A SPARQL Extension for Generating RDF from Heterogeneous Formats

Maxime Lefrançois(✉), Antoine Zimmermann, and Noorani Bakerally

Univ Lyon, MINES Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516, 42023 Saint-Étienne, France
{maxime.lefrancois,antoine.zimmermann,noorani.bakerally}@emse.fr

Abstract. RDF aims at being the universal abstract data model for structured data on the Web. While there is effort to convert data in RDF, the vast majority of data available on the Web does not conform to RDF. Indeed, exposing data in RDF, either natively or through wrappers, can be very costly. Furthermore, in the emerging Web of Things, resource constraints of devices prevent from processing RDF graphs. Hence one cannot expect that all the data on the Web be available as RDF anytime soon. Several tools can generate RDF from non-RDF data, and transformation or mapping languages have been designed to offer more flexible solutions (GRDDL, XSPARQL, R2RML, RML, CSVW, etc.). In this paper, we introduce a new language, SPARQL-Generate, that generates RDF from: (i) a RDF Dataset, and (ii) a set of documents in arbitrary formats. As SPARQL-Generate is designed as an extension of SPARQL 1.1, it can provably: (i) be implemented on top on any existing SPARQL engine, and (ii) leverage the SPARQL extension mechanism to deal with an open set of formats. Furthermore, we show evidence that (iii) it can be easily learned by knowledge engineers that know SPARQL 1.1, and (iv) our first naive open source implementation performs better than the reference implementation of RML for big transformations.

1 Introduction

We aim at lowering the overhead for web services and constrained things to embrace the Semantic Web formalisms and tool. A usual key step is to generate RDF from documents having various formats (or *triplify*). Indeed, companies and web services store and exchange documents in a multitude of data models and formats: the relational data model and XML (not RDF/XML) are still very present, data portals heavily rely on CSV, and web APIs on JSON. Furthermore, constrained things on the Web of things may be only able to support binary formats such as EXI or CBOR. Although effort has been made to define RDF data formats that are also compatible with the formats in use (e.g., RDF/XML is compatible with XML, JSON-LD is compatible with JSON, any EXI version

This paper has been partly financed by the ITEA2 12004 SEAS (Smart Energy Aware Systems) project, the ANR 14-CE24-0029 OpenSensingCity project, and a bilateral research convention with ENGIE R&D.

E. Blomqvist et al. (Eds.): ESWC 2017, Part I, LNCS 10249, pp. 35–50, 2017.
DOI: 10.1007/978-3-319-58068-5_3

of RDF/XML is compatible with EXI, etc.), it is unlikely that these formats will completely replace existing data formats one day. However, the RDF data *model* may still be used as a *lingua franca* to reach semantic interoperability and integration and querying of data having heterogeneous formats.

Several pieces of research and development focused on generating RDF from other models and formats, and sometimes led to the definition of standards. However, in the context of projects we participate in, we identified use cases and requirements that existing approaches satisfy only partially. These are reported in Sect. 2 and include:

- the solution must be expressive, flexible, and extensible to new data formats;
- the solution must generate RDF from several data sources with heterogeneous formats, potentially in combination with a RDF dataset;
- the solution should be easy to learn and to integrate in a typical semantic web engineering workflow, so that knowledge engineers can learn it easily to start prototyping triplifications.

Section 3 describes existing solutions and identify their limitations. In order to satisfy these requirements, we introduce SPARQL-Generate, an extension of SPARQL 1.1 that answers the aforementioned requirements and combines the following advantages: (1) it leverages SPARQL's expressivity and flexibility, including the standard extension mechanism for binding functions; (2) it may be implemented on top of any existing SPARQL engine.

The rest of this paper is organized as follows. Section 4 formally specifies the abstract syntax and semantics of the SPARQL-Generate language. These definitions enable to prove in Sect. 5.1 that it can be implemented on top of any existing SPARQL 1.1 engine, and propose a naive algorithm for this. Section 5.2 briefly describes a first open-source implementation on top of Apache ARQ, which has been tested on use cases from the related work and more. Finally, Sect. 5.3 proposes a comparative evaluation between SPARQL-Generate and RML on two aspects: performance of the reference implementations, and cognitive complexity of the query/mapping.

2 Use-Cases and Requirements

We identified two important use cases for generating RDF from heterogeneous data formats. They are originating from projects in which the stakeholders require strong interoperability in consuming and exchanging data, although data providers cannot afford the cost to move towards semantic data models.

Open Data. In the context of open data, organizations can rarely afford the cost of cleaning and reengineering their datasets towards more interoperable linked data. They sometimes also lack the expertise to do so. Therefore, data is published on a best effort basis in the formats that require least labour and resources. Yet, data consumers expect more uniform, self describing data sets that can be easily cross-related. In the case when a knowledge model has been agreed upon,

it is important for the users to be able to prototype transformations to RDF from one or more of these data sources, potentially in different formats. In addition, the solution should be flexible enough to allow for fine-grained control on the generated RDF and the links between data sets, and should be able to involve contextual RDF data. The list of formats from which RDF may be generated must be easily extended. Finally, the solution must be easily used by knowledge engineers that know RDF and SPARQL.

Web of Things. In the emerging Web of Things, constrained devices must exchange lightweight messages due to their inherent bandwidth, memory, storage, and/or battery constraints. Yet, RDF formats have to encode a lot of textual information such as IRIs and literals with datatype IRIs. Although some research is led to design lightweight formats for RDF (such as a CBOR version of JSON-LD), it is likely that companies and device vendors will continue to use and introduce new binary formats that are optimized for their usage.

From these use cases, we identify the following requirements:

R1: transform several sources having heterogeneous formats;
R2: contextualize the transformation with an RDF Dataset;
R3: be extensible to new data formats;
R4: be easy to use by Semantic Web experts;
R5: integrate in a typical semantic web engineering workflow;
R6: be flexible and easily maintainable;
R7: transform binary formats as well as textual formats.

