS/ferre/sparklis/

le principe de N<A>F. L'utilisateur est du côté NL et ne comprend pas FL. La machine, ici un point d'accès SPARQL, est du côté FL et ne comprend pas NL. L'élément central du pont est fait d'arbres de syntaxe abstraite (AST), appartenant à un langage intermédiaire (IL) entre NL et FL. IL est conçu pour rendre les traductions des AST vers NL (*verbalisation*) et FL (*formalisation*) aussi simple que possible. Il n'a pas besoin d'une syntaxe concrète, NL et FL jouent ce rôle respectivement pour l'utilisateur et la machine. Les AST sont des structures arborescentes où les nœuds représentent les sous-structures d'une requêtes. Chaque nœud $X = \mathbf{C}(X_1, \ldots, X_n)$ est caractérisé par un *constructeur* $\mathbf{C}$, le type du nœud, et un tuple de nœuds $X_1, \ldots, X_n$, les structures filles du nœud. N<A>F suit l'approche d'un constructeur de requêtes, où la structure qui est construite incrémentalement est précisément un AST. À la différence d'autres constructeurs de requêtes, la requête envoyée à la machine (FL) et la requête affichée à l'utilisateur (NL) peuvent fortement différer : IL joue un rôle de médiation. L'AST en construction est initialement la requête la plus simple et est incrémentalement affinée par application de *transformations*. Une transformation peut insérer ou supprimer une sous-structure de la requête. Chaque transformation s'applique au *focus* de la requête, où le focus est un nœud distingué de l'AST que l'utilisateur peut déplacer librement. Les transformations possibles sont suggérées par le système en fonction de la sémantique du langage de requêtes et des données, puis verbalisées en NL et enfin sélectionnées par l'utilisateur.

Nous illustrons N<A>F sur les motifs de graphe simples. L'AST

$$\mathbf{That}(\mathbf{A}(\texttt{:Film}), \mathbf{Has}(\texttt{:director}, \mathbf{Term}(\texttt{:Spielberg})))$$

est une imbrication de constructeurs inspirés des constructions syntaxique de la langue naturelle, avec des termes RDF comme sous-structures atomiques (e.g., la classe :Film, le terme :Spielberg). Les traductions en FL et NL sont réalisées par un parcours récursif de l'AST.

— SPARQL : `?x a :Film. ?x :director :Spielberg.`
— English : `'a film whose director is Spielberg'`
— French : `'un film dont le directeur est Spielberg'`

L'AST est construit via la séquence suivante de transformations. Après chaque transformation, nous donnons l'AST obtenu en soulignant la position du focus :

(0) AST initial : $\underline{\mathbf{Something}}$
(1) classe :Film : $\mathbf{A}(\underline{\texttt{:Film}})$
(2) propriété :director : $\mathbf{That}(\mathbf{A}(\texttt{:Film}), \mathbf{Has}(\texttt{:director}, \underline{\mathbf{Something}}))$
(3) terme :Spielberg : $\mathbf{That}(\mathbf{A}(\texttt{:Film}), \mathbf{Has}(\texttt{:director}, \underline{\mathbf{Term}(\texttt{:Spielberg})}))$
(4) déplacement du focus : $\mathbf{That}(\underline{\mathbf{A}(\texttt{:Film})}, \mathbf{Has}(\texttt{:director}, \mathbf{Term}(\texttt{:Spielberg})))$

L'AST **Something** est traduit en SPARQL par une variable, donnant généralement accès aux valeurs d'une propriété (ex., le directeur des films en (2)). Sa verbalization de base est `'something'` mais se simplifie souvent comme en (2) où la verbalization de la requête est `'a film that has a director'`. Les transformations sont suggérées en fonction du focus et des résultats de la requête courante. Par exemple, en (2) seules les propriétés ayant pour sujet ou objet des films sont suggérées, et en (3) seuls les réalisateurs de films sont suggérés comme termes. À chaque étape, l'interface utilisateur montre : (a) la verbalisation de la requête courante avec le focus souligné, (b) les résultats de la requête SPARQL générée et (c) la liste des transformations suggérées.

# 3 Construction guidée de requêtes analytiques

La contribution de ce papier est de jeter un pont entre les utilisateurs et les possibilités de calcul de SPARQL 1.1 dans le but de permettre les requêtes analytiques directement sur les graphes RDF non modifiés. Nous réalisons ce pont comme une nouvelle instance du patron de conception N<A>F, c'est-à-dire en définissant de nouveaux AST avec leurs transformations, leur formalisation et leur verbalisation. Comparé à l'instance sur les motifs de graphes (voir la section 2), cette nouvelle instance introduit deux nouvelles sortes d'AST couvrant les calculs de SPARQL, en plus des motifs de graphe : les *tables* et les *expressions*. Les AST des requêtes analytiques sont de type table et contiennent des AST de type motif de graphe simple et/ou expression. La sémantique d'un AST $P$ de type "motif de graphe simple" est un ensemble de valuations $M(P)$, où chaque valuation $\mu \in M$ est une fonction partielle des variables du motif vers des termes RDF. La sémantique d'un AST $E$ de type "expression" est un terme RDF résultant d'un calcul $eval(E)$. La sémantique d'un AST $T$ de type "table" est une structure tabulaire avec un ensemble de *colonnes* $C(T)$, qui sont des variables, et un ensemble de *lignes* $R(T)$, où chaque ligne $r \in R(T)$ est une fonction partielle des colonnes vers des termes RDF.

Dans les sous-sections suivantes, nous couvrons progressivement les possibilités de calcul de SPARQL en définissant de nouveaux constructeurs d'AST. Chaque sous-section présente un ou deux cas d'usage en guise de motivation, puis définit formellement le nouveau constructeur. Les cas d'usage sont basés sur un jeu de données réelles, MONDIAL (May, 1999), qui contient des données géographiques (ex., pays, villes, lacs) avec de nombreuses données numériques (ex., population, superficie).

Les constructeurs d'expressions sont définis d'un bloc dans la sous-section 3.2. Pour chaque constructeur d'un AST table $T$, nous définissons :

— $C(T)$ : l'ensemble des colonnes,
— $R(T)$ : l'ensemble des lignes,
— $sparql(T)$ : la formalisation en SPARQL,
— $nl(T)$ : la verbalisation en NL (ici, en anglais),
— un ensemble de transformations introduisant ce constructeur.

Une remarque importante est que la fonction $sparql()$ n'est pas toujours appliquée à l'AST table lui-même mais parfois à une variante de cet AST qui dépend du focus. Ces variantes sont expliquées ci-dessous quand nécessaire. La formalisation globale d'un AST table $T$ est `SELECT * WHERE { ` $sparql(T')$ ` }` où $T'$ est la variante de $T$ dépendant du focus.

## 3.1 Tables primitives

Tous nos constructeurs de calcul définissent une table comme fonction d'une autre table. Nous avons donc besoin de constructeurs de tables primitives pour démarrer. Des candidats valables de tables primitives sont les tables des bases relationnelles, les cubes OLAP et les feuilles de calcul. Néanmoins, afin de permettre les requêtes analytiques directement sur des graphes RDF, nous proposons d'utiliser les motifs de graphe simples pour extraire des tables quelconques. En effet, les résultats d'une requête SPARQL (de type SELECT) prennent la forme d'une table. On peut ainsi réutiliser des travaux antérieurs sur la construction de motifs de graphes avec N<A>F, implémentée dans l'outil Sparklis (Ferré, 2017), afin de guider l'utilisateur dans la construction de ces tables primitives.

**Définition 1** *Soit $P$ un AST de motif de graphe simple. L'AST table $T = \mathbf{GetAnswers}(P)$ représente la table primitive où les lignes sont les valuations solutions de $P$.*

— $C(T) = Vars(sparql(P))$ — $sparql(T) = sparql(P)$
— $R(T) = M(P) = Result(sparql(P))$ — $nl(T) =$ `'give me` $nl(P)$`'`

Par exemple, si l'utilisateur veut construire la table primitive des pays avec leur population et leur superficie, 4 étapes sont suffisantes pour construire l'AST suivant :

$$T0 \;:=\; \mathbf{GetAnswers}(\mathbf{That}(\mathbf{A}(\texttt{:Country}),\\ \mathbf{And}(\mathbf{Has}(\texttt{:population}, \mathbf{Something}), \mathbf{Has}(\texttt{:area}, \mathbf{Something}))))$$

qui se traduit en SPARQL : `?x1 a :Country ; :population ?x2 ; :area ?x3.`
et en anglais : `'give me every country that has a population and that has an area'.`

## 3.2 Expressions dans BIND et FILTER

Cas d'usage (E) : *Donne-moi la densité de population pour chaque pays, à partir de la population et de la superficie*. Les expressions SPARQL (ex., `?x2 / ?x3`) sont utilisées dans deux sortes de contextes en SPARQL : *filtrage* (`FILTER`) and *définition* (`BIND`, `SELECT`, `GROUP BY`). Étant donné une table, un filtrage effectue une sélection sur l'ensemble des lignes sur la base d'une expression booléenne ; tandis qu'une définition ajoute une colonne et calcule ses valeurs pour chaque ligne avec l'expression. Dans les deux cas, l'expression ne peut faire référence qu'aux colonnes de la table. Les définitions permettent de dériver des informations qui ne sont pas explicitement représentées dans le graphe, ex. la densité de population d'une pays dans le cas d'usage (E).

Nous définissons d'abord les AST expression car ce sont des sous-structures des filtrages et des définitions. Nous les définissons de façon classique, comme une composition de constantes, de variables et de d'opérateurs/fonctions.

