Génération de RDF à partir de sources de données aux formats hétérogènes

Maxime Lefrançois, Antoine Zimmermann, Noorani Bakerally

Univ Lyon, MINES Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516,
F-42023 Saint-Étienne, France
prenom.nom@emse.fr

Résumé. Contrairement à ce que promeut le Web des données, les données exposées par la plupart des organisations sont dans des formats non-RDF tels que CSV, JSON, ou XML. De plus sur le Web des objets, les objets contraints préféreront des formats binaires tels que EXI ou CBOR aux formats RDF textuels. Dans ce contexte, RDF peut toutefois servir de lingua franca pour l'interopérabilité sémantique, l'intégration de données aux formats hétérogènes, le raisonnement, et le requêtage. Dans ce but, plusieurs outils et formalismes permettent de transformer des documents non-RDF vers RDF, les plus flexibles étant basés sur des langages de transformation ou de correspondance (GRDDL, XSPARQL, R2RML, RML, CSVW, etc.). Cet article définit un nouveau langage, SPARQL-Generate, qui permet de générer du RDF à partir: (i) d'une base de données RDF, et (ii) d'un nombre quelconque de documents aux formats arbitraires. L'originalité de SPARQL-Generate est qu'il étend SPARQL 1.1, et peut donc (i) être appris facilement par les ingénieurs de la connaissance familiers de SPARQL, (ii) être implémenté au dessus de n'importe quel moteur SPARQL existant, (iii) tirer parti des mécanismes d'extension de SPARQL pour prendre en compte de futurs formats.

1 Introduction

Nous cherchons à faciliter l'accès aux formalismes et outils du Web Sémantique pour les entreprises, services Web, et objets contraints. Une étape clé pour utiliser ces formalismes est de générer du RDF à partir de documents ayant des formats variés. En effet, les entreprises et services web stockent et échangent les données dans une multitude de modèles et formats de données : les modèles de données relationnels ainsi que le format XML (pas RDF/XML) sont encore très présent. Les portails open data préfèrent CSV, et les API web : JSON. Quand aux objets contraints sur le Web des objets, ils préfèrent des formats légers, potentiellement binaires, tels que EXI ou CBOR. Dans ce contexte, les formats de données RDF (RDF/XML, Turtle, JSON-LD) ne remplaceront vraisemblablement jamais les formats de données existants. Par contre, le *modèle* de données RDF peut toujours servir de *lingua franca* pour l'interopérabilité sémantique, l'intégration de données aux formats hétérogènes, le raisonnement, et le requêtage.

Générer du RDF à partir d'autres modèles et formats est l'objet de plusieurs travaux de recherche et outils, qui ont parfois fait l'objet de standards. Cependant, dans le cadre des projets auxquels nous participons, nous avons identifié des cas d'utilisation et des besoins, écrits dans la section 2, que les approches existantes ne satisfont que partiellement. Notamment :

— la solution doit être expressive, flexible, et pouvoir prendre en compte de nouveaux formats de données à la demande ;

— elle doit permettre de générer du RDF à partir de plusieurs sources de données aux format hétérogènes, conjuguées à un ensemble de données RDF (*RDF dataset*).

— elle doit être aussi proche que possible du langage d'interrogation SPARQL, pour pouvoir être maîtrisée rapidement par les ingénieurs de la connaissance.

Nous décrirons les solutions existantes et leurs limitations dans la section 3. Afin de répondre à ces besoins, nous définissons le langage SPARQL-Generate, une extension de SPARQL 1.1 qui répond aux besoins ci-dessus et combine les avantages suivants : (i) il bénéficie de l'expressivité et de la flexibilité de SPARQL, notamment le mécanisme standard d'extension des fonctions de liaison ; (ii) il peut s'implémenter au dessus d'un moteur SPARQL existant.

La principale contribution de cet article est la formalisation de SPARQL-Generate, présentée dans la section 4. Cette formalisation permet alors de démontrer que SPARQL-Generate peut être implémenté au dessus de n'importe quel moteur SPARQL 1.1 existant (section 5.1). Nous décrivons ensuite brièvement l'implémentation qui a été réalisée sur Apache ARQ (section 5.2), et discutons finalement de l'évaluation de SPARQL-Generate dans la section 5.3.

2 Cas d'utilisation et besoins

Dans le cadre de sessions d'ingénierie de la connaissance organisées pour des partenaires industriels et académiques qui souhaitent bénéficier des avancées du Web Sémantique à moindre coût, nous avons identifié les cas d'utilisation et besoins suivants pour la génération de RDF à partir de modèles et formats non RDF.

Dans le contexte des données ouvertes, les organisations atteignent péniblement la troisième étoile sur les cinq requises pour le schéma de déploiement des données ouvertes liées [1]. Les données sont exposées sur le Web, plus ou moins structurées, et peuvent utiliser des formats non propriétaires. La quatrième étoile consiste à utiliser des IRI et les formalismes du Web Sémantique pour encoder les données, ce qui n'est pas le cas chez nos partenaires. Il s'agit donc de pouvoir générer du RDF à partir de plusieurs de ces sources, potentiellement dans des formats différents. Ceci tout en ayant un contrôle fin sur le RDF généré, et les liens entre jeux de données. Ce contrôle doit d'ailleurs pouvoir impliquer des données contextuelles en RDF. La liste des formats à partir desquels du RDF devra être généré doit pouvoir être étendue facilement. Et la solution doit être maîtrisable facilement par ceux qui ont des notions de RDF et SPARQL.