With these requirements in mind, the next section overviews existing solutions.

3 Related Work

Data publisher and consumer can implement *ad-hoc* transformation mechanisms to generate RDF from data with heterogeneous models and formats. Although this approach certainly leads to the most efficient solutions, it is also costly to develop and maintain, and inflexible. Several pieces of work aimed at simplifying this task.

Many *converters to RDF* have been listed by the W3C Semantic Web Education and Outreach interest group (SWEO): https://www.w3.org/wiki/ConverterToRdf. Most of them target a specific format or specific metadata, such as ID3tag, BibTeX, EXIT, etc. Some like Apache Any23, datalift, or Virtuoso Sponger are designed to convert multiple data formats to RDF. Direct Mapping [1] describes a default transformation for relational data. These solutions are very ad hoc, implementation specific and barely allow the control of how RDF is generated. They do not provide a formal language that would allow to explicit and customize the conversion to RDF. As a result, the output RDF is often more geared towards describing the structure of the data rather than the data itself. It is still possible to compose these solutions with SPARQL `construct`

rules that transform the generated RDF to the required RDF, but this requires to get familiar with the vocabulary used in the output of each of these tools. They hence do not satisfy most of the requirements listed in Sect. 2.

Other approaches propose to use a transformation or mapping language to tune the RDF generation. However, most of these solutions target only one or a few specific data models (e.g., the relational model) or formats (e.g., JSON). For instance GRDDL encourages the use of XSLT and targets XML inputs [2]. XSPARQL is based on XQuery and originally targeted XML [11], as well as the inverse transformation from RDF to XML, before being extended to the relational data model [10], then to JSON [4]. GRDDL and XSPARQL rely respectively on XSLT and XQuery, that have been proven to be Turing-complete. These languages are hence full-fledged procedural programming language with explicit algorithmic constructs to produce RDF.

Other formalisms have been designed to generate RDF from the relational data [7]. From these pieces of work originated R2RML [3], which proposes a RDF vocabulary to describe mappings to RDF. Finally, CSVW [12] also adopts this approach but targets the CSV data format.

One approach that stands out is RML [5], that extends the R2RML vocabulary to describe logical sources which are different from relational database tables. It generates RDF from JSON (exploiting JSONPath), XML (exploiting XPath), CSV[1], TSV, or HTML (exploiting CSS3 selectors). The approach is implemented on top of Sesame[2]. RML satisfies at least requirements R1, R3, R5. It would be possible to implement the support of binary data formats (R7), and ongoing research are led to integrate RDF sources on the Web of Linked Data (R2). Only RML and XSPARQL are specifically dedicated to the flexible generation of RDF from various formats.

In what follows, we propose an alternative to these approaches that is based on an extension of SPARQL 1.1, named SPARQL-Generate, that leverages its expressiveness and extensibility, and can be implemented on top of its engines.

4 SPARQL-Generate Specification

SPARQL-Generate is based on a query language that queries the combination of an RDF dataset and what we call a *documentset*, where each document is named and typed by an IRI. For illustration purposes, Fig. 1 is an example of a SPARQL-Generate query and the result of its execution on a RDF dataset that contains a default graph, and on a documentset that contains two documents identified by `<position.txt>` and `<measures.json>`. This query answers the question: *"What sensors are nearby, and what do they display?"*.[3] The concrete SPARQL-Generate syntax extends that of SPARQL 1.1 with three new clauses:

[1] RML is an implementation of the CSV on the Web standard [12].

[2] http://rdf4j.org/.

[3] Prefixes correspond to those registered at http://prefix.cc/ and are omitted to save space.

– The source clause is used to bind a variable to a document (here, `?pos` and `?measures` to the documents identified by `<position.txt>` and `<measures.json>`, respectively).
– The iterator clause allows to extract bits of documents using so-called *iterator functions*, duplicate a binding, and make a variable be successively bound to these extracted bits of documents (here, function `sgiter:JSONListKeys` is used to extract the set of keys of the JSON object that is bound to `?measures`, and successively bind `?sensorId` to these keys).
– Finally, the generate clause replaces and extends the construct clause with embedded SPARQL-Generate queries. This enables the modularization of queries and the factorization of the RDF generation.

Various data formats can be supported thanks to the extensible set of SPARQL 1.1 *binding* functions and SPARQL-Generate *iterator* functions.

Default graph (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 measures.json

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

Output (Turtle)

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

SPARQL-Generate query

```
GENERATE {
  ?sensor a :NearbySensor .

  GENERATE {
    ?sensorIRI :temp ?temp .
  }
  ITERATOR sgiter:JSONListKeys(?measures) 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 <measures.json> AS ?measures
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 )
}
```

Fig. 1. Example of a SPARQL-Generate query execution on a default graph and two documents. This running example illustrates requirements **R1** and **R2**