**Définition 2 (expression)** *Un AST expression $E$ est composé de termes RDF (constantes), de colonnes de table (variables) et d'opérateurs/fonctions SPARQL. Nous ajoutons aussi le constructeur d'*expression indéfinie *?? pour permettre la construction incrémentale d'expressions. Une expression $E$ est dite* définie *si elle ne contient pas de ??. On note $C(E)$ l'ensemble des colonnes utilisées par l'expression.*

La sémantique d'une expression $E$ est notée $eval_r(E)$, où $r$ est la ligne sur laquelle l'évaluation est effectuée. $eval_r(E)$ et $sparql(E)$ sont définies de façon évidente mais sont seulement définies quand $E$ est une expression définie. Si le focus est sur une sous-expression $E'$ alors seule cette sous-expression est évaluée et formalisée en SPARQL afin de montrer la valeur au focus. La verbalisation d'une expression résulte d'une imbrication des verbalisations de ses fonctions, opérateurs, colonnes et termes. Elle mélange des notations mathématiques et littérales selon lesquelles sont les plus claires : `'`$E_1$ `+` $E_2$`'` est clair pour tout le monde et moins verbeux que `'the addition of` $E_1$ `and` $E_2$`'`, tandis que `'`$E_1$ `or` $E_2$`'` est plus clair que `'`$E_1$ `||` $E_2$`'` pour de non informaticiens. La verbalisation des colonnes est dérivée des noms de classes/propriétés utilisées dans le motif de graphe qui les introduit en tant que variable :

ex., dans $sparql(T0)$, `'the population'` pour $x_2$, `'the area'` pour $x_3$. Les parties de l'expression qui ne sont pas sous le focus sont affichées en grisé pour indiquer qu'elles ne sont pas calculées.

Les transformations utilisées pour construire des AST expressions sont les suivantes : (a) insérer un terme RDF au focus, (b) insérer une colonne au focus, (c) appliquer un opérateur ou une fonction au focus. Quand un opérateur/fonction est appliqué, le focus est automatiquement déplacé à la prochaine sous-expression ?? s'il en existe une, typiquement un autre argument de la fonction si elle a plusieurs paramètres. Par exemple, la séquence de transformations et d'AST qui construit l'expression du cas (E) est la suivante :

(0) expression initiale : $E = \underline{??}$
(1) insérer la colonne $x_2$ (population) : $E = \underline{x_2}$
(2) appliquer l'opérateur / (division) : $E = x_2 \,/\, \underline{??}$
(3) insérer la colonne $x_3$ (area) : $E = \underline{x_2 \,/\, x_3}$

Il existe des contraintes sur les transformations applicables, de façon à éviter la construction d'expressions mal formées. Les contraintes sur les colonnes pouvant être insérées sont définies par les constructeurs de table les utilisant. Des contraintes de type sont aussi utilisées pour déterminer quels fonctions sont applicables en fonction du focus, des types de données des colonnes et des signatures des fonctions. Dans l'exemple précédent, à l'étape (2), seuls des fonctions numériques peuvent être appliquées car la colonne $x_2$ contient des entiers ; et à l'étape (3), seules les colonnes numériques peuvent être insérées.

Nous définissons maintenant les deux constructeurs de tables qui utilisent une expression.

**Définition 3 (filtrage)** *Soit $T_1$ un AST table, et $E$ un AST d'expression booléenne tel que $C(E) \subseteq C(T_1)$. L'AST table $T = \mathbf{SelectRows}(T_1, E)$ représente le filtrage des lignes de $T_1$ qui vérifient $E$.*

— $C(T) = C(T_1)$
— $R(T) = \begin{cases} \{r \mid r \in R(T_1), eval_r(E) = true\} & \textit{si } E \textit{ est définie}, \\ R(T_1) & \textit{sinon} \end{cases}$
— $sparql(T) = \begin{cases} sparql(T_1)\ \texttt{FILTER (}\ sparql(E)\ \texttt{)} & \textit{si } E \textit{ est définie} \\ sparql(T_1) & \textit{sinon} \end{cases}$
— $nl(T) =$ `'`$nl(T_1)$ `where` $nl(E)$`'`

**Définition 4 (définition)** *Soit $T_1$ un AST table, $x \notin C(T_1)$ une variable fraîche, et $E$ un AST expression tel que $C(E) \subseteq C(T_1)$. L'AST table $T = \mathbf{AddColumn}(T_1, x, E)$ représente l'ajout d'une colonne $x$ à $T_1$, et la définition de cette nouvelle colonne par $E$.*

— $C(T) = C(T_1) \cup \{x\}$
— $R(T) = \begin{cases} \{r \cup \{x \mapsto eval_r(E)\} \mid r \in R(T_1)\} & \textit{si } E \textit{ est définie}, \\ R(T_1) & \textit{sinon } (x \textit{ est non défini}) \end{cases}$
— $sparql(T) = \begin{cases} sparql(T_1)\ \texttt{BIND (}\ sparql(E)\ \texttt{AS ?}x\ \texttt{)} & \textit{si } E \textit{ est définie}, \\ sparql(T_1) & \textit{sinon} \end{cases}$
— $nl(T) = \begin{cases} \texttt{`}nl(T_1)\ \texttt{and give me}\ name(x)\ \texttt{=}\ nl(E)\texttt{'} & \textit{si } name(x) \textit{ est défini}, \\ \texttt{`}nl(T_1)\ \texttt{and give me}\ nl(E)\texttt{'} & \textit{sinon} \end{cases}$

Dans les deux constructeurs, les colonnes utilisables par l'expressions sont celles de $T_1$. Si le focus est sur une sous-expression $E'$, alors la fonction $sparql()$ s'applique respectivement à $T' = \mathbf{SelectRows}(T_1, E')$ et $T' = \mathbf{AddColumn}(T_1, x, E')$, ignorant ainsi le reste de l'expression dans le calcul. Il suffit de déplacer le focus à la racine de l'AST expression ou au-dessus afin de récupérer le calcul complet.

Les filtrages et définitions sont introduits dans l'AST en mettant le focus sur une colonne et en lui appliquant une fonction ou un opérateur, ce qui initie la construction d'une expression. Le choix entre **SelectRows** et **AddColumn** est basé sur le type de l'expression entière selon l'hypothèse que les expressions booléennes sont généralement utilisées pour filtrer les lignes d'une table. Cependant, une autre transformation permet de forcer le choix de **AddColumn** pour obtenir des colonnes de valeurs booléennes. Enfin, une autre transformation permet d'attribuer un nom utilisateur $name(x)$ à la nouvelle colonne introduite par une définition, lequel est ensuite utilisé dans la verbalisation. Par exemple, en partant de la table primitive $T0$ définie dans la section précédente, le cas (E) peut être construit via la séquence suivante de transformations d'AST :

| | | |
|---|---|---|
| (0-4) | ... | : $T = T0$ (voir la section 3.1) |
| (5) | focus sur $x_2$ (population) | |
| (6) | appliquer l'opérateur / | : $T = \mathbf{AddColumn}(T0, x_4, x_2 \ / \ \underline{??})$ |
| (7) | insérer la colonne $x_3$ (area) | : $T = \mathbf{AddColumn}(T0, x_4, \underline{x_2 \ / \ x_3})$ |
| (8) | $name(x_4) :=$ `'population density'` | |

L'AST table résultant peut être traduit en SPARQL :

```
?x1 a :Country ; :population ?x2 ; :area ?x3. BIND (?x2/?x3 AS ?x4)
```

et en anglais :

```
'give me every country that has a population and that has an area,
 and give me the population density = the population / the area'.
```

## 3.3 Agrégations

Une *agrégation de base* est l'application d'un opérateur d'agrégation (ex., somme, moyenne) sur un ensemble d'entités ou valeurs, résultant en une unique valeur. Cas d'usage (A1) : *Combien de pays existe-t-il en Europe ?*. Une *agrégation simple* consiste à faire des groupes à partir d'un ensemble de valeurs selon un ou plusieurs critères, puis à appliquer un opérateur d'agrégation sur chaque groupe de valeurs. Une *agrégation multiple* étend l'agrégation simple en produisant plusieurs valeurs agrégées par groupe. Cas d'usage (A2) : *Donne-moi les moyennes de la population et de la superficie des pays, pour chaque continent*. Une agrégation correspond à une vue OLAP où les critères de regroupement sont les *dimensions* du cube et où les valeurs agrégées en sont les *mesures*. En SPARQL, les agrégations reposent sur l'usage d'opérateurs d'agrégation dans la clause `SELECT` et sur les clauses `GROUP BY`. Nous introduisons maintenant un nouveau constructeur de table qui couvre tous les types d'agrégations ci-dessus.

**Définition 5 (agrégations)** *Soit $T_1$ un AST table, $X \subseteq C(T_1)$ un ensemble de colonnes (possiblement vide), et $G = \{(g_j, y_j, z_j)\}_{j \in 1..m}$ un ensemble de triplets de la forme (opérateur d'agrégation, colonne, colonne) tels que pour tout $j \in 1..m$, $y_j \in C(T_1) \setminus X$ and $z_j \notin C(T_1)$. L'AST table $T = Aggregate(T_1, X, G)$ représente la table obtenue en regroupant les lignes*

*de $T_1$ selon les colonnes $X$ et, pour chaque $(g_j, y_j, z_j) \in G$, en définissant la nouvelle colonne $z_j$ par l'application de $g_j$ au multi-ensemble des valeurs de $y_j$ dans chaque groupe.*

- $C(T) = X \cup \{z_j\}_{j \in 1..m}$
- $R(T) = \{r_X \cup \{z_j \mapsto g_j(V_j)\}_{j \in 1..m} \mid r_X \in \pi_X R(T_1)$, *pour chaque* $(g_j, y_j, z_j) \in G$, $V_j = \{\{r(y_j) \mid r \in R(T_1), \pi_X r = r_X\}\}\ \}$
- $sparql(T) =$ `{ SELECT ?`$x_1 \ldots$`?`$x_n$ `(`$g_1$`(?`$y_1$`) AS ?`$z_1$`) ... (`$g_m$`(?`$y_m$`) AS ?`$z_m$`) WHERE {` $sparql(T_1)$ `} GROUP BY ?`$x_1 \ldots$`?`$x_n$ `}`
- $nl(T) =$ `'`$nl(T_1)$ `and [for ...each` $nl(x_i)$`, ...] give me ...`$nl(z_j)$`...'`

Dans $C(T)$, les colonnes $y$ disparaissent et sont remplacées par les colonnes agrégées $z$. Dans la définition de $R(T)$, la notation $\{\{...\}\}$ représente un multi-ensemble. En effet, la distinction avec les ensembles est importante pour les opérateurs d'agrégation `SUM` et `AVG`. Comme les autres constructeurs de tables génèrent des motifs de graphe SPARQL, pas des requêtes, nous utilisons dans la formalisation des agrégations une sous-requête SPARQL (requête SPARQL entre accolades). Cela permet l'imbrication arbitraires des différents constructeurs de tables. Si le focus est dans $T_1$, alors les fonctions $R()$ et $sparql()$ sont appliquées à $T_1$ au lieu de $T$, ignorant ainsi l'agrégation. Cela permet d'accéder temporairement aux colonnes cachées par l'agrégation. La verbalisation de chaque colonne agrégée $z_j$ est la verbalisation de $g_j(y_j)$ : ex., `'the number of` $nl(y_j)$`'`, `'the average` $nl(y_j)$`'`. Les crochets autour de `'for each...'` indiquent une partie optionnelle, vide dans le case où $X = \emptyset$.

