Exploiting RDF Open Data Using NoSQL Graph Databases

Raouf Bouhali, Anne Laurent

Exploiting RDF Open Data
Using NoSQL Graph Databases

R. Bouhali and A. Laurent

University of Montpellier
LIRMM - CNRS UMR 5506
F-34392 Montpellier, France
`armbouhali@gmail.com,laurent@lirmm.fr`
`http://www.lirmm.fr`

Abstract. RDF, or Resource Description Framework, is a well-established format for representing data. Based on the so-called RDF triples, it allows to represent data and properties over the data through the subject-predicate-object framework. By linking objects defined by unique identifiers, it is widely used for Open Data and Linked Open Data. On the other hand, relationships between objects are more and more managed through NoSQL graph databases which use the property graph model to structure information. Such databases provide indeed very efficient tools for handling huge volumes of complex data. They are especially powerful for retrieving relationships between objects (as for instance the famous *friend of friend* query). While many works have dealt with the transformation from various data formats to RDF, no work has addressed, as far as we know, the transformation from RDF to NoSQL graph databases. We thus consider this topic in this paper by proposing transformation rules and by testing our approach on real data.

Keywords: RDF, Open Data, NoSQL Graph Databases.

1 Introduction

The Semantic Web is now widely recognised as a key for promoting data exchanges over the Web of data. Data exchanges rely on the RDF format, especially considered in the Open Data and Linked Open Data frameworks as a key component. These frameworks are studied and used by many researchers and practitioners. For this reason, works have been proposed to translate resources either into RDF format so as to ease the publication of the data, or from RDF to perform on them some application-specific processing. Linked open data gathers a huge number of datasets. It is often represented by pictures such as Fig. 1 taken from the EU research project PlanetData (`http://planet-data.eu/`).

Besides, NoSQL graph databases have become highly demanded to achieve the data management needed by some tasks mainly focusing on relationships. For example, social networks, and variety of scientific and business application tend to be easy to manage if data were stored and queried as graphs. NoSQL

Fig. 1. Linked Open Data

graph database engines are recognised as being scalable [2] and efficient for some complex data management operations.

We thus propose to discuss how RDF data can be managed in NoSQL graph databases. Our work relies on two main motivations explained below.

First, we consider the extension of proprietary data available in a graph database. RDF is a standard for data publishing on the Web. Therefore, there is a wide variety of public data sets available for applications, having their own permissions and restrictions. Such external data are a rich source of information and can be used as Background Knowledge by applications either to enhance the quality of service or to provide users with extra features.

Second, we consider the improvement of performance of data processing applications. RDF serialization formats, such as RDF/XML, Turtle, and JSON-LD, are stored as text documents and cannot be queried in an efficient manner unless loaded into a SPARQL engine. Although studies have driven to some improvements in SPARQL efficiency, there are still some limits with complex or large-scale queries, where some of the current Graph Database solutions generally show better performance and are open to [2]

The paper is organized as follows. Section 2 recalls the basics of NoSQL graph databases and RDF data and presents some of the existing work for data transformation in this framework. Section 3 introduces our proposition. Implementation issues and experiments are reported in Section 4 while Section 5 concludes the paper and discusses further works.

2   Related Work

2.1   RDF

RDF stands for Resource Description Framework. A resource may be any type of object, for instance a person, a picture, a Web page, etc. Resources are identified by Uniform Resource Identifiers (URI) which are unique sequences of characters. URLs are a type of URIs. Some examples can be found in [8].

RDF describes information through the $(Subject, Predicate, Object)$ framework. The predicate is a relationship linking the subject to the object. RDF can thus be seen as a labeled oriented graph, as well as NoSQL graph databases described below. They are queried through SPARQL queries [7].

2.2   NoSQL Graph Databases

NoSQL has recently emerged as a new framework for databases [5]. The main advantage of NoSQL databases is often viewed as the capacity to deal with Big Data. This does not only mean that such NoSQL databases are scalable and can thus handle huge volumes of data, but also means that it can manage more complex data. That's for instance the case with document-oriented NoSQL databases. That's also the case with NoSQL graph databases [4] that are especially designed for managing data that are intrinsically graph-based, such as social networks, biological data, etc.