4.1 SPARQL-Generate Concrete Syntax

The SPARQL-generate syntax is very close to the standard SPARQL 1.1 syntax with only slight additions to the EBNF [6, Sect. 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'.' )?
```

While the production of SPARQL Queries and SPARQL Updates respectively start at `QueryUnit` and `UpdateUnit`, the production of a SPARQL-Generate query starts at rule `GenerateUnit`. We wanted to not rewrite any of the SPARQL 1.1 production rules, this is why we do not use construct and introduce generate instead. This concrete syntax has two notable features.

Negotiating the Document Type. The first notable feature is in production rule `[181]`. The optional part `('ACCEPT' VarOrIri)` allows to specify a type IRI for the document to bind in the source clause. If a SPARQL-Generate implementation chooses to look up the IRI of a document on the Web, they may retrieve different actual documents corresponding to different representations of the same resource. The optional accept component in the source clause is thought of as a hint for the implementation to choose how to negotiate the content of that resource. We chose to represent it as a IRI that identifies a document type, because the concept of content negotiation here goes beyond the usual HTTP Accept request header. It may also encompass other HTTP Accept-* parameters, and it may also describe other preferences to look up IRIs not related to the HTTP protocol. After negotiation with the server, the retrieved document type may be different from the requested document type.

Modularization and Reuse of Queries. The second feature is in production rule `[182]`, and enables to modularize queries. A SPARQL-Generate sub-query (i.e., a query in the generate part of a parent query) may contain a generate template, including graph patterns and potentially other sub-queries. It can also refer to a IRI. As for the documentset, implementations are free to choose how this IRI must be looked up to retrieve the identified SPARQL-Generate query. This feature does not need to be described in the abstract syntax, but allows in practice (i) to publish queries on the Web and make them callable by other, and (ii) to modularize large queries and make them more readable. Of course, implementations need to take care about loops in query calls.

For now, SPARQL-Generate implementations are free to choose whether and how they use these informations. Section 5.2 describes the choices we made for our own implementation on top of Apache Jena. Let us now introduce the abstraction of the SPARQL-Generate syntax.

4.2 Abstract Syntax

We note $\mathbf{I}$, $\mathbf{B}$, $\mathbf{L}$, and $\mathbf{V}$ the pairwise disjoint sets of *IRIs*, *blank nodes*, *literals*, and *variables*. The set of *RDF terms* is $\mathbf{T} = \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. The set of *triple patterns* is defined as $\mathbf{T} \cup \mathbf{V} \times \mathbf{I} \cup \mathbf{V} \times \mathbf{T} \cup \mathbf{V}$, and a *graph pattern* is a finite set of triple patterns. The set of all graph patterns is denoted $\mathcal{P}$. We denote $\mathbf{F_0}$ the set of SPARQL 1.1 function names,[4] which is disjoint from $\mathbf{T}$. We write $\boldsymbol{Q}$ the set of

[4] SPARQL 1.1 defines built-in functions with names `IF`, `IRI`, `CONCAT`, and so on.

SPARQL 1.1 query patterns. Finally, for any set X, we note $X^* = \bigcup_{n \geqslant 0} X^n$ the set of lists of X.

The set of *function expressions* is noted $\mathcal{E}$ and is the smallest set such that:

$$\mathbf{T} \cup \mathbf{V} \subseteq \mathcal{E} \quad \text{(e.g., } \texttt{<position.txt>}\text{)} \quad (1)$$

$$(\mathbf{F_0} \cup \mathbf{I}) \times (\mathbf{T} \cup \mathbf{V})^* \subseteq \mathcal{E} \quad \text{(e.g., } \texttt{CONCAT("\$.",?id), sgiter:JSONListKeys(?m)}\text{)} \quad (2)$$

$$\forall E \subseteq \mathcal{E}, (\mathbf{F_0} \cup \mathbf{I}) \times E^* \subseteq \mathcal{E} \quad \text{(i.e., the set of nested function expressions)} \quad (3)$$

The abstraction of production rule `[181]` is the set of *source clauses*, and enable to select a document in the documentset and bind it to a variable. For instance in the query above, variable `?pos` is bound to the document identified by `<position.txt>`. Let us introduce a special element $\omega \notin \mathbf{T} \cup \mathbf{V}$, that represents *null*, and let us note $\hat{X} = X \cup \{\omega\}$ the *generalized set of* X.

Definition 1 (source clauses). *The set $\mathcal{S}$ of source clauses is defined by equation $\mathcal{S} = \mathcal{E} \times (\hat{\mathbf{I}} \cup \mathbf{V}) \times \mathbf{V}$. We use notation $v \overset{source}{\longleftarrow} \langle e, a \rangle \in \mathcal{S}$ for a specific source clause, where $v \in \mathbf{V}$, $e \in \mathcal{E}$, and $a \in \hat{\mathbf{I}} \cup \mathbf{V}$.*

In most use cases, at some point one needs a given variable to iterate over several parts of the same document. For instance in the illustrating request, variable `?sensorId` is successively bound to the keys of the JSON object bound to `?measures`: `"s25"` and `"s26"`. Other examples include the results of a XPath query evaluation over a XML document,[5] or the matches of a regular expression over a string.[6] In SPARQL, binding clauses involving binding functions are the only way through which one could extract a term from a literal. Yet, these functions output at most one RDF term. So they cannot be used to generate more solution bindings. Consequently, we introduce a second extension, the set of *iterator* clauses, which output a *set* of terms, and replace the current solution binding with as many solution bindings as there are elements in that set.