Sur le Web des objets, les dispositifs contraints doivent échanger des messages légers, ceci à cause de leurs contraintes inhérentes de bande passante, d'énergie, ou de mémoire. Les syntaxes RDF encodent beaucoup d'informations textuelles dont les IRI et les littéraux avec des IRI de types de données. Même si certains groupes de travail au W3C souhaitent définir des

1. http://5stardata.info/en/

syntaxes légères pour RDF (notamment une version EXI de RDF/XML ou CBOR de JSON-LD), il est à prédire que de nombreux autres formats binaires optimisés pour chaque objet ou application resteront utilisés.

De ces cas d'utilisation découlent donc les besoins suivants :

Besoin 1 : transformer plusieurs sources aux formats hétérogènes ;

Besoin 2 : contextualiser la transformation avec un ensemble de données RDF ;

Besoin 3 : être extensible à d'autres formats de données ;

Besoin 4 : simple cognitivement et facile à maîtriser par les experts en Web Sémantique.

Besoin 5 : implémentation facile au dessus de moteurs RDF ou SPARQL existants ;

Besoin 6 : la solution doit être la plus performante possible (temps, espace).

Besoin 7 : transformer aussi bien des formats binaires que des formats textuels ;

3 Approches existantes

Pour générer du RDF à partir de données aux modèles et formats hétérogènes, les fournisseurs ou consommateurs de données peuvent coder des mécanismes de traduction *ad hoc*, ce qui s'avère coûteux. Un certain nombre de travaux de recherche et d'outils permettent de simplifier cette tâche.

De nombreux *convertisseurs vers RDF* ont été référencés par le groupe d'intérêt SWEO (*Semantic Web Education and Outreach*) du W3C : `https://www.w3.org/wiki/ConverterToRdf`. Ils ciblent pour la plupart un format ou des métadonnées spécifiques, comme ID3tag, BibTeX, EXIT, etc. Certains peuvent convertir différents types de données en RDF, c'est le cas de Apache Any23, Datalift, ou bien Virtuoso Sponger. Citons également le standard Direct Mapping (Arenas et al., 2012), qui décrit une transformation par défaut pour les données relationnelles. Ces solutions restent ad hoc, et permettent peu ou prou de contrôler comment le RDF est généré. En conséquence, la sortie RDF décrit souvent la structure des données plutôt que les données elle-mêmes. Il serait possible de transformer cette sortie à l'aide de règles SPARQL CONSTRUCT, mais cela nécessiterait de se familiariser avec le vocabulaire utilisé pour la sortie de chacun de ces outils. Ils ne satisfont donc pas les besoins exprimés dans la section 2.

D'autres approches proposent d'utiliser un langage de transformation ou de correspondance pour paramétrer la génération de RDF. Cependant, la plupart de ces solutions s'intéressent à un ou quelques modèles (comme le modèle relationnel) ou formats (comme le format JSON) de données spécifiques. Par exemple GRDDL utilise XSLT et cible XML (Connolly, 2007). XSPARQL est basé sur XQuery et ciblait originellement XML (Polleres et al., 2009), avant d'être étendu pour le modèle relationnel (Lopes et al., 2011), puis pour JSON (Dell'Aglio et al., 2014). Plusieurs autres formalismes ont été proposés pour générer du RDF à partir de données relationnelles (Hert et al., 2011). Ces travaux sont à l'origine du standard R2RML (Das et al., 2012) [2], qui définit un vocabulaire RDF pour décrire la transformation des données en RDF. Enfin, CSVW (Tandy et al., 2015) adopte cette même dernière approche, mais cible le format de données CSV.

2. XSPARQL est une implémentation de R2RML (Lopes et al., 2011)

Une approche qui se distingue est RML (Dimou et al., 2014), qui étend le vocabulaire R2RML pour décrire des sources logiques différentes des tables de bases de données relationnelles : JSON (à l'aide de JSONPath), XML (à l'aide de XPath), CSV [3], TSV, ou HTML (à l'aide des sélecteurs CSS3). L'approche est implémentée sur Sesame [4]. RML répond au moins aux besoins 1, 3, 5. Il serait possible d'implémenter le support de types de données binaires (besoin 7), et des recherches sont en cours pour prendre en compte des sources RDF sur le linked data RDF (besoin 2).

Dans la suite de cet article, nous présentons une alternative à RML basée sur une extension de SPARQL 1.1, nommée SPARQL-Generate, qui bénéficie de l'expressivité et de l'extensibilité de SPARQL 1.1, et peut être implémenté au dessus de ses moteurs.

4 Spécification de SPARQL-Generate

SPARQL-Generate est basé sur un langage qui requête la combinaison d'un ensemble de données RDF (RDF *dataset*) [5] et de ce qu'on appelle un *ensemble de documents* (*document-set*), où chaque document est nommé et typé par une IRI. En guise d'illustration, voici une exécution de SPARQL-Generate à partir d'un dataset RDF qui contient un graphe par défaut, et de deux documents (identifiés ici par `<position.txt>` et `<mesures.json>`). Cette requête répond à la question : *"quels capteurs sont proches de moi, et qu'indiquent-ils ?"*.