De façon similaire aux filtrages et définitions de colonnes, une agrégation est introduite dans un AST en déplaçant le focus sur une colonne et en y appliquant un opérateur d'agrégation. Ensuite, d'autres colonnes peuvent être sélectionnées, soit comme critère de regroupement, soit comme autre colonne à agréger. Les contraintes de type sont aussi utilisées ici pour déterminer quels opérateurs peuvent être appliqués à une colonne. La séquence de transformations qui répond au cas (A2) à partir de la table primitive $T0$ est la suivante :

| | |
|---|---|
| (0-4) ... | : $T = T0$ (voir la section 3.1) |
| (5) propriété `:continent` | : $T = T1 = \mathbf{GetAnswers}(\ldots, \mathbf{Has}(\texttt{:continent}, \ldots)$ |
| (6) déplacer le focus sur $x_2$ (population) | |
| (7) appliquer agrégation `AVG` | : $T = \mathbf{Aggregate}(T1, \{\}, \{(\texttt{AVG}, x_2, x_5)\})$ |
| (8) grouper par $x_4$ (continent) | : $T = \mathbf{Aggregate}(T1, \{x_4\}, \{(\texttt{AVG}, x_2, x_5)\})$ |
| (9) appliquer `AVG` sur $x_3$ (area) | : $T = \mathbf{Aggregate}(T1, \{x_4\}, \{(\texttt{AVG}, x_2, x_5), (\texttt{AVG}, x_3, x_6)\})$ |

Différentes séquences sont possibles, les éléments pouvant être introduits dans presque n'importe quel ordre. Il est également possible de supprimer des éléments pour faire évoluer la requête sans recommencer de zéro. Par exemple, dans le cas (A2), il est possible de construire d'abord la requête sans la colonne $x_4$ (continent), et sans grouper par $x_4$. Cela calcule les moyennes des populations et superficies de tous les pays. Ensuite, on peut remettre le focus sur $x_1$ (country), insérer la propriété `:continent` pour ajouter la colonne $x_4$ dans la table avant agrégation, et pour finir mettre le focus sur l'agrégation et grouper par continent. L'AST résultant pour le cas (A2) se traduit en SPARQL par :

```
{ SELECT ?x4 (AVG(?x2) AS ?x5) (AVG(?x3) AS ?x6)
  WHERE { ?x1 a :Country. ?x1 :population ?x2.
          ?x1 :area ?x3. ?x1 :continent ?x4. }
  GROUP BY ?x4 }
```

et en anglais par :

```
‘ give me every country
      that has a population
      and that has an area
      and that has a continent
and for each continent,
      give me the average population and the average area ’
```

### 3.4 Combinaisons de constructeurs de tables

Le véritable atout de notre approche tient en la possibilité de combiner les différents constructeurs de tables (**SelectRows**, **AddColumn**, **Aggregate**) et donc les différents type de calculs correspondants. Nous illustrons cela avec trois nouveaux cas d'utilisation.

**Agrégation de définitions.** Cas d'usage (C1) : *Quel est le PIB par habitant moyen par continent ?* Ce cas nécessite de calculer le PIB par habitant via l'expresion (PIB total $\times 10^6$/ population) pour chaque pays (définition), puis de faire la moyenne des PIB par habitant pour chaque continent (agrégation simple). La requête peut être construite en 13 étapes, produisant une imbrication de 3 constructeurs de tables.

**Comparaison d'agrégations.** Cas d'usage (C2) : *Quels continents ont un PIB agricole moyen supérieur à leur PIB tertiaire moyen ?* Ce case nécessite de calculer deux agrégations pour le même critère 'continent' (agrégation multiple), puis d'exprimer une inégalité entre les deux (filtrage). La requête peut être construite en 9 étapes, produisant une imbrication de 3 constructeurs de tables.

**Agrégations imbriquées.** Cas d'usage (C3) : *Donne-moi, pour chaque nombre d'îles possible dans un archipel, combien d'archipels ont ce nombre d'îles.* Ce cas nécessite premièrement de calculer le nombre d'îles par archipel (agrégation simple), et deuxièmement de calculer pour chacun de ces nombres d'îles le nombre d'archipels ayant ce nombre d'îles (agrégation simple). La requête peut être construite en 6 étapes, avec 3 constructeurs de table imbriqués.

### 3.5 Implémentation

Nous avons complètement implémenté notre approche dans l'outil Sparklis. Sa version de base couvrait les motifs de graphes et fournissait donc tout ce qu'il faut pour construire nos tables primitives. Grâce à la généricité de `N<A>F`, aucun changement n'est nécessaire dans l'interface utilisateur. L'impact de notre approche apparaît seulement à l'utilisateur au travers d'un langage de requête plus riche et de suggestions supplémentaires. Cela facilite la transition vers la nouvelle version. En revanche, dans l'implémentation, un remaniement important du langage intermédiaire a été nécessaire avec l'introduction des AST de tables et d'expressions en plus des AST de motifs de graphe. De nouveaux composants ont été introduits, par exemple pour l'inférence et la vérification de types dans le calcul des suggestions.

## 4 Évaluation

Le challenge QALD-6 (*Questions Answering over Linked Data*) a introduit une nouvelle tâche intitulée “Questions-réponses statistiques sur des cubes de données RDF” (Unger et al.,

2016). Le jeu de données contient environ 4 millions de transactions sur des dépenses gouvernementales dans le monde, organisée en 50 cubes de données. Il contient environ 16M triplets au total. Bien que le jeu de données soit représenté sous forme d'un ensemble de cubes, nous avons répondu aux questions du challenge en construisant des tables primitives à partir de motifs de graphes, comme nous le ferions pour un quelconque jeu de données RDF. Nous avons officiellement participé au challenge en soumettant les réponses obtenues en construisant des requêtes dans notre implémentation.

**Expressivité.** Nous évaluons l'expressivité de notre approche en mesurant la couverture des questions de QALD-6. Sur les 150 questions (entraînement+test), 148 questions sont toutes des agrégations basiques ou simples et sont donc couvertes par notre approche ; et 2 questions (training-Q23 et test-Q23) sont des comparaisons de deux agrégations et ne sont pas couvertes par notre approche. La raison est que les deux agrégations utilisent des motifs de graphe disjoints, alors que notre approche est limitée à un seul motif de graphe connexe.

**Correction.** Dans le challenge, nous sommes donc parvenus à répondre à 49/50 questions tests. Sur les 49 réponses soumises, 47 étaient correctes, d'où un taux de succès de 94% (mesure officielle : $F_1 = 0.95$). Les deux erreurs proviennent d'une ambiguité dans les questions Q35 et Q42, qui admettent plusieurs réponses aussi plausibles l'une que l'autre du fait que plusieurs URI ont la même étiquette. Bien que notre approche ne soit pas directement comparable aux autres participants qui ont utilisé l'approche questions-réponses en langue naturelle, il est utile de mentionner que leurs taux de succès étaient de 50% pour QA$^3$ ($F_1 = 0.53$), et de 38% pour CubeQA ($F_1 = 0.44$). Même si notre approche n'est pas aussi intuitive que les questions-réponses, elle fait la démonstration de sa pertinence en terme de taux de correction, tout en étant bien plus intuitive qu'écrire des requêtes en SPARQL. De plus, les autres approches nécessitent une représentation sous forme de cubes de données et ne sont donc pas directement applicables aux graphes RDF classiques.

**Réactivité.** Nous évaluons la réactivité du système en mesurant le temps nécessaire pour construire les requêtes dans notre implémentation par quelqu'un en maîtrisant bien l'interface (l'auteur de ce papier). Nos mesures de temps incluent les interactions utilisateur et sont donc une borne supérieure du temps d'éxécution par le système. Le temps de construction des requêtes va de 31s à 6min20s, et la moitié des requêtes sont construites en moins de 1min30s (temps médian). La plupart des questions de QALD-6 nécessitent 5-10 étapes. Cela démontre que notre implémentation est suffisamment réactive pour satisfaire de véritables besoins d'information sur de grands jeux de données. Cependant, cela ne prouve rien concernant le temps nécessaire à de véritables utilisateurs pour construire les requêtes (utilisabilité).

