G2GML: Graph to Graph Mapping Language for Bridging RDF and Property Graphs

Hirokazu Chiba[1], Ryota Yamanaka[2], and Shota Matsumoto[3]

[1] Database Center for Life Science, Chiba 277-0871, Japan
chiba@dbcls.rois.ac.jp
[2] Oracle Corporation, Bangkok 10500, Thailand
ryota.yamanaka@oracle.com
[3] Lifematics Inc., Tokyo 101-0041, Japan
shota.matsumoto@lifematics.co.jp

Abstract. How can we maximize the value of accumulated RDF data? Whereas the RDF data can be queried using the SPARQL language, even the SPARQL-based operation has a limitation in implementing traversal or analytical algorithms. Recently, a variety of database implementations dedicated to analyses on the property graph (PG) model have emerged. Importing RDF datasets into these graph analysis engines will enable access to the accumulated datasets through various application interfaces. However, the RDF model and the PG model are not interoperable. Here, we developed a framework based on the Graph to Graph Mapping Language (G2GML) for mapping RDF graphs to PGs to make the most of accumulated RDF data. Using this framework, accumulated graph data described in the RDF model can be converted to the PG model, which can then be loaded to graph database engines for further analysis. For supporting different graph database implementations, we redefined the PG model and proposed its exchangeable serialization formats. We demonstrate several use cases, where publicly available RDF data are extracted and converted to PGs. This study bridges RDF and PGs and contributes to interoperable management of knowledge graphs, thereby expanding the use cases of accumulated RDF data.

Keywords: RDF · Property Graph · Graph Database

1 Introduction

Increasing amounts of scientific and social data are being published in the form of the Resource Description Framework (RDF) [1], which presently constitutes a large open data cloud. DBpedia [2] and Wikidata [3] are well-known examples of such RDF datasets. SPARQL [4] is a protocol and query language that serves as a standardized interface for RDF data. This standardized data model and interface enables the construction of integrated graph data. However, the lack of an interface for graph-based analysis and performant traversal limits use cases of the graph data.

Recently, the property graph (PG) model [5,6] has been increasingly attracting attention in the context of graph analysis. Various graph database engines, including Neo4j [7], Oracle Database [8], and Amazon Neptune [9] adopt this model. These graph database engines support algorithms for traversing or analyzing graphs. However, few datasets are published in the PG model and the lack of an ecosystem for exchanging data in the PG model limits the application of these powerful engines.

In the light of this situation, developing a method to transform RDF into PG would be highly valuable. One of the practical issues faced by this challenge is the lack of a standardized PG model. Another issue is that the transformation between RDF and PG is not straightforward due to the differences in their models. In RDF graphs, all information is expressed by triples (node-edge-node), whereas in PGs, arbitrary information can be contained in each node and edge as the key-value form. Although this issue was previously addressed on the basis of predefined transformations [10], users still cannot control the mapping for their specific use cases.

In this study, we redefine the PG model incorporating the differences in existing models and propose serialization formats based on the data model. We further propose a graph to graph mapping framework based on the Graph to Graph Mapping Language (G2GML). Employing this mapping framework, accumulated graph data described in RDF can be converted into PGs, which can then be loaded into several graph database engines for further analysis. We demonstrate several use cases, where publicly available RDF data is extracted and converted to PGs. Thus, this study provides the foundation for the interoperability between knowledge graphs.

The main contributions of this study are as follows: 1) language design of G2GML and 2) its prototype implementation. Furthermore, we propose 3) the common PG model and its serialization, which are essential to ensure that this mapping framework is independent from the implementations of databases.

2   Graph to Graph Mapping

2.1   Overview

We provide an overview of the graph to graph mapping (G2G mapping) framework (Figure 1).

In this framework, users describe mappings from RDF to PG in G2GML. According to this G2GML description, the input RDF dataset is converted into a PG dataset. The new dataset can also be optionally saved in specific formats for loading into major graph database implementations.

