IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 30, 2020, accepted April 26, 2020, date of publication May 7, 2020, date of current version May 20, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2993117

Mapping RDF Databases to Property Graph Databases

RENZO ANGLES[1], HARSH THAKKAR[2,4], AND DOMINIK TOMASZUK[3]

[1]Department of Computer Science, Faculty of Engineering, Universidad de Talca, Curicó 3340000, Chile
[2]OSTHUS GmbH, 52068 Aachen, Germany
[3]Institute of Informatics, University of Białystok, 15-245 Białystok, Poland
[4]Informatik III Department, University of Bonn, 53115 Bonn, Germany

Corresponding author: Renzo Angles (renzoangles@gmail.com)

The work of Renzo Angles was supported by the Millennium Institute for Foundational Research on Data (Chile). The work of Harsh Thakkar was supported in part by OSTHUS GmbH. The work of Dominik Tomaszuk was supported by the National Science Center, Poland (NCN) under Grant Miniatura 2.

ABSTRACT RDF triplestores and property graph databases are two approaches for data management which are based on modeling, storing and querying graph-like data. In spite of such common principle, they present special features that complicate the task of database interoperability. While there exist some methods to transform RDF graphs into property graphs, and vice versa, they lack compatibility and a solid formal foundation. This paper presents three direct mappings (schema-dependent and schema-independent) for transforming an RDF database into a property graph database, including data and schema. We show that two of the proposed mappings satisfy the properties of semantics preservation and information preservation. The existence of both mappings allows us to conclude that the property graph data model subsumes the information capacity of the RDF data model.

INDEX TERMS Database interoperability, direct mapping, RDF, property graph.

I. INTRODUCTION

The database systems based on graph-oriented models are gaining relevance in the industry due to their use in various application domains where complex data analytics is required [4]. RDF triple stores and graph database systems are two approaches for data management that are based on modeling, storing, and querying graph-like data.

RDF triplestores are based on the RDF data model [24], [41], their standard query language is SPARQL [19], and RDF Schema [11] allows to describe classes of resources and properties (i.e. the data schema). On the other hand, most graph database systems are based on the Property Graph (PG) data model [2], and a standard query language is in current development [23].

Although the RDF model and the PG model are based on a graph-oriented structure, they have particularities that complicate the task of data interoperability between them. Among the most important differences we can mention: a PG allows properties for nodes and edges, i.e. nodes and edges could have a set of name-value pairs which are used to introduce metadata; the RDF model use elements with special syntax and semantics (e.g. IRIs, blank nodes, literals, namespaces, reification, collections).

The associate editor coordinating the review of this manuscript and approving it for publication was Wajahat Ali Khan.

A. MOTIVATION

Considering the intrinsic connection between RDF triple stores and PG databases, and their popularity for representing knowledge databases, it becomes necessary to develop methods to allow interoperability among these types of systems.

The term ''Interoperability'' was introduced in the area of information systems, and is defined as the ability of two or more systems or components to exchange information, and to use the information that has been exchanged [35]. In the context of data management, interoperability is concerned with the support of applications that share and exchange information across the boundaries of existing databases [32].

Database interoperability is relevant for several reasons: promotes data exchange and data integration [30]; facilitates the reuse of available systems and tools [26], [32]; enables a fair comparison of database systems by using benchmarks [3], [37]; and supports the success of emergent systems and technologies [32].

B. THE PROBLEM

To the best of our knowledge, the research about the interoperability between RDF and PG databases is very restricted (cf. Section VII). While there exist some system-specific approaches, most of them are restricted to data transformation and lack of solid formal foundations.

C. OBJECTIVES & CONTRIBUTIONS

Database interoperability can be divided into syntactic interoperability (i.e. data transformations), semantic interoperability (i.e. data exchange via schema and instance mappings), and query interoperability (i.e. transformations among different query languages or data accessing methods) [5].

The main objective of this paper is to study the semantic interoperability between RDF and PG databases. Specifically, we propose three database mappings: a simple mapping which allows transforming an RDF graph into a PG without considering schema restrictions (in both sides); a generic mapping which allows transforming an RDF graph (without RDF schema) into a PG that follows the restrictions defined by a generic PG schema; and, a complete mapping which allows transforming a complete RDF database into a complete PG database (i.e. schema and instance).

We study three desirable properties of the above database mappings: computability, semantics preservation, and information preservation. Based on such analysis, we argued that any RDF database can be transformed into a PG database. In terms of data modeling, we conclude that the PG data model subsumes the information capacity of the RDF data model.

The remainder of this paper is as follows: a formal background is presented in Section IV; the simple database mapping is described in Section III; the generic database mapping is described in Section IV; the complete database mapping is described in Section V; the experimental evaluation of the mappings is presented in Section VI; the related work is discussed in Section VII; our conclusions are presented in Section VIII.

II. PRELIMINARIES

This section presents a formal background required to study the interoperability between RDF and PG databases. In particular, we formalize the notions of database mapping, RDF database, and PG database.

A. DATABASE MAPPINGS

In general terms, a database mapping is a method to translate databases from a source database model to a target database model. We can consider two types of database mappings: *direct database mappings*, which allow an automatic translation of databases without any input from the user [34]; and *manual database mappings*, which require additional information (e.g. an ontology) to conduct the database translation. In this paper, we focus on direct database mappings.

1) DATABASE SCHEMA AND INSTANCE

Let M be a database model. A *database schema* in M is a set of semantic constraints allowed by M. A *database instance* in M is a collection of data represented according to M. A *database* in M is an ordered pair $D^M = (S^M, I^M)$, where S^M is a schema and I^M is an instance.

If a database D^M does not define its schema, i.e. $D^M = (\emptyset, I^M)$, we will say that D^M is a schema-less database. If D^M does not define its instance, i.e. $(S^M, \emptyset)$, then D^M will be called an empty database. If D^M defines both, schema and instance, then we have a complete database.

Note that the above definition does not establish that the database instance satisfies the constraints defined by the database schema. Given a database instance I^M and a database schema S^M, we say that I^M is valid with respect to S^M, denoted $I^M \models S^M$, iff I^M satisfies the constraints defined by S^M. Given a database $D^M = (S^M, I^M)$, we say that D^M is a valid database iff it satisfies that $I^M \models S^M$.

2) SCHEMA, INSTANCE, AND DATABASE MAPPING

A database mapping defines a way to translate databases from a "target" database model to a "source" database model. For the rest of this section, assume that M_1 and M_2 are the source and the target database models respectively.

Considering that a database includes a schema and an instance, we first define the notions of schema mapping and instance mapping. A *schema mapping* from M_1 to M_2 is a function $\mathcal{SM}$ from the set of all database schemas in M_1, to the set of all database schemas in M_2. Similarly, an *instance mapping* from M_1 to M_2 is a function $\mathcal{IM}$ from the set of all database instances in M_1, to the set of all database instances in M_2.

A database mapping from M_1 to M_2 is a function $\mathcal{DM}$ from the set of all databases in M_1, to the set of all databases in M_2. Specifically, a database mapping is defined as the combination of a schema mapping and an instance mapping.

Definition 1 (Database Mapping): A database mapping is a pair $\mathcal{DM} = (\mathcal{SM}, \mathcal{IM})$ where $\mathcal{SM}$ is a schema mapping and $\mathcal{IM}$ is an instance mapping.

a: PROPERTIES OF DATABASE MAPPINGS

Every data model allows to structure the data in a specific way, or using a particular abstraction. Such abstraction determines the conceptual elements that the data model can represent, i.e. its representation power or information capacity [22].

Given two database models M_1 and M_2, the possibility to exchange databases between them depends on their information capacity. Specifically, we say that M_1 subsumes the information capacity of M_2 iff every database in M_2 can be translated to a database in M_2. Additionally, we say that M_1 and M_2 have the same information capacity iff M_1 subsumes M_2 and M_2 subsumes M_1.

The information capacity of two database models can be evaluated in terms of a database mapping satisfying

some properties. In particular, we consider three properties: computability, semantics preservation, and information preservation.

Assume that $\mathcal{D}_1$ is the set of all databases in a source database model M_1, and $\mathcal{D}_2$ is the set of all databases in a target database model M_2.

Definition 2 (Computable mapping): A database mapping $\mathcal{DM} : \mathcal{D}_1 \rightarrow \mathcal{D}_2$ is computable if there exists an algorithm $\mathcal{A}$ that, given a database $D \in \mathcal{D}_1$, $\mathcal{A}$ computes $\mathcal{DM}(D)$.

The property of computability indicates the existence and feasibility of implementing a database mapping from M_1 to M_2. This property also implicates that M_2 subsumes the information capacity of M_1.

Definition 3 (Semantics Preservation): A computable database mapping $\mathcal{DM} : \mathcal{D}_1 \rightarrow \mathcal{D}_2$ is semantics preserving if for every valid database $D \in \mathcal{D}_1$, there is a valid database $D' \in \mathcal{D}_2$ satisfying that $D' = \mathcal{DM}(D)$.

Semantics preservation indicates that the output of a database mapping is always a valid database. Specifically, the output database instance satisfies the constraints defined by the output database schema. In this sense, we can say that this property evaluates the correctness of a database mapping.

Definition 4 (Information preservation): A database mapping $\mathcal{DM} = (\mathcal{SM}, \mathcal{IM})$ from M_1 to M_2 is information preserving if there is a computable database mapping $\mathcal{DM}^{-1} = (\mathcal{SM}^{-1}, \mathcal{IM}^{-1})$ from M_2 to M_1 such that for every database $D = (S, I)$ in M_1, it applies that $D = \mathcal{DM}^{-1}(\mathcal{DM}(D))$.

Information preservation indicates that, for some database mapping $\mathcal{DM}_3$, there exists an "inverse" database mapping $\mathcal{DM}^{-1}$ which allows recovering a database transformed with $\mathcal{DM}_3$. Note that the above definition implies the existence of both a "inverse" schema mapping $\mathcal{SM}^{-1}$ and a "inverse" instance mapping $\mathcal{IM}^{-1}$.

Information preservation is a fundamental property because it guarantees that a database mapping does not lose information [34]. Moreover, it implies that the information capacity of the target database model subsumes the information capacity of the source database model.

Our goal is to define database mappings between the RDF data model and the PG data model. Hence, next, we will present a formal definition of the notions of instance, schema, and database for them.

B. RDF DATABASES

An RDF database is an approach for data management which is oriented to describe the information about Web resources by using Web models and languages. In this section we describe two fundamental standards used by RDF databases: the Resource Description Framework (RDF) [24], which is the standard data model to describe the data; and RDF Schema [11], which is a standard vocabulary to describe the structure of the data.

1) RDF GRAPH

Assume that I, B, and L are three disjoint infinite sets, corresponding to IRIs, blank nodes and literals respectively. An IRI identifies a concrete web resource, a blank node identifies an anonymous resource, and a literal is a basic value (e.g. a string, a number or a date). We will use the term RDF resource to indicate any element in the set $I \cup B$.

An RDF triple is a tuple $t = (v_1, v_2, v_3)$ where $v_1 \in I \cup B$ is called the *subject*, $v_2 \in I$ is called the *predicate* and $v_3 \in I \cup B \cup L$ is called the *object*. Here, the subject represents a resource, the predicate represents a relationship of the resource, and the object represents the value of such relationship. Given a set of RDF triples S, we will use sub(S), pred(S) and obj(S) to denote the sets of subjects, predicates, and objects in S respectively.

There are different data formats to encode a set of RDF triples, including Notation3 (N3) [12], RDF/XML [17], N-Triples [14], Turtle [8] and N-Quads [13]. The following example shows a set of RDF triples encoded using the Turtle data format.