# 5 Application

Persée [2] est une organisation (UMS CNRS) qui fournit un accès libre à une collection de plus de 600.000 publications scientifiques, notamment dans le domaine des humanités et sciences sociales. Il maintient un point d'accès SPARQL qui donne accès à leurs méta-données [3]. L'outil Sparklis a été officiellement adopté par Persée en février 2017 comme outil de recherche dans ces méta-données RDF. Nous avons consulté des membres de Persée pour savoir quels types de requêtes les intéressaient et nous avons été surpris de constater que parmi

2. `http://www.persee.fr/`
3. `http://data.persee.fr/`

celles-ci beaucoup étaient des requêtes analytiques. Elles couvrent d'ailleurs toutes les catégories de requêtes présentées dans ce papier. Nous donnons un échantillon représentatif de ces requêtes analytiques, avant de les commenter.

- Q1 *Combien de documents a une personne donnée (ex., Pierre Bourdieu) co-écrit avec chacun de ses co-auteurs ?*
- Q2 *Pour chaque auteur, obtenir le nombre et la liste des titres de ses documents.*
- Q3 *Comment a évolué le nombre moyen d'auteurs par document à travers le temps ?*
- Q4 *Y a-t-il des personnes dont la date de décès est antérieure à la date de naissance, révélant ainsi des erreurs dans les données ?*
- Q5 *Trier les auteurs par durée décroissante de leur période d'activité, laquelle est calculée comme la différence entre les dates maximale et minimale de publication de leurs documents.*
- Q6 *Pour chaque année de publication, obtenir le nombre d'articles publiés cette année, et dont le titre contient un terme donné (ex., "rural"), afin d'étudier l'évolution de ce terme dans le temps.*

Q1 et Q2 sont des agrégations, multiple dans le cas de Q2 ("la liste des" se traduit en SPARQL par `GROUP_CONCAT`). Q3 est une agrégation imbriquée, comptant tout d'abord les auteurs par document, puis faisant la moyenne de ces nombres d'auteurs par année de publication. Q4 implique un filtrage dont l'expression combine deux propriétés des personnes (dates de naissance et de décès). Q5 et Q6 sont des combinaisons complexes d'agrégations et de définitions. Dans Q5, on a d'abord deux agrégations, une pour le minimum et l'autre pour le maximum, puis une définition pour la différence entre le maximum et le minimum, et enfin un tri. Dans Q6, on a tout d'abord un filtrage par le terme choisi, puis une agrégation par année. Toutes les questions reçues ont pu être résolues dans notre implémentation. Cela conforte le niveau d'expressivité de notre approche.

Il est important de noter que le jeu de données de Persée n'est pas du tout organisé en cubes de données, et ne contient même pas de données numériques à part les dates. La propriété centrale est celle reliant les documents aux auteurs, qui est une relation $n$-$n$. Comme les questions ci-dessus le montrent, les documents et les auteurs servent autant de dimension que de mesure. Ces cas d'usage réel soutiennent donc notre approche directe des requêtes analytiques en RDF.

# 6 Conclusion

Nous avons montré que la puissance de SPARQL 1.1 pouvait être exploitée pour faire des requêtes analytiques sur des graphes RDF non modifiés au travers d'une interface de construction guidée de requêtes à la fois intuitive et réactive. L'approche a été implémentée dans Sparklis, évaluée sur le challenge QALD-6 et appliquée dans le graphe de connaissances de Persée.

# Références

Atzori, M., G. M. Mazzeo, et C. Zaniolo (2019). Qa3 : A natural language approach to question answering over rdf data cubes. *Semantic Web 10*(3), 587–604.

Chaudhuri, S. et U. Dayal (1997). An overview of data warehousing and OLAP technology. *ACM Sigmod record 26*(1), 65–74.

Colazzo, D., F. Goasdoué, I. Manolescu, et A. Roatiş (2014). RDF analytics : lenses over semantic graphs. In *Int. Conf. World Wide Web*, pp. 467–478. ACM.

Cyganiak, R., D. Reynolds, et J. Tennison (2013). The RDF data cube vocabulary.

Ferré, S. (2016). Bridging the gap between formal languages and natural languages with zippers. In H. Sack et al. (Eds.), *Extended Semantic Web Conf. (ESWC)*, pp. 269–284. Springer.

Ferré, S. (2017). Sparklis : An expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web : Interoperability, Usability, Applicability 8*(3), 405–418.

Hoefler, P., M. Granitzer, V. Sabol, et S. Lindstaedt (2013). Linked data query wizard : A tabular interface for the semantic web. In *The Semantic Web : ESWC 2013 Satellite Events*, pp. 173–177. Springer.

Höffner, K., J. Lehmann, et R. Usbeck (2016). CubeQA - question answering on RDF data cubes. In *Int. Semantic Web Conf.*, pp. 325–340. Springer.

Kaminski, M., E. V. Kostylev, et B. Cuenca Grau (2016). Semantics and expressive power of subqueries and aggregates in SPARQL 1.1. In *Int. Conf. World Wide Web*, pp. 227–238. ACM.

Kämpgen, B. et A. Harth (2011). Transforming statistical linked data for use in OLAP systems. In *Int. Conf. Semantic systems*, pp. 33–40. ACM.

Kovacic, I., C. G. Schuetz, S. Schausberger, R. Sumereder, et M. Schrefl (2018). Guided query composition with semantic OLAP patterns. In *EDBT/ICDT Workshops*, pp. 67–74.

Lopez, V., V. S. Uren, M. Sabou, et E. Motta (2011). Is question answering fit for the semantic web ? : A survey. *Semantic Web 2*(2), 125–155.

May, W. (1999). Information extraction and integration with FLORID : The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`.

Sherkhonov, E., B. C. Grau, E. Kharlamov, et E. V. Kostylev (2017). Semantic faceted search with aggregation and recursion. In C. d'Amato et al. (Eds.), *Int. Semantic Web Conf. (ISWC)*, LNCS 10587, pp. 594–610. Springer.

Unger, C., A.-C. N. Ngomo, et E. Cabrio (2016). 6th open challenge on question answering over linked data (QALD-6). In H. Sack et al. (Eds.), *Semantic Web Evaluation Challenge*, pp. 171–177. Springer.

# Summary

As more and more data are available as RDF graphs, the availability of tools for analytical queries beyond semantic search becomes a key issue of the Semantic Web. Previous work require the modelling of data cubes on top of RDF graphs. We propose an approach that directly answers analytical queries on unmodified RDF graphs by exploiting the computation features of SPARQL 1.1 (aggregations, expressions). We rely on the `N<A>F` design pattern to design a query builder user interface that is user-friendly by completely hiding SPARQL behind a verbalization in natural language; and responsive by giving intermediate results and suggestions at each step. Our evaluations show that our approach covers a large range of use cases, and scales well on large datasets.

# Anonymisation de graphes de connaissances par anatomisation

Maxime Thouvenot*,** Olivier Curé *
Lynda Temal ** Sarra Ben Abbès ** Philippe Calvez **

*LIGM (UMR 8049), CNRS, F-77454, MLV, France.
prénom.nom@u-pem.fr,
** ENGIE France.
prénom.nom@engie.com

**Résumé.** Le besoin d'anonymisation de données digitales s'accentue dans un contexte de Big Data. Ce constat concerne tous les types de données et implique donc les graphes de connaissances. Dans cet article, nous présentons une extension de l'approche d'anatomisation aux données représentées avec le modèle de données RDF.

## 1 Introduction

Avec l'émergence du phénomène Big Data, d'énormes quantités de données sont produites chaque jour et peuvent porter sur des sujets divers tels que des organisations, des entreprises, des personnes et bien d'autres encore. Ces données sont majoritairement stockées dans des centres de données (data centers) dans le but d'être analysées afin d'en extraire de nouvelles informations. La dernière décennie a également vu la naissance du mouvement Open Data qui s'est traduit par la volonté de nombreux acteurs, publiques comme privés, de diffuser leurs données jugées d'intêret général.

Cependant, une part non négligeable de ces données concerne des individus et constitue des informations sensibles qui peuvent menacer la vie privée de ces personnes. Si l'on prend l'exemple du domaine médical, la distribution de certains jeux de données posent des problèmes pour la protection du secret médical.

Beaucoup d'études ont été effectuées en vue de trouver une solution à ce problème d'anonymisation des données. Ce domaine consiste à modifier le contenu ou la structure de ces données afin de rendre très difficile voire impossible la « ré-identification » des personnes ou des entités concernées. De manière générale, nous appellerons "entités d'intêret" (*EI*) les cibles d'une technique d'anonymisation. Chaque EI est identifiée par un ensemble de valeurs que l'on appelle des attributs que l'on peut séparer en quatre catégories différentes :

— **Les identifiants explicites** (*IDE*) : ils permettent d'identifier explicitement une entité. Exemple : le numéro de sécurité sociale.
— **Les quasi-identifiants** (*QID*) : un ensemble d'attributs qui, lorsqu'ils sont utilisés ensemble, rendent possible la ré-identification. Exemple : l'âge, le code postal, etc..

— **Les attributs sensibles** (*AS*) : ils correspondent à des informations sensibles à propos d'une entité que l'on voudrait exploiter lors de nos analyses. Exemple : maladie, salaire, opinion politique, etc..
— **Les attributs non-sensibles** (*ANS*) : ils sont inutiles pour identifier une entité.

Parmi ces catégories, seules les trois premières correspondent à des attributs privés que nous devons considérer avec notre stratégie. Les IDE permettant une ré-identification sans ambiguïté, doivent d'office être supprimés du jeu de données tandis que les QID doivent être anonymisés. Les AS constituent des informations importantes que nous souhaitons pouvoir exploiter pour effectuer des analyses, nous ne les modifierons donc pas.

À ce jour, plusieurs méthodes d'anonymisation ont été développées, le plus souvent appliquées au modèle relationnel en délaissant le modèle de données RDF du Web Sémantique. Quelque soit le modèle de données ciblé, l'objectif est double : pouvoir garantir l'anonymité des entités présentes dans le jeu de données et assurer que ces données soient toujours utilisables (il est encore possible d'en tirer des informations intéressantes). Ce compromis est difficile à atteindre du fait qu'il existe une énorme quantité de connaissances externes, facilement accessibles, qui peuvent être utilisées afin de briser une stratégie d'anonymisation. Dans cet article, nous proposons une nouvelle approche, dénotée anatomisation, dans le contexte des graphes de connaissances.