G2GML is a declarative language comprising pairs of RDF patterns and PG patterns. The core concept of a G2GML description can be represented by a map from RDF subgraphs, which matches specified SPARQL patterns, to PG components.
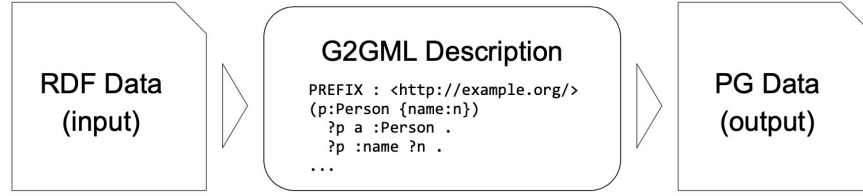
Fig. 1. Overview of mapping from RDF to PG

2.2   G2GML

G2GML is a language to describe maps from RDF graphs to PGs. It represents a domain-specific declarative language, where programmers only have to describe patterns of RDF and PG. The RDF pattern is described in terms of the SPARQL syntax, and the PG pattern is described in a syntax derived from openCypher [24].

Preliminaries. *An RDF triple (s, p, o) is an element of $(I \cup B) \times I \times (I \cup B \cup L)$, where I, L, and B are a set of IRIs, a set of literals, and a set of blank nodes, respectively, which are considered pairwise disjoint. For an RDF triple (s, p, o), s is the subject, p the predicate, and o the object. An RDF graph is defined as a finite set of RDF triples.*

A G2GML description comprises one or more *node maps* and *edge maps*. A *node map* is a function that maps resources from an RDF graph to a set of *PG nodes*. Similarly, an *edge map* is a function that maps resources from an RDF graph to a set of *PG edges*. Here, we define the syntax of G2GML as follows.

Definition 1 (EBNF notation of G2GML)**.**

```
g2gml   ::= SPARQL_PREFIX* mapping+
mapping ::= ( node_pat | edge_pat ) NEWLINE
            INDENTED_RDF_PATTERN NEWLINE
```

In Definition 1, `SPARQL_PREFIX` and `INDENTED_RDF_PATTERN` are almost the same as *PrefixDecl* and *GroupGraphPattern* in SPARQL syntax (see EBNF notation of SPARQL grammar [4]). The only difference is that `INDENTED_RDF_PATTERN` must have one or more white spaces at the beginning of each line. The symbols `node_pat` and `edge_pat` are described according to Definition 2 and 3. `NEWLINE` is used as an abstract symbol corresponding to the appropriate character to start a new line. Minimal examples of G2GML are shown in Section 2.5.

An RDF graph pattern in G2GML specifies the resources that should be mapped to PG components. Variables in RDF graph patterns are embedded into the corresponding location in the preceding PG patterns, which yield resultant PGs.

A node map is described as a pair of a PG node pattern and an RDF graph pattern. Here, we define the syntax of the PG node pattern as follows.

Definition 2 (EBNF notation of PG node pattern)**.**

```
node_pat  ::= "(" NODE  ":" LABEL  ("{" prop_list "}")? ")"
prop_list ::= property "," prop_list | property
property  ::= PROP_NAME ":" PROP_VAL
```

In Definition 2, `NODE`, `LABEL`, `PROP_NAME`, and `PROP_VAL` are all identifiers. A PG node pattern specifies a node, a label, and properties to be generated by the map. The `NODE` symbol specifies the ID of the generated node, which is replaced by the resource retrieved from the RDF graph. The `LABEL` symbol specifies the label of the resultant node. A label also serves as the name of the node map, which is referred from edge maps. Therefore, the identifier of each label must be unique in a G2GML description. Each node pattern contains zero or more properties in a `prop_list`. A `property` is a pair of `PROP_NAME` and `PROP_VAL` that describes the properties of nodes. The identifiers `NODE` and `PROP_VAL` must be names of variables in the succeeding RDF graph pattern, while `LABEL` and `PROP_NAME` are regarded as text literals.

In contrast, a PG edge pattern specifies the pattern of the edge itself, its label, and properties, as well as the source and destination nodes. Here, we define the syntax of the PG edge pattern as follows.

Definition 3 (EBNF notation of PG edge pattern)**.**

```
edge_pat ::= "(" SRC  ":" SRC_LAB ") -"
             "[" EDGE?  ":" EDGE_LAB  ("{" prop_list "}")? "]"
             ("->" | "-") "(" DST ":" DST_LAB ")"
```

In Definition 3, `SRC`, `SRC_LAB`, `EDGE`, `EDGE_LAB`, `DST`, and `DST_LAB` are all identifiers. Identifiers of `SRC` and `DST` specify the variables in the succeeding RDF graph pattern that should be mapped to the endpoint nodes of resultant edges. A `SRC_LAB` and a `DST_LAB` serve not only as labels of resultant nodes, but also as implicit constraints of the edge map. The resultant source and destination nodes must match patterns in the corresponding node maps (described in Section 2.6 with a concrete example). For this reason, `SRC_LAB` and `DST_LAB` must be defined in other node maps. `EDGE`, `EDGE_LAB`, and `prop_list` can be described in the same manner as their counterparts in node maps.