NoSQL graph databases implement data management systems based on oriented graphs. These graphs are labeled with a complex structure that contains information as a set of $(key : value)$ properties. For this reason, we consider such data as a *Property Graph* (hereafter PG).

Fig. 2 depicts an example where people and accommodations are managed. Every object, should it be a node or a relationship, can be associated with properties. For instance, the $RENTS$ relationship has some properties such as the fees and the dates of renting.

Serveral graph database engines exist, such as Neo4j, OrientDB, FlockDB, InfiniteGraph, etc.. In our work, we consider Neo4j (`http://neo4j.com/`) for experiments.

Those database management systems are not only data repositories but allow to deal with some constraints and provide querying tools. Queries can be run from a program or can rely on a declarative language. For instance, Neo4j provides a declarative query language called *Cypher*.

In this paper, we focus on how to build a bridge between the RDF format and the concept of property graphs, allowing then to manage these data by a chosen NoSQL Graph DBMS.

2.3   Graph Transformations

Graphs are usual data structures. Graph transformations have thus been studied for many years in graph theory and in both the fields of artificial intelligence and databases.
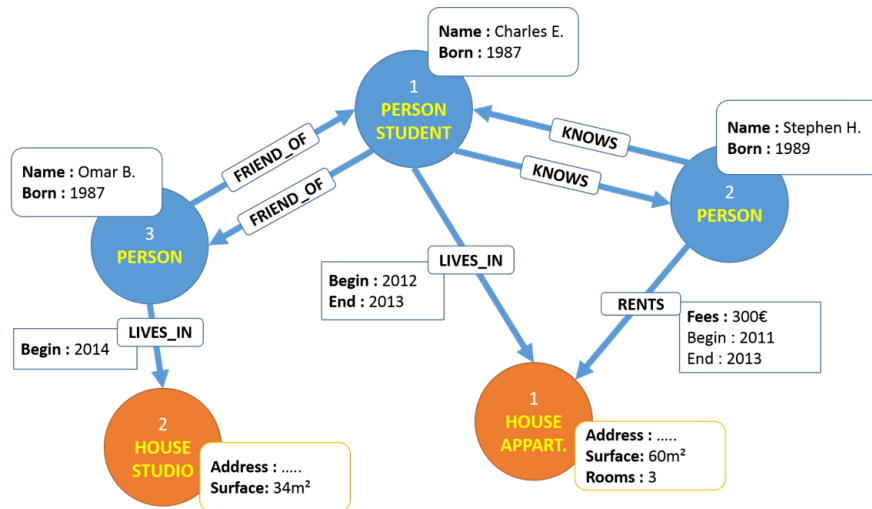
Fig. 2. Example of a Property Graph illustrating all the characteristics of the model

Some other works have focused on the transformation of several data formats to RDF in order to publish information on the Web [3]. One of the main used languages is R2RML, which provides a markup language to describe how to transform a given relational database into an RDF triple set[6]. Mappings are defined between relational data and graph patterns so as to allow the automatic exportation of table rows to RDF. Examples are provided by Figures 3, 4, 5, taken from `http://www.w3.org/TR/r2rml/`.

Fig. 3. Example of a relational database to be transformed with R2RML (http://www.w3.org/TR/r2rml)

Regarding the transformation of RDF to other formats, little works exist.

SPARQL queries return information in either XML or JSON, or Notation3 (N3) or textual/HTML formats, which allows to export RDF data into several other formats provided the fact that some transformation tools are available. CONSTRUCT queries return RDF/XML format. However, direct exportation has been less addressed in the literature. [1] addresses the topic of transforming RDF to conceptual graphs.

```
Example output data
<http://data.example.com/employee/7369> rdf:type ex:Employee.
<http://data.example.com/employee/7369> ex:name "SMITH".
<http://data.example.com/employee/7369> ex:department <http://data.example.com/department/10>.

<http://data.example.com/department/10> rdf:type ex:Department.
<http://data.example.com/department/10> ex:name "APPSERVER".
<http://data.example.com/department/10> ex:location "NEW YORK".
<http://data.example.com/department/10> ex:staff 1.
```