# 2 Anatomisation de graphes de connaissances

L'idée de l'anatomisation pour les graphes RDF a été introduite pour la première fois par Radulovic et al. [6]. Leur article ne contient aucune implantation mais offre des pistes sur la façon dont cette technique pourrait être adaptée à cette structure de données. L'objectif de notre travail a consisté à s'appuyer sur ce papier et de l'étendre en prenant en compte la capacité d'inférence propre aux graphes RDF.

Dans cette section, nous commencerons par expliquer le principe de base de l'anatomisation puis nous poursuivrons avec nos contributions avant de finir par la présentation de notre approche en pratique.

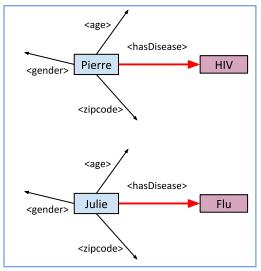
## 2.1 Présentation générale de l'anatomisation

L'anatomisation consiste à briser les relations entre les QID et leurs AS par l'insertion de noeuds vides intermédiaires (blank nodes). Ces derniers vont être utilisés pour pointer sur des groupes qui rassemblent les différents AS. Pour rappel, on définit un attribut comme l'objet d'un triplet dont le sujet est une EI. On peut noter que contrairement à d'autres méthodes d'anonymisation qui modifient la valeur des données (souvent par généralisation) ou les suppriment (e.g., [3] et [2]), nous ne modifions le graphe qu'au niveau structurel en ajoutant de nouveaux noeuds et arêtes.

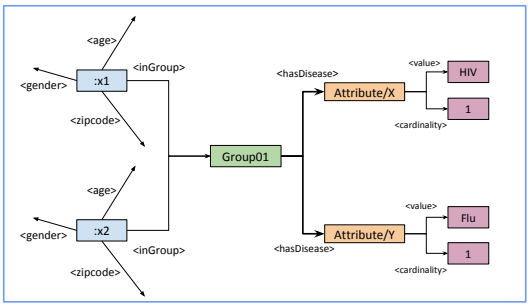
Nous donnons un exemple dans les Figures 1(a) et (b) avec des données médicales contenant des personnes reliées à leur maladie. Notons que deux types de noeuds ont été insérés :

— Les noeuds de type *groupes* utilisés pour casser la relation sensible (ici *Group1*)

FIG. 1: Anatomisation : couper les liens directs entre entités et attributs sensibles




(a) Ressources connectés à un attribut sensible (une maladie)




(b) Application de l'approche.

— Les noeuds de type *attributes* permettant de conserver des informations supplémentaires sur les attributs sensibles comme leur cardinalité (i.e., le nombre de fois qu'ils apparaissent dans le jeu de données).

## 2.2 Anatomisation guidée par l'ontologie

### 2.2.1 Principe

En préambule, nous rappelons que nous manipulons des graphes RDF donc des données structurées et associées à des ontologies : des classes ont été définies et sont organisées sous une forme hiérarchique. Ces outils constituent un moyen de comparer les différents concepts dans notre jeu de données, on souhaite donc les exploiter pour créer des groupes qui soient cohérents du point de vue sémantique.

Dans la Figure 2(a), nous proposons une ontologie pour les maladies où celles-ci peuvent être plus ou moins sérieuses (*Critical* et *Regular*) et concerner des parties ciblées du corps comme le coeur ou bien les poumons. Nous donnons aussi des instances de ces maladies tout en bas de l'arborescence. Nous présentons dans la Figure 2(b) un nouvel exemple d'anatomisation basée sur cette ontologie : le premier groupe contient les différentes maladies liées au coeur, le second les maladies des poumons et enfin le dernier regroupe les pathologies bénignes.

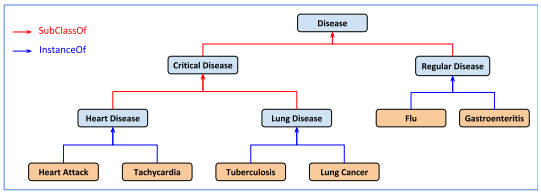
Il est également important de remarquer que les groupes possèdent désormais une sémantique propre car ils appartiennent chacun à une classe correspondant au *least common ancestor* des concepts résultant du raisonnement de *realization* de nos maladies. Nous revenons sur ces notions dans la section suivante.

### 2.2.2 Realization et Least Common Ancestor

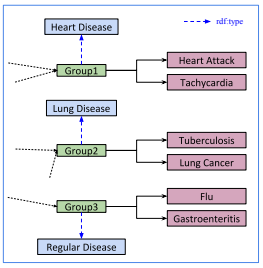
Le service de raisonnement *realization* (REA) [1] d'un invididu (ou d'un fait) correspond au concept le plus spécifique dont il est une instance. À titre d'exemple, dans la Figure 2(a) :

— $REA(Tachycardia) = HeartDisease$
— $REA(Tuberculosis) = LungDisease$

FIG. 2: Anatomisation guidée par une ontologie des maladies.




(a) Exemple de graphe de connaissances pour les maladies.




(b) Création de groupes spécifiques pour les types de maladie.

— $REA(Flu) = RegularDisease$

Par ailleurs, le *least common ancestor* (LCA) de deux concepts *A* et *B* correspond au concept le plus spécifique qui est un ancêtre de *A* et *B*. Nous étendons ce principe aux instances en considèrant que le LCA d'une instance correspondra au LCA de son REA. Si l'on reprend encore une fois le graphe de la figure 2(a) :

— $LCA(HeartAttack, Tachycardia) = HeartDisease$
— $LCA(HeartAttack, Tuberculosis) = CriticalDisease$
— $LCA(Flu, Gastroenteritis) = RegularDisease$
— $LCA(Flu, HeartAttack) = Disease$

### 2.2.3 Une mesure pour évaluer la similarité

Dans les exemples précédents, nous avions systématiquement plusieurs instances d'un même concept que l'on pouvait réunir dans un groupe. Cependant cette configuration n'est pas toujours présente dans la réalité : il n'est pas rare de trouver des concepts ne comptant qu'une seule instance.

Dans cette situation, notre approche précédente présente des limites pour déterminer le type d'un groupe. Ainsi, il faut définir une mesure générique afin d'évaluer la similarité entre deux instances, quand bien même elles appartiennent à des classes différentes.

Maedche et al. [4] introduisent dans leur papier la notion de *taxonomy similarity* qui calcule la similarité entre deux instances en se basant sur leur concept respectif et leur position dans l'ontologie (autrement dit l'arborescence des classes).

Cette mesure repose sur plusieurs notions que nous présentons maintenant.

**Upward Cotopy.** Tout d'abord, [4] définit une hiérarchie de concepts *H* comme une relation prenant en arguments deux concepts $C_1$ et $C_2$ : $H(C_1, C_2)$ signifie que "$C_1$ *est un sous-concept de* $C_2$".

Ils poursuivent ensuite en introduisant la notion d'Upward Cotopy : soient $C_i$ un concept quelconque appartenant à un ensemble de concepts $C$ et $H$ la hiérarchie de concepts associés. On définit alors l'Upward Cotopy comme :

$$UC(C_i,\, H) = \{C_j \in C \mid H(C_i, C_j) \ \vee \ C_j = C_i\} \tag{1}$$

Concrètement, cela correspond à l'ensemble des super-concepts de $C_i$ ainsi que $C_i$ lui-même. Par extension, on considère que l'UC d'une instance correspond à l'UC de son REA.

A partir de l'exemple de la Figure 2(a), nous obtenons :

— $UC(Heart\ Attack,\, H) = \{Disease, Critical\ Disease, Heart\ Disease\}$

— $UC(Tuberculosis,\, H) = \{Disease, Critical\ Disease, Lung\ Disease\}$

— $UC(Regular\ Disease,\, H) = \{Disease, Regular\ Disease\}$

**Concept Match.** Basé sur l'upward cotopy, le concept match entre deux concepts $C_1$ et $C_2$ appartenant à une hiérarchie *H* est défini comme :

$$CM(C_1,\, C_2) = \frac{|(UC(C_1,\, H) \ \cap \ UC(C_2,\, H))|}{|(UC(C_1,\, H) \ \cup \ UC(C_2,\, H))|} \tag{2}$$

On effectue donc un rapport entre la taille de l'intersection des UC et celle de leur union. De la même manière que pour l'upward cotopy, nous étendons ce principe aux instances et nous présentons un exemple avec *Tachycardia* et *Tuberculosis* :

$$CM(Tachycardia,\, Tuberculosis) = \frac{|(UC(Tachycardia,\, H) \ \cap \ UC(Tuberculosis,\, H))|}{|(UC(Tachycardia,\, H) \ \cup \ UC(Tuberculosis,\, H))|}$$

$$= \frac{|\{Heart\ Disease, Critical\ Disease, Disease\} \ \cap \ \{Lung\ Disease, Critical\ Disease, Disease\})|}{|\{Heart\ Disease, Critical\ Disease, Disease\} \ \cup \ \{Lung\ Disease, Critical\ Disease, Disease\})|}$$

$$= \frac{|\{Critical\ Disease, Disease\}|}{|\{Heart\ Disease, Lung\ Disease, Critical\ Disease, Disease\}|} = \frac{2}{4} = 0.5 \tag{3}$$

**Taxonomy similarity.** Enfin, cette notion entre deux instances $I_1$ et $I_2$ est définie comme :

$$TS(I_1, I_2) = \begin{cases} 1 & \text{si } I_1 = I_2 \\ \frac{CM(I_1, I_2)}{2} & \text{sinon} \end{cases} \tag{4}$$

Nous avons présenté le moyen de comparer les attributs sensibles, il faut maintenant les regrouper.

