Grouping multiple RDF graphs in the collections

Dominik Tomaszuk[1] and Henryk Rybiński[2]

[1] Faculty of Mathematics and Informatics,
University of Bialystok, Poland
dtomaszuk@ii.uwb.edu.pl
[2] Institute of Computer Science,
Warsaw University of Technology, Poland
h.rybinski@ii.pw.edu.pl

Abstract. This paper defines a document-oriented Resource Description Framework (RDF) graph store. It proposes collections for grouping multiple graphs. We define a lightweight representation of graphs which emphasizes legibility and brevity. We also present an implementation of our system, an algorithm of mapping to a pure RDF model and algorithms of generation and normalization. Our proposal supports knowledge metrics for RDF graphs.

Keywords: Semantic Web, graph store, document-oriented database, serialization, provenance, metric, semistructural data

1 Introduction

Knowledge representation deals with how knowledge is represented, in the case at hand based on Semantic Web standards including RDF [9] and OWL [1], while knowledge storing is the way in which the knowledge is retained in a computer: RDF, one of the foundations of Linked Data and the Semantic Web at large, is used for knowledge representation on the Web. The tools supporting processing and storage of RDF appeared in the beginning of the 21^{st} century, but they have a number of drawbacks and limitations:

- No mechanisms to store access and subgraph selection in compliance with the Linked Data principles.
- No possibility of grouping graphs, which make graph provenance and other related metrics hard if not impossible to realise.
- Problems with capacity related to data processing and access to data, resulting from no normalisation of structures; the existing proposals do not allow for generating optimal structures, needed in certain use cases.

The current solutions do not offer complete storage access mechanisms. The RDF graph store in connection with such proposals as semi-structured documents together with serialisation means a complete solution of problems related to knowledge storing and processing in a Linked Data environment, resulting in an improvement of the possibilities to access it in the graph store.

In this paper is presented an approach of grouping multiple RDF graphs in the collections, which providing various metrics. Our proposal extends RDF graphs to values, which can symbolize metrics such as temporal, uncertainty and trust. We also introduce implementation and algorithms of generating and normalization for this approach. Our proposal allows to store knowledge metrics near RDF graphs. Moreover, we propose document serialisation, which can contain additional metadata about stored RDF triples.

2 Collections in RDF Graph Store

In this section we introduce a document-oriented graph store with collections and document serialization for the graph store.

2.1 Collections and Graph Store

In this subsection we propose a document-oriented graph store that is not bound to any predefined database types. Instead, it is close to the RDF data, so that no predefined structure is needed. The graph store can be thought of as a store including containers so called *data collections* or simply *collections*. A data collection is similar to a relation from relational databases. A collection is represented by a graph, provenance and list of metrics. These collections include multiple *documents* and documents store serialized RDF statements. The concept of a *document* is a central element of the graph store. The documents consist of RDF data. For the sake of generality in our considerations, we define here a document as an ordered set of keys with associated values, which can be one of several different datatypes.

Hence, a collection can be seen as a group of RDF triples (representing documents). A *collection* is a tuple $C = \langle r, [v_1, v_2, \ldots, v_i], G \rangle$, where:

1. $r \in \mathcal{I}$ is the provenance of a graph, which can be interpreted as IRI,
2. $[v_1, v_2, \ldots, v_i]$ is a list of metrics ($v \in \mathcal{L}$), which can be interpreted as temporal [13], uncertainty [18] and/or trust metrics [20],
3. G is an RDF graph.

A provenance provides information about a graph's origin, such as who created it, when it was modifed, or how it was created. It is used for building representations of entities, involved in producing a piece of data. Special metrics provide information about RDF graph characteristics.

A *document-oriented graph store* is $GS_D = \{C_1, C_2, \ldots, C_i\}$, where every C_i is a collection, $i \geq 1$.