The resultant edges can be either directed or undirected. If '`->`' is used as a delimiter between an edge and a destination part, the edge will be directed.

In node and edge maps, a variable bound to `PROP_VAL` may have multiple candidates for a single node or edge. If multiple candidates exist, they are concatenated into an array in the same manner as GROUP_CONCAT in SPARQL. If `EDGE` is omitted in an edge map, such candidates are concatenated within groups of tuples of `SRC` and `DST`.

2.3 Property Graph Model

We define the PG model independent of specific graph database implementations. For the purpose of interoperability, we incorporate differences in PG models, taking into consideration multiple labels or property values for nodes and edges, as well as mixed graphs with both directed and undirected edges. The PG model we redefine here requires the following characteristics:

- A PG contains nodes and edges.
- Each node and edge can have zero or more labels.
- Each node and edge can have properties (key-value pairs).
- Each property can have multiple values.
- Each edge can be directed or undirected.

Formally, we define the PG model as follows.

Definition 4 (Property Graph Model).

A property graph *is a tuple* $PG = \langle N, E_d, E_u, e, l_n, l_e, p_n, p_e \rangle$, *where:*

1. N *is a set of nodes.*
2. E_d *is a set of directed edges.*
3. E_u *is a set of undirected edges.*
4. $e : E \rightarrow \langle N \times N \rangle$ *is a function that yields the endpoints of each directed or undirected edge where* $E := E_d \cup E_u$. *If the edge is directed, the first node is the source, and the second node is the destination.*
5. $l_n : N \rightarrow 2^S$ *is a function mapping each node to its multiple labels where* S *is a set of all strings.*
6. $l_e : E \rightarrow 2^S$ *is a function mapping each edge to its multiple labels.*
7. $p_n : N \rightarrow 2^P$ *is a function used to assign nodes to their multiple properties.* P *is a set of properties. Each property assumes the form* $p = \langle k, v \rangle$, *where* $k \in S$ *and* $v \in 2^V$. *Here* V *is a set of values of arbitrary data types.*
8. $p_e : E \rightarrow 2^P$ *is a function used to assign edges to their multiple properties.*

2.4 Serialization of Property Graphs

According to our definition of the PG model, we propose serialization in flat text and JSON. The flat text format (PG format) performs better in terms of human readability and line-oriented processing, while the JSON format (JSON-PG format) is best used for server–client communication.

The PG format has the following characteristics. An example is given in Figure 5.

- Each line describes a node or an edge.
- All elements in a line are separated by spaces or tabs.
- The first column of a node line contains the node ID.
- The first three columns of an edge line contain the source node ID, direction, and destination node ID.

```
# NODES
101  :Person  name:Alice  age:15  country:"United States"
102  :Person  :Student  name:Bob  country:Japan  country:Germany

# EDGES
101 -- 102  :sameSchool  :sameClass  since:2012
102 -> 101  :likes  since:2015
```

Fig. 2. Example of PG format

- Each line can contain an arbitrary number of labels.
- Each line can contain an arbitrary number of properties (key-value pairs).

More formally, we describe the PG format in the EBNF notation as follows.

Definition 5 (EBNF notation of the PG format)**.**

```
pg         ::= (node | edge)+
node       ::= NODE_ID labels properties NEWLINE
edge       ::= NODE_ID direction NODE_ID labels properties NEWLINE
labels     ::= label*
properties ::= property*
label      ::= ":" STRING
property   ::= STRING ":" value
value      ::= STRING | NUMBER
direction  ::= "--" | "->"
```

Furthermore, we implemented command-line tools to convert formats between the flat PG and JSON-PG, where the latter follows the JSON syntax in addition to the above definition. We further transform them into formats for well-known graph databases such as Neo4j, Oracle Database, and Amazon Neptune. The practical use cases of our tools demonstrate that the proposed data model and formats have the capability to describe PG data used in existing graph databases (see `https://github.com/g2glab/pg`).

2.5   Minimal Examples

Figure 4 shows the minimal example of G2G mapping from RDF to PG data, representing the following five types of typical mapping.