### 2.2.4 Regroupement hiérarchique

Les algorithmes de regroupement hiérarchique sont des techniques de classification automatiques cherchant à répartir un ensemble d'individus en des classes distinctes.

[4] défend l'idée que ces algorithmes sont préférables car ils produisent des hiérarchies de clusters et contiennent par conséquent plus d'informations que les techniques non-hiérarchiques.

Ils reprennent dans leur papier un algorithme dit "*bottom up*" (littéralement "*du bas vers le haut*") décrit dans Manning et al. [5].

L'algorithme part d'une situation de départ où tous les individus sont seuls dans une classe et où l'on dispose d'une mesure de dissimilarité entre les individus. La méthode va ensuite

```
1  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2  PREFIX owl: <http://www.w3.org/2002/07/owl#>
3
4  SELECT DISTINCT ?concept
5  WHERE {
6      {?concept rdfs:type rdfs:Class.}
7      UNION
8      {?concept rdfs:type owl:Class .}
9  }
```

FIG. 3: Requête SPARQL pour la récupération des concepts de l'ontologie.

procéder de manière itérative en fusionnant à chaque étape les deux classes les plus proches au sens de cette mesure et va ainsi créer une hiérarchie de ces individus.

Dans notre cas, les individus seront bien entendus les attributs sensibles et la mesure de dissimilarité sera la taxonomy similarity décrite plus haut. Dans la section suivante, nous présentons en détail notre méthode mettant en oeuvre ces outils.

## 2.3 Notre approche en pratique

Notre algorithme prend deux paramètres en entrée : le graphe de connaissances à anonymiser et l'URI de la relation qui relie les entités d'intêret à leur attribut sensible.

Notre méthode consiste à briser ces liens et de les remplacer par des groupes d'attributs afin d'assurer à la fois l'anonymité des individus dans le jeu de données et de conserver au maximum l'utilité des données.

Dans la suite, nous expliquons les différentes étapes de notre algorithme pour parvenir à ce but.

### 2.3.1 Phase de prétraitement

**Récupérer les concepts et calculer l'upward cotopy**

Avant de pouvoir entreprendre l'anonymisation, il nous faut déjà récupérer tous les concepts contenus dans l'ontologie du graphe du connaissances et il nous faut un moyen de pouvoir calculer leur upward cotopy. Nous avons mis au point deux requêtes SPARQL à cet effet que nous présentons respectivement dans les Figures 3 et 4.

Pour la récupération des concepts, nous avons recours à une union de façon à combiner des patterns alternatifs dans une même requête, tous les triplets satisfaisant l'un ou l'autre (ou les deux) des patterns seront retenus dans le résultat. On peut traduire cette requête par "nous voulons obtenir toutes les ressources dont le type est *rdfs :Class* ou (non exclusif) *owl :Class*".

La requête pour récupérer les ancêtres est plus compliquée car elle emploie des *property paths* au niveau du prédicat "*subClassOf*" à la ligne 7.

Sans l'opérateur "*", la requête se traduit simplement par "obtenir le super-concept du concept en entrée" mais avec le property path, nous pouvons remonter dans l'arborescence en suivant les liens "*subClassOf*" entre les concepts et récupérer tous les ancêtres.

Nous ne retenons pas les ancêtres *owl :Thing* et *rdfs :Resource* car s'il agit des concepts racines c'est-à-dire que tout individu dans le vocabulaire OWL (respectivement RDFS) est un

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ex: <http://example.org/examples#>

SELECT ?ancestor
WHERE {
    ex:concept  rdfs:subClassOf*  ?ancestor .
    FILTER(?ancestor  !=  owl:Thing) .
    FILTER(?ancestor  !=  rdfs:Resource)
}
```

FIG. 4: Requête SPARQL pour le calcul de l'upward cotopy.

| | **Disease** | **Critical Disease** | **Regular Disease** | **Heart Disease** | **Lung Disease** |
|---|---|---|---|---|---|
| Disease | 1 | | | | |
| Critical Disease | 0.25 | 1 | | | |
| Regular Disease | 0.25 | 0.166 | 1 | | |
| Heart Disease | 0.166 | 0.333 | 0.125 | 1 | |
| Lung Disease | 0.166 | 0.333 | 0.125 | 0.25 | 1 |

TAB. 1: Matrice symétrique des distances pour les concepts de l'ontologie des maladies dans 2(a)

descendant de *owl :Thing* (resp. *rdfs :Resource*). Ces concepts ne contiennent aucune valeur sémantique, c'est pourquoi nous les filtrons dans notre requête.

**Matrice de distances**

Au cours du processus d'anatomisation, nous allons être amenés à calculer la similarité entre les mêmes concepts à plusieurs reprises : on veut donc optimiser cette partie en évitant les redondances. Pour se faire, nous réalisons le calcul de la similarité au préalable et nous réutilisons cette valeur à chaque fois que l'on en a besoin.

Les taxonomy similarities sont contenues dans une structure de données que nous appelons la matrice de distance, nous garantissant un accès beaucoup plus rapide. Nous donnons un exemple de cette structure dans la Table 1 pour l'ontologie des maladies.

Du point de vue programmation, nous utilisons des tables de hachage imbriquées afin de représenter la matrice : chaque entrée correspond à un concept et pointe sur une nouvelle table de hachage où les entrées sont des concepts qui pointent sur la valeur de similarité.

Un schéma est fourni dans la Figure 5 : $H[Heart\ Disease][Disease]$ correspond à la valeur de similarité entre ces deux concepts et vaut donc $0.166$.

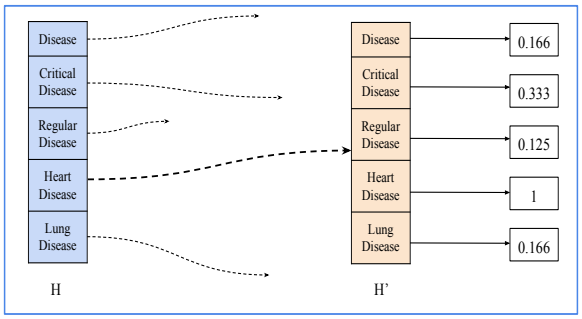



FIG. 5: Tables de hachage imbriquées pour la matrice de distances : exemple avec le concept *Heart Disease*.

Finalement, notre prétraitement se déroule en deux temps :

1. En premier lieu, nous récupérons tous les concepts dans l'ontologie du graphe de connaissances et nous calculons leur upward cotopy grâce aux requêtes des Figures 3 et 4.
2. On calcule ensuite la taxonomy similarity entre toutes les paires de concepts possibles et nous les stockons dans la matrice de distances.

On peut maintenant poursuivre avec la formation des groupes.

### 2.3.2 Obtenir les groupes initiaux

Durant la phase de clusterisation, nous manipulons deux types d'objets :
— **Sensitive attribute** : utilisé pour représenter les attributs sensibles à anonymiser, il est constitué de deux champs à savoir sa valeur ainsi que sa cardinalité dans le graphe.
— **Cluster** : sert à représenter les groupes lors de l'anatomisation, il est constitué de deux champs : la liste des sensitive attributes qu'il contient et son type qui correspond au LCA des attributs.

Nous avons décidé de reprendre l'algorithme bottom-up pour le hierachical clustering proposé par [5], de ce fait, les clusters initiaux contiendront tous un seul attribut sensible.

À l'aide du prédicat fourni par l'utilisateur et d'une requête SPARQL que nous avons écrite, nous sommes capables de récupérer les valeurs des attributs sensibles, leur concepts issus de la realization et leur cardinalité. Nous présentons la requête en question dans la Figure 6(a) et un exemple de résultats possibles dans la Figure 6(b).

On peut remarquer l'usage de la fonction d'agrégation *COUNT* et de l'opérateur *GROUP BY* permettant, exactement comme en SQL, de calculer des opérations statistiques sur des ensembles d'enregistrements. Dans notre cas, nous récupérons tous les attributs sensibles et leur *REA* puis nous calculons leur nombre d'occurrences dans le jeu de données.

En nous basant sur les résultats obtenus de la requête de la Figure 6(a), nous créons les SensitiveAttributes et les englobons directement dans un objet Cluster dont le type sera le *REA* de l'attribut.

FIG. 6: Construction des clusters initiaux.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX disease: <http://examples.ontotext.com/disease#>

SELECT ?super ?attribute (COUNT(?attribute) as ?count)
WHERE {
    ?s  disease:hasDisease ?attribute .
    ?attribute a ?super .
} GROUP BY ?super ?attribute
```

(a) Requête SPARQL

| # | super | attribute | count |
|---|---|---|---|
| 1 | Heart Disease | Heart Attack | 3 |
| 2 | Heart Disease | Tachycardia | 2 |
| 3 | Lung Disease | Tuberculosis | 2 |
| 4 | Regular Disease | Flu | 1 |

(b) Résultats possibles pour la requête

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ex: <http://example.org/examples#>

SELECT  ?super
WHERE {
    ex:concept1  rdfs:subClassOf*  ?super .
    ex:concept2  rdfs:subClassOf*  ?super .
    FILTER  NOT  EXISTS  {
        ?moreSpecificClass  rdfs:subClassOf  ?super .
        ex:concept1  rdfs:subClassOf*  ?moreSpecificClass .
        ex:concept2  rdfs:subClassOf*  ?moreSpecificClass .
    }
    FILTER(?super  !=  owl:Thing) .
    FILTER(?super  !=  rdfs:Resource)
}
```

FIG. 7: Requête SPARQL utilisée calculer le LCA entre deux concepts.

### 2.3.3 Fusion des groupes

À présent que les premiers clusters sont formés, nous les stockons dans une liste $L_1$ et les regroupons de manière itérative :

1. Nous prenons le premier cluster dans $L_1$ (et l'enlevons de la liste).
2. Nous calculons sa similarité avec tous les autres clusters, y compris ceux étant déjà le résultat d'une fusion.
3. Nous fusionnons les deux clusters les plus proches, ce qui revient à :
   (a) Concaténer leur liste d'attributs sensibles.
   (b) Calculer le LCA de leur concept respectif afin d'obtenir le concept du nouveau cluster.
   (c) Si le second cluster appartient à $L_1$, il est enlevé de la liste.
4. Le nouveau cluster est ajouté dans une liste $L_2$.

Nous répétons cette procédure jusqu'à ce que $L_1$ soit vide.

Une requête SPARQL est de nouveau utilisée afin de calculer le LCA de deux concepts, nous l'exposons dans la Figure 7.

Celle-ci fait à la fois intervenir des property paths et une nouvelle variante du *FILTER* avec la commande *NOT EXISTS*.