Fig. 4. Example of the resulting RDF data from an R2RML mapping (http://www.w3.org/TR/r2rml)

```
Example R2RML mapping
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://example.com/ns#>.

<#TriplesMap1>
    rr:logicalTable [ rr:tableName "EMP" ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{EMPNO}";
        rr:class ex:Employee;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:name;
        rr:objectMap [ rr:column "ENAME" ];
    ].
```

Fig. 5. Example of an R2RML Mapping (http://www.w3.org/TR/r2rml)

However, as far as we know, no work has addressed the transformation of RDF to NoSQL graph databases. That is the reason why we propose in this paper a method to transform RDF data to property graphs. The next sections detail the rules we propose and the experiments we have conducted in order to assess our proposition.

3   From RDF Data to NoSQL Graph Databases

The PG model is very general. It extends the basic graph model and adds some concepts from the class model where objects and attributes are equivalent to nodes properties. This model has almost no restrictions except the fact that a relationship can have one and only type. Indeed, if a relationship is multi-typed, its types are either equivalent and only one should be kept, or different and in this case a new relationship should be created for every distinct type.

RDF is a very basic model allowing to represent information as triples (Subject, Predicate, Object). It can be extended to something more complex by adding a semantic layer.

Since RDF is nothing but a sub-model of PG, a straightforward transition is possible from RDF to PG. In other words, any set of RDF triples can be translated into a property graph without any loss in terms of structure.

However, the opposite transition, namely from PG to RDF, is generally structure-lossy, because some concepts found in property graphs, such as relationship properties and collection properties, cannot be directly represented

using the (Subject, Predicate, Object) paradigm without a decomposition into atomic triples.

In both directions, the description of the final usage of the data is what really determines the appropriate transformations and the semantics of structures, rather than the trivial correspondences between the two models.

We propose describing in this work a solution to convert RDF data to fit the Property Graph model. Our solution comprises two approaches: the first is to establish direct correspondences between RDF triples and the primitives of property graphs; the second approach offers more flexibility by integrating a detailed mapping for each category of RDF triples.

3.1   Approach 1: Direct Correspondences

This approach processes RDF triples in the most intuitive manner and uses correspondences that are generally correct for the majority of known datasets. Its objective is to simply afford a conversion that meets the nature of the data, and thus allow to start processing them as soon as possible using a graph database.
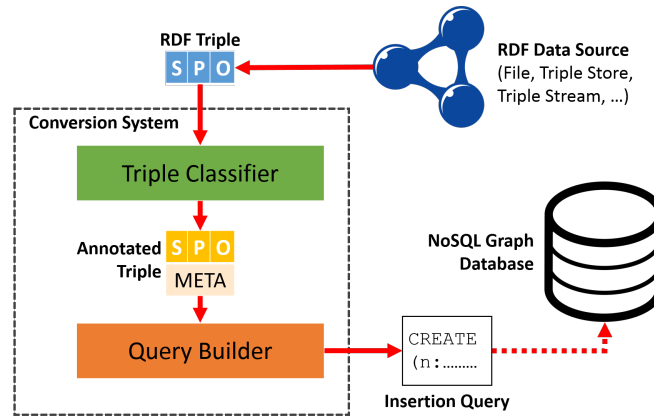
The conversion system is illustrated by Fig. 6

Fig. 6. Description of RDF to PG Conversion System - Approach 1

First, the system takes an input triple (S, P, O) and injects it into the Classifier. The Classifier then annotates the triple according to its structure (i.e. the nature of the object O and the predicate P's name). The annotated triple is then passed to the Query Builder which integrates the three parts of the triple in an insertion query, taking in account the previously added meta-data. The consequent query is finally sent to the output of the system where the graph database engine is awaiting for it to be executed.

Below, we concisely define the chosen matching between RDF and Property Graph entities, and explain the rules used to convert any given set of triples into insertion queries intended to be run on a NoSQL graph database.