- Resource to node: In lines 2–4, the RDF resources with type `:Person` are mapped into the PG nodes using their IRIs as node IDs.
- Datatype property to node property: In lines 2–4, the RDF datatype property `:name` is mapped onto the PG node property key `name`. The literal objects 'Alice' and 'Bob' are mapped onto the node property values.
- Object property to edge: In lines 5–6, the RDF object property `:supervised_by` is mapped onto the PG edge `supervised_by`.
- Resource to edge: In lines 7–10, the RDF resource with type `:Email` is mapped onto the PG edge `emailed`.

```
@prefix : <http://example.org/> .
:person1 a :Person ;
        :name 'Alice' .
:person2 a :Person ;
        :name 'Bob' .
:person1 :supervised_by :person2 .
[] a :Email ;
   :sender     :person1 ;
   :receiver   :person2 ;
   :year       2017 .
   :attachment '01.pdf'
```

Fig. 3. Example of input RDF data

```
PREFIX : <http://example.org/>
(p:Person {name:n})
    ?p a :Person .
    ?p :name ?n .
(p1:Person)-[:supervised_by]->(p2:Person)
    ?p1 :supervised_by ?p2 .
(p1:person)-[:emailed {year:y, attachment:a}]->(p2:person)
    ?f a :Email
        :sender   ?p1 ;
        :receiver ?p2 ;
        :year     ?y .
    OPTIONAL { ?f :attachment ?a }
```

Fig. 4. Example of G2G mapping definition

```
"http://example.org/person1" :person name:Alice
"http://example.org/person2" :person name:Bob
"http://example.org/person1" -> "http://example.org/person2" :supervised_by
"http://example.org/person1" -> "http://example.org/person2" :emailed year:2017 attachment:"01.pdf"
```

Fig. 5. Example of output PG data

– Datatype property to edge property: In lines 7–10, the RDF datatype property :year and :attachment are mapped onto the PG edge property year and attachment. The literal objects 2017 and 01.pdf are mapped onto the edge property values.

2.6   Mapping Details

The G2G mapping based on G2GML is designed to be intuitive in general as shown above; however, there are several discussions regarding the details of mapping.

Referencing Node Labels The node labels in edge maps must be defined in node maps (described in Section 2.2) to reference the conditions. This means that the conditions for defining Person are imported into the definition of
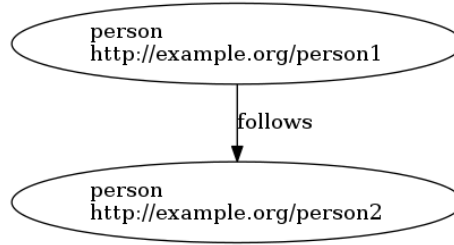
Fig. 6. Example of output PG data

```
@prefix : <http://example.org/> .
:person1 a :Person ;
         :name 'Alice' .
:person2 a :Person ;
         :name 'Bob' .
[] a :Email ;
   :sender   :person1 ;
   :receiver :person2 ;
   :year     2017 .
[] a :Email ;
   :sender   :person1 ;
   :receiver :person2 ;
   :year     2018 .
```

Fig. 7. Example of input RDF data (multi-edges)

```
"http://example.org/person1" :Person name:Alice
"http://example.org/person2" :Person name:Bob
"http://example.org/person1" -> "http://example.org/person2" :emailed year:2017
"http://example.org/person1" -> "http://example.org/person2" :emailed year:2018
```

Fig. 8. Example of output PG data (multi-edges)

supervised_by and emailed relationships. Therefore, the supervised_by relationship will be generated only between the nodes satisfying the Person condition. In this manner, users can retrieve the PG datasets that are intuitively consistent (it is also possible to avoid specifying node labels). In contrast, edge conditions are not inherited to nodes, such that the nodes are retrieved without all relationships, supervised_by and emailed. Although the nodes with no relationship also satisfy the conditions, the prototype implementation does not retrieve such orphan nodes by default from a practical perspective.