Example 2.1:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix voc: <http://www.example.org/voc/> .
@prefix ex: <http://www.example.org/data/> .
ex:Tesla_Inc rdf:type voc:Organisation .
ex:Tesla_Inc voc:name "Tesla, Inc." .
ex:Tesla_Inc voc:creation "2003-07-01"^^xsd:date .
ex:Tesla_Inc voc:ceo ex:Elon_Musk .
ex:Tesla_Inc voc:location _:b1 .
ex:Elon_Musk rdf:type voc:Person .
ex:Elon_Musk voc:birthName "Elon Musk" .
ex:Elon_Musk voc:age "46"^^xsd:int .
_:b1~rdf:type voc:City .
_:b1~voc:name "Palo Alto" .
_:b1~voc:country _:b2 .
_:b2~rdf:type voc:Country .
_:b2~voc:name "US" .
_:b2~voc:is_location_of ex:Tesla_Inc .
```

The lines beginning with `@prefix` are prefix definitions and the rest are RDF triples. A prefix definition associates a prefix (e.g. `voc`) with an IRI (e.g. `http://www.example.org/voc/`). Hence, a full IRI like `http://www.example.org/voc/Person` can be abbreviated as a prefixed name `voc:Person`. We will use prefix(r) and name(r) to extract the prefix and the name of an IRI r respectively.

In order to facilitate readability, we will use prefixed names instead of full URIs. Moreover, we will assume that there exists a standard way to transform a full URI into a prefixed name, and vice versa (e.g. by using an internal index or an external service like DRPD [42]).

A blank node is usually represented as `_:` followed by a blank node label which is a series of name characters (e.g. `_:b1`). There are other ways to encode blank nodes (e.g. `[]`), but we will use the above for simplicity. Given a blank node b, the function lab(b) returns the label of b.

We will consider two types of literals: a simple literal which is a Unicode string (e.g. `"Elon Musk"`), and a typed literal which consists of a string and a datatype IRI (e.g. `"46"^^xsd:int`). Numbers can be unquoted and

boolean values may be written as either `true` or `false`. Given a literal l, the function val(l) returns the string of l.

The example shows the six types of valid RDF triples: (*iri*, *iri*, *iri*) in line 5, (*iri*, *iri*, *literal*) in line 6, (*iri*, *iri*, *bnode*) in line 9, (*bnode*, *iri*, *iri*) in line 13, (*bnode*, *iri*, *literal*) in line 14, and (*bnode*, *iri*, *bnode*) in line 15. Any other combination is considered invalid.

A set of RDF triples can be visualized as a graph where the nodes represent the resources, and the edges represent properties and values. However, the RDF model has a particular feature: an IRI can be used as an object and predicate in an RDF graph. For instance, the triple (`voc:ceo`, `rdfs:label`, `"Chief Executive Officer"`) can be added to the graph shown in Example 2.1 to include metadata about the property `voc:ceo`. It implies that an RDF graph is not a traditional graph because it allows edges between edges, and consequently an RDF graph cannot be visualized in a traditional way. Next, we introduce a formal definition of the RDF data model which is able to support the above feature.

Definition 5 (RDF Graph): An RDF graph is defined as a tuple $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ where:

- N_R is a finite set of nodes representing RDF resources (i.e. resource nodes divided in IRI nodes and blank nodes);
- N_L is a finite set of nodes representing RDF literals (i.e. literal nodes), satisfying that $N_R \cap N_L = \emptyset$;
- E_O is a finite set of edges called object property edges;
- E_D is a finite set of edges called datatype property edges,[1] satisfying that $E_O \cap E_D = \emptyset$.;
- $\alpha_R : N_R \rightarrow I \cup B$ is a total one-to-one function that associates each resource node with a resource identifier (i.e. either a IRI or a blank node identifier);
- $\alpha_L : N_L \rightarrow L$ is a total one-to-one function that associates each literal node with a single literal;
- $\beta_O : E_O \rightarrow (N_R \times N_R)$ is a total function that associates each object property edge with a pair of resource nodes;
- $\beta_D : E_D \rightarrow (N_R \times N_L)$ is a total function that associates each datatype property edge with a resource node and a literal node;
- δ: $(N_R \cup N_L \cup E_O \cup E_D) \rightarrow I$ is a partial function that assigns a resource class label to each node or edge.

Note that the function δ has been defined as being partial in order to support a partial connection between schema and data (which is usual in real RDF datasets). However, it is possible to define the following simple procedure to make the function δ total: For each resource $r \in N_R$, if $r \notin \text{dom}(\delta)$ then assign $\delta(r) =$ `rdfs:Resource`. Therefore, we will assume that every resource in an RDF graph defines its resource class.

Concerning the issue about an IRI u occurring as both resource and property, note that u will occur as resource and property separately. In such a case, we will have a bipartite graph. The same applies for blank nodes.

[1]The terms "object property" and "datatype property" have been taken from the Web Ontology Language (OWL) [18]

Given a set of RDF triples S, the procedure to create a formal RDF graph $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ from S is defined as follow:

- For every resource $r \in \text{sub}(S)$, there is a node $n \in N_R$ with $\alpha_R(n) = r$;
  - If $(r,$ `rdf:type`$, c) \in S$ then $\delta(n) = c$, else $\delta(n) =$ `rdfs:Resource`;
- For every literal $l \in \text{obj}(S) \cap L$, there is a node $n \in N_L$;
  - If l is a simple literal then $\alpha_L(n) = l$ and $\delta(n) =$ `xsd:string`;
  - If l is a typed literal of the form `value^^datatype` then $\alpha_L(n) =$ `value` and $\delta(n) =$ `datatype`;
- For every triple $(s, p, o) \in S$ where $o \in I \cup B$, there is an edge $e \in E_O$ with $\delta(e) = p$ and $\beta_O(e) = (n, n')$, such that $\alpha_R(n) = s$ and $\alpha_R(n') = o$;
- For every triple $(s, p, o) \in S$ where $o \in L$, there is an edge $e \in E_D$ with $\delta(e) = p$ and $\beta_D(e) = (n, n')$, such that $\alpha_R(n) = s$ and $\alpha_L(n') = o$.

For example, the RDF graph obtained from the set of RDF triples shown in Example 2.1 is defined as follows:

1 $N_R = \{n_1, n_2, n_3, n_4\}$,
2 $N_L = \{n_5, n_6, n_7, n_8, n_9, n_{10}\}$,
3 $E_O = \{e_1, e_2, e_3, e_4\}$,
4 $E_D = \{e_5, e_6, e_7, e_8, e_9, e_{10}\}$,
5 $\alpha_R(n_1) =$ `ex:Tesla_Inc`, $\alpha_R(n_2) =$ `ex:Elon_Musk`,
$\alpha_R(n_3) =$ `_:b1`, $\alpha_R(n_4) =$ `_:b2`,
6 $\alpha_L(n_5) =$ `"Tesla, Inc."`, $\alpha_L(n_6) =$ `"2003-07-01"`,
$\alpha_L(n_7) =$ `"Elon Musk"`, $\alpha_L(n_8) =$ `"46"`,
$\alpha_L(n_9) =$ `"Palo Alto"`, $\alpha_L(n_{10}) =$ `"US"`,
7 $\beta_O(e_1) = (n_1, n_2)$, $\beta_O(e_2) = (n_1, n_3)$, $\beta_O(e_3) = (n_3, n_4)$,
$\beta_O(e_4) = (n_4, n_1)$,
8 $\beta_D(e_5) = (n_1, n_5)$, $\beta_D(e_6) = (n_1, n_6)$, $\beta_D(e_7) = (n_2, n_7)$,
$\beta_D(e_8) = (n_2, n_8)$, $\beta_D(e_9) = (n_3, n_9)$, $\beta_D(e_{10}) = (n_4, n_{10})$,
9 $\delta(n_1) =$ `voc:Organisation`, $\delta(n_2) =$ `voc:Person`,
10 $\delta(n_3) =$ `voc:City`, $\delta(n_4) =$ `voc:Country`
11 $\delta(n_5) =$ `xsd:string`, $\delta(n_6) =$ `xsd:date`,
12 $\delta(n_7) =$ `xsd:string`, $\delta(n_8) =$ `xsd:int`,
13 $\delta(n_9) =$ `xsd:string`, $\delta(n_{10}) =$ `xsd:string`,
14 $\delta(e_1) =$ `voc:ceo`, $\delta(e_2) =$ `voc:location`,
$\delta(e_3) =$ `voc:country`, $\delta(e_4) =$ `voc:is_location_of`,
$\delta(e_5) =$ `voc:name`, $\delta(e_6) =$ `voc:creation`,
$\delta(e_7) =$ `voc:birthName`, $\delta(e_8) =$ `voc:age`,
$\delta(e_9) =$ `voc:name`, $\delta(e_{10}) =$ `voc:name`.

Additionally, Figure 1 shows a graphical representation of the RDF graph described above. The IRI nodes are represented as ellipses, the blank nodes are represented as dotted ellipses and literal nodes are presented as rectangles. Each node is labeled with two IRIs: the inner IRI indicates the resource identifier, and the outer IRI indicates the resource class of the node. Each edge is labeled with an IRI that indicates its property class. We use balloons to indicate the object identifiers.

2) RDF GRAPH SCHEMA

RDF Schema (RDFS) [11] defines a standard vocabulary (i.e., a set of terms, each having a well-defined meaning) which enables the description of resource classes and property classes. From a database perspective, RDF Schema can be used to describe the structure of the data in an RDF database.

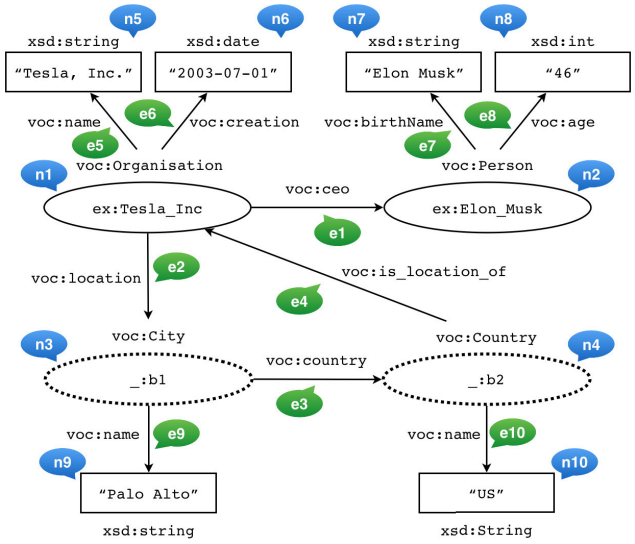

FIGURE 1. Graphical illustration of an RDF graph.

In order to describe classes of resources and properties, the RDF Schema vocabulary defines the following terms: `rdfs:Class` and `rdf:Property` represent the classes of resources, and properties respectively; `rdf:type` can be used (as property) to state that a resource is an instance of a class; `rdfs:domain` and `rdfs:range` allow to define domain resource classes and range domain classes for a property, respectively. Note that `rdf:` and `rdfs:` are the prefixes for RDF and RDFS respectively.

An RDF Schema description consists into a set of RDF triples, so it can be encoded using RDF data formats. The following example shows an RDF Schema document which describes the structure of the data shown in Example 2.1, using the Turtle data format.

Example 2.2:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix voc: <http://www.example.org/voc/> .
voc:Organisation rdf:type rdfs:Class .
voc:Person rdf:type rdfs:Class .
voc:City rdf:type rdfs:Class .
voc:Country rdf:type rdfs:Class .
xsd:string rdf:type rdfs:Class .
xsd:date rdf:type rdfs:Class .
xsd:int rdf:type rdfs:Class .
voc:ceo rdf:type rdf:Property .
voc:ceo rdfs:domain voc:Organisation .
voc:ceo rdfs:range voc:Person .
voc:location rdfs:domain voc:Organisation .
voc:location rdfs:range voc:City .
voc:country rdf:type rdf:Property .
voc:country rdfs:domain voc:City .
voc:country rdfs:range voc:Country .
voc:is_location_of rdf:type rdf:Property .
voc:is_location_of rdfs:domain voc:City .
voc:is_location_of rdfs:range voc:Organisation .
voc:name rdf:type rdf:Property .
voc:name rdfs:domain voc:Organisation .
voc:name rdfs:domain voc:City .
voc:name rdfs:domain voc:Country .
voc:name rdfs:range xsd:string .
voc:creation rdf:type rdf:Property .
voc:creation rdfs:domain voc:Organisation .
voc:creation rdfs:range xsd:date .
voc:location rdf:type rdf:Property .
voc:birthName rdf:type rdf:Property .
voc:birthName rdfs:domain voc:Person .
voc:birthName rdfs:range xsd:string .
voc:age rdf:type rdf:Property .
voc:age rdfs:domain voc:Person .
voc:age rdfs:range xsd:int .
```

Note that: a resource class rc is defined by a triple of the form (rc `rdf:type rdfs:Class`); a property class pc is defined by a triple of the form (pc `rdf:type rdf:Property`); a triple (pc `rdfs:domain` rc_1) indicates that the resource class rc_1 is part of the domain of pc (i.e. a resource of class rc_1 could have an outgoing property pc); a triple (pc `rdfs:range` rc_2) indicates that the resource class rc_2 is part of the range of pc (i.e. a resource of class rc_1 could have an incoming property pc).

If the range of a property class pc is a resource class (defined by the user), then pc is called an object property (e.g. `voc:ceo`). If the range is a datatype class, defined by RDF Schema or another vocabulary, then pc is called a datatype property (e.g. `age`). The IRIs `xsd:string`, `xsd:integer` and `xsd:dateTime` are examples of datatypes defined by XML Schema [9]. Let $I_{DT} \subset I$ be the set of RDF datatypes.

Note that the RDF schema presented in Example 2.2 provides a complete description of resource classes and property classes. However, in practice, it is possible to find incomplete or partial RDF schema descriptions. In particular, a datatype could not be defined as a resource class, and property could not define its domain or its range.

We will assume that a partial schema can be "normalized" to be a total schema. In this sense, we will use the term `rdfs:Resource`[2] to complete the definition of properties without domain or range. For instance, suppose that our sample RDF Schema does not define the range of the property class `voc:ceo`. In such case, we will include the triple (`voc:ceo`, `rdfs:range`, `rdfs:Resource`) to complete the definition of `voc:ceo`.

Now, we introduce the notion of RDF graph schema as a formal way to represent an RDF schema description. Assume that $I_V \subset I$ is the set that includes the RDF Schema terms `rdf:type`, `rdfs:Class`, `rdfs:Property`, `rdfs:domain` and `rdfs:range`.

Definition 6 (RDF Graph Schema): An RDF graph schema is defined as a tuple $S^R = (N_S, E_S, \phi, \varphi)$ where:

- N_S is a finite set of nodes representing resource classes;
- E_S is a finite set of edges representing property classes;
- $\phi : (N_S \cup E_S) \rightarrow I \setminus I_V$ is a total function that associates each node or edge with an IRI representing a class identifier;

[2]According to the RDF Schema specification [11], `rdfs:Resource` denotes the class of everything.

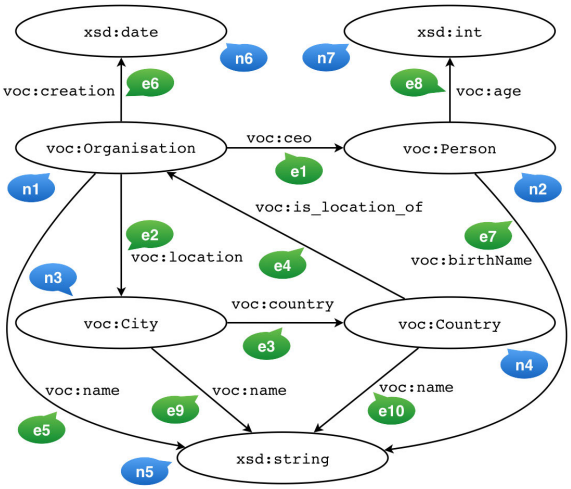

FIGURE 2. Graphical illustration of an RDF graph schema.

- φ: $E_S \rightarrow (N_S \times N_S)$ is a total function that associates each property class with a pair of resource classes.

Recall that I_{DT} denotes the set of RDF datatypes. Given an RDF Schema description D, the procedure to create an RDF graph schema $S^R = (N_S, E_S, \phi, \varphi)$ from D is given as follows:

1) Let $C = \{rc \mid (rc, \texttt{rdf:type}, \texttt{rdfs:Class}) \in D \vee (pc, \texttt{rdfs:domain}, rc) \in D \vee (pc, \texttt{rdfs:range}, rc) \in D\}$
2) For each $rc \in C$, we create $n \in N_S$ with $\phi(n) = rc$
3) For each pair of triples $(pc, \texttt{rdfs:domain}, rc_1)$ and $(pc, \texttt{rdfs:range}, rc_2)$ in D, we create $e \in E_S$ with $\phi(e) = pc$ and $\varphi(e) = (n_1, n_2)$, satisfying that $n_1, n_2 \in N_S$, $\phi(n_1) = rc_1$ and $\phi(n_2) = rc_2$.