2.2 Document Serialization

In this subsection we introduce a concept of *RDF in JSON Document* (in the following sections denoted by RDFJD) and their serialization. Serialization is the process of converting a data structure into a format that can be stored and transmitted across the web and reconstructed later in the same or another computer environment. We define a document as a resource that serves as the container of semistructural data. One of the semistructural data formats is JavaScript Object Notation (JSON) [8], which is a syntax designed for human-readable data interchange and easy for machines to generate. It uses both simple datatypes, such as number, string or boolean and composite data types, such as array and object.

We propose serialization based on JSON, which is equivalent to the RDF model. The proposal is a lightweight textual syntax that can easily be modified by humans, servers and clients. The advantage of this syntax is that it can easily be transformed from other syntaxes. Another benefit of serializing RDF graphs in JSON is that there are many software libraries and built-in functions, which support the serialization.

The difference between regular JSON and RDFJD is that the above RDFJD object uniquely identifies itself on the World Wide Web and can be used, without introducing ambiguity across the Web Service using a document-oriented graph store.

The proposed structure can be modeled as a set of an abstract data structure with two operations:

1. $\mathcal{U} = get(C, \mathcal{Y})$ – returning a list of objects $\mathcal{U}$, where C is a collection, $\mathcal{Y}$ is a key,
2. $set(C, \mathcal{Y}, \mathcal{U})$ – causes a key $\mathcal{Y}$ and a list of objects $\mathcal{U}$ to be stored at a collection C.

We propose two types of RDFJD documents:

1. ***directive document***, which expresses the context of statement documents,
2. ***statement document***, which expresses RDF statements.

A *directive document* is associated with a collection and implements the knowledge metrics, provenance, and defines the short-hand names that are used throughout an RDFJD statement document. The directive document is a metadata package of a collection. This document should be unique. All the possible keys in a directive document are presented in Table 1. The list of metrics and provenance keys should impose a unique key constraint.

In the Listing 1 we present a directive document. The RDFJD document contains fields which define the provenance (`http://example.org/g1`) and trust (`0.9`) of the collection. It also defines a `foaf` prefix as an abbreviation for `http://xmlns.com/foaf/0.1/`.

```
{
  "_prov": "http://example.org/g1",
  "_metric": [0.9],
  "foaf": "http://xmlns.com/foaf/0.1/"}
```

Listing 1. Directive document

Table 1. RDFJD directive document keys

Key	Description
`_metric`	a predefined value of collection metric
`_prov`	a predefined value of collection provenance
prefix ID	abbreviating IRIs

A *statement document* is the main part, which stores RDF statements with extensions. A statement document uses subject-centric syntax, and it represents one or more properties of a subject. Often these documents occur more than once in the context of collection. They implement the subject as predefined keys, predicates as keys and objects as values. Plain literals with a language tag and typed literals are supported by special predefined keys. All the possible keys in a statement document are presented in Table 2.

In the Listing 2 we present a statement document. Key `foaf.name` is expand to value from a directive document (see Listing 1). The RDFJD document contains fields which define RDF statements:

1. triple 1: `http://example/voc#me`, `rdf:type`, `http://example/voc#Teacher`
2. triple 2: `http://example/voc#me`, `http://xmlns.com/foaf/0.1/name`, `John Smith`

```
{
 "_subject": "http://example/voc#me",
 "_type": {"_value": "http://example/voc#Teacher"},
 "foaf.name": {"_value": "John Smith"}
}
```

Listing 2. Statement document

3 Generating Algorithms

In this section we propose algorithms for serialization, normalization, and mapping into named graph model.

Table 2. RDFJD statement document keys

Key	Description
`_subject`	Used to identify subject that are being described
`_type`	Used to set the datatype of a subject
predicate key	Used to describe object
Possible values of predicate key	
`_value`	Used to specify the data that is associated with a particular predicate
`_lang`	Used to specify the native language for a particular object
`_datatype`	Used to specify the datatype for a particular object

3.1 Serialization and Normalization

Algorithm 2 shows the process of generating RDFJD statement document. The algorithm creates triples. The algorithm takes into account the simple literals without a language tag, simple literals with a language tag and typed literals.