RDF Entities	Property Graph Primitives
Resource (URI)	Node {Key}
Literal	Node Property value
RDF Triple Classes	**Property Graph Structures**
(Resource, Property, Literal)	(Node {Key, Property : Value)})
(Resource, Property, Resource)	(Node {Key})[r :Relationship]-(Node {Key})
(Resource, rdf:type, Type)	(Node:Type {Key})

Table 1. Direct correspondences between RDF and PG model entities

The rules are :

- **Rule #1:** Every RDF resource becomes a PG node. Since the subject in RDF is always a resource, there is at least one PG node created or merged into an existing node.
- **Rule #2:** If the object is a literal, the predicate and object become respectively a propertys name and value. The property is added to the created or existing node corresponding to the resource.
- **Rule #3:** If the object is a resource, then the subject and object are each transformed into a node. A relationship between them is created and it holds the predicate's name as a relationship type.
- **Rule #4:** If the predicate is rdf:type, the subject becomes a node having a label set to the objects name (which is actually the resource type).

This rule set is minimal. It guarantees a coverage of the three different classes defined DF triples. A concise example of the application of these rules is described in Fig. 7

The main advantage of this method is that all the rules are triple-intrinsic. in other words, the conversion system processes only one triple at a time and doesn't take in account many triples. This makes it possible to handle the input as a serial or continuous stream of RDF triples and achieve conversions on-the-fly. This system can fit well with continuously evolving data such as data series, dynamic databases and transactional databases in general.

The main drawback of this approach is the fact it doesn't make full use of the Property Graph model. In fact, the conversion system cannot produce any properties over relationships. Relationships are provided only by applying **Rule #3** defined above. But since all the data are present, it is possible to extend this conversion with refactoring queries executed by the graph database engine: their role is to match specific graph patterns and apply on them some modifications to create new structures.
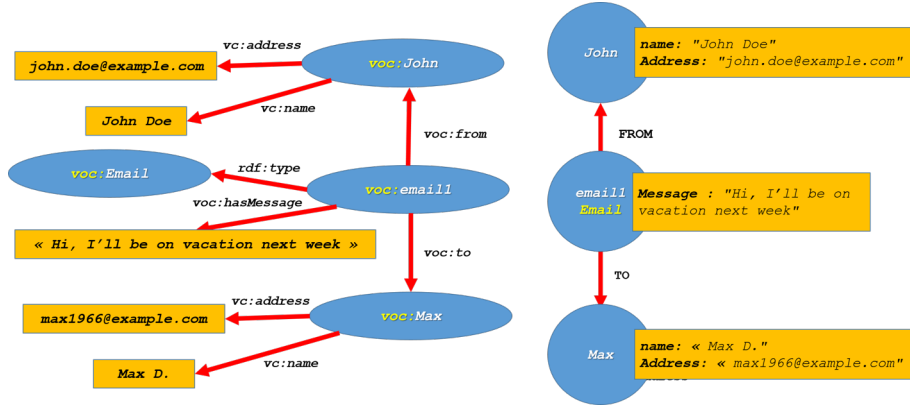
Fig. 7. Description of RDF to PG Conversion System - Approach 1

3.2   Approach 2: Correspondences by Mapping

This approach is an extension of the previous system. The Query Builder is enhanced with an input mapping that associates to each RDF entity a chosen correspondence among many possibilities.

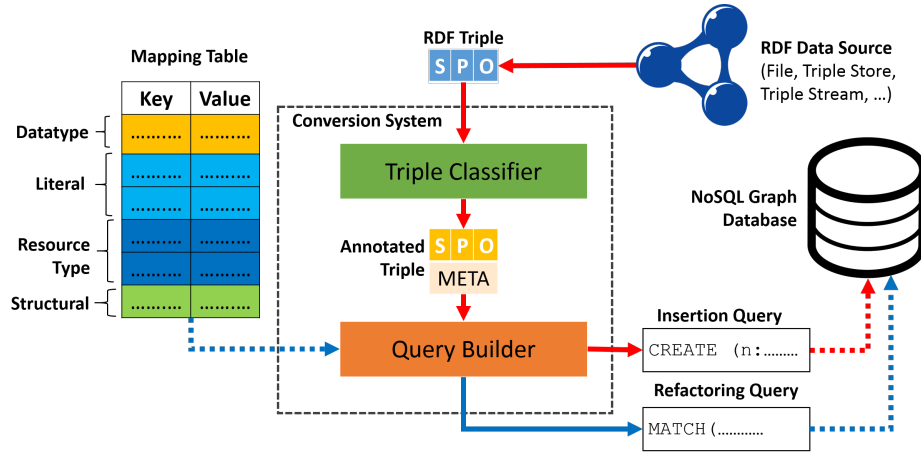The conversion system for this approach is described by Fig. 8