Following the above procedure, the RDF schema shown in Example 2.2 can be formally described as follows:

$N_S = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$,
$E_S = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$;
$\phi(n_1) = \{\texttt{voc:Organisation}\}$, $\phi(n_2) = \{\texttt{voc:Person}\}$,
$\phi(n_3) = \{\texttt{voc:City}\}$, $\phi(n_4) = \{\texttt{voc:Country}\}$,
$\phi(n_5) = \{\texttt{xsd:string}\}$, $\phi(n_6) = \{\texttt{xsd:date}\}$,
$\phi(n_7) = \{\texttt{xsd:int}\}$
$\phi(e_1) = \{\texttt{voc:ceo}\}$, $\phi(e_2) = \{\texttt{voc:location}\}$,
$\phi(e_3) = \{\texttt{voc:country}\}$, $\phi(e_4) = \{\texttt{voc:is\_location\_of}\}$
, $\phi(e_5) = \{\texttt{voc:name}\}$, $\phi(e_6) = \{\texttt{voc:creation}\}$,
$\phi(e_7) = \{\texttt{voc:birthName}\}$, $\phi(e_8) = \{\texttt{voc:age}\}$,
$\phi(e_9) = \{\texttt{voc:name}\}$, $\phi(e_{10}) = \{\texttt{voc:name}\}$,
$\varphi(e_1) = (n_1, n_2)$, $\varphi(e_2) = (n_1, n_3)$, $\varphi(e_3) = (n_3, n_4)$,
$\varphi(e_4) = (n_4, n_1)$, $\varphi(e_5) = (n_1, n_5)$, $\varphi(e_6) = (n_1, n_6)$,
$\varphi(e_7) = (n_2, n_5)$, $\varphi(e_8) = (n_2, n_7)$, $\varphi(e_9) = (n_3, n_5)$,
$\varphi(e_{10}) = (n_4, n_5)$.

Additionally, Figure 2 shows a graphical representation of the RDF schema graph described above.

Given an RDF graph schema $S^R = (N_S, E_S, \phi, \varphi)$ and an RDF graph $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$, we say that G^R is valid with respect to S^R, denoted as $G^R \models S^R$, iff:

1) for each $r \in N_R \cup N_L$, it applies that there is $rc \in N_S$ where $\delta(r) = \phi(rc)$;

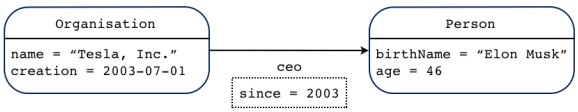

FIGURE 3. Graphical illustration of a property graph.

2) for each $e \in E_O$ with $\beta_O(e) = (n, n')$, it applies that there is $pc \in E_S$ where $\delta(e) = \phi(pc)$, $\varphi(pc) = (rc, rc')$, $\delta(n) = \phi(rc)$ and $\delta(n') = \phi(rc')$.
3) for each $e \in E_D$ with $\beta_D(e) = (n, n')$, it applies that there is $pc \in E_S$ where $\delta(e) = \phi(pc)$, $\varphi(pc) = (rc, rc')$, $\delta(n) = \phi(rc)$ and $\delta(n') = \phi(rc')$.

Here, condition (1) validates that every resource node is labeled with a resource class defined by the schema; condition (2) verifies that each object property edge, and the pairs of resource nodes that it connects, are labeled with the corresponding resource classes; and condition (3) verifies that each datatype property edge, and the pairs of nodes that it connects (i.e. a resource node and a literal node), are labeled with the corresponding resource classes

Finally, we present the notion of RDF database.

Definition 7 (RDF Database): An RDF database is a pair (S^R, G^R) where S^R is an RDF graph schema and G^R is an RDF graph satisfying that $G^R \models S^R$.

C. PROPERTY GRAPH DATABASES

A Property Graph (PG) is a labeled directed multigraph whose main characteristic is that nodes and edges can contain a set (possibly empty) of *name-value* pairs referred to as *properties*. From the point of view of data modeling, each node represents an entity, each edge represents a relationship (between two entities), and each property represents a specific characteristic (of an entity or a relationship).

Figure 3 presents a graphical representation of a PG. The circles represent nodes, the arrows represent edges, and the boxes contain the properties for nodes and edges.

Currently, there are no standard definitions for the notions of PG and PG Schema. However, we present formal definitions that resemble most of the features provided by current PG database systems.

1) PROPERTY GRAPH

Assume that $\mathbb{L}$ is an infinite set of labels (for nodes, edges and properties), $\mathbb{V}$ is an infinite set of (atomic or complex) values, and $\mathbb{T}$ is a finite set of data types (e.g. *string*, *integer*, *date*, etc.). A value in $\mathbb{V}$ will be distinguished as a quoted string. Given a value $v \in \mathbb{V}$, the function type(v) returns the datatype of v. Given a set S, $\mathcal{P}^+(S)$ denotes the set of non-empty subsets of S.

Definition 8 (Property Graph): A Property Graph is defined as a tuple $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ where:

- $\mathbb{N}$ is a finite set of nodes, $\mathbb{E}$ is a finite set of edges, $\mathbb{P}$ is a finite set of properties, and $\mathbb{N}$, $\mathbb{E}$, $\mathbb{P}$ are mutually disjoint sets;

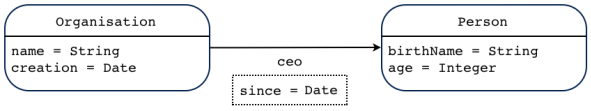

FIGURE 4. Graphical illustration of a property graph schema.

- $\Gamma : (\mathbb{N} \cup \mathbb{E}) \rightarrow \mathbb{L}$ is a total function that associates each node or edge with a label;
- $\Upsilon : \mathbb{P} \rightarrow (\mathbb{L} \times \mathbb{V})$ is a total function that assigns a label-value pair to each property.
- $\Sigma : \mathbb{E} \rightarrow (\mathbb{N} \times \mathbb{N})$ is a total function that associates each edge with a pair of nodes;
- $\Delta : (\mathbb{N} \cup \mathbb{E}) \rightarrow \mathcal{P}^+(\mathbb{P})$ is a partial function that associates a node or edge with a non-empty set of properties, satisfying that $\Delta(o_1) \cap \Delta(o_2) = \emptyset$ for each pair of objects $o_1, o_2 \in \text{dom}(\Delta)$;

The above definition supports PGs with the following features: a pair of nodes can have zero or more edges; each node or edge has a single label; each node or edge can have zero or more properties; and a node or edge can have the same label-value pair one or more times.

On the other side, the above definition does not support multiple labels for nodes or edges. We have two reasons to justify this restriction. First, this feature is not supported by all graph database systems. Second, it makes complex the definition of schema-instance consistency.

Given two nodes $n_1, n_2 \in N$ and an edge $e \in E$, satisfying that $\Sigma(e) = (n_1, n_2)$, we will use $e = (n_1, n_2)$ as a shorthand representation for e, where n_1 and n_2 are called the "source node" and the "target node" of e respectively.

Hence, the formal description of the PG presented in Figure 3 is given as follows:

$$
\begin{aligned}
&\mathbb{N} = \{n_1, n_2\},\\
&\mathbb{E} = \{e_1\},\\
&\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5\},\\
&\Gamma(n_1) = \{\texttt{Organisation}\},\ \Gamma(n_2) = \{\texttt{Person}\},\\
&\Gamma(e_1) = \{\texttt{ceo}\},\\
&\Upsilon(p_1) = (\texttt{name}, \texttt{"Tesla, Inc."}),\\
&\quad \Upsilon(p_2) = (\texttt{creation}, \texttt{2003-07-01}),\\
&\quad \Upsilon(p_3) = (\texttt{birthName}, \texttt{"Elon Musk"}),\ \Upsilon(p_4) = (\texttt{age}, 46),\\
&\quad \Upsilon(p_5) = (\texttt{since}, 2003)\\
&\Sigma(e_1) = \{n_1, n_2\},\\
&\Delta(n_1) = \{p_1, p_2\},\ \Delta(n_2) = \{p_3, p_4\},\ \Delta(e_1) = \{p_5\}.
\end{aligned}
$$

2) PROPERTY GRAPH SCHEMA

A Property Graph Schema defines the structure of a PG database. Specifically, it defines types of nodes, types of edges, and the properties for such types.

For instance, Figure 4 shows a graphical representation of a PG schema. The formal definition of PG schema is presented next.

Definition 9 (Property Graph Schema): A property graph schema is defined as a tuple $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ where:

- $\mathbb{N}_S$ is a finite set of node types;
- $\mathbb{E}_S$ is a finite set of edge types;
- $\mathbb{P}_S$ is a finite set of property types;
- $\Theta : (\mathbb{N}_S \cup \mathbb{E}_S) \rightarrow \mathbb{L}$ is a total function that assigns a label to each node or edge;
- $\Pi : \mathbb{P}_S \rightarrow (\mathbb{L} \times \mathbb{T})$ is a total function that associates each property type with a property label and a data type;
- $\Phi : \mathbb{E}_S \rightarrow (\mathbb{N}_S \times \mathbb{N}_S)$ is a total function that associates each edge type with a pair of node types;
- $\Psi : (\mathbb{N}_S \cup \mathbb{E}_S) \rightarrow \mathcal{P}^+(\mathbb{P}_S)$ is a partial function that associates a node or edge type with a non-empty set of property types, satisfying that $\Psi(o_1) \cap \Psi(o_2) = \emptyset$, for each pair of objects $o_1, o_2 \in \text{dom}(\Psi)$.

Hence, the formal description of the PG schema shown in Figure 4 is the following:

$$
\begin{aligned}
&\mathbb{N}_S = \{n_1, n_2\},\\
&\mathbb{E}_S = \{e_1\},\\
&\mathbb{P}_S = \{p_1, p_2, p_3, p_4, p_5\},\\
&\Theta(n_1) = \{\texttt{Organisation}\},\ \Theta(n_2) = \{\texttt{Person}\},\ \Theta(e_1) = \{\texttt{ceo}\},\\
&\Pi(p_1) = (\texttt{name}, \texttt{String}),\ \Pi(p_2) = (\texttt{creation}, \texttt{Date}),\\
&\quad \Pi(p_3) = (\texttt{birthName}, \texttt{String}),\ \Pi(p_4) = (\texttt{age}, \texttt{Integer}),\\
&\quad \Pi(p_5) = (\texttt{since}, \texttt{Date}),\\
&\Phi(e_1) = (n_1, n_2),\\
&\Psi(n_1) = \{p_1, p_2\},\ \Psi(n_2) = \{p_3, p_4\},\ \Psi(e_1) = \{p_5\}
\end{aligned}
$$

Given a PG schema $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ and a PG $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$, we say that G^P is valid with respect to S^P, denoted as $G^P \models S^P$, iff:

1) for each $n \in \mathbb{N}$, it applies that there is $nt \in \mathbb{N}_S$ satisfying that:
   (a) $\Gamma(n) = \Theta(nt)$;
   (b) for each $p \in \Delta(n)$, there is $pt \in \Psi(nt)$ satisfying that $\Upsilon(p) = (l, v)$ and $\Pi(t_p) = (l, \text{type}(v))$.
2) for each $e = (n, n') \in \mathbb{E}$, it applies that there is $et \in \mathbb{E}_S$ with $\Phi(et) = (nt, nt')$ satisfying that:
   (a) $\Gamma(e) = \Theta(et)$, $\Gamma(n) = \Theta(nt)$, $\Gamma(n') = \Theta(nt')$;
   (b) for each $p' \in \Delta(e)$, there is $pt' \in \Psi(et)$ satisfying that $\Upsilon(p') = (l', v')$ and $\Pi(pt') = (l', \text{type}(v'))$.

Here, condition (1a) validates that every node is labeled with a node type defined by the schema; condition (1b) verifies that each node contains the properties defined by its node type; condition (2a) verifies that each edge, and the pairs of nodes that it connects, are labeled with an edge type, and the corresponding node types; and condition (2b) verifies that each edge contains the properties defined by the schema.

Finally, we present the notion of PG database.

Definition 10 (Property Graph Database): A property graph database D^P is a pair (S^P, G^P) where S^P is a PG schema and G^P is a PG satisfying that $G^P \models S^P$.

D. RDF DATABASES VERSUS PG DATABASES

Upon comparison of RDF graphs and PGs, we see that both share the main characteristics of a traditional labeled directed graph, that is, nodes and edges contain labels, the edges are directed, and multiple edges are possible between a given pair of nodes. However, there are also some differences between them:

- An RDF graph allows three types of nodes (IRIs, blank nodes and literals) whereas a PG allows a single type of node;

- Each node or edge in an RDF graph contains just a single value (i.e. a label), whereas each node or edge in a PG could contain multiple labels and properties respectively;
- An RDF graph supports multi-value properties, whereas a PG usually just support mono-value properties;
- An RDF graph allows to have edges between edges, a feature which isn't supported in a PG (by definition);
- A node in an RDF graph could be associated with zero or more classes or resources, while a node in a PG usually has a single node type.

In addition to the above structural differences, RDF Schema gives special semantics to the terms in its vocabulary. For example, the terms `rdf:Statement`, `rdf:subject`, `rdf:predicate` and `rdf:object` can be used to describe explicitly RDF statements. This feature, called "reification", is not studied in this article as it is rarely used in practice.

A very interesting feature of both, RDF and PG databases, is the support for schema-less databases, i.e. the databases could not have a fixed data structure. In the particular case of RDF, it is possible to find three types of datasets: datasets without schema definitions, datasets that merge data and schema; and datasets that separate schema and instance.

Depending on whether or not the input RDF dataset has a schema, the database mappings can be classified into two types: **(i)** *schema-dependent*: one that generates a target PG schema from the input RDF graph schema, and then transforms the RDF graph into a PG; and **(ii)** *schema-independent*: one that creates a generic PG schema (based on a predefined structure) and then transforms the RDF graph into a PG. In this paper, we developed these two types of database mappings.

III. SIMPLE DATABASE MAPPING (SDM)

This section describes the schema-independent database mapping $\mathcal{DM}_1$ which allows to transform an schema-less RDF database into a schema-less PG database. $\mathcal{DM}_1$ is just composed of an instance mapping which allows to transform the input RDF graph into a PG graph.

Given an RDF database $D^R = (\emptyset, G^R)$, we define the database mapping $\mathcal{DM}_1 = (\emptyset, \mathcal{IM}_1)$ such that $\mathcal{DM}_1(D^R) = (\emptyset, G^P)$ where $G^P = \mathcal{IM}_1(G^R)$. The instance mapping $\mathcal{IM}_1$ is defined next.

A. INSTANCE MAPPING $\mathcal{IM}_1$

The instance mapping $\mathcal{IM}_1$ is defined as follows:

Definition 11 (Instance mapping $\mathcal{IM}_1$): Let $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph and $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a PG. The instance mapping $\mathcal{IM}_1(G^R) = G^P$ is defined as follows:

1) For each $r \in N_R$
   - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \text{name}(\delta(r))$
2) For each $op \in E_O$ satisfying that $\beta_O(op) = (r, r')$ where $r, r' \in N_R$
   - There will be $e \in \mathbb{E}$ with $\Gamma(e) = \text{name}(\delta(op))$ and $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to r, r' respectively.
3) For each $dp \in E_D$ satisfying that $\beta_D(dp) = (r, l)$ where $r \in N_R$ and $l \in N_L$
   - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\text{name}(\delta(dp)), \alpha_L(l))$
   - $\Delta(n) = \Delta(n) \cup p$ such that $n \in \mathbb{N}$ corresponds to r.