Definition 2 (iterator clauses). *The set of iterator clauses is defined as $\mathcal{I} = \mathbf{I} \times \mathcal{E}^* \times \mathbf{V}$. We use notation $v \overset{iterator}{\longleftarrow} (u, e_0, \ldots, e_k) \in \mathcal{I}$ for a specific iterator clause, where $v \in \mathbf{V}$, $u \in \mathbf{I}$, $e_0, \ldots, e_k \in \mathcal{E}$, and $k \in \mathbb{N}$.*

We then extend the query pattern of SPARQL 1.1 queries $\mathcal{Q}$ with a list of source and iterator clauses, in any number and any order. We purposely do not change the definition of $\mathcal{Q}$ in order to facilitate the reuse of existing SPARQL implementations.

Definition 3 (SPARQL-Generate query patterns). *The set of SPARQL-Generate query patterns is defined as a sequence of source or iterator clauses followed by a query pattern: $\mathcal{Q}^+ = (\mathcal{S} \cup \mathcal{I})^* \times \mathcal{Q}$.*

[5] See test case *rmlproeg1* - http://w3id.org/sparql-generate/tests-reports.html.
[6] See test case *regexeg1* - http://w3id.org/sparql-generate/tests-reports.html.

Finally, the set of SPARQL-Generate queries augments $\mathcal{Q}^+$ with a basic graph pattern, and potentially other SPARQL-Generate sub-queries.

Definition 4 (SPARQL-Generate queries). *The set of SPARQL-Generate queries is noted $\mathcal{G}$, and defined as the least set such that:*

$$\mathcal{P} \times \mathcal{Q}^+ \subseteq \mathcal{G} \qquad \textit{(simple SPARQL-Generate queries)} \qquad (4)$$

$$\forall G \subseteq \mathcal{G}, \mathcal{P} \times G^* \times \mathcal{Q}^+ \subseteq \mathcal{G} \qquad \textit{(nested SPARQL-Generate queries)} \qquad (5)$$

SPARQL-Generate queries defined by Eq. 4 are comparable to SPARQL CONSTRUCT queries, where a basic graph pattern will be instantiated with respect to a set of solution bindings. Those defined by Eq. 5 contain nested SPARQL-Generate queries, which are used to factorize the generation of RDF. For example, this enables to first generate RDF from the name of all the JSON object keys, and then iterate over the values for these keys, which may be arrays.

4.3 SPARQL-Generate Semantics

This section reuses some concepts from the SPARQL 1.1 semantics, that we redefine in an uncommon, yet equivalent, way for convenience in notations and definitions.

Definition of the SPARQL-Generate Data Model. A SPARQL-Generate query is issued against a data model that extends the one of SPARQL, namely RDF dataset. An *RDF dataset* is a pair $\langle D, N \rangle$ such that D is an RDF graph, called the *default graph*, and N is a finite set of pairs $\langle u, G \rangle$ where u is an IRI and G is an RDF graph, such that no two pairs contain the same IRI. In order to allow the querying of arbitrary data formats, we introduce the notion of a *documentset*, analogous to RDF datasets.

Definition 5 (Documentset). *A documentset is a finite set of triples $\Delta \subseteq \mathbf{I} \times \hat{\mathbf{I}} \times \mathbf{L}$. An element of Δ is a triple $\langle u, a, \langle d, t \rangle \rangle$ where: u is the name of the document; a is the requested type for the document; literal $\langle d, t \rangle$ models the document; and the literal datatype IRI t is the document type. Δ must be such that no pair of distinct triples share the same two first elements.*

In order to lighten formulas, we also note $\Delta : \hat{\mathbf{T}} \times \hat{\mathbf{T}} \rightarrow \hat{\mathbf{L}}$ the mapping that associates a pair $\langle u, a \rangle$ to a literal l if and only if $\langle u, a, l \rangle \in \Delta$, and to ω otherwise. A set of documents can hence be stored internally, or represent the Web: u represents where a look up (e.g., a series of HTTP GET following redirections) must be issued, a describes how the content must be negotiated, d is the content of the successfully obtained representation, and t describes the representation type (its media type, language, encoding, etc.).

Mappings. The set of *mappings* is noted $\mathcal{M}$, and is defined by Eq. (6) as a function from $\mathbf{T} \cup \mathbf{V}$ to the generalized set of terms. As opposed to standard SPARQL 1.1, we use a total function defined on the full set of terms and variables, and rely on the element ω to represent the image of unbound variables. As in SPARQL, The *domain* of a mapping is the set of variables that are bound to a term (see Eq. (7)).

$$\mu : \mathbf{T} \cup \mathbf{V} \to \hat{\mathbf{T}} \text{ s.t., } \forall t \in \mathbf{T}, \mu(t) = t \tag{6}$$

$$\forall \mu \in \mathcal{M}, \mathsf{dom}(\mu) = \{v \in \mathbf{V} | \mu(v) \in \mathbf{T}\} \tag{7}$$

We introduce a distinguished set of mappings called *substitution mappings*, whose domain is a singleton. i.e., $\forall v \in \mathbf{V}$ and $t \in \hat{\mathbf{T}}$, $[v/t]$ is a substitution mapping with:

$$\forall t' \in \mathbf{T}, [v/t](t') = t', \quad [v/t](v) = t, \quad \text{and } \forall x \in \mathbf{V}, x \neq v, [v/t](x) = \omega \tag{8}$$

Then, the *left composition* operator $\fatsemi$ is defined such that in $\mu_1 \fatsemi \mu_2$, any variable that is commonly bound by μ_1 and μ_2 is finally bound to its value in mapping μ_1. In practice, this may be used to override bindings for variables in source or iterator clauses.

$$\mu_1 \fatsemi \mu_2 : \begin{cases} x \mapsto \mu_1(x) & \text{if } x \in \mathsf{dom}(\mu_1) \\ x \mapsto \mu_2(x) & \text{if } x \in \mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1) \\ x \mapsto \omega & \text{otherwise} \end{cases} \tag{9}$$