Graphe par défaut (Turtle)

```
<s25> a :TempSensor ;
      geo:lat 38.677220 ;
      geo:long -27.212627 .
<s26> a :TempSensor ;
      geo:lat 37.790498 ;
      geo:long -25.501970 .
<s27> a :TempSensor ;
      geo:lat 37.780768;
      geo:long -25.496294 .
```

Document position.txt

```
37.780496,-25.495157
```

Document mesures.json

```
{ "s25": 14.24,
  "s26": 18.18 }
```

Résultat (Turtle)

```
  <s26> a :NearbySensor ;
        :temp 18.18 .
  <s27> a :NearbySensor .
```

Requête SPARQL-Generate

```
GENERATE {

  ?sensor a :NearbySensor .

  GENERATE {
    ?sensorIRI :temp ?temp .
  }
  ITERATOR sgiter:JSONListKeys(?mesures) AS ?sensorId
  WHERE {
    BIND( IRI( ?sensorId ) AS ?sensorIRI )
    FILTER( ?sensor = ?sensorIRI )
    BIND( CONCAT( "$." , ?sensorId ) AS ?jsonPath )
    BIND( sgfn:JSONPath( ?measures , ?jsonPath  ) AS ?temp
        )
  }
}
SOURCE <position.txt> AS ?pos
SOURCE <mesures.json> AS ?mesures
WHERE {
  BIND( sgfn:SplitAtPosition(?pos,"(.*),(.*)",1) AS ?long )
  BIND( sgfn:SplitAtPosition(?pos,"(.*),(.*)",2) AS ?lat )
  ?sensor a :TempSensor .
  ?sensor geo:lat ?slat .
  ?sensor geo:long ?slong .
  FILTER( ex:distance(?lat, ?long, ?slat, ?slong) < 10 )
}
```

3. RML est une implémentation du standard CSV on the Web (Tandy et al., 2015)
4. `http://rdf4j.org/`
5. Nous utilisons la terminologie des traductions françaises des documents W3C `https://www.w3.org/2003/03/Translations/byLanguage?language=fr`

La syntaxe concrète de SPARQL-Generate étend légèrement celle de SPARQL 1.1, en y ajoutant trois nouvelles clauses. La clause source permet de lier un document à une variable (p.ex. ici, ceux identifiés par `<position.txt>` et `<mesures.json>` à `?pos` et `?mesures`).

La clause iterator permet d'extraire des sous-éléments à l'aide de fonctions d'*itération* et de les lier successivement à une variable (p.ex. ici, la fonction `sgiter:JSONListKeys` est utilisée pour extraire l'ensemble des clés de l'objet JSON lié à `?mesures` et les lier à `?sensorId`).

Enfin, la clause generate remplace la clause construct pour l'étendre et permettre de factoriser la génération de RDF à l'aide de requêtes imbriquées. Différents formats de données peuvent être supportés à l'aide de l'ensemble extensible de fonctions de *liaison* et d'*itération*.

4.1 Syntaxe concrète

Pour faciliter l'apprentissage de SPARQL-Generate par les ingénieurs de la connaissance, nous n'étendons que légèrement la EBNF de SPARQL 1.1 (Harris et Seaborne, 2013, §19.8) :

```
[174] GenerateUnit ::= Generate
[175] Generate ::= Prologue GenerateQuery
[176] GenerateQuery ::= 'GENERATE' GenerateTemplate DatasetClause* IteratorOrSourceClause
    * WhereClause? SolutionModifier
[177] GenerateTemplate ::= '{' GenerateTemplateSub '}'
[178] GenerateTemplateSub ::= ConstructTriples? ( SubGenerateQuery ConstructTriples? )*
[179] IteratorOrSourceClause ::= IteratorClause | SourceClause
[180] IteratorClause ::= 'ITERATOR' FunctionCall 'AS' Var
[181] SourceClause ::= 'SOURCE' FunctionCall ( 'ACCEPT' VarOrIri )? 'AS' Var
[182] SubGenerateQuery ::= 'GENERATE' ( SourceSelector | GenerateTemplate ) (
    IteratorOrSourceClause* WhereClause? SolutionModifier '.' )?
```

Alors que la production des requête SPARQL Query et SPARQL Update commencent à `QueryUnit` et `UpdateUnit`, la production d'une requête SPARQL-Generate commence à la règle `GenerateUnit`. Cette syntaxe concrète contient deux caractéristiques notoires.

La première caractéristique notoire réside dans la règle `[181]`. La partie optionnelle `('ACCEPT' VarOrIri)` permet de spécifier l'IRI du type du document à lier dans la clause source. Dans le cas où l'implémentation irait récupérer le document ainsi nommé u et typé t sur le Web, l'IRI t désigne la manière dont le contenu de la ressource identifiée par u devrait être négocié. Cette IRI peut décrire différentes sortes de négociation, en rapport avec le *media type*, la langue ou encore l'encodage. Le type du document obtenu peut être différent de ce qui a été demandé.

La seconde réside dans la règle `[182]`, et permet, en pratique, de modulariser la requête. Une sous-requête SPARQL-Generate (i.e., une requête dans la partie generate d'une requête parente) peut contenir un gabarit generate (*generate template*), incluant un motif de graphe et potentiellement d'autres sous-requêtes. Il peut également faire référence à un SourceSelector, i.e., une IRI. Les implémentations sont libres de choisir comment cette IRI doit être déréférencée pour obtenir une nouvelle requête SPARQL-Generate. Ceci n'a pas besoin d'être représenté dans la syntaxe abstraite de SPARQL-Generate, mais permet par exemple : (1) de modulariser de grosses requêtes pour les rendre plus lisibles, ou (2) de permettre à une requête d'"appeler" une requête existante qui serait déréférencée sur le Web. Charge à l'implémentation de faire attention aux boucles d'appels de requêtes.

4.2 Syntaxe abstraite

Nous utilisons les notations $\mathbf{I}$, $\mathbf{B}$, $\mathbf{L}$, et $\mathbf{V}$ pour les ensembles d'*IRI*, *nœud anonyme* (*blank node*), *littéraux*, et *variables*, disjoints deux à deux. L'ensemble des *termes RDF* est $\mathbf{T}$ =

$\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. L'ensemble des *motifs de triplets* est défini par $\mathbf{T} \cup \mathbf{V} \times \mathbf{I} \cup \mathbf{V} \times \mathbf{T} \cup \mathbf{V}$, et un *motif de graphe* est un ensemble fini de motifs de triplets. L'ensemble des motifs de graphes est noté $\mathcal{P}$. Nous notons $\mathbf{F_0}$ l'ensemble des noms de fonctions SPARQL 1.1 [6], disjoint de $\mathbf{T}$. L'ensemble des patrons de requête SPARQL 1.1 est noté $\mathcal{Q}$. Enfin, pour tout ensemble X, nous notons $X^* = \bigcup_{n \geqslant 0} X^n$ l'ensemble des listes de X.