```
1  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX owl: <http://www.w3.org/2002/07/owl#>
4  PREFIX ex: <http://example.org/examples#>
5
6  DELETE {?s  ex:predicate   ex:attributeValue.}
7  INSERT {
8      ?s   ex:inGroup  ex:groupX  .
9      ex:groupX  rdf:type  ex:concept .
10     ex:groupX  ex:predicate   ex:attributeX .
11     ex:attributeX  ex:value  ex:attributeValue .
12     ex:attributeX  ex:cardinality  ex:attributeCardinality .
13 }
14 WHERE {?s  ex:predicate  ex:attributeValue . }
```

FIG. 8: Requête SPARQL de mise à jour pour appliquer l'anatomisation

Empruntée directement au langage de requête SQL, la commande *EXISTS* renvoie un booléen (vrai ou faux) et est utilisée pour savoir si des triplets existent dans le graphe. L'opérateur *NOT* est quant à lui utilisé pour exprimer la négation.

La requête peut en fin de compte être divisée en deux parties principales :

— La première aux lignes 7 et 8 où l'on recherche l'ensemble des ancêtres communs de *Concept1* et *Concept2*.

— La seconde des lignes 9 à 13 : on filtre tous les ancêtres communs afin de garder uniquement le plus spécifique. Pour tous les ancêtres qui ne sont pas le LCA, la clause *NOT EXISTS* renverra *False*, ils seront donc captés par le FILTER et retirés des résultats de la requête.

Par ailleurs, on filtre de nouveau les concepts *owl :Thing* et *rdfs :Resource* car ils sont les ancêtres de toutes les classes et sont trop généraux.

### 2.3.4 Création des requêtes de mise à jour

Une fois que les clusters sont formés, l'algorithme peut produire les opérations de mise à jour à exécuter pour appliquer l'anatomisation. Nous itérons sur chacun des clusters et sur chacun des attributs qu'ils contiennent afin de créer les requêtes SPARQL adéquates.

Nous donnons un exemple de requête dans la Figure 8 ainsi qu'un schéma représentant son exécution dans la Figure 9. Les valeurs de *GroupX* et *AttributeX* sont simplement des indices dont la valeur évolue au fur et à mesure que l'on itère sur les clusters et les attributs.

La clause $DELETE$ nous sert à supprimer le lien direct entre un individu et son attribut sensible tandis que la clause $INSERT$ est utilisée dans le but d'ajouter les noeuds intermédiaires et les relations entre ces derniers.

## 2.4 L'implantation

Nous avons développé notre programme en Java en utilisant la librairie libre *Apache Jena* conçue pour les technologies du Web sémantique. Elle offre un large panel de modules et de classes utilitaires afin de rendre plus facile la manipulation de graphes RDF.

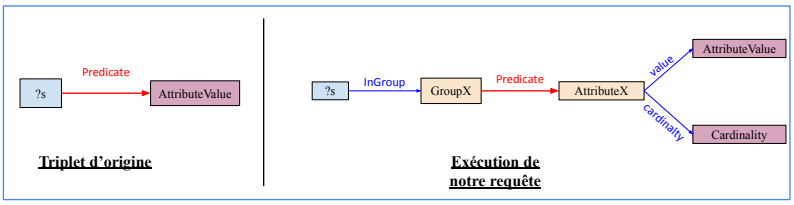
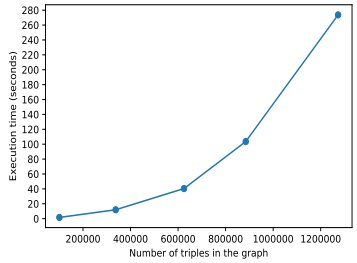
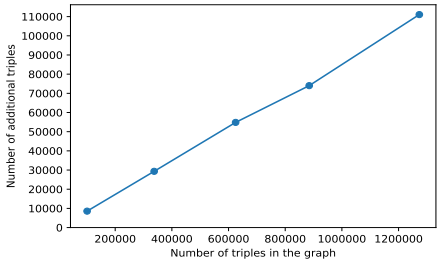



FIG. 9: Exécution d'une requête de mise à jour : exemple sur un triplet

FIG. 10: Évaluation de notre approche




(a) Temps d'exécution




(b) Nombre de triplets additionnels

Les graphes y sont représentés par l'intermédiaire d'une interface appelée "*Model*" et peuvent être construits à partir de fichiers, d'autres modèles ou bien manuellement. En outre, ce système d'interface est intéressant car il offre un plus haut niveau d'abstraction pour différents types de structures de données tels que des graphes stockés en mémoire, ceux stockés sur disque ou encore les modèles d'inférence rendant possible le raisonnement sur les données.

Nous avons également eu recours à *ARQ*, le moteur de requête de Jena pour créer et exécuter les différentes requêtes présentées précédemment.

# 3 Evaluation

Dans cette section, nous présentons une évaluation préliminaire de notre approche d'anatomisation sur une extension du jeu de données LUBM [1]. A partir de 5 graphes RDF de tailles différentes, allant de 100.000 à 1 million de triplets, nous avons testé (sur un laptop équipé de 16Go de RAM) les temps d'exécution de notre approche et l'évolution du nombre total de triplets des jeux de données anonymisés.

La Figure 10(a) montre que l'exécution de notre algorithme d'anatomisation croît de manière exponentielle avec le nombre distinct d'attributs sensibles. Ce résultat peut s'expliquer par notre approche bottom-up en complexité O($n^2$) : à chaque étape, nous prenons un groupe et nous le comparons avec tous les autres. Par contre, la Figure 10(b) met en évidence une crois-

1. http ://swat.cse.lehigh.edu/projects/lubm/

sance linéaire du nombre de triplets du graphe anonymisé, en effet, nous ajoutons 4 triplets à chaque attribut sensible (groupe, type de groupe, valuer et cardinalité).

# 4 Conclusion

Dans cet article, nous avons présenté une approche d'anonymisation basée sur la solution d'anatomisation pour des graphes de connaissances. A partir de relations sensibles, notre approche déclarative est capable de casser les liens entre les ressources et ses attributs sensibles tout en préservant la connectivité des noeuds du graphe et sans altérer de valeurs contenues dans le graphe, en dehors des QID. Nos futurs travaux concernerons l'impact du raisonnement à partir d'ontologies dans un contexte d'anonymisation, par exemple en considérant les propriétés inverses et transitives.

# Références

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook : Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[2] Remy Delanaux, Angela Bonifati, Marie-Christine Rousset, and Romuald Thion. Query-based linked data anonymization. In *ISWC, 2018*, pages 530–546.

[3] Benjamin Heitmann, Felix Hermsen, and Stefan Decker. k-RDF-neighbourhood anonymity : Combining structural and attribute-based anonymisation for linked data. In *PrivOn co-located with ISWC*, 2017.

[4] Alexander Maedche and Valentin Zacharias. Clustering ontology-based metadata in the semantic web. In *PKDD 2002*, pages 348–360, 2002.

[5] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[6] Filip Radulovic, Raúl García-Castro, and Asunción Gómez-Pérez. Towards the anonymisation of RDF data. In *SEKE*, pages 646–651, 2015.

# Summary

The need to anonymize digital data is increasing in a Big Data context. This fact concerns all data types including knowledge graphs. In this paper, we present an extension of the anatomization method for data represented with the RDF data model.

# SemanticBot: Intégration Semi-Automatique de Données au Web des Données

Benjamin Moreau*,** Nicolas Terpolilli**,
Patricia Serrano-Alvarado*

*GDD-LS2N – Nantes University, France
{Name.LastName@}univ-nantes.fr,
https://www.ls2n.fr
**OpenDataSoft
{Name.LastName}@opendatasoft.com
https://www.opendatasoft.com

**Résumé.** L'intégration de données structurées au Web des données est possible avec des *Mappings RDF*. Cependant, pour le développement de ces mappings il est nécessaire d'être familier avec RDF, en plus de connaître parfaitement le jeu de données. Le faible nombre de personnes satisfaisant ces deux conditions est un obstacle à la démocratisation du Web des données. Nous présentons ici un outil capable de générer, semi-automatiquement, des mappings RDF pour des jeux de données structurés. Le défi consiste à automatiser la partie du processus d'intégration qui nécessite que l'utilisateur soit familiarisé avec RDF.

## 1 Introduction et Motivation

Le Web des données est un ensemble de bonnes pratiques pour publier des données au format RDF [1]. Les données publiées dans le Web des données sont décrites à l'aide d'ontologies. Une ontologie est un ensemble de concepts (i.e., classes) et de relations (i.e., propriétés) représentant un domaine de connaissance. RDFS [2] et OWL [3] sont des langages permettant de décrire ces ontologies en RDF. Utiliser des ontologies déjà existantes est une bonne pratique qui permet d'accroitre l'interopérabilité entre les jeux de données du web des données.

Un *Mapping RDF* définit la transformation d'un jeu de données structuré (relationel, JSON, etc.) vers un jeu de données au format RDF. Cependant, développer un mapping RDF n'est pas trivial. Le Tableau 1 représente un extrait d'un jeu de données orienté colonnes. La Figure 1 est un mapping RDF permettant de transformer ce jeu de données en RDF. Pour écrire un tel mapping, il est nécessaire d'avoir la réponse à certaines questions, par exemple: (i) A quels concepts appartiennent les instances des colonnes *Name* et *Birth city* ? Ici, *Name* contient des personnes et *Birth city* des lieux. (ii) Quels sont les relations entre ces concepts ? Dans cet exemple, les lieux sont les villes de naissance des personnes. (iii) Quels sont les ontologies existantes pour décrire ce jeu de données. Dans ce cas, nous pourrions utiliser DBpedia, Schema.org, etc.