Fig. 8. Description of RDF to PG Conversion System - Approach 2

The mapping is divided into 4 sub-mappings, each of them is stored in a (key:value)-style vector. We define them as follows:

1. **Literal Datatype Mapping:** RDF data types are not always rigorously attributed to literals. We thus propose this mapping as a correction measure

Mapping	Key	Possible values
Literal Datatype Mapping	Predicate Name	Primitive data types : Integer, Float, String
Literal Mapping	Predicate Name	Property, Node with Property, Collection Property
Resource Type Mapping	rdf:type Object	Label, Label Collection Property
Structural Mapping	Graph Pattern	Graph Pattern replacement

Table 2. Table of Mappings

in order to store literal values in the most efficient manner. The output type is by default *String* and it can be set to an *Integer* or *Float* value.

2. **Literal Mapping:** A literal found in a triple always causes a property to be created. The goal of this mapping is to define the correct structure of this property. Fig. 9 shows an example with a bibliographic dataset. The choice of the mapping is established according to the relevance of the information to the user: If authors are auxiliary they can be grouped in a Collection Property as in (C), and if they are important, they can be represented with Nodes as in (B).

3. **Resource Type Mapping:** rdf:type objects are by default mapped to labels over the nodes. But when many rdf:type statements are defined for the same resource, this may pollute the label space and affect performance. This mapping offers the possibility to store insignificant rdf:type objects within a Collection Property instead of a Label.

4. **Structural Mapping:** The three previous mappings are all triple-intrinsic and are not designed to handle multi-triple based conversions. The role of Structural mapping is to achieve any missing transformations. It concerns the post-import phase and works directly on the graph database. It defines a set of match-replace rules to build the necessary refactoring queries. . Fig. 10 takes the output of the previous email data sample conversion shown in Fig. 7, and runs a refactoring query on it by replacing every match of the pattern:

$$(p1) \leftarrow [: FROM] - (e : EMAIL\{\$1\}) - [: TO] \rightarrow (p2)$$

by the pattern:

$$(p1) - [: EMAILED\{\$1\}] \rightarrow (p2)$$

where \$1 is used to capture all the properties of nodes labeled EMAIL and adds them as properties to the new relationship EMAILED. This mapping reduces a node with two relationships into a relationship holding the node's properties.

The use of a mapping is common in most data conversion solutions. However, it assumes the user has a good knowledge, not only about his input dataset, but also the final graph processing application, and this is generally not the case.
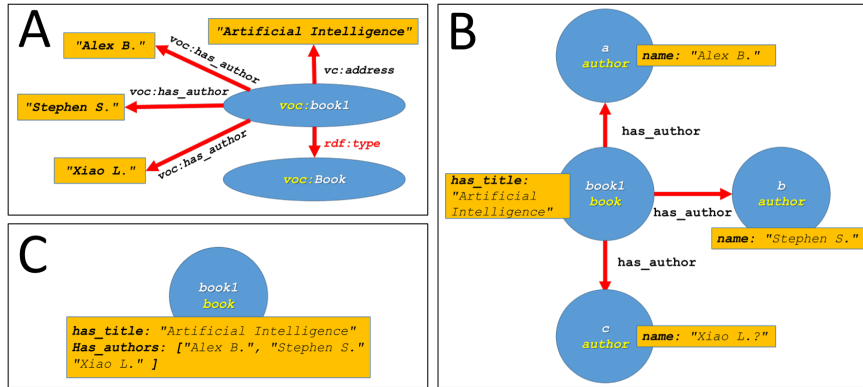
Fig. 9. Mapping RDF literals to PG structures. A is the original RDF sample. B and C are PG conversions of A with different mapping parameters.
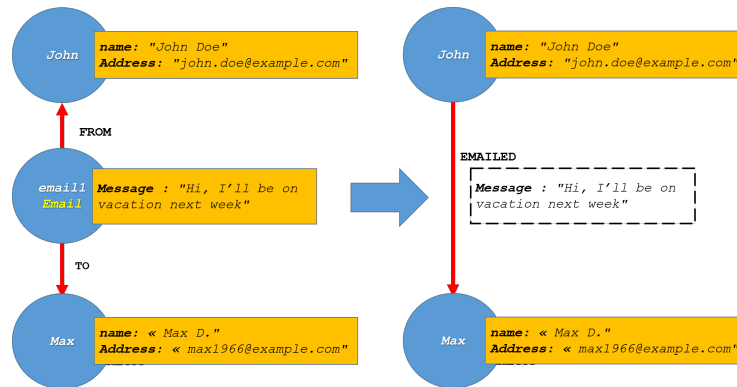