L'ensemble des *expressions de fonctions* est noté $\mathcal{E}$, et est le plus petit ensemble tel que : (i) $\mathbf{T} \cup \mathbf{V} \subseteq \mathcal{E}$ (p. ex., `<position.txt>`), (ii) $(\mathbf{F_0} \cup \mathbf{I}) \times \mathbf{T}^* \subseteq \mathcal{E}$ (p. ex., `CONCAT("$.", ?sensorId)`, `sgiter:JSONListKeys(?mesures)`), et (iii) les imbriquées : $\forall E \subseteq \mathcal{E}, (\mathbf{F_0} \cup \mathbf{I}) \times E^* \subseteq \mathcal{E}$.

L'abstraction de la règle de grammaire `[181]` est l'ensemble des *clauses* **source**, et permettra de sélectionner un document dans l'ensemble de documents pour le lier à une variable. Par exemple dans la requête ci-dessus la variable `?pos` est liée au document identifié par `<position.txt>`. Nous introduisons un élément spécial $\omega \notin \mathbf{T} \cup \mathbf{V}$, qui représente *null*, et notons $\hat{X} = X \cup \{\omega\}$ l'ensemble *généralisé* d'un ensemble X.

Définition 1 (Clause source)**.** *L'ensemble $\mathcal{S}$ des clauses* **source** *est défini par l'équation $\mathcal{S} = \mathcal{E} \times (\hat{\mathbf{I}} \cup \mathbf{V}) \times \mathbf{V}$. On note $v \xleftarrow{source} \langle e, a \rangle \in \mathcal{S}$ une clause* **source** *particulière, avec $v \in \mathbf{V}$, $e \in \mathcal{E}$, $a \in \hat{\mathbf{I}} \cup \mathbf{V}$.*

Dans la plupart des cas d'utilisation pour SPARQL-Generate, une variable doit itérer sur plusieurs parties d'un même document. Par exemple dans la requête d'illustration, la variable `?sensorId` est liée successivement aux clés de l'objet JSON `?mesures`. [7] Dans SPARQL 1.1, la seule manière d'extraire un terme d'un littéral serait d'utiliser une clause bind impliquant une fonction de liaison. Cependant, ces fonctions sortent au plus un terme, et ne peuvent donc pas être utilisées pour générer plus de liaisons (*bindings*). En conséquence, nous introduisons une seconde extension, les *clauses* **iterator**, qui sortent un *ensemble de termes*, et remplacent la liaison courante par autant de liaisons qu'il y a d'éléments dans cet ensemble.

Définition 2 (Clause iterator)**.** *L'ensemble $\mathcal{I}$ des clauses* **iterator** *est défini par l'équation $\mathcal{I} = \mathbf{I} \times \mathcal{E}^* \times \mathbf{V}$. On note $v \xleftarrow{iterator} \langle u, e_0, \ldots, e_k \rangle \in \mathcal{I}$ une clause* **iterator** *particulière, avec $v \in \mathbf{V}$, $u \in \mathbf{I}$, $e_0, \ldots, e_k \in \mathcal{E}$, et $k \in \mathbb{N}$.*

On peut alors étendre les patrons de requête SPARQL 1.1 $\mathcal{Q}$ avec une liste de clauses source et iterator, en nombre et ordre quelconque. Nous faisons exprès de ne pas changer la définition de $\mathcal{Q}$ pour faciliter la réutilisation des implémentations SPARQL existantes.

Définition 3 (Patron de requête SPARQL-Generate)**.** *L'ensemble des patrons de requête SPARQL-Generate est une séquence de clauses* **source** *et* **iterator***, suivie d'un patron de requête SPARQL 1.1 : $\mathcal{Q}^+ = (\mathcal{S} \cup \mathcal{I})^* \times \mathcal{Q}$.*

Finalement, l'ensemble des requêtes SPARQL-Generate étend $\mathcal{Q}^+$ avec un gabarit de graphe, et potentiellement d'autres requêtes SPARQL-Generate.

Définition 4 (Requête SPARQL-Generate)**.** *L'ensemble des requêtes SPARQL-Generate est noté $\mathcal{G}$, et défini comme le plus petit ensemble tel que : (i) $\mathcal{P} \times \mathcal{Q}^+ \subseteq \mathcal{G}$, et (ii) $\forall G \subseteq \mathcal{G}, \mathcal{P} \times G^* \times \mathcal{Q}^+ \subseteq \mathcal{G}$.*

Les requêtes SPARQL-Generate définies par (ii) sont des requêtes imbriquées, qui permettent de factoriser la génération de RDF.

6. SPARQL 1.1 définit des fonctions intégrées nommées `IF`, `IRI`, `CONCAT`, etc.

7. Autre exemple : les résultats d'une évaluation XPath sur un document XML, voir test unitaire *rmlproeg1* - `http://w3id.org/sparql-generate/tests-reports.html`

4.3 Sémantique de SPARQL-Generate

Une requête SPARQL-Generate est exécutée sur un modèle de données qui étend celui de SPARQL : l'ensemble de données RDF. Un *ensemble de données RDF* est une paire $\langle D, N\rangle$ tel que D est un graphe RDF, appelé le *graphe par défaut*, et N est un ensemble fini de paires $\langle u, G\rangle$ où u est une IRI et G est un graphe RDF, de sorte qu'aucune paire ne contient la même IRI. Afin de requêter des documents aux formats arbitraires, nous introduisons la notion d'*ensemble de documents (*documentset*)* par analogie aux ensembles de données RDF.