1. https://www.w3.org/RDF/
2. https://www.w3.org/TR/rdf-schema/
3. https://www.w3.org/TR/owl2-overview/

| string | date | string | string | float | float |
|---|---|---|---|---|---|
| **Name** | **Birth** | **Birth City** | **Birth Province** | **Lat** | **Long** |
| Augustus | 0062-09-23 | Rome | | | |
| Caligula | 0012-08-31 | Antitum | | | |
| Claudius | 0009-08-01 | Lugdunum | Gallia Lugdunensis | 47.932559 | 0.191854 |
| ... | ... | ... | ... | ... | ... |

TAB. 1 – *Un extrait de jeu de données structuré représentant des empereurs romains.*

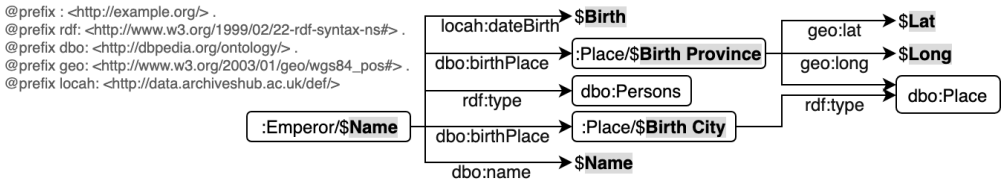



FIG. 1 – *Un mapping RDF du jeu de données des empereurs romains. Les informations en gras commençant par $ référencent les colonnes du dataset.*

Pour répondre à ces questions, il est à la fois nécessaire de bien connaitre le jeu de données et son domaine de connaissance, mais aussi, d'être familier avec RDF (RDFS, OWL et les langages de mapping RDF). Malheureusement, un nombre important de producteurs de données ne sont pas familiarisés avec le Web des données et ne sont pas encore prêts à s'y investir. Notre problème de recherche est le suivant: *Comment simplifier l'intégration de données structurées dans le Web des données ?* Le défi auquel nous sommes confrontés est d'automatiser la partie du processus d'intégration qui nécessite que l'utilisateur soit familiarisé avec RDF.

RML(1) et SPARQL-Generate(6) sont deux langages de mapping RDF. Même si il existe des syntaxes simplifiées tel que YARRRML(3), écrire un mapping RDF nécessite d'être familier avec RDF. Récemment, des outils tels que KARMA(2), RMLeditor(4) ou Juma(5) ont été proposés pour assister les utilisateurs pendant la création de mappings RDF. Cependant, ces outils ne sont pas encore adaptés aux utilisateurs n'étant pas familiers avec RDF.

Nous proposons *SemanticBot*, un outil capable de générer un mapping RDF pour un jeu de données structuré. Il fonctionne de manière semi-automatique en ne posant que des questions à l'utilisateur à propos de son jeu de données. Une version anglaise de cette démonstration a été publiée à ISWC 2019 [4].

## 2 L'outil SemanticBot

Afin de générer un mapping RDF pour un jeu de données structuré, *SemanticBot* se base sur les graphes de connaissances DBpedia et YAGO, les ontologies existantes dans LOV [5] et les langages OWL et RDFS.

4. https://hal.archives-ouvertes.fr/hal-02194315v1
5. https://lov.linkeddata.es/dataset/lov/

En résumé, pour chaque colonne du jeu de données, *SemanticBot* extrait un ensemble d'instances et cherche les entités correspondantes dans DBpedia et YAGO. L'objectif est de trouver les classes pouvant décrire les instances de chaque colonne. Ensuite, LOV est utilisé pour trouver les propriétés les plus cohérentes avec le nom et le types de chaque colonne. Ainsi, chaque instance de la colonne correspondra à l'objet de la propriété. Pour confirmer et enrichir ces correspondances, *SemanticBot* pose des questions simples à l'utilisateur. Les réponses aux questions servent à générer un premier mapping RDF qui est, par la suite, saturé selon des règles OWL et RDFS.

Nous détaillons ici les différentes étapes de notre outil en utilisant comme exemple le jeu de données des empereurs romains du Tableau 1.

**Identifier les classes.** Dans la première étape, l'instance *Augustus* de la colonne *Name* correspond à l'entité `http://dbpedia.org/resource/Augustus` de classe `dbo:Person`. *Augustus* est donc identifié comme entité de classe `dbo:Person`. Quand cette étape est terminée, nous obtenons deux correspondances suggérant que les colonnes *Name* et *Birth City* contiennent respectivement des entités de classes `dbo:Person` et `dbo:Place`. Ces suggestions sont ensuite proposées à l'utilisateur sous la forme de questions simples: "La colonne Name dans votre dataset contient-elle des Personnes ?". Pour cacher les URIs, les questions sont formulées en utilisant l'attribut `rdfs:label` des classes.

**Identifier les propriétés et leurs objets.** À l'étape suivante, la recherche des propriétés dans LOV, à l'aide du nom et type des colonnes, nous donne 5 résultats. Les colonnes *Name*, *Birth*, *Birth City*, *Lat* et *Long* sont respectivement identifiées comme contenant les objets des propriétés `dbo:name`, `locah:dateBirth`, `dbo:birthPlace`, `geo:lat` et `geo:long`. Comme précédemment, ces suggestions sont proposées à l'utilisateur: "La colonne Lat représente t-elle la Latitude d'un objet localisé ?". La question est formulée avec le `rdfs:label` et le `rdfs:domain` des propriétés.

**Identifier les sujets des propriétés.** Pour compléter, l'outil demande à l'utilisateur, pour chaque propriété validée, de sélectionner la colonne contenant les sujets de la propriété. Si l'utilisateur confirme la propriété `geo:lat` sur la colonne *Lat*, l'outil demande: "latitude est un attribut d'un objet localisé. Sélectionnez la colonne contenant ces objets localisés.". Dans notre exemple, si l'utilisateur répond correctement à la question, la colonne *Birth Province* sera associée au sujet de la propriété `geo:lat`.

Notre outil utilise des heuristiques pour limiter le nombre de questions posées à l'utilisateur: (i) Il ne suggère qu'au plus une classe et une propriété par colonne. (ii) Seule la classe correspondant au plus grand nombre d'instances dans une colonne est suggérée. (iii) La propriété suggérée pour une colonne est celle qui a le meilleur score de popularité sur LOV. (iv) Enfin, une propriété n'est pas proposée à l'utilisateur si son score LOV est inférieur à une valeur fixée.

L'ensemble des suggestions confirmées par l'utilisateur (classes et propriétés) sont utilisées pour générer un premier mapping RDF, lequel est saturé en lui appliquant les règles d'inférence RDFS et OWL [6].

6. Nous considérerons uniquement les règles RDFS `https://www.w3.org/TR/rdf11-mt/#rdfs-entailment` 2, 3, 5, 7, 9 et 11 et les règles OWL `https://www.w3.org/TR/owl-ref` basées sur `owl:equivalentClass` et `owl:equivalentProperty`.

Le mapping RDF de notre exemple est disponible en YARRRML à l'adresse `https://git.io/fjKY6` et le résultat de la transformation selon ce mapping est disponible ici: `https://git.io/fjKYo`.

# 3 Démonstration

*SemanticBot* encourage de nouveaux utilisateurs à faire leur premier pas dans le Web sémantique. Il peut être testé ici `https://semanticbot.opendatasoft.com/` avec des jeux de données d'Opendatasoft [7]. Des explications supplémentaires sur le fonctionnement de l'outil ainsi que son code source sont disponibles sur GitHub [8].

# Références

[1] Dimou, A., M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, et R. Van de Walle (2014). RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Workshop on Linked Data on the Web (LDOW) collocated with WWW*.

[2] Gupta, S., P. Szekely, C. A. Knoblock, A. Goel, M. Taheriyan, et M. Muslea (2012). Karma: A System for Mapping Structured Sources Into the Semantic Web. In *Extended Semantic Web Conference (ESWC), Poster&Demo*.

[3] Heyvaert, P., B. De Meester, A. Dimou, et R. Verborgh (2018). Declarative Rules for Linked Data Generation at Your Fingertips ! In *Extended Semantic Web Conference (ESWC), Poster&Demo*.

[4] Heyvaert, P., A. Dimou, A.-L. Herregodts, R. Verborgh, D. Schuurman, E. Mannens, et R. Van de Walle (2016). RMLEditor: a Graph-Based Mapping Editor for Linked Data Mappings. In *Extended Semantic Web Conference (ESWC)*.

[5] Junior, A. C., C. Debruyne, et D. O'Sullivan (2018). An Editor that Uses a Block Metaphor for Representing Semantic Mappings in Linked Data. In *Extended Semantic Web Conference (ESWC), Poster&Demo*.

[6] Lefrançois, M., A. Zimmermann, et N. Bakerally (2017). A SPARQL Extension For Generating RDF From Heterogeneous Formats. In *Extended Semantic Web Conference (ESWC)*.

# Summary

The integration of structured data to the web of data is possible with *RDF Mappings*. However, for the development of these mappings it is necessary to be familiar with RDF, in addition to knowing perfectly the dataset. The small number of people satisfying these two conditions is an obstacle to the democratization of the Web of data. We present here a tool able to generate, semi-automatically, RDF mappings for structured datasets. The challenge is to automate the part of the integration process that requires the user to be familiar with RDF.

7. `https://data.opendatasoft.com`
8. `https://github.com/opendatasoft/semantic-bot`

# Index