There is the possibility that the same subject could occur in different RDFJD statement documents (e.g. because of the insertion of new statements). To improve the speed of data retrieval operations on a subject-centric statement there is the necessity to merge two or more statement documents with the same subject. Algorithm 1 presents the process of merging RDFJD documents. After this action an index may be applied to the *subject*.

```
  input  : set of statement document $SD$
  output: statement document $SD_M$
1 SDt ← sort($SD$);
2 foreach $s \in SDt$ do
3     if equal(current(), next()) then
4         merge(current(), next());
```

Algorithm 1: Merging statement documents

3.2 Mapping into Named Graph Model

In this subsection the mapping from our approach to the named graph model [6] is presented. A collections $C = (r, [], G)$ is equivalent to named graph $ng = (n, G)$, where $n \in \mathcal{I}$ is name of graph G. To case where $C = (r, [v_1, v_2, \ldots, v_i], G)$ we proposed to use `value` object property defined in [20], which allows to include metric values. Algorithm 3 presents the process of transformation, which uses named graphs.

```
input  : set of RDF triples $T$
output: set of statement documents $SD$
1  create root object;
2  foreach $t \in T$ do
3      get subject s from t;
4      insert s into "_subject" key;
5      get predicate p from t;
6      get object o from t;
7      if equal(p, "rdf:type") then
8          create "_type" key;
9          insert o into "_type" key;
10     else
11         add prefix(p) to directive document;
12         create key abbreviation(p);
13         if o is literal without a language tag then
14             insert o into abbreviation(p) key;
15         else if o is literal with a language tag then
16             create "_value" key in abbreviation(p) key;
17             insert o into "_value" key;
18             get language lg from o;
19             create "_language" key in abbreviation(p) key;
20             insert lg into "_language" key;
21         else
22             create "_value" key in abbreviation(p) key;
23             insert o into "_value" key;
24             get datatype dt from o;
25             create "_datatype" key in abbreviation(p) key;
26             insert dt into "_datatype" key;
```

Algorithm 2: Generating statement document

The RDF graph store can also be mapped to an RDF dataset. Following [14], RDF dataset DS consists of one graph, called the default graph, which does not have a name, and zero or more named graphs, each identified by IRI. We assume that $\mathcal{NG} = \{(u_1, G_1), (u_2, G_2), \ldots, (u_n, G_n)\}$ is a set of named graphs, where all IRI references are disjoint. An *RDF dataset* is $DS = \{G, \mathcal{NG}\}$, where G is called default graph and $\mathcal{NG}$ is a set of named graphs. If in $GS_D = \{C_1, C_2, \ldots, C_i\}$, $C_1 = (\varnothing, \varnothing, G)$ then DS is equivalent to GS_D. Otherwise, we suggest to map from C_i to (u_{i+1}, G_{i+1}) and use Algorithm 3. It is also possible to use RDF reification with the same metric in all statements, but this solution is much more verbose than our proposal.

```
   input  : collection $C$
   output: named graph $NG$, default graph $G$
 1 get r from $C$;
 2 get v from $C$;
 3 create $NG$ with r as a name;
 4 create default graph $G$;
 5 foreach $q \in C$ do
 6     get triple t from triple with metric;
 7     insert t into $NG$;
 8     get metric m;
 9     if equal(m, $\varnothing$) then
10         insert (r, "value", v) into $G$;
11     else
12         insert (r, "value", m) into $G$;
```

Algorithm 3: Mapping to Named Graphs

4 Implementation and Experiments

In this section we present the implementation and experiments of our approach. We used NoSQL database MongoDB[3] as the development platform. The testbed consists of the following three parts: query engine (applying matching Application Programming Interface), resources stored in collections (a part of the RDF graph store), and Representational State Transfer (REST) [11] client. The main part of the prototype is the matching API, which maps Hypertext Transfer Protocol (HTTP) request methods to object-oriented imperative query language.

Now we present load tests, which we performed on the Berlin SPARQL Benchmark [3]. We also discuss the results of these tests. The load experiment measures the time required to load on the testbed and Virtuoso Open-Source Edition 6.1, which is the leading graph store supporting the biggest Linked Data knowledge base DBpedia[4].