Définition 5 (Ensemble de documents)**.** *Un ensemble de documents est un ensemble fini de triplets $\Delta \subseteq \mathbf{I} \times \hat{\mathbf{I}} \times \mathbf{L}$. Un élément de Δ est un triplet $\langle u, a, \langle d, t\rangle\rangle$ où : u est le nom du document ; a est le type demandé pour le document ; le littéral $\langle d, t\rangle$ modélise le document ; et t (l'IRI du type de donnée du littéral) est le type du document. Δ doit être tel qu'aucune paire de triplets distincts n'a les mêmes deux premiers éléments.*

Pour alléger les formules, on note également $\Delta : \hat{\mathbf{T}} \times \hat{\mathbf{T}} \to \hat{\mathbf{L}}$ l'application qui associe à un couple $\langle u, a\rangle$ un littéral l si et seulement si $\langle u, a, l\rangle \in \Delta$, et ω sinon. Un ensemble de documents peut donc être stocké en interne, ou représenter le Web : u représente là où un appel (ex. HTTP GET) doit être fait, a décrit comment le contenu doit être négocié, d est le contenu de la représentation obtenue en cas de succès, et t décrit le type de la représentation (son media type, sa langue, son encodage, etc.).

Nous réutilisons certains concepts de la sémantique de SPARQL 1.1, mais avec des notations plus concises. L'ensemble des *liaisons* est noté $\mathcal{M}$, et est défini par l'équation (1). Contrairement au standard SPARQL 1.1, nous utilisons une fonction totale sur l'ensemble des termes et des variables, et utilisons l'élément ω pour représenter l'image d'une *variable non liée*. Comme pour SPARQL, le *domaine* d'une liaison est l'ensemble des variables qui sont liées à un terme (voir Eq. (2)).

$$\mu : \mathbf{T} \cup \mathbf{V} \to \hat{\mathbf{T}} \text{ t.q., } \forall t \in \mathbf{T}, \mu(t) = t \tag{1}$$

$$\forall \mu \in \mathcal{M}, \mathsf{dom}(\mu) = \{v \in \mathbf{V} | \mu(v) \in \mathbf{T}\} \tag{2}$$

Nous introduisons un ensemble particulier de liaisons appelées *liaisons de substitution*, dont le domaine est un singleton. i.e., $\forall v \in \mathbf{V}$ et $t \in \hat{\mathbf{T}}$, $[v/t]$ est une liaison de substitution avec :

$$\forall t' \in \mathbf{T}, [v/t](t') = t', \quad [v/t](v) = t, \quad \text{et } \forall x \in \mathbf{V}, x \neq v, [v/t](x) = \omega \tag{3}$$

L'opérateur de *composition à gauche* $\fatsemi$ est défini tel que dans $\mu_1 \fatsemi \mu_2$, la priorité de liaison revient à μ_1. En d'autres termes, toute variable v liée à la fois dans μ_1 et μ_2 est finalement liée à $\mu_1(v)$ dans $\mu_1 \fatsemi \mu_2$.

$$\mu_1 \fatsemi \mu_2 : \begin{cases} x \mapsto \mu_1(x) & \text{si } x \in \mathsf{dom}(\mu_1) \\ x \mapsto \mu_2(x) & \text{si } x \in \mathsf{dom}(\mu_2) \backslash \mathsf{dom}(\mu_1) \\ x \mapsto \omega & \text{dans les autres cas} \end{cases} \tag{4}$$

Chaque moteur SPARQL 1.1 reconnaît un ensemble F_b d'IRI de fonctions de liaison SPARQL 1.1 (p.ex. ici, au moins `sgfn:JSONPath` et `ex:distance`). Une fonction de liaison associe à une expression de fonction son évaluation, i.e., un terme RDF. Formellement, pour un

moteur SPARQL 1.1 donné, l'équation (5) définit l'*application des fonctions de liaison* f_b, qui associe à une IRI de fonction de liaison reconnue sa fonction de liaison. L'*application des fonctions d'itération* est définie de manière analogue pour un moteur SPARQL-Generate (p.ex. ici, il reconnait au moins `sgiter:JSONListKeys`), sauf que l'évaluation d'une expression de fonction est ici un ensemble de termes RDF. Etant donné un ensemble F_i d'IRI de fonctions d'itération reconnues, l'équation (6) définit *l'application des fonctions d'itération* f_i.

$$f_b : F_b \to \left(\hat{\mathbf{T}}^* \to \hat{\mathbf{T}}\right) \qquad (5) \qquad\qquad f_i : F_i \to \left(\hat{\mathbf{T}}^* \to 2^{\hat{\mathbf{T}}}\right) \qquad (6)$$

Nous généralisons la définition des liaisons pour que leur domaine inclue l'ensemble des fonctions d'expression. L'ensemble des *liaisons généralisées* est noté $\bar{\mathcal{M}}$. Il contient la *généralisation* $\bar{\mu}$ des liaisons $\mu \in \mathcal{M}$, où $\bar{\mu} : \mathbf{T} \cup \mathbf{V} \cup \mathcal{E} \to \hat{\mathbf{T}}$ est défini récursivement par :

$$\forall t \in \mathbf{T} \cup \mathbf{V}, \bar{\mu}(t) = \mu(t) \qquad (7)$$

$$\forall \langle u, e_1, \ldots, e_n \rangle \in \mathcal{E} \text{ t.q. } u \in F_b,\ \bar{\mu}(\langle u, e_1, \ldots, e_n \rangle) = f_b(u)(\bar{\mu}(e_1), \ldots, \bar{\mu}(e_n)) \qquad (8)$$

Une clause source $v \xleftarrow{\text{source}} \langle e, a \rangle \in \mathcal{S}$ permet de modifier la liaison μ pour lier la variable v à un document de Δ (p.ex. `?pos` à `"37.780496,-25.495157"`). Une clause iterator $v \xleftarrow{\text{iterator}} \langle t, e_0, \ldots, e_k \rangle \in \mathcal{I}$ sert typiquement à extraire les parties importantes d'un document : elle permet, à partir d'une liaison μ, de générer plusieurs autres liaisons où la variable v est liée aux éléments de l'évaluation de $f_i(t)$ sur $e_0, \ldots, e_k$ (p.ex. ici, `?sensorId` sera liée successivement à `"s25"` puis à `"s26"`). Ces clauses peuvent être combinées en liste quelconques.