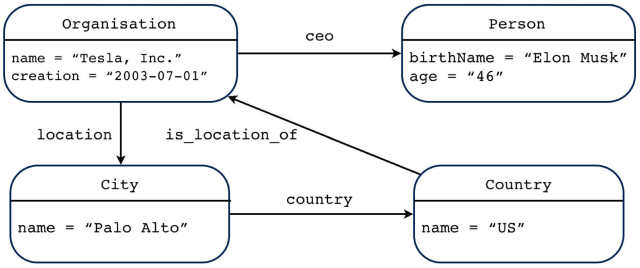

FIGURE 5. Property graph obtained after applying the instance mapping $\mathcal{IM}_1$ to the RDF graph shown in Figure 1.

In general terms, the instance mapping $\mathcal{IM}_1$ creates PG nodes from resource nodes, PG properties from datatype properties, and PG edges from object properties. Nodes, edges and properties are labeled with the name of the corresponding resource class label (defined by the function δ) or the name of the resource identifier (when function δ is undefined).

For example, the PG obtained after applying $\mathcal{IM}_1$ over the RDF graph shown in Figure 1 is given as follows:

$\mathbb{N} = \{n_1, n_2, n_3, n_4\}$,
$\mathbb{E} = \{e_1, e_2, e_3, e_4\}$,
$\mathbb{P} = \{p_5, p_6, p_7, p_8, p_9, p_{10}\}$,
$\Gamma(n_1) = \texttt{Organisation}$, $\Gamma(n_2) = \texttt{Person}$, $\Gamma(n_3) = \texttt{City}$, $\Gamma(n_4) = \texttt{Country}$, $\Gamma(e_1) = \texttt{ceo}$, $\Gamma(e_2) = \texttt{location}$, $\Gamma(e_3) = \texttt{country}$, $\Gamma(e_4) = \texttt{is\_location\_of}$
$\Upsilon(p_5) = (\texttt{name}, \texttt{"Tesla, Inc."})$, $\Upsilon(p_6) = (\texttt{creation}, \texttt{"2003-07-01"})$, $\Upsilon(p_7) = (\texttt{birthName}, \texttt{"Elon\_Musk"})$, $\Upsilon(p_8) = (\texttt{age}, \texttt{"46"})$, $\Upsilon(p_9) = (\texttt{name}, \texttt{"Palo Alto"})$, $\Upsilon(p_{10}) = (\texttt{name}, \texttt{"US"})$
$\Sigma(e_1) = \{n_1, n_2\}$, $\Sigma(e_2) = \{n_1, n_3\}$, $\Sigma(e_3) = \{n_3, n_4\}$, $\Sigma(e_4) = \{n_4, n_1\}$,
$\Delta(n_1) = \{p_5, p_6\}$, $\Delta(n_2) = \{p_7, p_8\}$, $\Delta(n_3) = \{p_9\}$, $\Delta(n_4) = \{p_{10}\}$.

Figure 5 shows a graphical representation of the PG described above.

B. PROPERTIES OF $\mathcal{DM}_1$

In this section we evaluate the properties of the database mapping $\mathcal{DM}_1$, i.e. computability, semantics preservation and information preservation. Recall that $\mathcal{DM}_1$ just contains the instance mapping $\mathcal{IM}_1$, and the output is a PG database without RDF graph schema.

Proposition 1: The database mapping $\mathcal{DM}_1$ is computable.

It is easy to see that the procedure presented in Definition 11 can be implemented as an algorithm.

Proposition 2: The database mapping $\mathcal{DM}_1$ is semantic preserving.

Note that $\mathcal{DM}_1$ assumes that there is no RDF graph schema, i.e. no schema restrictions are considered. Moreover, the output PG database does not contain a PG schema. Hence, it is straightforward to see that $\mathcal{DM}_1$ is semantic preserving.

Proposition 3: The database mapping $\mathcal{DM}_1$ is not information preserving.

Note that the instance mapping $\mathcal{IM}_1$ loses multiple pieces of information from the input RDF graph. In particular, it extract simple labels from IRIs and blank nodes (e.g. by removing the namespace part of a IRI). Hence, it is not possible to define an inverse mapping which is able to reconstruct all the original information.

Although the database mapping $\mathcal{DM}_1$ does not satisfy the information preservation property, it is a simple method to transform RDF datasets that contains a merge of data and schema. In particular, it works well with RDF graphs where each resource defines its resource class by means of the `rdf:type` term.

IV. GENERIC DATABASE MAPPING (GDM)

This section describes the schema-independent database mapping $\mathcal{DM}_2$ which allows to transform a schema-less RDF database into a complete PG database. $\mathcal{DM}_2$ is composed of a schema mapping $\mathcal{SM}_2$ and an instance mapping $\mathcal{IM}_2$ such that $\mathcal{SM}_2$ generates a "generic" PG schema (always the same) and $\mathcal{IM}_2$ allows to generate a PG graph from the input RDF graph.

Given an RDF database $D^R = (\emptyset, G^R)$, we define the database mapping $\mathcal{DM}_2 = (\mathcal{SM}_2, \mathcal{IM}_2)$ such that $\mathcal{DM}_2(D^R) = (S^P, G^P)$ where S^P is a generic PG schema and $G^P = \mathcal{IM}_2(G^R)$. The schema mapping $\mathcal{SM}_2$ and the instance mapping $\mathcal{IM}_2$ are defined next.

A. GENERIC PROPERTY GRAPH SCHEMA

First we introduce a property graph schema which is able to model any RDF graph.

Definition 12 (Generic Property Graph Schema): Let $S^* = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ be the PG schema defined as follows:

$\mathbb{N}_S = \{n_1, n_2, n3\}$,
$\mathbb{E}_S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$,
$\mathbb{P}_S = \{p_1, p_2, p_3, p_4, p_5, p_6\}$,
$\Theta(n_1) = \{\texttt{Resource}\}$, $\Theta(n_2) = \{\texttt{BlankNode}\}$, $\Theta(n_3) = \{\texttt{Literal}\}$,
$\Theta(e_1) = \{\texttt{ObjectProperty}\}$, $\Theta(e_2) = \{\texttt{ObjectProperty}\}$, $\Theta(e_3) = \{\texttt{ObjectProperty}\}$, $\Theta(e_4) = \{\texttt{ObjectProperty}\}$, $\Theta(e_5) = \{\texttt{DatatypeProperty}\}$, $\Theta(e_6) = \{\texttt{DatatypeProperty}\}$,
$\Pi(p_1) = (\texttt{iri}, \texttt{String})$, $\Pi(p_2) = (\texttt{type}, \texttt{String})$, $\Pi(p_3) = (\texttt{id}, \texttt{String})$, $\Pi(p_4) = (\texttt{type}, \texttt{String})$, $\Pi(p_5) = (\texttt{value}, \texttt{String})$, $\Pi(p_6) = (\texttt{type}, \texttt{String})$, $\Pi(p_7) = (\texttt{type}, \texttt{String})$, $\Pi(p_8) = (\texttt{type}, \texttt{String})$, $\Pi(p_9) = (\texttt{type}, \texttt{String})$,
$\Phi(e_1) = (n_1, n_1)$, $\Phi(e_2) = (n_2, n_2)$, $\Phi(e_3) = (n_1, n_2)$, $\Phi(e4) = (n_2, n_1)$, $\Phi(e_5) = (n_1, n_3)$, $\Phi(e_6) = (n_2, n_3)$,
$\Psi(n_1) = \{p_1, p_2\}$, $\Psi(n_2) = \{p_3, p_4\}$, $\Psi(n_3) = \{p_5, p_6\}$,
$\Psi(e_1) = \{p_7\}$, $\Psi(e_2) = \{p_8\}$, $\Psi(e_3) = \{p_9\}$.

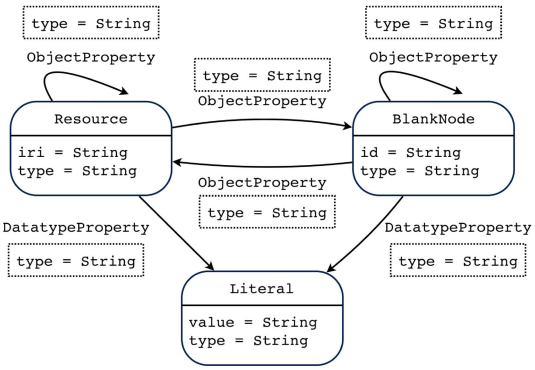

FIGURE 6. A generic property graph schema.

In the above definition: the node type `Resource` will be used to represent RDF resources, the node type `Literal` will be used to represent RDF literals, the edge type `ObjectProperty` allows to represent object properties (i.e. relationships between RDF resources), and the edge type `DatatypeProperty` allows representing datatype properties (i.e. relationships between an RDF resource an a literal). Figure 6 shows a graphical representation of the generic PG schema.

B. INSTANCE MAPPING $\mathcal{IM}_2$

Now, we define the instance mapping $\mathcal{IM}_2$ which takes an RDF graph and produces a PG following the restrictions established by the generic PG schema defined above.

Definition 13 (Instance mapping $\mathcal{IM}_2$): Let $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph and $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a PG. The instance mapping $\mathcal{IM}_2(G^R) = G^P$ is defined as follows:

1) For each $r \in N_R$
   - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \texttt{Resource}$
   - There will be $p \in \mathbb{P}$
   - If $\alpha_R(r) \in I$ then $\Upsilon(p) = (\texttt{iri}, \alpha_R(r))$
   - If $\alpha_R(r) \in B$ then $\Upsilon(p) = (\texttt{id}, \alpha_R(r))$
   - There will be $p' \in \mathbb{P}$ with $\Upsilon(p') = (\texttt{type}, \delta(r))$
   - $\Delta(n) = \{p, p'\}$
2) For each $l \in N_L$
   - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \texttt{Literal}$
   - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\texttt{value}, \alpha_L(l))$
   - There will be $p' \in \mathbb{P}$ with $\Upsilon(p') = (\texttt{type}, \delta(l))$
   - $\Delta(n) = \{p, p'\}$
3) For each $op \in E_O$ satisfying that $\beta_O(op) = (r_1, r_2)$ where $r_1, r_2 \in N_R$
   - There will be $e \in \mathbb{E}$ with $\Gamma(e) = \texttt{ObjectProperty}$, and $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to $r_1, r_2 \in N_R$ respectively
   - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\texttt{type}, \delta(op))$
   - $\Delta(e) = \{p\}$
4) For each $dp \in E_D$ satisfying that $\beta_D(dp) = (r, l)$ where $r \in N_R$ and $l \in N_L$

- There will be $e \in \mathbb{E}$ with $\Gamma(e) =$ `DatatypeProperty`, and $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to r and l respectively
- There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\texttt{type}, \delta(dp))$
- $\Delta(e) = \{p\}$

According to the above definition, the instance mapping $\mathcal{IM}_2$ creates PG nodes from resource nodes and literal nodes, and PG edges from datatype properties and object properties. The property `type` is used to maintain resource class identifiers and RDF datatypes. The property `iri` is used to store the IRI of RDF resources and properties. The property `value` is used to maintain a literal value.

For example, the PG obtained after applying $\mathcal{IM}_2$ over the RDF graph shown in Figure 1 is given as follows:

$\mathbb{N} = \{n_1, \ldots, n_{10}\}$,
$\mathbb{E} = \{e_1, \ldots, e_{10}\}$,
$\mathbb{P} = \{p_1, \ldots, p_{30}\}$,
$\Gamma(n_1) =$ `Resource`, $\Upsilon(p_1) =$ (`iri`, `"ex:Tesla_Inc"`), $\Upsilon(p_2) =$ (`type`, `"voc:Organisation"`), $\Delta(n_1) = \{p_1, p_2\}$,
$\Gamma(n_2) =$ `Resource`, $\Upsilon(p_3) =$ (`iri`, `"ex:Elon_Musk"`), $\Upsilon(p_4) =$ (`type`, `"voc:Person"`), $\Delta(n_2) = \{p_3, p_4\}$,
$\Gamma(n_3) =$ `BlankNode`, $\Upsilon(p_5) =$ (`id`, `"_:b1"`), $\Upsilon(p_6) =$ (`type`, `"voc:City"`), $\Delta(n_3) = \{p_4, p_5\}$,
$\Gamma(n_4) =$ `BlankNode`, $\Upsilon(p_7) =$ (`id`, `"_:b2"`), $\Upsilon(p_8) =$ (`type`, `"voc:City"`), $\Delta(n_4) = \{p_6, p_7\}$,
$\Gamma(n_5) =$ `Literal`, $\Upsilon(p_9) =$ (`value`, `"Tesla, Inc."`), $\Upsilon(p_{10}) =$ (`type`, `"xsd:string"`), $\Delta(n5) = \{p_9, p_{10}\}$,
$\Gamma(n_6) =$ `Literal`, $\Upsilon(p_{11}) =$ (`value`, `"2003-07-01"`), $\Upsilon(p_{12}) =$ (`type`, `"xsd:date"`), $\Delta(n_6) = \{p_{11}, p_{12}\}$,
$\Gamma(n_7) =$ {`Literal`}, $\Upsilon(p_{13}) =$ (`value`, `"Elon Musk"`), $\Upsilon(p_{14}) =$ (`type`, `"xsd:string"`), $\Delta(n_7) = \{p_{13}, p_{14}\}$,
$\Gamma(n_8) =$ {`Literal`}, $\Upsilon(p_{15}) =$ (`value`, `"46"`), $\Upsilon(p_{16}) =$ (`type`, `"xsd:int"`), $\Delta(n_8) = \{p_{15}, p_{16}\}$,
$\Gamma(n_9) =$ {`Literal`}, $\Upsilon(p_{17}) =$ (`value`, `"Palo Alto"`), $\Upsilon(p_{18}) =$ (`type`, `"xsd:string"`), $\Delta(n_9) = \{p_{17}, p_{18}\}$,
$\Gamma(n_{10}) =$ {`Literal`}, $\Upsilon(p_{19}) =$ (`value`, `"US"`), $\Upsilon(p_{20}) =$ (`type`, `"xsd:string"`), $\Delta(n_{10}) = \{p_{19}, p_{20}\}$,
$\Gamma(e_1) =$ `ObjectProperty`, $\Sigma(e_1) = \{n_1, n_2\}$, $\Upsilon(p_{21}) =$ (`type`, `"voc:ceo"`), $\Delta(e_1) = \{p_{21}\}$,
$\Gamma(e_2) =$ `ObjectProperty`, $\Sigma(e_2) = \{n_1, n_3\}$, $\Upsilon(p_{22}) =$ (`type`, `"voc:location"`), $\Delta(e_2) = \{p_{22}\}$,
$\Gamma(e_3) =$ `ObjectProperty`, $\Sigma(e_3) = \{n_3, n_4\}$, $\Upsilon(p_{23}) =$ (`type`, `"voc:country"`), $\Delta(e_3) = \{p_{23}\}$,
$\Gamma(e_4) =$ `ObjectProperty`, $\Sigma(e_4) = \{n_3, n_4\}$, $\Upsilon(p_{24}) =$ (`type`, `"voc:is_location_of"`), $\Delta(e_4) = \{p_{24}\}$,
$\Gamma(e_5) =$ `DatatypeProperty`, $\Sigma(e_5) = \{n_3, n_5\}$, $\Upsilon(p_{25}) =$ (`type`, `"voc:name"`), $\Delta(e_5) = \{p_{25}\}$,
$\Gamma(e_6) =$ `DatatypeProperty`, $\Sigma(e_6) = \{n_1, n_6\}$, $\Upsilon(p_{26}) =$ (`type`, `"voc:creation"`), $\Delta(e_6) = \{p_{26}\}$,
$\Gamma(e_7) =$ `DatatypeProperty`, $\Sigma(e_7) = \{n_2, n_7\}$, $\Upsilon(p_{27}) =$ (`type`, `"voc:birthName"`), $\Delta(e_7) = \{p_{27}\}$,
$\Gamma(e_8) =$ `DatatypeProperty`, $\Sigma(e_8) = \{n_2, n_8\}$, $\Upsilon(p_{28}) =$ (`type`, `"voc:age"`), $\Delta(e_8) = \{p_{28}\}$,
$\Gamma(e_9) =$ `DatatypeProperty`, $\Sigma(e_9) = \{n_3, n_9\}$, $\Upsilon(p_{29}) =$ (`type`, `"voc:name"`), $\Delta(e_9) = \{p_{29}\}$,
$\Gamma(e_{10}) =$ `DatatypeProperty`, $\Sigma(e_{10}) = \{n_4, n_10\}$, $\Upsilon(p_{30}) =$ (`type`, `"voc:name"`), $\Delta(e_{10}) = \{p_{30}\}$.