Multi-Edges When a pair of RDF resources has multiple relationships, those will be converted to multiple PG edges instead of a single PG edge with multiple properties. The G2GML in Figure 4 generates two PG edges for the RDF data in Figure 7 to maintain the information that Alice emailed Bob twice.

```
@prefix : <http://example.org/> .
:person1 a :Person ;
         :name 'Alice' .
:person2 a :Person ;
         :name 'Bob' .
[] a :Email ;
   :sender      :person1 ;
   :receiver    :person2 ;
   :year        2017 ;
   :attachment '01.pdf' ;
   :attachment '02.pdf' .
```

Fig. 9. Example of input RDF data (list of property values)

```
"http://example.org/person1" :Person name:Alice
"http://example.org/person2" :Person name:Bob
"http://example.org/person1" -> "http://example.org/person2" :emailed year:2017 attachment:"01.pdf" attachment:"02.pdf"
```

Fig. 10. Example of output PG data (list of property values)

List of Property Values Each PG property can assume multiple values. In an RDF that includes the same data property multiple times, the values are assumed to be the members of a list. The G2GML in Figure 4 generates one PG edge with two properties for the RDF data in Figure 9, keeping the information that Alice emailed Bob once, and the email contained two attachments.

3   Use Cases

We present several use cases to show how to apply the G2G mapping framework, where we utilize publicly available RDF datasets through SPARQL endpoints.

3.1   Using Wikidata

Wikidata is a useful source of open data from various domains. We present an example of mapping using a disease dataset from Wikidata, where each disease is associated with zero or more genes and drugs. Figure 11 illustrates the schematic relationships between those entities. This subset of Wikidata can be converted as a PG using the G2G mapping framework by writing a mapping file in G2GML (see Figure 12). Here, we focus on human genes and specify the necessary conditions in G2GML. Each instance of `Q12136` (disease) can have a property `P2176` (drug used for treatment), which is thereby linked to items of `Q12140` (medication). Further, each disease can have a property `P2293` (genetic association), which is thereby linked to items of `Q7186` (gene). The resultant PG includes 4696 diseases, 4496 human genes, and 1287 drugs. The total numbers of nodes and edges are summarized in Table 1. As a result of G2G mapping, the numbers of nodes and edges are reduced to almost half and one third, respectively.

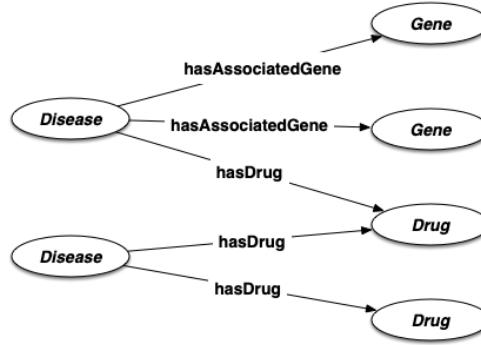**Fig. 11.** Schematic relations of Wikidata entities

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

(g:HumanGene {symbol: s})
    ?g wdt:P31 wd:Q7187 ;       # "instance of" "gene"
       wdt:P703 wd:Q15978631 ; # "found in taxon" "Homo sapiens"
       wdt:P353 ?s .            # "HGNC gene symbol"

(d:Disease {name: n})
    ?d wdt:P31 wd:Q12136 ;      # "instance of" "disease"
       rdfs:label ?l .
    FILTER(lang(?l) = "en")
    BIND(str(?l) AS ?n)

(m:Drug {name: n})
    ?m wdt:P31 wd:Q12140 ;      # "instance of" "medication"
       rdfs:label ?l .
    FILTER(lang(?l) = "en")
    BIND(str(?l) AS ?n)

(d:Disease)-[:hasAssociatedGene]->(g:HumanGene)
    ?d wdt:P2293 ?g .           # "genetic association"