Soit $\Sigma \in (\mathcal{S} \cup \mathcal{I})^n$, et $n \geqslant 1$. Définissons par induction l'évaluation d'une telle liste $[\![\Sigma]\!]^\mu_\Delta$:

$$[\![v \xleftarrow{\text{source}} \langle e, a \rangle]\!]^\mu_\Delta = [v/\Delta(\bar{\mu}(e), a)] \fatsemi \mu \qquad (9)$$

$$[\![v \xleftarrow{\text{iterator}} \langle t, e_0, \ldots, e_k \rangle]\!]^\mu_\Delta = \left\{ [v/t'] \fatsemi \mu | t' \in f_i(t)(\bar{\mu}(e_0), \ldots, \bar{\mu}(e_k)) \right\} \qquad (10)$$

$$[\![\langle \Sigma, v \xleftarrow{\text{source}} e \rangle]\!]^\mu_\Delta = \left\{ [\![v \xleftarrow{\text{source}} e]\!]^{\mu'}_\Delta | \mu' \in [\![\Sigma]\!]^\mu_\Delta \right\} \qquad (11)$$

$$[\![\langle \Sigma, v \xleftarrow{\text{iterator}} e \rangle]\!]^\mu_\Delta = \bigcup_{\mu' \in [\![\Sigma]\!]^\mu_\Delta} [\![v \xleftarrow{\text{iterator}} e]\!]^{\mu'}_\Delta \qquad (12)$$

Soit $Q \in \mathcal{Q}$ un patron de requête SPARQL 1.1, D un ensemble de données RDF, et $[\![Q]\!]^\mu_D$ l'ensemble des liaisons solutions de Q pour une liaison μ, définie par SPARQL 1.1. Soit également Σ une liste de clauses source et iterator. L'évaluation du patron de requête SPARQL-Generate $Q^+ = \langle \Sigma, Q \rangle \in (\mathcal{S} \cup \mathcal{I})^* \times \mathcal{Q}$ sur D et l'ensemble de documents Δ est défini par l'équation (13). Si l'on définit maintenant une liaison initiale $\mu_0 : v \mapsto \omega, \forall v \in \mathbf{V}$, alors l'ensemble des liaisons solution de la requête Q^+ sur D et Δ est définie par l'équation (14).

$$[\![Q^+]\!]^\mu_{\Delta,D} = \bigcup_{\mu' \in [\![\Sigma]\!]^\mu_\Delta} [\![Q]\!]^{\mu'}_D \qquad (13) \qquad\qquad [\![Q^+]\!]_{\Delta,D} = [\![Q^+]\!]^{\mu_0}_{\Delta,D} \qquad (14)$$

Pour tout gabarit de graphe $P \in \mathcal{P}$ et toute liaison $\mu \in \mathcal{M}$, nous notons $\mathfrak{G}^\mu(P)$ le graphe RDF généré par l'instanciation de P par rapport à μ selon Harris et Seaborne (2013, §16.2.1). On peut finalement définir le résultat de l'évaluation d'une requête SPARQL-Generate récursivement. Soient une requête simple $\langle P, Q \rangle \in \mathcal{P} \times \mathcal{Q}^+$, une requête quelconque $G =$

$\langle P, G_0, \ldots, G_j, Q\rangle \in \mathcal{P} \times \mathcal{G}^* \times \mathcal{Q}^+$, et une liaison μ. Les trois équations suivantes définissent le graphe RDF généré par G.

$$\mathfrak{G}^{\mu}_{\Delta,D}(\langle P, Q\rangle) = \bigcup_{\mu' \in [\![Q]\!]^{\mu}_{\Delta,D}} \mathfrak{G}^{\mu'}(P) \tag{15}$$

$$\mathfrak{G}^{\mu}_{\Delta,D}(\langle P, G_0, \ldots, G_j, Q\rangle) = \bigcup_{\mu' \in [\![Q]\!]^{\mu}_{\Delta,D}} \left(\mathfrak{G}^{\mu'}(P) \cup \bigcup_{0 \leqslant i \leqslant j} \mathfrak{G}^{\mu'}_{\Delta,D}(G_i)\right) \tag{16}$$

$$\mathfrak{G}_{\Delta,D}(G) = \mathfrak{G}^{\mu_0}_{\Delta,D}(G) \tag{17}$$

5 Implémentation et évaluation de SPARQL-Generate

5.1 Approche générique

Il serait avantageux de pouvoir implémenter SPARQL-Generate sur n'importe quel moteur SPARQL existant. En effet, un tel moteur fournit déjà : (i) l'application des fonctions de liaison f_b (on peut donc connaître pour toute liaison $\mu \in \mathcal{M}$ sa version généralisée $\bar{\mu}$) ; (ii) une fonction SELECT qui prend un patron de requête SPARQL 1.1 en entrée, et renvoie un ensemble de liaisons solution ; (iii) une fonction INSTANTIATE qui prend un gabarit de graphe $P \in \mathcal{P}$ et une liaison $\mu \in \mathcal{M}$ en entrée, et renvoie le graphe RDF correspondant à l'instanciation de P par rapport à μ ; (iv) la gestion des ensembles de données RDF D. L'implémentation de SPARQL-Generate pourrait se contenter de fournir : (i) l'ensemble de documents Δ, et (ii) l'application des fonctions d'itération f_i.