Fig. 10. Example of structural mapping applied on an e-mail graph database

3.3   Discussion

NoSQL Graph Database Engines are characterized by multi-level cache management policies. In other words, data primitives (node, relationship, properties ...) are not always stored in the same cache or accessed in same manner. This leads to a contrast in the time necessary to retrieve a specific information. For example, Neo4j defines the following cache priority order over its different primitives (1 being the highest priority) :

1. Nodes + Labels, Relationships + Types
2. Indexed Properties
3. Unindexed properties:
   (a) Static size properties (int, float, etc.)
   (b) Dynamic size properties (Collections, Strings)

This access policy and caching should be taken into account to make the good mapping decisions that will likely improve query performance. Indeed, any frequently accessed information should be put forward as a node or a relationship if possible. or as an indexed priority. The same goes for irrelevant information that should be kept away from being cached by putting them in Dynamic size properties, or even by removing them if necessary.

As mentioned at the end of section 3 as a drawback for the second approach, when the user is inexperienced, it is preferable to employ user-friendly solutions with a higher abstraction level to help him understand the needs of the system, and assist him to make the good decisions. Thus we propose two ideas worth working on :

1. One keypoint is to deal with the structure of the data. Some works in the literature have dealt with schema extraction in the context of semi-structured graph data, i.e., XML data.
2. Another part of the issue is how to know more about the application. In this case, instead of taking a mapping as an input, we can generate a mapping from a given application profile. The profile is supposed to provide information about Graph Database queries, their extent, frequency, and their estimated running time. If such information is not available, it is still possible to ask the user in a more user-friendly way about the application through a set of artificially generated questions. The user's answers on those questions will allow to define specific constraints on the nature of the queries and make, according to that, the necessary refactoring and optimizations to improve the overall performance.

4   Implementation Issues

We have implemented the conversion system detailed by the Approach 2 (see Fig. 8). The solution is coded in Java. It uses Apache Jena packages to parse and process RDF triples, and imports data into a Neo4j Graph database by

generating and running Cypher queries directly using the Neo4j API. It is also possible to provide the queries in an output file instead of injecting them into the database engine directly.

The experiments are led on the DBLP database dump provided in RDF format by RKBExplorer (`http://dblp.rkbexplorer.com/`).

4.1   Evaluation Measures

In order to check the reliability of our method and provide the user with a relevant feedback to make some changes on his input mapping, we define three quality measures. Given a Graph Database $G = (N, R)$ where an N and R are respectively the node set and the relationship set of G:

1. **Conciseness:** this measure simply calculates the sum of the number of nodes and the number of relationship instances in the Graph database. The lower this number, the better the performance.

$$Conciseness(G) = |N| + |R|$$

   Can be used can be used to calculate the variation of the graph size when the user has to make a choice among different mappings.
2. **Connectivity:** it is defined as the the number of relationships among the total number of nodes.

$$Connectivity(G) = \frac{|R|}{|N|}$$

   A connected graph has a connectivity around 1 and a graph having at least $Connectivity(G) > 1.5$ is considered good for processing. Higher values mean the graph database contains strong connections between its nodes. This measure can be seen as the average rank of a vertex in graph theory.
3. **Key Connectivity:** instead of calculating the connectivity of the whole graph database, we focus here only on a set of predefined key relationship types. We note R' as the set of key relationship instances, and N' the set of distinct Nodes related by the elements of R'. R' and N' are by definition subsets of R and N respectively.

$$KeyConnectivity(G) = Connectivity((N', R')) = \frac{|R'|}{|N'|}$$

   The smallest value of $KeyConnectivity(G)$ is 0.5 and is obtained when we choose a single relationship type, or many unrelated relationship types. Any value larger than this reference means that elements of N' are related by more than one relationship. This measure can be interpreted in several ways and depends of the elements of R'. The principle is similar to independence calculations of statistical variables.