(d:Disease)-[:hasDrug]->(m:Drug)
    ?d wdt:P2176 ?m .           # "drug used for treatment"
```

Fig. 12. G2GML for Wikidata mapping

3.2   Using DBpedia

Figure 13 schematically illustrates an example of the G2G mapping to convert
RDF data retrieved from DBpedia into PG data. Focusing on a relationship
where two musicians (`?m1` and `?m2`) belong to the same group, this information
can be represented in the PG shown on the right side of the figure. The relation-
ships are originally presented as independent resources `?g`. In the mapping, we
map `?g`, `?n`, and `?s` onto a PG edge labeled 'sameGroup' with attributes. Such

compaction by mapping is useful in many use cases, for example, when users are just interested in relationships between musicians.
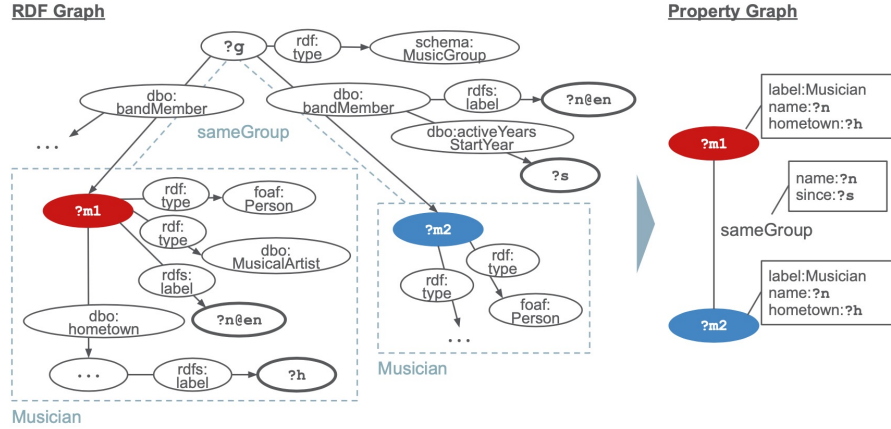

Fig. 13. Schematic example of DBpedia mapping

The mapping rules for this example are specified in the G2GML description shown in Figure 14. The G2GML description contains node mapping for `Musician` entities and edge mapping for `same_group` relationships. In each specified PG pattern, {m, n, h} and {m1, m2, n, s} are used as variables to reconstruct resources and literals extracted from RDF graphs.

Figure 15 shows queries to retrieve the pairs of musicians that are in the same group in SPARQL and graph query languages [24] [11]. The queries in Cypher and PGQL are more succinct owing to the simple structure of the PG obtained by G2G mapping.

	RDF nodes	RDF edges	PG nodes	PG edges
Wikidata disease	20,692	36,826	10,477	11,770
DBpedia musician	23,846	32,808	7,069	10,755

Table 1. Number of nodes and edges in use cases

4   Availability

The prototype implementation is written in JavaScript and can be executed using Node.js in the command line. It has an endpoint mode and a local file mode. The local file mode uses Apache Jena ARQ to execute SPARQL queries

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX schema: <http://schema.org/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

# Node mapping
(m:Musician {name:n, hometown:h})                          # PG Pattern
    ?m rdf:type foaf:Person , dbo:MusicalArtist ;          # RDF Pattern
        rdfs:label ?n . FILTER(lang(?n) = "en") .
    OPTIONAL { ?m dbo:hometown/rdfs:label ?h . FILTER(lang(?h) = "en") }

# Edge mapping
(m1:Musician)-[:sameGroup {name:n, since:s}]-(m2:Musician)  # PG Pattern
    ?g rdf:type schema:MusicGroup ;                        # RDF Pattern
        dbo:bandMember ?m1 , ?m2 . FILTER(?m1 != ?m2)
    OPTIONAL { ?g rdfs:label ?n . FILTER(lang(?n) = "en")}
    OPTIONAL { ?g dbo:activeYearsStartYear ?s }
```

Fig. 14. G2GML for DBpedia mapping

```
# SPARQL
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX schema: <http://schema.org/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?nam1 ?nam2
WHERE {
    ?mus1 rdf:type foaf:Person , dbo:MusicalArtist .
    ?mus2 rdf:type foaf:Person , dbo:MusicalArtist .
    ?mus1 rdfs:label ?nam1 . FILTER(lang(?nam1) = "ja") .
    ?mus1 rdfs:label ?nam2 . FILTER(lang(?nam2) = "ja") .
    ?grp a schema:MusicGroup ;
        dbo:bandMember ?mus1 , ?mus2 .
    FILTER(?mus1 != ?mus2)
}

# Cypher
MATCH (m1)-[:same_group]-(m2) RETURN DISTINCT m1.name, m2.name

# PGQL
SELECT DISTINCT m1.name, m2.name MATCH (m1)-[:same_group]-(m2)
```

Fig. 15. SPARQL, Cypher, and PGQL

internally, while the endpoint mode accesses SPARQL endpoints via the Internet. An example of the usage in the endpoint mode is as follows:

    $ g2g musician.g2g http://dbpedia.org/sparql

where the first argument is a G2GML description file, and the second argument is the target SPARQL endpoint, which provides the source RDF dataset.

Furthermore, a Docker image (`https://hub.docker.com/r/g2glab/g2g`) and a demonstration site of the G2G mapping framework (`http://g2g.fun`) are also available (Figure 16).