Soit $\mathcal{V} = 2^{\mathcal{M}}$ l'ensemble des blocs de données en ligne (*inline data block*). Soit $\langle V, Q\rangle \in \mathcal{Q}$ le résultat de préfixer le patron de requête Q par un bloc de données en ligne $V \in \mathcal{V}$. Le théorème 1 ci-dessous permet de proposer l'algorithme naïf 1 qui peut être utilisé pour implémenter SPARQL-Generate au dessus de n'importe quel moteur SPARQL 1.1.

Théorème 1. *Soit une requête SPARQL 1.1 $Q \in \mathcal{Q}$, et une liste de clauses* source *et* iterator *$\Sigma \in (\mathcal{S} \cup \mathcal{I})^*$. L'évaluation du patron de requête SPARQL-Generate $\langle \Sigma, Q\rangle \in \mathcal{Q}^+$ est équivalente à l'évaluation de $\langle [\![\Sigma]\!]_{\Delta,D}, Q\rangle$, où $[\![\Sigma]\!]_{\Delta,D}$ est le résultat de l'évaluation de Σ.*

Démonstration. Le résultat est direct à partir des équations 13, 14, et de la traduction dans nos notations de la phrase `https://www.w3.org/TR/sparql11-query/#data-block` de la sémantique de SPARQL 1.1 : $[\![\langle V, Q\rangle]\!] = \bigcup_{\mu \in V} [\![Q]\!]^{\mu}$. □

5.2 Implémentation sur Jena

Une première implémentation de SPARQL-Generate a été développée au dessus du moteur SPARQL de Apache Jena[8], décrite plus précisément par Lefrançois et al. (2016), documentée en ligne[9], et rendue disponible en code open source, archive exécutable, service web, et formulaire en ligne. Cette implémentation intègre pour l'instant des fonctions de liaison et d'itération pour générer du RDF à partir des formats de données suivants : XML, CSV, TSV, HTML, JSON, CBOR, et texte brut.

8. `http://jena.apache.org/`
9. `https://w3id.org/sparql-generate`

Algorithm 1 Implémentation naïve de SPARQL-Generate sur un moteur SPARQL 1.1. [10]

procedure GENERATE($\langle P, G_0, \ldots, G_j, \langle E_0, \ldots, E_n \rangle, Q \rangle, \mu$) ▷ Voir Def. 4
  $M \leftarrow \{\mu\}$ ▷ M est un singleton qui contient une liaison
  for $0 \leqslant i \leqslant n$ **do**
    if $E_i = v \xleftarrow{\text{source}} e$ **then** ▷ Voir Def. 1
      for all $\mu \in M$ **do**
        $\mu(v) \leftarrow \Delta(\bar{\mu}(e))$ ▷ Voir Def. 5 et Eq. 7
      end for
    else if $E_i = v \xleftarrow{\text{iterator}} \langle t, e_0, \ldots, e_k \rangle$ **then** ▷ Voir Def. 2
      $M' \leftarrow \varnothing$
      for all $\mu \in M$ **do**
        for all $t' \in f_i(t)(\bar{\mu}(e_0), \ldots, \bar{\mu}(e_k))$ **do** ▷ Voir Eq. 8
          $\mu' \leftarrow \mu$; $\mu'(v) \leftarrow t'$; and $M' \leftarrow M' \cup \{\mu'\}$
        end for
      end for
      $M \leftarrow M'$ ▷ remplacer M par M'
    end if
  end for
  $M \leftarrow$ SELECT($\langle M, Q \rangle$) ▷ le patron de requête préfixé par le bloc de données en ligne
  $G \leftarrow \varnothing$ ▷ le graphe RDF vide
  for $\mu \in M$ **do**
    $G \leftarrow G \cup$ INSTANTIATE(P, μ) ▷ opérer une union de graphe RDF (pas fusion : ne pas fusionner les nœuds anonymes même si ils ont le même nom)
    for $0 \leqslant i \leqslant j$ **do**
      $G \leftarrow G \cup$ GENERATE(G_i, μ)
    end for
  end for
  return G
end procedure

5.3 Evaluation préliminaire

L'évaluation de SPARQL-Generate n'est pas la contribution principale de cet article. Nous pouvons néanmoins rapporter un certain nombre de comparaisons qualitatives, qui sont autant de pistes pour développer une étude comparative approfondie entre SPARQL-Generate et RML en terme de complexité cognitive, d'expressivité, de performance, ou encore de passage à l'échelle. Tout d'abord, l'implémentation de SPARQL-Generate propose une suite de test unitaires. En particulier, chacun des exemples rencontrés dans les spécifications de RML a été transposé en une requête SPARQL-Generate, de même que chacun des tests unitaires de l'implémentation de référence de RML. Ceci suggère que l'expressivité de SPARQL-Generate est suffisante pour la plupart des cas envisagés par RML. Les requêtes semblent d'ailleurs être plus concises que leur équivalent en RML. Enfin, SPARQL-Generate bénéficie de l'expressivité de

10. Cet algorithme est simplifié et ne montre pas les subtilités liées à la gestion des nœuds anonymes, qui feront l'objet d'un autre article.