Total Nodes	3406
Total Relationships	9862
Total Relationships for *has_author*	2364
Total Nodes related by *has_author*	4728
Conciseness	13268
Connectivity	2.895
Key Connectivity for *has_author*	0.5

Table 3. Import results and evaluation measures

We applied these measures on an imported chunk of the DBLP dataset in Neo4j and we obtained the results shown in Table 3

Connectivity is around 3 which means our graph is rich in terms of relationships. it can thus be used and explored in different ways since its nodes have many connections.

Key connectivity equals 0.5. As seen in definition, considering *has_author* as the only key relationship type, this value means there are no double relationship instances between nodes related by *has_author*. It would be interesting if the measure is applied on more relationship types, but since the database is small, other relationship type are under-represented.

5   Conclusion

In this paper, we discuss how to transform RDF data to Property Graph (PG) in the NoSQL graph database context. Transformation rules are proposed. They have been implemented and tested on real data. There is still room for improvements in theory as well as practice. For example, we can define explicitly what kind characteristic information should reflected by the application's and the dataset's profiles. And then how to use those profiles to achieve conversion in a more automatic way with a minimum user intervention.

Further work will also include the study of transformations in the opposite direction, i.e. from property graphs to RDF. A transition from the property graph model to RDF is mainly motivated by:

- The need to publish data: RDF is a standard of the Semantic Web. It is not only a data representation convention but also a set of ready-to-use tools which have been studied and developed in way that guarantees interoperability with different applications.
- The need to link the published dataset with other existing open data sources on the Web (for instance, the LOD project) to make it more useful. This can be done by enriching datasets with a semantic RDFS/OWL layer named ontology.

Our main research is about using NoSQL graph databases as support for summarizing Open Data, and this paper has details one preliminary phase in this process and provides a way to import data from RDF triplestores to Graph Databases built upon the Property Graph paradigm.

References

1. Baget, J.F., Chein, M., Croitoru, M., Fortin, J., Genest, D., Gutierrez, A., Leclère, M., Mugnier, M.L., Salvat, E.: RDF to Conceptual Graphs Translations. In: CS-TIW'09: 3rd Conceptual Structures Tool Interoperability Workshop @ICCS'09: 17h International Conference on Conceptual Structures. p. 17. No. 5662 in LNAI, Springer, Moscow, Russia (Aug 2009), `http://hal-lirmm.ccsd.cnrs.fr/lirmm-00410621`
2. Barmpis, K., Kolovos, D.S.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. Journal of Object Technology 13(3), 3: 1–26 (2014), `http://dx.doi.org/10.5381/jot.2014.13.3.a3`
3. Braatz, B., Brandt, C.: Graph transformations for the resource description framework. ECEASST 10 (2008), `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/158`
4. Castelltort, A., Laurent, A.: In: Eighth International Conference on Digital Information Management (ICDIM 2013)
5. Cattell, R.: Scalable sql and nosql data stores. SIGMOD Rec. 39(4), 12–27 (2011)
6. Das, S., Sundara, S., Cyganiak, R.: R2rml: Rdb to rdf mapping language (w3c working draft) (2011), `http://www.w3.org/TR/r2rml/`
7. Prud'hommeaux, E., Seaborne, A.: Sparql query language for rdf (2008), `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
8. Sauermann, L., Cyganiak, R., Völkel, M.: Cool uris for the semantic web. Technical Memo TM-07-01, DFKI GmbH (2007), `http://www.dfki.uni-kl.de/dfkidok/publications/TM/07/01/tm-07-01.pdf`, written by 29.11.2006