[3] `http://www.mongodb.org/`
[4] `http://dbpedia.org/`

In Fig. 1a. we show loading of normalized RDFJD serialization into our testbed and RDF into Virtuoso. This plot shows that loading triples into Virtuoso is much faster than loading statements into testbed. For the loading 40000 statements Virtuoso is up to 60 times faster. The testbed times are nearly quadratic to the number of quads and the coefficient of determination $R^2 \approx 0.99$. The Virtuoso times are nearly linear to the number of triples and the coefficient of determination $R^2 \approx 0.98$.

Taking into consideration that the times of textual RDFJD are nearly quadratic, we propose binary representation of RDFJD. The design goals for it emphasized performance. In particular, it is designed to be smaller and faster than textual version and it is fully compatible. Compared to textual RDFJD, binary RDFJD is designed to be efficient both in storage space and scan-speed. Our proposal represents data types in little-endian format. Large elements are prefixed with a length field to facilitate scanning. In Fig. 1b. we show the loading of binary normalized RDFJD serialization into the testbed. This plot shows that loading statements into the testbed is much faster than loading statements into Virtuoso. The load times of the testbed with binary serialization are approximately 2.4 times faster than the load times of Virtuoso. At 40000 statements the loading of binary RDFJD into the testbed is up to 10 times faster than the loading of RDF into Virtuoso.

Fig. 1a. Textual

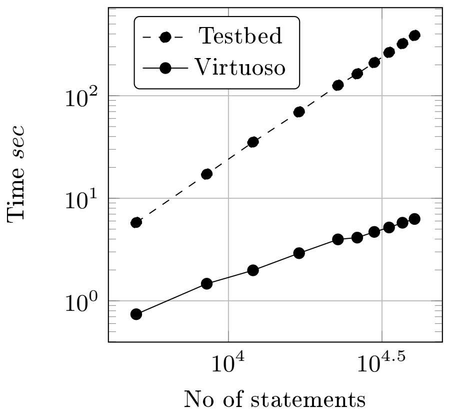

Fig. 1b. Binary

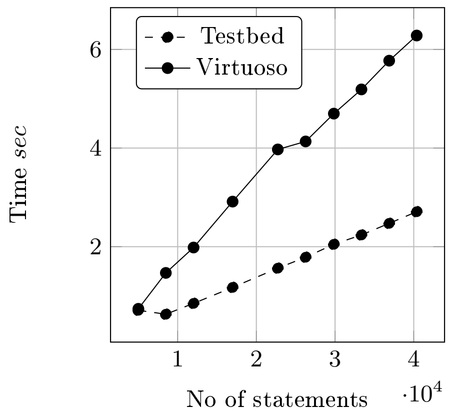

Fig. 1. Load test: Testbed comparison to Virtuoso

5 Related work

While the pure RDF does not allow referring to whole RDF graphs, named graphs introduced in [6] provide the means to group a set of statements in a graph. This approach may be sufficient for RDF graph stores only with provenance metrics. Unfortunately, it is not satisfactory for other metrics. Schenk et al. propose Networked Graphs [16]. It allows a user to define RDF graphs by using a SPARQL CONSTRUCT clause and a named graph model. Unfortunately, this approach may be insufficient for RDF graph stores, which do not support SPARQL queries or named graphs. Shaw et al. [17] propose vSPARQL which allows to define virtual graphs and use recursive subqueries to iterate over paths of arbitrary lengths. It also extends SPARQL by allowing to create new entities based upon the data encoded in existing datasets.

On the other hand there are RDF serializations [12, 7, 2, 15, 5, 4]. RDF/XML [12] is XML compatible syntax, which nodes and predicates must be represented in the names of elements, names of attributes, contents of elements or values of attributes. RDF/XML may not be fully described by such schemes as DTD or XML Schema. Another disadvantage of this syntax is its incapability of encoding all legal RDF graphs. It not handle named graphs, while Triples In XML (TriX) [6] serialisation does. TriX used XML syntax as well but it is not compatible with [12]. Another proposal refers to Terse RDF Triple Language (Turtle) [15] is simplification and subset of [2]. This solution offers textual syntax that makes it possible to record RDF graphs in a completely compact form. The drawbacks of this proposal include the fact that it is not capable of handling named graphs and its possibility to represent RDF triples in an unnormalised form. N-Triples [5] and N-Quards [4] are also a textual format of RDF serialisation. It is based on Turtle. Unfortunately, there are sign restrictions imposed on older version by US-ASCII standard and it does not handle named graphs. There are also serializations based on the JSON syntax [21, 19]