SPARQL. Contrairement à RML, il est donc possible de filtrer, d'agréger les résultats, ou bien de modifier les séquences de solutions. Enfin, SPARQL-Generate bénéficie des fonctions intégrées à SPARQL, ainsi que du mécanisme d'extension standard des fonctions de liaison. Un équivalent de ce mécanisme d'extension pour les fonctions d'itération est par ailleurs proposé par l'implémentation de SPARQL-Generate, rendant possible aux tiers de développer et d'utiliser des fonctions d'itération personnalisées pour n'importe quel format.

6 Conclusion

Le problème de l'exploitation des données de sources et formats hétérogènes est commun sur le Web, et certaines solutions existent pour intégrer des données dans le modèle de données RDF. Dans cet article, nous avons défini un nouveau langage, SPARQL-Generate, qui permet de décrire la génération de RDF à partir d'un ensemble de données RDF et d'un ensemble de documents aux formats hétérogènes. SPARQL-Generate étend SPARQL 1.1 et bénéficie donc de son expressivité et de son extensibilité, et peut être maîtrisé rapidement pas les ingénieurs de la connaissance. Nous avons défini les syntaxes concrètes et abstraites, ainsi que la sémantique de ce nouveau langage. Nous avons alors démontré qu'il peut être implémenté au dessus de n'importe quel moteur SPARQL 1.1 existant. Une première implémentation sur Apache Jena démontre que les cas d'utilisation des approches concurrentes sont couverts, et permet déjà de générer du RDF à partir des formats de données suivants : XML, CSV, TSV, HTML, JSON, CBOR, et texte brut. Les travaux futurs incluent notamment un étude comparative approfondie entre SPARQL-Generate et RML en terme de complexité cognitive, d'expressivité, de performance, ou encore de passage à l'échelle.

Remerciements

Ce travail a été partiellement financé par le projet ITEA2 12004 Smart Energy Aware Systems (SEAS), le projet ANR 14-CE24-0029 OpenSensingCity, et une convention bilatérale de recherche avec ENGIE R&D.

Références

Arenas, M., A. Bertails, E. Prud'hommeaux, et J. Sequeda (2012). A Direct Mapping of Relational Data to RDF, W3C Recommendation 27 September 2012. W3C Recommendation, World Wide Web Consortium (W3C).

Connolly, D. (2007). Gleaning Resource Descriptions from Dialects of Languages (GRDDL), W3C Recommendation 11 September 2007. W3C Recommendation, World Wide Web Consortium (W3C).

Das, S., S. Sundara, et R. Cyganiak (2012). R2RML : RDB to RDF Mapping Language, W3C Recommendation 27 September 2012. W3C Recommendation, World Wide Web Consortium (W3C).

Dell'Aglio, D., A. Polleres, N. Lopes, et S. Bischof (2014). Querying the web of data with XSPARQL 1.1. In R. Verborgh et E. Mannens (Eds.), *Proceedings of the ISWC Developers*

Workshop 2014, co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 19, 2014., Volume 1268 of *CEUR Workshop Proceedings*. Sun SITE Central Europe (CEUR).

Dimou, A., M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens, et R. V. de Walle (2014). RML : A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In C. Bizer, T. Heath, S. Auer, et T. Berners-Lee (Eds.), *Proceedings of the Workshop on Linked Data on the Web, co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea, April 8, 2014*, Volume 1184 of *CEUR Workshop Proceedings*. Sun SITE Central Europe (CEUR).

Harris, S. et A. Seaborne (2013). SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013. W3C Recommendation, World Wide Web Consortium (W3C).

Hert, M., G. Reif, et H. C. Gall (2011). A comparison of RDB-to-RDF mapping languages. In C. Ghidini, A.-C. N. Ngomo, S. N. Lindstaedt, et T. Pellegrini (Eds.), *Proceedings the 7th International Conference on Semantic Systems, I-SEMANTICS 2011, Graz, Austria, September 7-9, 2011*, ACM International Conference Proceeding Series, pp. 25–32. ACM Press.

Lefrançois, M., A. Zimmermann, et N. Bakerally (2016). Flexible RDF generation from RDF and heterogeneous data sources with SPARQL-Generate. In *Proc. of the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW'16)*.

Lopes, N., S. Bischof, et A. Polleres (2011). On the semantics of heterogeneous querying of relational, XML, and RDF data with XSPARQL. In *Proceedings of the 15th Portuguese Conference on Artificial Intelligence (EPIA2011) - Computational Logic with Applications Track, Lisbon, Portugal, October 2011*.

Polleres, A., T. Krennwallner, N. Lopes, J. Kopecký, et S. Decker (2009). XSPARQL Language Specification, W3C Member Submission 20 January 2009. W3C Member Submission, World Wide Web Consortium (W3C).

Tandy, J., I. Herman, et G. Kellogg (2015). Generating RDF from Tabular Data on the Web, W3C Recommendation 17 December 2015. W3C Recommendation, World Wide Web Consortium (W3C).

Summary

Unlike what is promoted by the Web of Data initiative, data published by most organizations are in non-RDF formats such as CSV, JSON, or XML. Furthermore in the Web of Things, constrained objects prefer binary formats such as EXI or CBOR over textual RDF formats. In this context, RDF can still be used as a *lingua franca* to enable semantic interoperability, integration of data with heterogeneous formats, reasoning, and querying. Several tools and formalisms have been designed to transform non-RDF documents to RDF. The most flexible ones are based on transformation or mapping languages (GRDDL, XSPARQL, R2RML, RML, CSVW, etc.). This paper defines a new such language, SPARQL-Generate, designed as an extension of SPARQL 1.1 to generate RDF from a RDF dataset and a set of documents with arbitrary formats. We show it can be implemented on top of any existing SPARQL 1.1 engine, mention a first implementation on top of Apache Jena, and show that it leverages the expressivity and the extensibility of SPARQL 1.1 to open the set of supported formats.