Figure 7 shows a graphical representation of the PG described above.

C. PROPERTIES OF $\mathcal{DM}_2$

In this section we evaluate the properties of the database mapping $\mathcal{DM}_2$. Recall that $\mathcal{DM}_2$ is a formed by the schema mapping $\mathcal{SM}_2$ and the instance mapping $\mathcal{IM}_2$, where $\mathcal{SM}_2$ always creates a generic PG schema S^*.

Proposition 4: The database mapping $\mathcal{DM}_2$ is *computable*.

It is not difficult to see that an algorithm can be created from the description of the instance mapping $\mathcal{IM}_2$, presented in Definition 13.

Lemma 1: The database mapping $\mathcal{DM}_2$ is *semantics preserving*.

It is straightforward to see (by definition) that any PG graph created with the instance mapping $\mathcal{IM}_2$ will be valid with respect to the generic PG schema S^*.

Theorem 1: The database mapping $\mathcal{DM}_2$ is *information preserving*.

In order to prove that $\mathcal{DM}_2$ is information preserving, we need to provide a database mapping $\mathcal{DM}_2^{-1}$ which allows to transform a PG database into an RDF database, and show that for every RDF database D^R, it applies that $D^R = \mathcal{DM}_2^{-1}(\mathcal{DM}_2(D^R))$.

Recalling that the objective of this section is to provide a schema-independent database mapping, we will assume that for any RDF database $D^R = (S^R, G^R)$, the RDF graph schema S^R is null or irrelevant to validate G^R. Hence, we just define an instance mapping $\mathcal{IM}_2^{-1}$ which allows to transform a PG graph into an RDF database, such that for every RDF graph G^R, it must satisfy that $G^R = \mathcal{IM}_2^{-1}(\mathcal{IM}_2(G^R))$.

Definition 14 (Instance mapping $\mathcal{IM}_2^{-1}$): Let $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a property graph and $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph. The instance mapping $\mathcal{IM}_2^{-1}(G^P) = G^R$ is defined as follows:

1) For each $n \in \mathbb{N}$ satisfying that $\Gamma(n) =$ `Resource`, $p_1, p_2 \in \Delta(n)$, $\Upsilon(p_1) = (\texttt{iri}, v_1)$ and $\Upsilon(p_2) = (\texttt{type}, v_2)$, then there will be $r \in N_R$ with $\alpha_R(r) = v_1$ and $\delta(r) = v_2$
2) For each $n \in \mathbb{N}$ satisfying that $\Gamma(n) =$ `BlankNode`, $p_1, p_2 \in \Delta(n)$, $\Upsilon(p_1) = (\texttt{id}, v_1)$ and $\Upsilon(p_2) = (\texttt{type}, v_2)$, then there will be $r \in N_R$ with $\alpha_R(r) = v_1$ and $\delta(r) = v_2$
3) For each $n \in \mathbb{N}$ satisfying that $\Gamma(n) =$ `Literal`, $p_1, p_2 \in \Delta(n)$, $\Upsilon(p_1) = (\texttt{value}, v_1)$ and $\Upsilon(p_2) = (\texttt{type}, v_2)$, then there will be $r \in N_L$ with $\alpha_L(r) = v_1$ and $\delta(r) = v_2$
4) For each $e \in \mathbb{E}$ satisfying that $\Gamma(e) =$ `ObjectProperty`, $p \in \Delta(e)$, $\Upsilon(p) = (\texttt{type}, v)$, $\Sigma(e) = (n_1, n_2)$, then there will be $op \in E_O$ with $\delta(op) = v$, $\beta_O(op) = (r_1, r_2)$ where $r_1 \in N_R$ corresponds to $n_1 \in \mathbb{N}$, and $r_2 \in N_R$ corresponds to $n_2 \in \mathbb{N}$
5) For each $e \in \mathbb{E}$ satisfying that $\Gamma(e) =$ `DatatypeProperty`, $p \in \Delta(e)$, $\Upsilon(p) = (\texttt{type}, v)$, $\Sigma(e) = (n_1, n_2)$, then there will be $dp \in E_D$ with $\delta(dp) = v$, $\beta_D(dp) = (r_1, r_2)$ where $r_1 \in N_R$ corresponds to $n_1 \in \mathbb{N}$, and $r_2 \in N_L$ corresponds to $n_2 \in \mathbb{N}$

Hence, the above method defines that for each node labeled with `Resource` or `BlankNode` is transformed into a resource node, each node labeled with `Literal` is transformed into a literal node, each edge labeled with

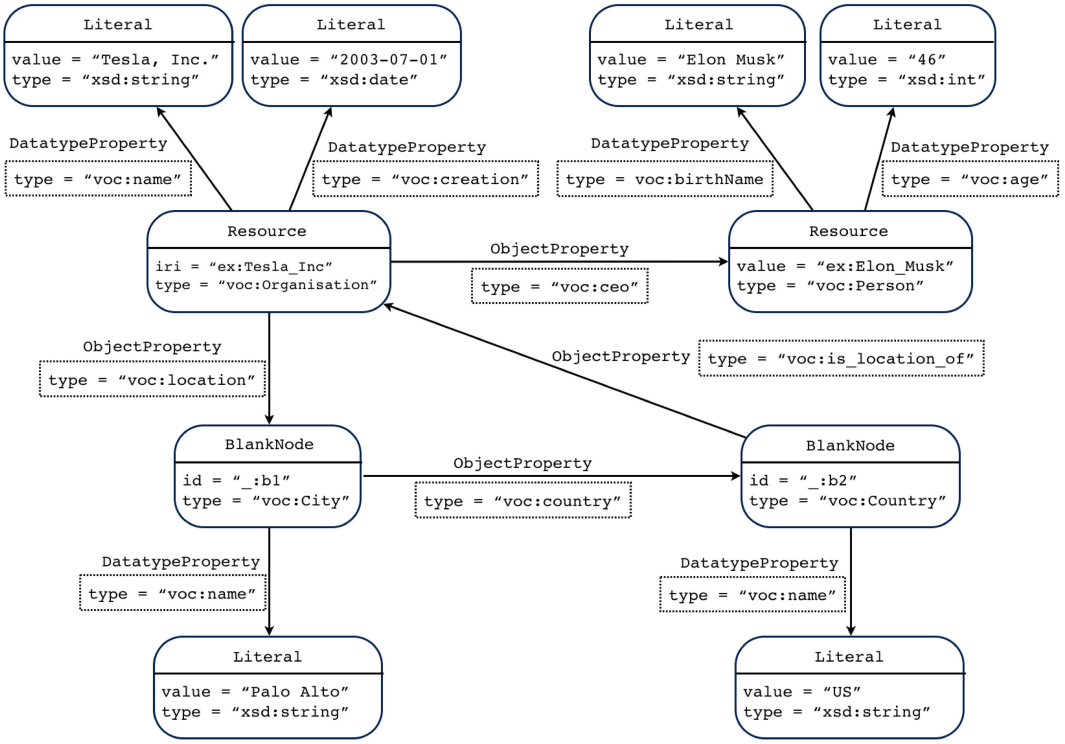

FIGURE 7. Property graph obtained after applying the instance mapping $\mathcal{IM}_2$ to the RDF graph shown in figure 1.

`ObjectProperty` is transformed into a resource-resource edge, and each edge labeled with `DatatypeProperty` is transformed into a resource-literal edge. Additionally, the property `iri` is used to recover the original IRI identifier for `Resource` nodes, the property `id` is used to recover the original identifier for `BlankNode` nodes, and the property `type` allows us to recover the IRI identifier of the resource class associated to each node or edge.

It is not difficult to verify that for any RDF graph G^R, we can produce a PG graph $G^P = \mathcal{SM}_3(G^R)$, and then recover G^R by using $\mathcal{IM}_2^{-1}(G^P)$.

V. COMPLETE DATABASE MAPPING (CDM)

This section describes the schema-dependent database mapping $\mathcal{DM}_3$ which allows to transform a complete RDF database into a complete PG database. $\mathcal{DM}_3$ is composed of a schema mapping $\mathcal{SM}_3$ and an instance mapping $\mathcal{IM}_3$ such that $\mathcal{SM}_3$ generates a PG schema from the input RDF graph schema, and $\mathcal{IM}_3$ generates a PG graph from the input RDF graph.

Recall that I_{DT} is the set of IRIs referencing RDF datatypes, and $\mathbb{T}$ is the set of PG datatypes. Assume that there is a total function $f : I_{DT} \rightarrow \mathbb{T}$ which maps RDF datatypes into PG datatypes. Additionally, assume that f^{-1} is the inverse function of f, i.e. f^{-1} maps PG datatypes into RDF datatypes.

Given an RDF database $D^R = (S^R, G^R)$, we define the database mapping $\mathcal{DM}_3 = (\mathcal{SM}_3, \mathcal{IM}_3)$ such that $\mathcal{DM}_3(D^R) = (S^P, G^P)$ where $S^P = \mathcal{SM}_3(S^R)$ and $G^P = \mathcal{IM}_3(G^R)$. The schema mapping $\mathcal{SM}_3$ and the instance mapping $\mathcal{IM}_3$ are defined next.

A. SCHEMA MAPPING $\mathcal{SM}_3$

We define a schema mapping $\mathcal{SM}_3$ which takes an RDF graph schema as input and returns a PG Schema as output.

Definition 15 (Schema Mapping $\mathcal{SM}_3$): Let $S^R = (N_S, E_S, \phi, \varphi)$ be an RDF schema and $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ be a PG schema. The schema mapping $\mathcal{SM}_3(S^R) = S^P$ is defined as follows:

1) For each $rc \in N_S$ satisfying that $\phi(rc) \notin I_{DT}$
   - There will be $nt \in \mathbb{N}_S$ with $\Theta(nt) = \phi(rc)$
2) For each $pc \in E_S$ satisfying that $\varphi(pc) = (rc_1, rc_2)$
   - If $\phi(rc_2) \in I_{DT}$ then
     - There will be $pt \in \mathbb{P}_S$ with $\Pi(pt) = (\phi(pc), f(\phi(rc_2)))$, $\Psi(nt) = \Psi(nt) \cup pt$ where $nt \in \mathbb{N}_S$ corresponds to $rc_1 \in N_S$.
   - If $\phi(rc_2) \notin I_{DT}$ then
     - There will be $et \in \mathbb{E}_S$ with $\Theta(et) = \phi(pc)$, $\Phi(et) = (nt_1, nt_2)$ where $nt_1, nt_2 \in \mathbb{N}_S$ correspond to $rc_1, rc_2 \in N_S$ respectively.

Hence, the schema mapping $\mathcal{SM}_3$ creates a node type for each resource type (with exception of RDF data types);, creates a property type for each object property, and creates an edge type for each value property.

Assume that the function f is defined by the following datatype assignments: $f(\texttt{xsd:string}) = \texttt{String}$, $f(\texttt{xsd:int}) = \texttt{Integer}$ and $f(\texttt{xsd:date}) = \texttt{Date}$. Hence, the PG schema obtained from the RDF graph schema shown in Figure 2 is given as follows:

$\mathbb{N}_S = \{n_1, n_2, n_3, n_4\}$,
$\mathbb{E}_S = \{e_1, e_2, e_3, e_4\}$,
$\mathbb{P}_S = \{p_5, p_6, p_7, p_8, p_9, p_{10}\}$,
$\Theta(n_1) = \texttt{voc:Organisation}$, $\Theta(n_2) = \texttt{voc:Person}$,
$\Theta(n_3) = \texttt{voc:City}$, $\Theta(n_4) = \texttt{voc:Country}$,
$\Theta(e_1) = \texttt{voc:ceo}$, $\Theta(e_2) = \texttt{voc:location}$,
$\Theta(e_3) = \texttt{voc:country}$, $\Theta(e_4) = \texttt{voc:is\_location\_of}$,
$\Pi(p5) = (\texttt{voc:name}, \texttt{String})$, $\Pi(p_6) = (\texttt{voc:creation}, \texttt{Date})$,
$\Pi(p_7) = (\texttt{voc:birthName}, \texttt{String})$,
$\Pi(p_8) = (\texttt{voc:age}, \texttt{Integer})$,
$\Pi(p_9) = (\texttt{voc:name}, \texttt{String})$,
$\Pi(p_{10}) = (\texttt{voc:name}, \texttt{String})$,
$\Phi(e_1) = (n_1, n_2)$, $\Phi(e_2) = (n_1, n_3)$, $\Phi(e_3) = (n_3, n_4)$,
$\Phi(e_4) = (n_4, n_1)$,
$\Psi(n_1) = \{p_5, p_6\}$, $\Psi(n_2) = \{p_7, p_8\}$, $\Psi(n_3) = \{p_9\}$, $\Psi(n_4) = \{p_{10}\}$.

B. INSTANCE MAPPING $\mathcal{IM}_3$

Now, we define the instance mapping $\mathcal{IM}_3$ which takes an RDF graph as input and returns a PG as output.