Examples:

```
mini-05 -- RDF datatype property >> PG edge property                                          ▾
```

RDF:

```
@prefix : <http://example.org/> .
:person1 a :Person .
:person2 a :Person .
[] a :Follow ;
  :follower :person1 ;
  :followed :person2 ;
  :since 2017 .
```

G2Gml:

```
PREFIX : <http://example.org/>

(p:person)
  ?p a :Person .

(p1:person)-[:follows {since:s}]->(p2:person)
  ?f :follower ?p1 ;
    :followed ?p2 ;
    :since ?s .
```

Select output format: PG   Oracle PGX   Neo4j   Graphviz   Amazon Neptune   All   Submit

Fig. 16. G2G mapping demonstration site

5   Related Work

5.1   Property Graph Model and Serialization

Recently, graph data has increasingly attracted attention, leading to a plethora of database implementations. Thus far, different data models and query languages have been available for these graph database implementations. Consequently, there have been community-based activities for standardizing the graph query language for interoperable use of PG databases [12]. Similarly, the standardization of the PG model for various database implementations enhances the interoperable use of graph data. There is indeed a demand for graph standardization in the community, which was recently discussed in a W3C workshop [13]. Another proposal for the PG model and serialization [14] that was similar to ours was presented in the workshop, and it is noteworthy that the two independent studies converged to a similar solution. They seem to be interchangeable; however, this remains to be tested. Future studies should address collaboration and standardization of the data model. In particular, our serialization has implementations for some of the major database engines and has the potential to further cover various database engines. The serialization formats that are independent of specific database implementations will increase the interoperability of graph databases and make it easier for users to import accumulated graph data.

5.2   Graph to Graph Mapping

A preceding study on converting existing data into graph data included an effort to convert relational databases into graph databases [15]. However, given that RDF has prevailed as a standardized data model in scientific communities, considering mapping based on the RDF model is crucial. The interoperability of RDF and PG [10,16,17,18] has been discussed, and efforts were made to develop methods to convert RDF into PG [19,20]. However, considering the flexibility regarding the type of information that can be expressed by edges in property graphs, a novel method for controlling the mapping is necessary. We discuss the comparison of controlled mapping and direct mapping later in this section.

   To the best of our knowledge, this study presents the first attempt to develop a framework for controlled mapping between RDF and PG. Notably, the designed G2GML is a declarative mapping language. As a merit of the declarative description, we can concentrate on the core logic of mappings. In the sense that the mapping process generates new graph data on the basis of existing graph data, it has a close relation to the semantic inference. A similar concept is found in the SPARQL CONSTRUCT queries. While the SPARQL CONSTRUCT clause defines mapping on the same data model, G2GML defines mapping between different data models. Thus, G2GML is considered as a specific extension of the SPARQL CONSTRUCT clause for generation or inference of PG data.

   A previous study compared the performance of RDF triple stores, such as Blazegraph and Virtuoso, and Neo4j [21]. They concluded that RDF has an advantage in terms of performance and interoperability. However, as PG model, they tested only Neo4j. It will be necessary to update such benchmarks using additional implementations, which requires improved interoperability and standardization of the PG model. Our study is expected to contribute to the improved interoperability of PG models, although a continuous discussion is necessary for the standardization of query language for the PG model, in order to achieve an interoperable ecosystem of both RDF and PG, and a fair benchmarking.

Direct Mapping   Other mapping frameworks, such as Neosemantics (a Neo4j plugins), propose a method to convert RDF datasets without mapping definitions (for convenience, we call such methods direct mapping). However, the following capabilities are essential in practical usage.

   1. Filtering data - RDF is designed for the web of data. When the source RDF dataset is retrieved from the public web space, it is inefficient (or even unrealistic) to convert the whole connected dataset. In G2GML, users can specify the resources they need, such that the SPARQL endpoints can return the filtered dataset. (This design is described in Section 2.2.)

   2. Mapping details - There is no common ruleset to uniquely map RDF terms to PG elements (labels, key-value pairs of properties) and to name new PG elements. Further, the conversion rules of multi-edges and lists of property values (discussed in Section 2.6) are not always obvious to users. Therefore,

defining a method for mapping details is necessary to create precise data in actual use cases.