Binding and Iterator Function Map. Each SPARQL engine recognizes a set of binding function IRIs F_b (e.g. here, at least `sgfn:JSONPath`, `sgfn:SplitAtPosition`, and `ex:distance`). A binding function maps function expressions used in binding clauses to their evaluation, i.e., a RDF term. Formally, for a given SPARQL engine, Eq. (10) defines a *binding functions map* f_b, that associates to any *recognized binding functions* its SPARQL binding function. The *SPARQL-Generate iterator functions map* is defined analogously for a SPARQL-Generate engine (e.g. here, it recognizes at least `sgiter:JSONListKeys`), except the evaluation of a function expression is a *set* of RDF terms. Given a set F_i of *recognized iterator functions*, Eq. (11) defines the *iterator functions map* f_i:

$$f_b : F_b \to \left(\hat{\mathbf{T}}^* \to \hat{\mathbf{T}}\right) \tag{10}$$

$$f_i : F_i \to \left(\hat{\mathbf{T}}^* \to 2^{\hat{\mathbf{T}}}\right) \tag{11}$$

Generalized Mappings. We generalize the definition of mappings so that their domains include the set of function expression. The set of *generalized mappings* is noted $\bar{\mathcal{M}}$. It contains the *generalization* $\bar{\mu}$ of every mapping $\mu \in \mathcal{M}$, where $\bar{\mu} : \mathbf{T} \cup \mathbf{V} \cup \mathcal{E} \to \hat{\mathbf{T}}$ is defined recursively as follows:

$$\forall t \in \mathbf{T} \cup \mathbf{V}, \bar{\mu}(t) = \mu(t) \tag{12}$$

$$\forall \langle u, e_1, \ldots, e_n \rangle \in \mathcal{E} \text{ s.t. } u \in F_b, \ \bar{\mu}(\langle u, e_1, \ldots, e_n \rangle) = f_b(u)(\bar{\mu}(e_1), \ldots, \bar{\mu}(e_n)) \tag{13}$$

Evaluation of source *and* iterator *Clauses.* A source clause $v \overset{\text{source}}{\longleftarrow} \langle e, a\rangle \in \mathcal{S}$ is used to modify the binding μ so that variable v is bound to a document in Δ (e.g., `?pos` is bound to `"37.780496,-25.495157"`). An iterator clause $v \overset{\text{iterator}}{\longleftarrow} \langle t, e_0, \ldots, e_k\rangle \in \mathcal{I}$ is typically used to extract important parts of a document: from a binding μ, it enables, to generate several other bindings where variable v is bound to elements of the evaluation of $f_i(t)$ over $e_0, \ldots, e_k$ (e.g. here, `?sensorId` will be successively bound to `"s25"` then to `"s26"`). Any number of source or iterator clauses can be combined in a list. Let $\Sigma \in (\mathcal{S} \cup \mathcal{I})^n$, and $n \geqslant 1$. The set of solution mappings (i.e., the evaluation) for any list of source and iterator clauses $[\![\Sigma]\!]^\mu_\Delta$ can be defined by induction as follows:

$$[\![v \overset{\text{source}}{\longleftarrow} \langle e, a\rangle]\!]^\mu_\Delta = [v/\Delta(\bar{\mu}(e), a)] \fatsemi \mu \tag{14}$$

$$[\![v \overset{\text{iterator}}{\longleftarrow} \langle t, e_0, \ldots, e_k\rangle]\!]^\mu_\Delta = \left\{[v/t'] \fatsemi \mu | t' \in f_i(t)(\bar{\mu}(e_0), \ldots, \bar{\mu}(e_k))\right\} \tag{15}$$

$$[\![\langle \Sigma, v \overset{\text{source}}{\longleftarrow} e\rangle]\!]^\mu_\Delta = \left\{[\![v \overset{\text{source}}{\longleftarrow} e]\!]^{\mu'}_\Delta | \mu' \in [\![\Sigma]\!]^\mu_\Delta\right\} \tag{16}$$

$$[\![\langle \Sigma, v \overset{\text{iterator}}{\longleftarrow} e\rangle]\!]^\mu_\Delta = \bigcup_{\mu' \in [\![\Sigma]\!]^\mu_\Delta} [\![v \overset{\text{iterator}}{\longleftarrow} e]\!]^{\mu'}_\Delta \tag{17}$$

Evaluation of SPARQL-Generate Query Patterns. Let $Q \in \mathcal{Q}$ be a SPARQL 1.1 query pattern, D be an RDF dataset, and $[\![Q]\!]^\mu_D$ be the set of solution mappings for Q that are compatible with a mapping μ, as defined by the SPARQL 1.1 semantics. Let also Σ be a list of source and iterator clauses. Then the evaluation of the SPARQL-Generate query pattern $Q^+ = \langle \Sigma, Q\rangle \in (\mathcal{S} \cup \mathcal{I})^* \times \mathcal{Q}$ over D and a documentset Δ is defined by Eq. (18). We introduce a special *initial* mapping, $\mu_0 : v \mapsto \omega, \forall v \in \mathbf{V}$. Then, the set of solution mappings of any SPARQL Generate query Q^+ over Δ and D is defined by Eq. (19).

$$[\![Q^+]\!]^\mu_{\Delta,D} = \bigcup_{\mu' \in [\![\Sigma]\!]^\mu_\Delta} [\![Q]\!]^{\mu'}_D \tag{18}$$

$$[\![Q^+]\!]_{\Delta,D} = [\![Q^+]\!]^{\mu_0}_{\Delta,D} \tag{19}$$

Generate Part of the SPARQL Generate Query. For any graph pattern $P \in \mathcal{P}$ and any mapping $\mu \in \mathcal{M}$, we note $\mathfrak{t}^\mu(P)$ the RDF Graph generated by instantiating the graph pattern with respect to a mapping μ, following [6, Sect. 16.2.1]. We then define the evaluation of SPARQL-Generate queries recursively. Let be a simple SPARQL-Generate query $\langle P, Q\rangle \in \mathcal{P} \times \mathcal{Q}^+$, another query $G = \langle P, G_0, \ldots, G_j, Q\rangle \in \mathcal{P} \times \mathcal{G}^* \times \mathcal{Q}^+$, and a mapping μ. The following three equations define the RDF graph generated by G.

$$\mathfrak{t}^\mu_{\Delta,D}(\langle P, Q\rangle) = \bigcup_{\mu' \in [\![Q]\!]^\mu_{\Delta,D}} \mathfrak{t}^{\mu'}(P) \tag{20}$$

$$\mathfrak{t}^\mu_{\Delta,D}(\langle P, G_0, \ldots, G_j, Q\rangle) = \bigcup_{\mu' \in [\![Q]\!]^\mu_{\Delta,D}} \left(\mathfrak{t}^{\mu'}(P) \cup \bigcup_{0 \leq i \leq j} \mathfrak{t}^{\mu'}_{\Delta,D}(G_i)\right) \tag{21}$$

$$\mathfrak{t}_{\Delta,D}(G) = \mathfrak{t}^{\mu_0}_{\Delta,D}(G) \tag{22}$$

5 Implementation and Evaluation

5.1 Generic Approach