Definition 16 (Instance Mapping $\mathcal{IM}_3$): Let $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph and $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a PG. The instance mapping $\mathcal{IM}_3(G^R) = G^P$ is defined as follows:

1) For each $r \in N_R$
   - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \delta(r)$
   - There will be $p \in \mathbb{P}$
   - If $\alpha_R(r) \in I$ then $\Upsilon(p) = (\text{iri}, \alpha_R(r))$
   - If $\alpha_R(r) \in B$ then $\Upsilon(p) = (\text{id}, \alpha_R(r))$
   - $\Delta(n) = \{p\}$.
2) For each $op \in E_O$ satisfying that $\beta_O(op) = (r_1, r_2)$
   - There will be $e \in \mathbb{E}$ with $\Gamma(e) = \delta(op)$, $\Sigma(e) = (n_1, n_2)$ where $n1, n2 \in \mathbb{N}$ correspond to $r1, r2 \in N_R$ respectively.
3) For each $dp \in E_D$ satisfying that $\beta_D(dp) = (r_1, r_2)$
   - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\delta(dp), \alpha_L(r_2))$, $\Delta(n) = \Delta(n) \cup \{p\}$ where $n \in \mathbb{N}$ corresponds to $r_1 \in N_R$.

According to the above definition, the instance mapping $\mathcal{IM}_3$ creates a node in G_R for each resource node, creates a property in G_R for each datatype property, and creates an edge in G_R for each object property.

For example, the PG obtained after applying $\mathcal{IM}_3$ over the RDF graph shown in Figure 1 is given as follows:

$\mathbb{N} = \{n_1, n_2, n3_3, n_4\}$,
$\mathbb{E} = \{e_1, e_2, e_3, e_4\}$,
$\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$,
$\Gamma(n_1) = \texttt{voc:Organisation}$, $\Gamma(n_2) = \texttt{voc:Person}$,
$\Gamma(n_3) = \texttt{voc:City}$, $\Gamma(n_4) = \texttt{voc:Country}$,
$\Gamma(e_1) = \texttt{voc:ceo}$, $\Gamma(e_2) = \texttt{voc:location}$,
$\Gamma(e_3) = \texttt{voc:country}$, $\Gamma(e_4) = \texttt{voc:is\_location\_of}$,
$\Upsilon(p_1) = (\texttt{iri}, \texttt{"ex:Tesla\_Inc"})$,
$\Upsilon(p_2) = (\texttt{iri}, \texttt{"ex:Elon\_Musk"})$, $\Upsilon(p_3) = (\texttt{id}, \texttt{"\_:b1"})$,
$\Upsilon(p_4) = (\texttt{id}, \texttt{"\_:b2"})$,
$\Upsilon(p_5) = (\texttt{voc:name}, \texttt{"Tesla, Inc."})$,
$\Upsilon(p_6) = (\texttt{voc:creation}, \texttt{"2003-07-01"})$,

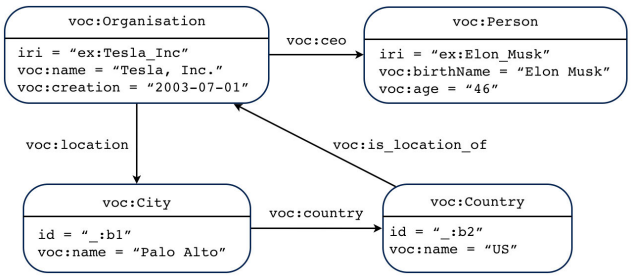

FIGURE 8. Property graph obtained after applying the instance mapping $\mathcal{IM}_3$ to the RDF graph shown in Figure 1.

$\Upsilon(p_7) = (\texttt{voc:birthName}, \texttt{"Elon Musk"})$,
$\Upsilon(p_8) = (\texttt{voc:age}, \texttt{"46"})$,
$\Upsilon(p_9) = (\texttt{voc:name}, \texttt{"Palo Alto"})$,
$\Upsilon(p_{10}) = (\texttt{voc:name}, \texttt{"US"})$
$\Sigma(e_1) = \{n_1, n_2\}$, $\Sigma(e_2) = \{n_1, n_3\}$, $\Sigma(e_3) = \{n_3, n_4\}$,
$\Sigma(e_4) = \{n_4, n_1\}$,
$\Delta(n_1) = \{p_1, p_5, p_6\}$, $\Delta(n_2) = \{p_2, p_7, p_8\}$, $\Delta(n_3) = \{p_3, p_9\}$,
$\Delta(n_4) = \{p_4, p_{10}\}$.

Figure 8 shows a graphical representation of the PG described above.

C. PROPERTIES OF $\mathcal{DM}_3$

In this section we will evaluate the properties of the database mapping $\mathcal{DM}_3$. Recall that $\mathcal{DM}_3$ is formed by the schema mapping $\mathcal{SM}_3$ and the instance mapping $\mathcal{IM}_3$.

Proposition 5: The database mapping $\mathcal{DM}_3$ is computable.

It is straightforward to see that Definition 15 and Definition 16 can be transformed into algorithms to compute $\mathcal{SM}_3$ and $\mathcal{IM}_3$ respectively.

Lemma 2: The database mapping $\mathcal{DM}_3$ is *semantics preserving*.

Note that the schema mapping $\mathcal{SM}_3$ and the instance mapping $\mathcal{IM}_3$ have been designed to create a PG database that maintains the restrictions defined by the source RDF database. On one side, the schema mapping $\mathcal{SM}_3$ allows transforming the structural and semantic restrictions from the RDF graph schema to the PG schema. On the other side, any PG generated by the instance mapping $\mathcal{IM}_3$ will be valid with respect to the generated PG schema.

The following facts support the semantics preservation property of $\mathcal{DM}_3$:

- We provide a procedure to create a complete RDF graph schema S^R from a set of RDF triples describing an RDF schema, i.e. each property defines its domain and range resource classes.
- We provide a procedure to create an RDF graph G^R from a set of RDF triples, satisfying that every node and edge in G^R is associated with a resource class; it allows a complete connection between the RDF instance and the RDF schema.
- The schema mapping $\mathcal{SM}_3$ creates a node type for each user-defined resource type, a property type for each datatype property, and an edge for each object property.

- Similarly, the instance mapping $\mathcal{IM}_3$ creates a node for each resource, a property for each resource-literal edge, and an edge for each resource-resource edge.

Theorem 2: The database mapping $\mathcal{DM}_3$ is *information preserving*.

In order to prove that $\mathcal{DM}_3$ is information preserving, we will define a database mapping $\mathcal{DM}_3^{-1} = (\mathcal{SM}_3^{-1}, \mathcal{IM}_3^{-1})$ which allows to transform a PG database into an RDF database. The inverse mapping $\mathcal{DM}_3^{-1}$ must satisfy that $D = \mathcal{DM}_3^{-1}(\mathcal{DM}_3(D))$ for any RDF database D. Next we define the schema mapping $\mathcal{SM}_3^{-1}$ and the instance mapping $\mathcal{IM}_3^{-1}$.

Definition 17 (Schema mapping $\mathcal{SM}_3^{-1}$): Let $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ be a PG schema and $S^R = (N_S, E_S, \phi, \varphi)$ be an RDF schema. The schema mapping $\mathcal{SM}_3^{-1}(S^R) = S^P$ is defined as follows:

- Let $C = \{n \mid n \in \text{range}(\Theta)\} \cup \{n = f^{-1}(t) \mid (u, t) \in \text{range}(\Pi)\}$
- Let $\omega : C \rightarrow N_S$ be a function that maps IRIs to resource classes
- For each $n \in C$
  - There will be $rc \in N_S$ with $\phi(rc) = n$
  - $\omega(n) = rc$
- For each $et \in \mathbb{E}_S$ with $\Phi(et) = (nt_1, nt_2)$
  - There will be $pc \in E_S$ with $\phi(pc) = \Theta(et)$ and $\varphi(pc) = (\omega(nt_1), \omega(nt_2))$
- For each $nt \in \mathbb{N}_S$
  - For each $pt \in \Psi(nt)$ with $\Phi(pt) = (n, t)$
    - There will be $pc \in E_S$ with $phi(pc) = n$ and $\varphi(pc) = (\omega(nt), \omega(f^{-1}(t)))$

In general terms, the schema mapping $\mathcal{SM}_3^{-1}$ creates a resource class for each node type, an object property for each edge type, and a datatype property for each property type. Given a PG schema $S^P = \mathcal{SM}_3(S^R)$, the schema mapping $\mathcal{SM}_3^{-1}$ allows to "recover" all the schema constraints defined by S^R, i.e $\mathcal{SM}_3^{-1}(S^P) = S^R$.

An issue of $\mathcal{SM}_3^{-1}$, is the existence of RDF datatypes which are not supported by PG databases. For example, `rdfs:Literal` has no equivalent datatype in PG database systems. The solution to this issue is to find a one-to-one correspondence between RDF datatypes and PG datatypes.

Definition 18 (Instance mapping $\mathcal{IM}_3^{-1}$): Let $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a property graph and $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph. The instance mapping $\mathcal{IM}_3^{-1}(G^P) = G^R$ is defined as follows:

1) For each $n \in \mathbb{N}$, there will be $r \in N_R$ where
   a) $\alpha_R(r) = v$ such that $p \in \Delta(n)$ and $\Upsilon(p) = (\texttt{iri}, v)$ or $\Upsilon(p) = (\texttt{id}, v)$
   b) $\delta(r) = \Gamma(n)$
   c) For each $p \in \Delta(n)$ satisfying that $\Upsilon(p) = (lab, val)$ and $lab \notin \{\texttt{iri}, \texttt{id}\}$, there will be $l \in N_L$ and $dp \in E_D$ with $\alpha_L(l) = val$, $\delta(l) = f^{-1}(\text{type}(val))$, $\delta(dp) = lab$ and $\beta(dp) = (r, l)$
2) For each $e \in \mathbb{E}$ where $\Sigma(e) = (n_1, n_2)$, there will be $op \in E_O$ with $\delta(op) = \Gamma(e)$ and $\beta(op) = (r_1, r_2)$ such that r_1, r_2 correspond to n_1, n_2 respectively.

Hence, the above method defines that each node in G^P is transformed into a resource node in G^R, each property in G^P is transformed into a datatype property in G^R, and each edge in G^P is transformed into an object property in G^R. Given a PG $G^P = \mathcal{IM}_3(G^R)$, the instance mapping $\mathcal{IM}_3^{-1}$ allows to "recover" all the data in G^R, i.e $\mathcal{IM}_3^{-1}(G^P) = G^R$.

Note that each RDF graph produced by the instance mapping $\mathcal{IM}_3^{-1}$ will be valid with respect to the schema produced with the corresponding schema mapping $\mathcal{SM}_3^{-1}$. Hence, any RDF database D^R can be transformed into a PG database by using the database mapping $\mathcal{DM}(D^R)$, and D^R could be recovered by using the database mapping $\mathcal{DM}_3^{-1}$.

VI. EXPERIMENTAL EVALUATION

The objective of our experimental evaluation is to examine the performance and scalability of the database mappings presented in this work. This section includes a description of the implementation, the evaluation methodology, the experimental results, and the corresponding discussion.

A. IMPLEMENTATION

We have developed a java application called `rdf2pg` which implements the mappings described in this article. The source code and the executable jar file of `rdf2pg` can be downloaded from Github (`https://github.com/renzoar/rdf2pg`). The tool can be executed in command line by using an expression with the structure

```
java -jar rdf2pg.jar <m> <i> <s>
```

where `<m>` indicates the database mapping (`-sdm` = simple database mapping, `-gdm` = generic database mapping, `-cdm` = complete database mapping), `<i>` indicates the input instance RDF graph file, and `<s>` indicates the input RDF schema file (in case of using `-gdm` or `-cdm`).

The output of the simple database mapping is a file encoding a PG. In addition, the generic and the complete instance mappings produce a second file containing the PG schema. The current implementation uses the YARS-PG [40] data format for both output files.

The `rdf2pg` API includes an interface named `PGWriter` which can be implemented to support other data formats. The use of `PGWriter` is very simple as it provides the methods `WriteNode(PGNode node)` and `WriteEdge(PGEdge edge)` which should be implemented with the corresponding instructions to write nodes and edges in the output data format.

In order to support the processing of large RDF data files, `rdf2pg` uses the StreamRDF class provided by Apache Jena. Additionally, `rdf2pg` implements two methods for writing the output file: a memory-based method which creates a PG object (which follows the definition presented in Section II-C.1); and a disk-based method which writes the output by using a minimal set of structures.

TABLE 1. Datasets used in the experimental evaluation.

Dataset	Domain	Nature	Structure	Website
SP2B	Bibliographic data	Synthetic	Regular	http://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/
Linked GeoData	Spatial data	Real	Regular	http://linkedgeodata.org/
WatDiv	Products sale	Synthetic	Irregular	https://dsg.uwaterloo.ca/watdiv/
BSBM	Products sale	Synthetic	Regular	http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
Wikidata	Knowledge base	Real	Irregular	https://www.wikidata.org/

TABLE 2. RDF Graphs used in the experimental evaluation.

Graph ID	Dataset	#Triples	File Size (*.nt)
G1	SP2B	328	42 KB
G2	SP2B	1,285	206 KB
G3	SP2B	10,303	1.6 MB
G4	SP2B	100,073	16.2 MB
G5	SP2B	1,000,009	165 MB
G6	GeoData	4,914,217	740.7 MB
G7	WatDiv	7,159,355	1.01 GB
G8	BSBM	38,333,972	10.01 GB
G9	Wikidata	41,191,235	6.27 GB

Additionally, we have developed a Java application called `rdfs-processor` which provides three functionalities: analysis of an RDF Schema file to obtain basic information (i.e. the number of resource classes, number of property classes, and number of datatypes); normalization of an RDF Schema, in the case of incomplete definitions (e.g. empty domains); and schema discovery from an RDF data file.

The functionality of schema discovery is very relevant for this paper because most of the available RDF datasets do not provide an RDF Schema file. Our method for schema discovery follows the approach described in [31]. In general terms, the method reads the set of RDF triples two times: in the first pass, it identifies resource classes and property classes; in the second pass, it determines the domain and range for each property class. The output is an RDF file containing a basic description of the RDF Schema by means of the terms `rdf:type`, `rdfs:Class`, `rdf:Property`, `rdfs:domain` and `rdfs:range`. The source code of the `rdfs-processor` is available in Github (`https://github.com/renzoar/rdfs-processor`).

B. METHODOLOGY AND EXPERIMENTAL SETUP

The experimental evaluation consists of a series of experiments that combine three variables: database mapping, data source, RDF graph size, and processing power. We evaluate the three database mappings defined in this paper: simple mapping, generic mapping, and complete mapping.

We consider four sources of RDF data whose characteristics (domain, nature, and structure) are shown in Table 1. We use nine RDF graphs (obtained from the data sources) whose size[3] goes from 328 triples to 41,191,235 triples, as shown in Table 2.

[3]The size of an RDF graph is expressed in terms of the number of triples. Note that the disk space occupied by a set of triples depends on the RDF data format used [27].