3. Schema definition - We assume that mapping is often used for developing specific applications on top of PG datasets. In this development, the schema of the dataset should be known, while the original RDF data source could contain more information that is not covered by the schema. G2GML helps developers understand the data schema in its intuitive definition (separately for nodes and edges, and their referencing - discussed in Section 2.6), while direct mapping has the potential to generate PGs without defining a schema.

We observe a similar discussion in the conversion from the relational model to RDF, where are two W3C standards, i.e., Direct Mapping [22] and R2RML [23].

6   Conclusion

We designed the G2GML for mapping RDF graphs to PGs and developed its prototype implementation. To ensure a clear definition of this mapping, we defined the PG model independent of specific graph database implementations and proposed its exchangeable serialization formats.

The advantage of using RDF is that different applications can retrieve necessary information from the same integrated datasets owing to the interoperable nature of semantic modeling.

For such increasing RDF resources, graph databases are potentially the ideal data storage as the property graph model can naturally preserve the relationship information semantically defined in RDF. G2GML therefore plays an important role in this data transformation process.

Various graph database implementations are actively developed and standardization of query languages is currently ongoing. We expect that the G2GML or its enhanced mapping framework will be supported by database management systems and other software. We believe that our efforts of generalization and prototype implementation will promote further discussion.

Acknowledgements

References

1. RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. `http://www.w3.org/TR/rdf11-concepts/`
2. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., ... Bizer, C. (2015). DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia. Semantic Web, 6(2), 167-195.

3. Vrandečić, D., Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. Communications of the ACM, 57(10), 78-85.
4. SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013. `http://www.w3.org/TR/sparql11-query/`
5. Angles, R., Gutierrez, C.: An introduction to Graph Data Management. arXiv preprint arXiv:1801.00036 (2017)
6. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoc, D.: Foundations of Modern Query Languages for Graph Databases. ACM Computing Surveys (CSUR), 50(5), 68. (2017)
7. The Neo4j Graph Platform. `https://neo4j.com/`
8. Oracle Database Property Graph. `https://www.oracle.com/goto/propertygraph`
9. Amazon Neptune. `https://aws.amazon.com/neptune/`
10. Hartig, O.: Reconciliation of RDF* and property graphs. arXiv preprint arXiv:1409.3288 (2014)
11. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, p. 7. ACM (2016)
12. Angles, R., Arenas, M., Barceló, P., Boncz, P., Fletcher, G., Gutierrez, C., *et al.*: G-CORE: A core for future graph query languages. In Proceedings of the 2018 International Conference on Management of Data, pp. 1421–1432. (2018)
13. W3C Workshop on Web Standardization for Graph Data. `https://www.w3.org/Data/events/data-ws-2019/`
14. Tomaszuk D., Angles R., Szeremeta L., Litman K., Cisterna D.: Serialization for property graphs. International Conference: Beyond Databases, Architectures and Structures, pp. 57–69. (2019)
15. De Virgilio, R., Maccioni, A., Torlone, R.: Converting relational to graph databases. In First International Workshop on Graph Data Management Experiences and Systems, p. 1. ACM (2013)
16. Angles, R., Thakkar, H., Tomaszuk, D.: RDF and Property Graphs Interoperability: Status and Issues. Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management. (2019)
17. Das, S., Srinivasan, J., Perry, M., Chong, E. I., Banerjee, J.: A Tale of Two Graphs: Property Graphs as RDF in Oracle. In EDBT, pp. 762–773. (2014)
18. Thakkar, H., Punjani, D., Keswani, Y., Lehmann, J., Auer, S.: A Stitch in Time Saves Nine–SPARQL querying of Property Graphs using Gremlin Traversals. arXiv preprint arXiv:1801.02911 (2018)
19. Tomaszuk, D.: RDF data in property graph model. In Research Conference on Metadata and Semantics Research, pp. 104–115. (2016)
20. De Virgilio, R.: Smart RDF data storage in graph databases. In 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 872–881. IEEE (2017)
21. Alocci, D., Mariethoz, J., Horlacher, O., Bolleman, J. T., Campbell, M. P., Lisacek, F. (2015). Property graph vs RDF triple store: A comparison on glycan substructure search. PloS one, 10(12), e0144578.
22. A Direct Mapping of Relational Data to RDF, W3C Recommendation 27 September 2012 `https://www.w3.org/TR/r2rml/`
23. R2RML: RDB to RDF Mapping Language, W3C Recommendation 27 September 2012 `https://www.w3.org/TR/r2rml/`
24. openCypher, `https://www.opencypher.org/`