It is advantageous to be able to implement SPARQL-Generate on top of any existing SPARQL 1.1 engine. In fact, such an engine already provides us with: (i) the binding functions map f_b (thus one can know for any mapping $\mu \in \mathcal{M}$ its generalization $\bar{\mu}$ to any binding function expression); (ii) a function SELECT that takes a SPARQL 1.1 query pattern as input, and returns a set of solution mappings; (iii) a function INSTANTIATE that takes a graph pattern $P \in \mathcal{P}$ and a mapping $\mu \in \mathcal{M}$ as input, and returns the RDF Graph corresponding to the instantiation of P with respect to μ; (iv) the management of RDF datasets D. Then an implementation of SPARQL-Generate would just need to provide: (1) the management of a documentset Δ, and (2) the iterator functions map f_i.

Let $\mathcal{V} = 2^{\mathcal{M}}$ be the set of *inline data blocks*. Then we note $\langle V, Q\rangle \in \mathcal{Q}$ the result of prefixing some SPARQL query $Q \in \mathcal{Q}$ by some inline data block $V \in \mathcal{V}$. Theorem 1 below allows us to design a naive algorithm[7] (Algorithm 1) that can be used to implement SPARQL-Generate on top of a SPARQL 1.1 engine.

Theorem 1. *Let be a SPARQL 1.1 query $Q \in \mathcal{Q}$, and a list of* source *and* iterator *clauses $\Sigma \in (\mathcal{S} \cup \mathcal{I})^*$. The evaluation of the SPARQL-Generate query pattern $\langle \Sigma, Q\rangle \in \mathcal{Q}^+$ is equal to the evaluation of $\langle [\![\Sigma]\!]_\Delta, Q\rangle$, where $[\![\Sigma]\!]_\Delta$ is the evaluation of Σ.*

Proof. First note that in the SPARQL 1.1 semantics, the evaluation of a SPARQL 1.1 query pattern Q prefixed by an inline data block V is a join between the evaluation of V (i.e., $[\![V]\!]_D = V$), and the evaluation of Q (i.e., $[\![Q]\!]_D$). With our notations, this translates to: $[\![\langle V, Q\rangle]\!] = \bigcup_{\mu \in V} [\![Q]\!]^\mu$. Substituting V by $[\![\Sigma]\!]_\Delta = [\![\Sigma]\!]_\Delta^{\mu_0}$ and combining with Eqs. 18 and 19 leads to the proof:

$$[\![\langle [\![\Sigma]\!]_\Delta^{\mu_0}, Q\rangle]\!]_{\Delta,D} = \bigcup_{\mu' \in [\![\Sigma]\!]_\Delta^{\mu_0}} [\![Q]\!]_{\Delta,D}^{\mu'} = [\![\langle \Sigma, Q\rangle]\!]_{\Delta,D}^{\mu_0} = [\![\langle \Sigma, Q\rangle]\!]_{\Delta,D} \tag{23}$$

5.2 Implementation on Top of Apache Jena

This section overviews a first implementation of SPARQL-Generate with Algorithm 1 over the Jena ARQ SPARQL 1.1 engine.

[7] This algorithm is simplified and does not show subtleties in the management of blank nodes, which will be the focus of a future paper. On the other hand, the implementation already addresses this, see unit tests **bnode1** and **bnode2** at http://w3id.org/sparql-generate/tests-reports.html.

Algorithm 1. Naive implementation of SPARQL-Generate on top of any SPARQL 1.1 engine.

procedure GENERATE($\langle P, G_0, \ldots, G_j, \langle E_0, \ldots, E_n \rangle, Q \rangle, \mu$) ▷ See also Def. 4
  $M \leftarrow \{\mu\}$ ▷ M is a singleton containing one mapping
  for $0 \leq i \leq n$ **do**
    if $E_i = v \overset{\text{source}}{\longleftarrow} e$ **then** ▷ See also Def. 1
      for all $\mu \in M$ **do**
        $\mu(v) \leftarrow \mathbf{\Delta}(\bar{\mu}(e))$ ▷ See also Def. 5 and Eq. 12
      end for
    else if $E_i = v \overset{\text{iterator}}{\longleftarrow} \langle t, e_0, \ldots, e_k \rangle$ **then** ▷ See also 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** ▷ See also Eq. 11
          $\mu' \leftarrow \mu$; $\mu'(v) \leftarrow t'$; and $M' \leftarrow M' \cup \{\mu'\}$
        end for
      end for
      $M \leftarrow M'$ ▷ replace M by M'
    end if
  end for
  $M \leftarrow$ SELECT($\langle M, Q \rangle$) ▷ evaluate the query pattern prefixed by the computed inline data block
  $G \leftarrow \varnothing$ ▷ the empty RDF graph
  for $\mu \in M$ **do**
    $G \leftarrow G \cup$ INSTANTIATE(P, μ) ▷ operate a RDF graph union (not merge), i.e., do not merge blank nodes even if they share the same name
    for $0 \leq i \leq j$ **do**
      $G \leftarrow G \cup$ GENERATE(G_i, μ)
    end for
  end for
  return G
end procedure

Open-Source Code and Online Testbed. This implementation is open-source and available on GitHub,[8,9] released as a Maven dependency,[10] can also be used as an executable jar, or as a Web API. SPARQL-Generate can also be tested online using a web form that calls the Web API.[11] The SPARQL-Generate editor uses the YASGUI library,[12] which has been modified to support the SPARQL-Generate syntax. Finally, one can load any of the library unit tests in this web form. These unit tests cover use cases from related work and more.[13]

[8] http://w3id.org/sparql-generate/get-started.html.
[9] https://github.com/thesmartenergy/sparql-generate.
[10] http://search.maven.org/#search|ga|1|sparql-generate.
[11] http://w3id.org/sparql-generate/language-form.html.
[12] http://yasqe.yasgui.org/.
[13] http://w3id.org/sparql-generate/tests-reports.html.

Supported Data Formats, and Extensibility. Binding and iterator functions are available for the following formats: JSON and CBOR (exploiting JSONPath, thus satisfying requirement **R7**), CSV and TSV (conforming to the RFC 4180, or custom), XML (exploiting XPath), HTML (exploiting CSS3 selectors), and plain text (exploiting regular expressions). A complete documentation of the available binding and iterator functions is available along with the documentation of the API.[14] The implementation relies on Jena's SPARQL binding function extension mechanism, and copies it for iterator functions. Therefore, covering a new data format in this implementation merely consists in implementing new binding and iterator functions like in Jena. This satisfies requirement **R3**. Even what is not covered by existing query languages can be implemented as an iterator function. For example, iterator function `iter:JSONListKeys` iterates on key names of a JSON object, which is not feasible using JSONPath. As another example, polymorphic binding function `fn:CustomCSV` enables to parse a CSV document with or without a header. Parameters guide the parsing and data extraction from CSV documents with sparse structures, but the function itself checks for the existence of a header. If present, it treats the parameter column as a string to refer to a column. Else, it treats is as the column index. This function hence covers the *Dialect Description* of CSVW.