TABLE 3. Virtual machines (google cloud platform) used in the experimental evaluation.

Machine ID	Type	#vCPUs	RAM	SSD
VM1	n1-standard-2	2	7.5GB	100GB
VM2	n1-standard-4	4	15GB	100GB
VM3	n1-standard-8	8	30GB	100GB
VM4	n1-standard-16	16	60GB	100GB

The processing power variable indicates the use of machines with different characteristics in terms of hardware. In this case, we used four virtual machines hosted in the Google Cloud Platform, having a varying number of CPUs (Intel Skylake), main/primary memory size (RAM), and secondary memory size (SSD). The technical specification of each machine is shown in Table 3. All the machines worked with Debian GNU/Linux 9 (amd64 built on 20200309) as the operating system and Java OpenJDK 1.8.0_242 (64-Bit) without a graphic environment.

Based on the above variables, we evaluated the database mappings in terms of performance and scalability. The performance is measured as the running time (or runtime) required to execute a mapping and construct the corresponding output database (schema graph and instance graph). To do this, the `rdf2pg` application uses the built-in Java function `System.currentTimeMillis` to register the runtime. The objective is to determine the computational complexity of the mappings in practice.

Each mapping is evaluated under two notions of scalability. Former, we measure the scalability with respect to the size of the input data (i.e. the number of triples). The objective is to determine the behavior of the mappings with RDF graphs of different sizes. Later, we analyze the scalability with respect to the computational resources. The objective is to determine the dependency of each mapping with respect to the hardware.

C. EXPERIMENTAL RESULTS

Our experimental evaluation begins with the extraction of the RDF Schema for each RDF graph. This task was performed by using the `rdfs-processor` tool described in Section VI-A. Table 4 shows information about the corresponding RDF Schemas. We can observe that: the SP2B graphs do not change too much in terms of the number of classes and properties; the number of classes in GeoData and BSBM is larger than the number of properties; a small number of datatypes are defined in the graphs.

TABLE 4. RDF schemas used in the experimental evaluation. This table shows the number of resource classes, property classes, and datatype definitions.

Graph ID	#Classes	#Properties	#Datatypes
G1	5	19	2
G2	6	20	2
G3	8	54	2
G4	9	55	2
G5	12	64	2
G6	393	60	3
G7	40	85	1
G8	1292	36	3
G9	24	44	2

TABLE 5. Runtimes (in milliseconds) for the simple database mapping. Undefined runtimes are represented with "?".

Graph	VM1	VM2	VM3	VM4
G1	3544	1247	1226	1263
G2	1644	1153	1057	1072
G3	2203	1516	1282	1364
G4	4245	3154	2428	2524
G5	16714	14133	11578	12079
G6	89930	69582	66290	67458
G7	193389	156963	148793	149272
G8	?	?	782718	741916
G9	?	593914	616252	625932

TABLE 6. Runtimes (in milliseconds) for the generic database mapping. Undefined runtimes are represented with "?".

Graph	VM1	VM2	VM3	VM4
G1	1359	1015	960	1099
G2	1642	1101	1030	1093
G3	1953	1411	1218	1260
G4	3545	2564	1904	2170
G5	14636	10772	8566	8483
G6	68390	50471	49977	47064
G7	96449	77464	81579	83639
G8	?	550476	486106	487969
G9	?	413090	427542	381572

Once we had the RDF Schema files for each dataset, we executed the experiments in the virtual machines. Every execution of the `rdf2pg` application was configured to use the maximum amount of primary memory allowed by the machine. To do this, we use the -Xmx parameter defined by Java. Table 5, Table 6 and Table 7 show the runtimes for the simple mapping, the generic mapping and the complete mapping respectively.

In general terms, we can observe that the mappings worked well with most of the graphs (i.e. G1 to G7). However, there were problems to complete the task for graphs G8 and G9, running on virtual machines VM1 and VM2. Specifically, the execution of `rdf2pg` produced an error of ''insufficient memory for the Java Runtime Environment''. Hence, our current implementation has a restriction to process large input graphs with small-memory machines.

The above problem is related to the main memory (RAM) required to manage the intermediate objects used by the mappings. Being more specific, the mappings create a HashMap

TABLE 7. Runtimes (in milliseconds) for the complete database mapping. Undefined runtimes are represented with "?".

Graph	VM1	VM2	VM3	VM4
G1	1659	1083	1004	1062
G2	1642	1194	1111	1117
G3	1984	1515	1318	1403
G4	4269	2920	2442	2505
G5	20685	1524	12629	12737
G6	97101	78374	74898	69639
G7	200952	173607	161105	145027
G8	?	?	817471	800303
G9	?	?	609577	603040

TABLE 8. Size (in bytes) of the output files produced during the experimental evaluation of the database mappings (SDM, GDM and CDM). PG and PGS mean property graph and property graph schema respectively.

	SDM	GDM		CDM	
Graph	PG	PG	PGS	PG	PGS
G1	12K	39K	352	15K	553
G2	43K	190K	352	61K	710
G3	320K	1.5M	352	466K	2.3K
G4	3.1M	15M	352	4.5M	3.1K
G5	32M	149M	352	46M	4.0K
G6	125M	731M	352	214M	22K
G7	224M	711M	352	276M	19K
G8	2.6G	7.7G	352	3.3G	872K
G9	1.1G	5.0G	352	1.7G	2.8K

to store all the nodes and their properties, and such a structure could be very large for some graphs. Note that the number of nodes is not directly related to the number of triples. For example, we observed that the number of nodes generated for G8 was higher than G9, even when G9 has more triples than G8. It explains why the simple mapping was able to process G9, but it was not able to process G8, both using VM2.

In order to analyze the scalability of the mappings with respect to the size of the input data, we selected the runtimes obtained with VM4. As shown in Figure 9, the execution time of all the mappings grows up in concordance with the size of the input graphs, i.e., the larger the size of the graph, the larger the runtime. Note also that the runtimes of the mappings are under the baseline defined by the graph sizes. Hence, we can conclude that the complexity of the mappings is linear with respect to the size of the input.

In order to analyze the scalability of the mappings with respect to the computational power, we prepare a plot for each mapping showing the runtimes for all the virtual machines (see Figures 10, 11 and 12). The plots show that the runtimes decrease for VM1, VM2, and VM3; however, the runtimes for VM3 and VM4 are not so different. The latter implies that there is a threshold in which the computational power does not reduce the execution time of the mapping.

As a general conclusion, we can say that the three database mappings presented in this work have an efficient implementation to process large datasets and work under middle-size computational resources.

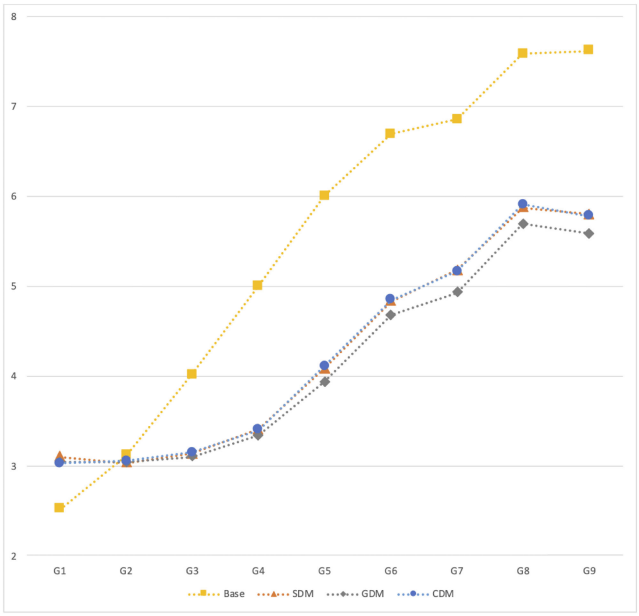

FIGURE 9. Scalability of the database mappings with respect to the size of the input data. The X-axis has graph sizes and the Y-axis has runtimes (in log scale). The "Base" line indicates the sizes of the input graphs in the log scale.

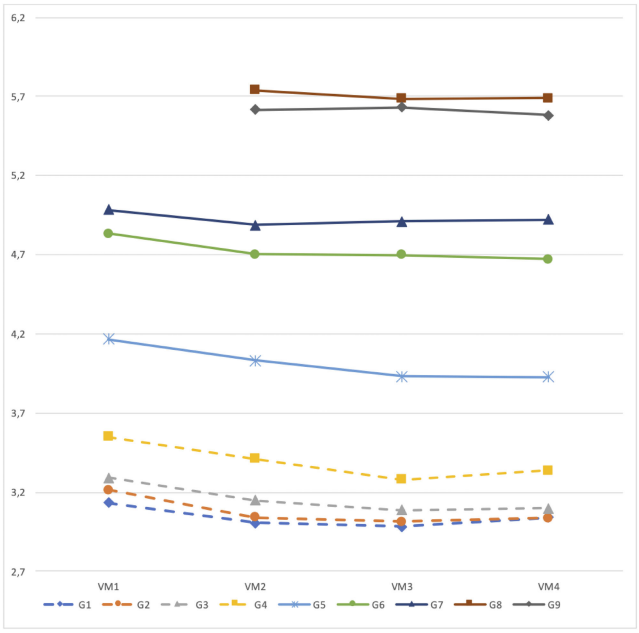

FIGURE 11. Scalability of the generic database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).

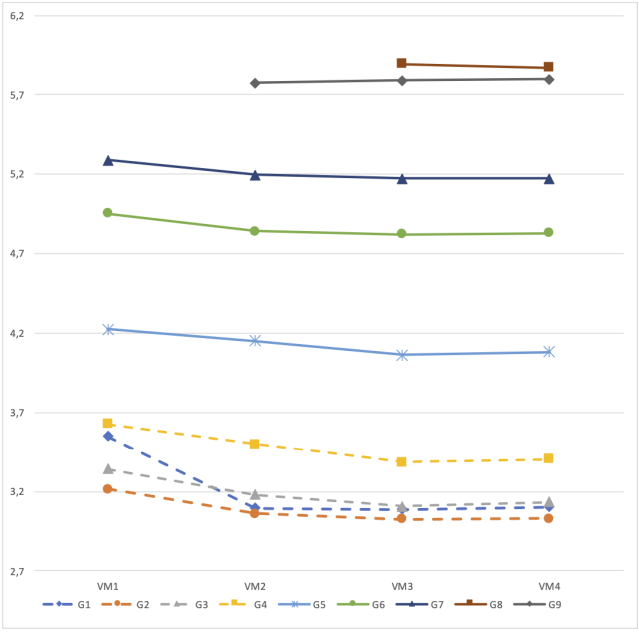

FIGURE 10. Scalability of the simple database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).

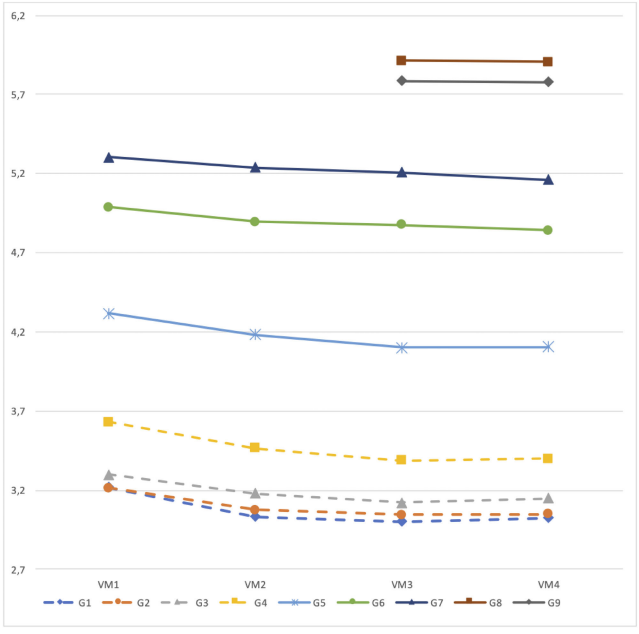

FIGURE 12. Scalability of the complete database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).

All the input and output files described in the above experiments are available in Figshare [6].

D. INTEROPERABILITY IN PRACTICE

In order to show the practical use of the database mappings proposed in this article, we conducted a complete ETL process that involves: Extracting RDF data from an RDF dataset; Transforming the extracted RDF data to PG data, and Loading the transformed data into a property graph database system. Due to its popularity and availability of features for data loading, we selected Neo4j as the target database system.

The main issue in this experiment is the configuration of `rdf2pg` to generate and encode property graphs into a data format that can be consumed by the Neo4j system. To do this,

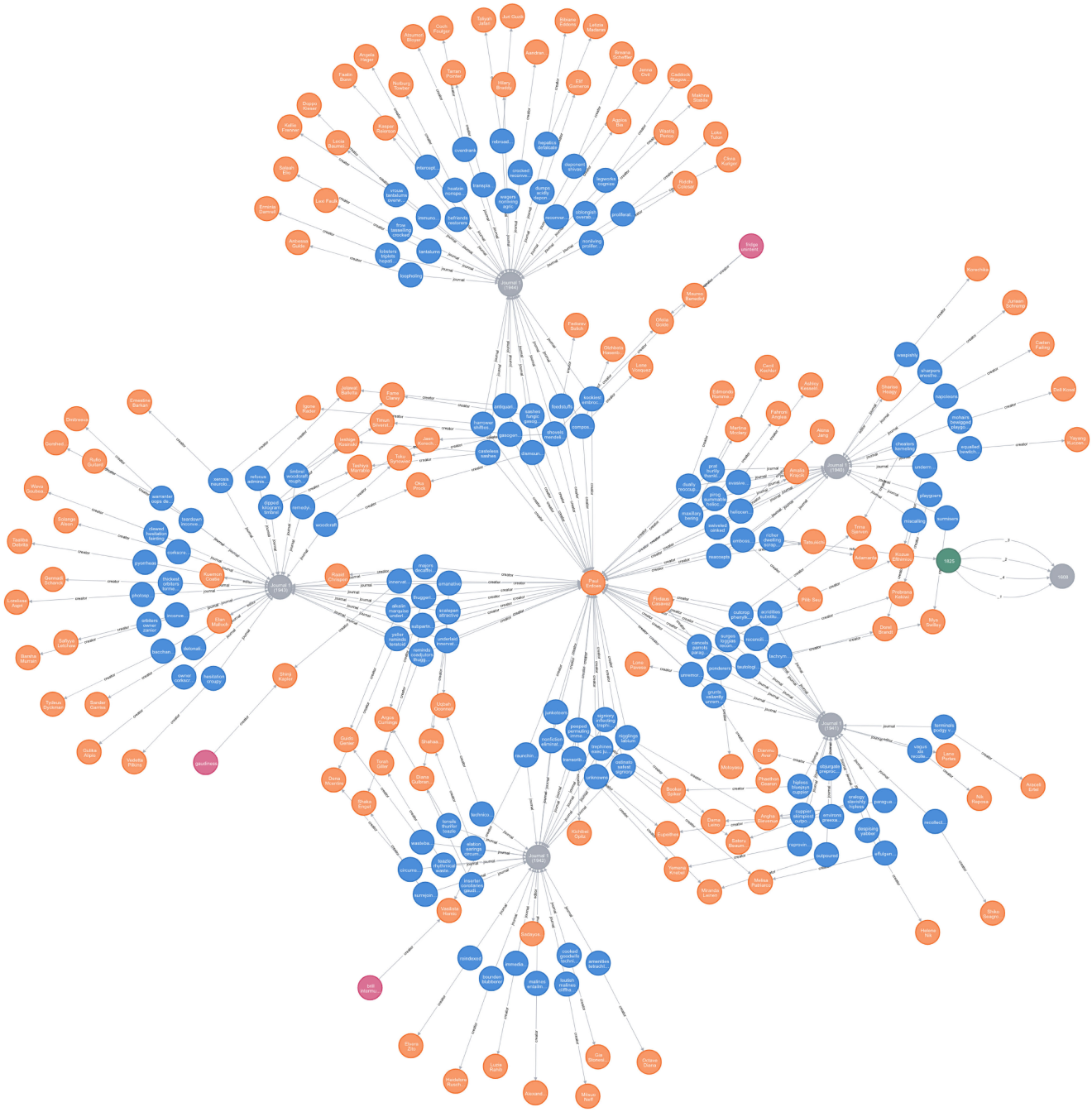

FIGURE 13. Graphical representation of the property graph produced by applying the simple database mapping over the RDF graph G2.