Foregoing serializations are supported by various graph stores. One of them is Virtuoso [10]. It is a row-wise transaction oriented database. It is re-targeted as an RDF store and inference. It is also revised to column-wise compressed storage and vectored execution.

6 Conclusions

The problem of how to group RDF triples and support metrics in these groups has produced many proposals. We assume that RDF, being more functional, should provide a method to set the metrics and provenance at the graph level.

We have produced a simple and thought-out proposal for grouping multiple RDF graphs in collections. We propose how our approach can be used in combination with various metrics. We believe that our idea is an interesting approach, because it can be transformed to the pure RDF and named graphs models. More importantly, we have provided algorithms for the generation and normalization of these semistructural data. Our approach extends the classical case of RDF with collections. The implementation shows its great potential.

We believe that our approach offers a flexible way to represent RDF data, we acknowledge, however, that there are areas that are subject to future investigation, such as replication of collections, versioning and access control.

References

1. Jie Bao, Elisa F. Kendall, Deborah L. McGuinness, and Peter F. Patel-Schneider. Owl 2 web ontology language quick reference guide (second edition). Technical report, World Wide Web Consortium, 2013.
2. Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable rdf syntax. Technical report, World Wide Web Consortium, 2008.
3. Christian Bizer and Andreas Schultz. Benchmarking the performance of storage systems that expose sparql endpoints. *World Wide Web Internet And Web Information Systems*, 2008.
4. Gavin Carothers. Rdf 1.1 n-quads. Technical report, World Wide Web Consortium, 2014.
5. Gavin Carothers and Andy Seaborne. Rdf 1.1 n-triples. Technical report, World Wide Web Consortium, 2014.
6. Jeremy J Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th international conference on World Wide Web*, pages 613–622. ACM, 2005.
7. Jeremy J Carroll and Patrick Stickler. Rdf triples in xml. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 412–413. ACM, 2004.
8. Douglas Crockford. The application/json media type for javascript object notation (json). Technical report, Internet Engineering Task Force, 2006.
9. Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. Technical report, World Wide Web Consortium, 2014.
10. Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.
11. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.
12. Fabien Gandon and Guus Schreiber. Rdf 1.1 xml syntax. Technical report, World Wide Web Consortium, 2014.
13. Claudio Gutierrez, Carlos A Hurtado, and Alejandro Vaisman. Introducing time into rdf. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, 2007.
14. Steve Harris and Andy Seaborne. Sparql 1.1 query language. Technical report, World Wide Web Consortium, 2012.
15. Eric Prud'hommeaux and Gavin Carothers. Rdf 1.1 turtle. Technical report, World Wide Web Consortium, 2014.
16. Simon Schenk and Steffen Staab. Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In *Proceedings of the 17th international conference on World Wide Web*, pages 585–594. ACM, 2008.
17. Marianne Shaw, Landon T Detwiler, Natalya Noy, James Brinkley, and Dan Suciu. vsparql: A view definition language for the semantic web. *journal of biomedical informatics*, 44(1):102–117, 2011.
18. Umberto Straccia. A minimal deductive system for general fuzzy rdf. In *Web Reasoning and Rule Systems*, pages 166–181. Springer, 2009.
19. Dominik Tomaszuk. Named graphs in rdf/json serialization. *Zeszyty Naukowe Politechniki Gdańskiej*, pages 273–278, 2011.
20. Dominik Tomaszuk, Karol Pąk, and Henryk Rybiński. Trust in rdf graphs. In *Advances in Databases and Information Systems*, pages 273–283. Springer, 2013.
21. World Wide Web Consortium. *Flat triples approach to RDF graphs in JSON*, 2010.