Specific Implementation Choices (see Sect. 4.1*).* For the documentset Δ, this implementation uses the `FileLocator` Jena utility. It hence looks up a IRI depending on its scheme, except if a configuration file explicitly specifies a mapping to a local file. For now, the `FileLocator` does not look up for IRIs with schemes other than `http` and `file`. The implementation still covers these cases in two ways: (a) they may be explicitly mapped to local files, or (b) they may be provided to the engine through some initial binding. For instance, test case named *cbor-venueeg1*, featuring CBOR, uses option (a).

If the source clause accept option is set to some IANA media-type URI of the form http://www.iana.org/assignments/media-types/text/csv, then the library negotiates the specified media type with the server.[15] In any other case, the datatype of retrieved documents defaults to `xsd:string`.

Similarly, when a query calls another query with its IRI, the implementation uses the `FileLocator` Jena utility. If not explicitly mapped to a local file, then the implementation uses the SPARQL-Generate registered media type `application/vnd.sparql-generate` (file extension `.rqg`) as the Accepted media type to fetch it on the Web.[16]

5.3 Evaluation

As RML is the language that most closely satisfies the identified needs, we conducted a comparative evaluation of it and SPARQL-Generate. This evaluation

[14] http://w3id.org/sparql-generate/functions.html.

[15] There is no consensus on the mapping between URIs and Internet Media Types, althought this is the object of a W3C TAG finding [13].

[16] https://w3id.org/sparql-generate/language.html#IANA_considerations..

focuses on two aspects: performances of the reference implementations, and cognitive complexity of the query/mapping. For this purpose, we chose to focus on a very simple transformation from CSV documents generated by `GenerateData.com` to RDF. For every line, a few triples with the same subject, fixed predicates, and objects computed from one column, are generated. The report and the instructions to reproduce this experiment are available online.[17]

Performance of the Reference Implementations. Figure 2 shows that for this simple transformation, the execution time with `sparql-generate-jena` becomes faster than `RML-Processor` above ∼1,500 rows, and linear. It is slightly above 3 min for 20,000 rows for `sparql-generate-jena`, when `RML-Processor` takes more than 6 min for 5,000 rows. Granted, comparing implementations does not necessarily highlight the true qualities of the approaches since optimizations, better choices of software libraries, and so on, could dramatically impact the results. Yet, with these experiments, we show that a straightforward and relatively naive implementation on top of Jena ARQ we achieve competitive performances. We argue that ease of implementation and use is the key benefit of our approach.

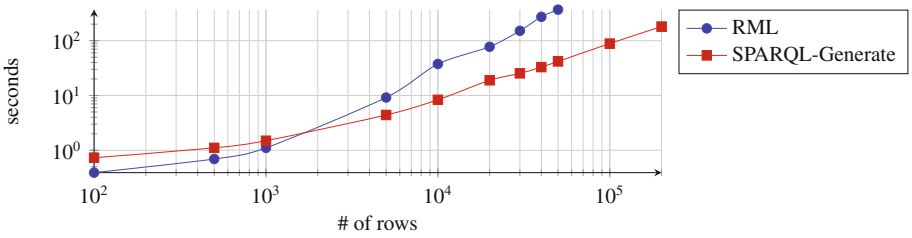

Fig. 2. Execution time for a simple transformation from CSV documents to RDF. Comparison between the current `RML-Processor` and `sparql-generate-jena` implementations.

Cognitive Complexity of the Query/Mapping. We conducted a limited study of the cognitive complexities of the languages we are comparing. On the experiment transformations, there are 12 terms from the R2RML and RML vocabularies, while SPARQL-Generate adds only 4 tokens to SPARQL 1.1 (`source`, `iterator`, `sgiter:CSV` and `sgfn:CSV`). Moreover, we realized that semantic web experts that have to carry on a triplification task usually observe the input data to identify the parts that have to be selected and formalize it with a selection pattern, such as a XPath or JSONPath query; then they draw an RDF graph or a graph pattern where they place the selected data from the input. This closely matches the structure of a SPARQL-generate query. The where clause contains the bindings that correspond to the select parts of the input documents; the generate clause contains the output graph patterns that reuse the extracted data. We also noticed

[17] https://w3id.org/sparql-generate/evaluation.html.

that when RML mappings get complex, they tend to grow to larger files than the equivalent SPARQL generate query, as can be witnessed by comparing our equivalent test cases.[18] These limitations in RML may be explained by the fact it extends R2RML whose triple maps are subject-centric. If one requires several triples to share the same object, then one must write several triple maps, that would have the same object map. This limitation impacts the cognitive complexity of the language. On the other hand, as the SPARQL-Generate concrete syntax is very close to that of SPARQL 1.1, we claim it makes it easy to learn and use by people that are familiar with the Semantic Web formalisms, satisfying requirement **R4** and **R5**. Nevertheless, from our own experience writing SPARQL-Generate queries, we identified some syntactic sugars that could strongly improve readability and conciseness of the queries. For instance one could use binding functions directly in the generate pattern, or use curly-bracket expressions instead of concatenating literals. Using such techniques, the running example query could be simplified as follows:

```
GENERATE {
  <http://example.com/person/{sgfn:CSV(?person,"PersonId")}> a foaf:Person ;
    foaf:name sgfn:CSV(?person, "Name" ) ;
    foaf:mbox <mailto:{sgfn:CSV(?person,"Email")}> ;
    foaf:phone <tel:{sgfn:CSV(?person,"Phone")}> ;
    schema:birthDate "{sgfn:CSV(?person,"Birthdate")}"^^xsd:dateTime ;
    schema:height "{sgfn:CSV(?person,"Height")}"^^xsd:decimal ;
    schema:weight "{sgfn:CSV(?person,"Weight")}"^^xsd:decimal .
} SOURCE <http://example.org/persons.csv> AS ?persons
  ITERATOR sgiter:CSV(?persons) AS ?person
```

Flexibility and Extensibility of the Languages. Work has been led to make RML be able to call external functions [8]. This is not necessary for SPARQL-Generate, and we believe that knowledge engineers are already familiar with SPARQL 1.1 functions, filtering capabilities, and solution sequence modifiers. This satisfies requirement **R6**.

6 Conclusion and Future Work

The problem of exploiting data from heterogeneous sources and formats is common on the Web, and Semantic Web technologies can help in this regard. However, adopting Semantic Web technologies does not automatically clear up those strong integration issues. Different solutions have been proposed to generate RDF from documents in heterogeneous formats. In this paper, we introduced a lightweight extension of SPARQL 1.1 called SPARQL-Generate, and compared it with the related work. We formally defined SPARQL-Generate and proved that it is (i) easily implementable on top of existing SPARQL engines; (ii) modular since extensions to new formats do not require a redefinition of the language (thanks to the use of SPARQL custom functions); (iii) easy to use by knowledge engineers because of its resemblance to normal SPARQL; and (iv) powerful

[18] See unit tests starting with RML⋆ at http://w3id.org/sparql-generate/tests-reports.html.

and flexible thanks to the custom function mechanism, the filtering capabilities, and the solution sequence modifiers of SPARQL 1.1. Our open-source implementation on top of Apache Jena covers many use cases, an is proven to be more efficient than the reference implementation of RML on a simple use case. Future plans consist of implementing more functions for more data formats, and extending the implementation to enable on the fly function integration (with an approach similar to [9]).

References

1. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J.: A direct mapping of relational data to RDF. W3C Recommendation, W3C, 27 September 2012
2. Connolly, D.: Gleaning resource descriptions from dialects of languages (GRDDL). W3C Recommendation, W3C, 11 September 2007
3. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recommendation, W3C, 27 September 2012
4. Dell'Aglio, D., Polleres, A., Lopes, N., Bischof, S.: Querying the web of data with XSPARQL 1.1. In: Proceedings of the ISWC Developers Workshop 2014, Co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy (2014)
5. Dimou, A., Sande, M.V., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: a generic language for integrated RDF mappings of heterogeneous data. In: Proceedings of the Workshop on Linked Data on the Web, Co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea (2014)
6. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Recommendation, W3C, 21 March 2013
7. Hert, M., Reif, G., Gall, H.C.: A comparison of RDB-to-RDF mapping languages. In: Proceedings the 7th International Conference on Semantic Systems, I-SEMANTICS 2011, Graz, Austria, pp. 25–32 (2011)
8. Junior, A.C., Debruyne, C., O'Sullivan, D.: Incorporating functions in mappings to facilitate the uplift of CSV files into RDF. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 55–59. Springer, Cham (2016). doi:10.1007/978-3-319-47602-5_12
9. Lefrançois, M., Zimmermann, A.: Supporting arbitrary custom datatypes in RDF and SPARQL. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 371–386. Springer, Cham (2016). doi:10.1007/978-3-319-34129-3_23
10. Lopes, N., Bischof, S., Polleres, A.: On the semantics of heterogeneous querying of relational, XML, and RDF data with XSPARQL. In: Proceedings of the 15th Portuguese Conference on Artificial Intelligence - Computational Logic with Applications Track (2011)
11. Polleres, A., Krennwallner, T., Lopes, N., Kopecký, J., Decker, S.: XSPARQL language specification. W3C Member Submission, W3C, 20 January 2009
12. Tandy, J., Herman, I., Kellogg, G.: Generating RDF from tabular data on the web. W3C Recommendation, W3C, 17 December 2015
13. Williams, S.: Mapping between URIs and internet media types. TAG Finding, W3C, 27 May 2002