we created the Neo4jWriter class as an implementation of the PGWriter interface provided by `rdf2pg`. The Neo4jWriter class allows exporting a property graph as a set of Cypher instructions to create nodes and edges. For example, the property graph shown in Figure 5 will be exported as follows:

```
CREATE (n15:City {name:'Palo Alto'} )
CREATE (n12:Person {birthName:'Elon Musk',age
    :'46'} )
CREATE (n33:Country {name:'US'} )
CREATE (n1:Organisation {name:'Tesla, Inc.',
    creation:'2003-07-01'} )
CREATE (n1)-[e5:ceo]->(n12)
CREATE (n1)-[e6:location]->(n15)
CREATE (n15)-[e7:country]->(n33)
CREATE (n33)-[e8:is_location_of]->(n1)
```

To demonstrate the validity of our mappings, we used the property graph obtained by applying the simple database mapping over the RDF graph G2, i.e. the SP2B file containing 1,285 triples. The output file containing Cypher instructions was loaded in Neo4j Desktop 1.2.3 by using the browser-based user interface. The loading process took 7 ms, resulting in a property graph with 270 nodes, 348 relationships, 677 properties, and 260 labels. A graphical representation of

TABLE 9. Summary of the related work addressing data interoperability between RDF and property graphs. The lack of information or evidence is denoted as "?".

Work	Target	Direct mapping	Formal definition	Instance mapping	Schema mapping	Information preserving
RDF⋆ [20]	RDF↔PG	No	Yes	Yes	No	?
URS [43]	RDF↔PG	No	No	Yes	Yes	No
NSMNTX [7]	RDF↔PG	No	No	Yes	Yes	No
S2X [33]	RDF→PG	?	No	Yes	No	No
LDM-3N [28]	RDF→PG	No	No	Yes	No	No
rdf2neo [10]	RDF→PG	No	No	Yes	Yes	No
SRDS [16]	RDF→PG	?	No	Yes	Yes	No
Das et al. [15]	PG→RDF	No	No	Yes	No	No
YARS [39]	PG→RDF	No	No	Yes	No	No
Our approach	RDF↔PG	Yes	Yes	Yes	Yes	Yes

the loaded property graph, obtained from the Neo4j browser, is shown in Figure 13.

The above experiment was repeated for the generic and the complete database mappings. The generic mapping produced, after 55 ms, a property graph containing 949 nodes, 1285 relationships, 2911 properties, and 949 labels. The complete mapping took 8 ms, producing a property graph with the same number of elements produced by the simple mapping.

All the information related to this data loading experiment, including the output files and the charts of the property graphs, are available in Figshare [6].

VII. RELATED WORK

In this section we present the related work that targets the interoperability issue between RDF and property graphs. We group the efforts based on the direction of the mapping, i.e. RDF → PG and PG → PG.

A. FROM RDF TO PROPERTY GRAPHS

Hartig and Thompson [20] proposes two formal transformations between RDF⋆ and PGs. RDF⋆ is a conceptual extension of RDF which is based on reification. The first transformation maps any RDF triple as an edge in the resulting PG. Each node has the "kind" property to describe the node type (e.g. IRI). The second transformation distinguishes data and object properties. The former is transformed into node properties and latter into edges of a PG. The shortcoming of this approach is that RDF⋆ – (i) does not support mapping an RDF graph schema, and (ii) adds an extra step of an intermediate mapping and; (iii) isn't supported by major RDF stores.

In *S2X*, Schätzle *et al.* [33] propose a GraphX-specific RDF-PG transformation. The mapping uses attribute *label* to store the node and edge identifiers, i.e. each triple $t = (s, p, o)$ is represented using two vertices v_s, v_0, an edge (v_s, v_o) and labels $v_s.label = s$, $v_o.label = o$, $(v_s, v_o).label = p$. Apart from being only GraphX-specific, this approach misses the concept of properties and also does not cover RDF graph schema.

Nguyen *et al.* [28] propose the *LDM-3N* (labeled directed multigraph - three nodes) graph model. This data model represents each triple element as separate nodes, thus the three nodes (3N). The *LDM-3N* graph model is used to address the Singleton Property (SP) [29] based on reified RDF data. The problem with this approach is that: (i) it adds adds an extra computation step (and $2n$ triples); (ii) does not cover RDF graph Schema; and misses the concept of properties.

Tomaszuk [39] proposes *YARS* serialization for transforming RDF data into PGs. This approach performs only a syntactic transformation between encoding schemes and does not cover RDF Schema.

Brandizi *et al.* [10] propose rdf2neo, a tool that can be used to map any RDF schema to the desired PG schema. This hybrid architecture facilitates access to knowledge networks based on shared data models. However, the disadvantage of this solution is that it maintains a more complex infrastructure that works well in the paper use case, but not for more general applications.

Another approach is presented in [16]. In this paper, the author presents a proposal for converting an RDF data store to a graph database by exploiting the ontology and the constraints of the source.

B. FROM PROPERTY GRAPHS TO RDF

There exist very few proposals for the PG-to-RDF transformation, such as Das *et al.* [15] and Hartig and Thompson [20], that mainly use RDF reification methods (including Blank Nodes) to convert nodes and edge properties in a PG to RDF data. While [20] propose an in-direct mapping that requires converting to the RDF⋆ model (as mentioned earlier), [15] lacks a formal foundation. Both approaches do not consider the presence of a PG schema.

Another approach is Unified Relational Storage (URS) [43]. It focuses on interchangeably managing RDF and PGs, and this is not a strict transformation method.

Barrasa [7] proposes NSMNTX, a plugin that enables the use of RDF in Neo4j. This plugin allows the import and export of both schema and data. The problem with this approach is that NSMNTX is not formally defined and the mappings do not satisfy the property of information preservation.

Table 9, presents a summary of the related work and the features they address. It should be mentioned that some works have studied the problem of mapping RDF to PGs in the scope of specific use cases, e.g. disease networks [25], protein structure exploration [1], and Wikidata reification [21].

VIII. CONCLUSIONS

In this article, we have proposed a novel approach, which consists of three direct mappings, to transform RDF databases into PG databases. We demonstrate, empirically and formally, that the mappings have an efficient implementation to process large datasets.

We showed that two of the proposed mappings satisfy the property of information preservation, i.e. there exist inverse mappings that allow recovering the original databases without losing information. These results allow us to present the following conclusion about the information capacity of the PG data model with respect to the RDF data model.

Corollary 1: The property graph data model subsumes the information capacity of the RDF data model.

Although our methods assume some condition for the input RDF databases, they are generic and can be easily extended (by overloading the mapping functions) to provide support for features such as inheritance and reification. Furthermore, our formal definitions will be very useful to study query interoperability [36], [38] and query preservation between RDF and PG databases (i.e. transformations among SPARQL and PG query languages). Thus, with this paper, we take a substantial step by laying the core formal foundation for supporting interoperability between RDF and PG databases.

Among the limitations of the mappings presented in this paper we can mention: the simple mapping is not suitable for RDF datasets with complex vocabularies as the common names will be merged in the resulting property graph; the generic mapping works with any RDF dataset, but the size of the output property graph will be bigger than the other two mappings; the complete mapping is suitable for any RDF dataset, but in practice, it requires the special directory to map prefixes to namespaces. A general limitation of the three mappings is that they are not able to deal with the special semantics defined by the RDF model (e.g. reification) and the inference rules supported by RDF Schema (e.g. subclass, sub-property). We plan to study these features in the future.

REFERENCES

[1] D. Alocci, J. Mariethoz, O. Horlacher, J. T. Bolleman, M. P. Campbell, and F. Lisacek, "Property graph vs RDF triple store: A comparison on glycan substructure search," *PLoS ONE*, vol. 10, no. 12, 2015, Art. no. e0144578.

[2] R. Angles, "The property graph database model," in *Proc. Alberto Mendelzon Int. Workshop Found. Data Manage. (AMW)*, vol. 2100, 2018.

[3] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma, "The linked data benchmark council: A graph and RDF industry benchmarking effort," *ACM SIGMOD Rec.*, vol. 43, no. 1, pp. 27–31, May 2014.

[4] R. Angles and C. Gutierrez, "An introduction to graph data management," in *Graph Data Management: Fundamental Issues and Recent Developments*. Cham, Switzerland: Springer, 2018, pp. 1–32.

[5] R. Angles, H. Thakkar, and D. Tomaszuk, "RDF and property graphs interoperability: Status and issues," in *Proc. Alberto Mendelzon Int. Workshop Found. Data Manage. (AMW)*, vol. 2369, 2019, pp. 1–11.

[6] R. Angles, H. Thakkar, and D. Tomaszuk. (Mar. 2020). *rdf2pg Experimental Datasets*. [Online]. Available: https://doi.org/10.6084/m9.figshare.12021156.v5

[7] J. Barrasa. *NSMNTX—Neo4J RDF & Semantics Toolkit*. Accessed: Mar. 27, 2020. [Online]. Available: https://neo4j.com/labs/nsmtx-rdf/

[8] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. *RDF 1.1 Turtle, Terse RDF Triple Language, W3C Recommendation*. [Online]. Available: https://www.w3.org/TR/turtle/

[9] V. P. Biron, M. Sperberg-McQueen, S. Gao, A. Malhotra, H. Thompson, and D. Peterson. (Apr. 5, 2012). *XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, W3C Recommendation*. [Online]. Available: http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/

[10] M. Brandizi, A. Singh, and K. Hassani-Pak, "Getting the best of linked data and property graphs: rdf2neo and the KnetMiner use case," in *Proc. Semantic Web Appl. Tools Health Care Life Sci. (SWAT4LS)*, 2018, pp. 1–11.

[11] D. Brickley and R. V. Guha. (Feb. 25, 2014). *RDF Schema 1.1, W3C Recommendation*. [Online]. Available: https://www.w3.org/TR/rdf-schema/

[12] G. Carothers. *Notation3 (N3): A Readable RDF Syntax, W3C Team Submission*. Accessed: Mar. 28, 2011. [Online]. Available: https://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/

[13] G. Carothers. (Feb. 25, 2014). *RDF 1.1 N-Quads, A Line-Based Syntax for RDF Datasets, W3C Recommendation*. [Online]. Available: https://www.w3.org/TR/2014/REC-n-quads-20140225/

[14] G. Carothers. (Feb. 25, 2014). *RDF 1.1 N-Triples, A Line-Based Syntax for an RDF Graph, W3C Recommendation*. [Online]. Available: https://www.w3.org/TR/2014/REC-n-triples-20140225/

[15] S. Das, J. Srinivasan, M. Perry, E. I. Chong, and J. Banerjee, "A tale of two graphs: Property graphs as RDF in oracle," in *Proc. Int. Conf. Extending Database Technol. (EDBT)*, 2014.

[16] R. D. Virgilio, "Smart RDF data storage in graph databases," in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2017, pp. 872–881.

[17] F. Gandon and G. Schreiber. (Feb. 25, 2014). *RDF 1.1 XML Syntax, W3C Recommendation*. [Online]. Available: https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/

[18] W3C OWL Working Group. (Dec. 11, 2012). *OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation*. [Online]. Available: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/

[19] S. Harris and A. Seaborne. (Mar. 21, 2013). *SPARQL 1.1 Query Language, W3C Recommendation*. [Online]. Available: http://www.w3.org/TR/sparql11-query/

[20] O. Hartig and B. Thompson, "Foundations of an alternative approach to reification in RDF," 2014, *arXiv:1406.3399*. [Online]. Available: http://arxiv.org/abs/1406.3399

[21] D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega, "Querying wikidata: Comparing SPARQL, relational and graph databases," in *Proc. Int. Semantic Web Conf. (ISWC)*. Cham, Switzerland: Springer, 2016.

[22] R. Hull, "Relative information capacity of simple relational database schemata," *SIAM J. Comput.*, vol. 15, no. 3, pp. 856–886, Aug. 1986.

[23] *Information Technology—Database Languages GQL*, Standard ISO/IEC JTC 1/SC 32 Data Management and Interchange, 2019. [Online]. Available: https://www.iso.org/standard/76120.html

[24] G. Klyne and J. Carroll. (Feb. 25, 2014). *RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation*. [Online]. Available: http://www.w3.org/TR/rdf11-concepts/

[25] A. Lysenko, I. A. Roznovăţ, M. Saqi, A. Mazein, C. J. Rawlings, and C. Auffray, "Representing and querying disease networks using graph databases," *BioData Mining*, vol. 9, no. 1, p. 23, Dec. 2016.

[26] M. N. Mami, D. Graux, H. Thakkar, S. Scerri, S. Auer, and J. Lehmann, "The query translation landscape: A survey," *CoRR*, no. 1910.03118, pp. 1–25, Oct. 2019.

[27] A. M. Martínez-Prieto, D. J. Fernández, A. Hernández-Illera, and C. Gutiérrez, "RDF compression," in *Encyclopedia of Big Data Technologies*. Cham, Switzerland: Springer, 2018, pp. 1–11.

[28] V. Nguyen, J. Leeka, O. Bodenreider, and A. Sheth, "A formal graph model for RDF and its implementation," 2016, *arXiv:1606.00480*. [Online]. Available: http://arxiv.org/abs/1606.00480

[29] V. Nguyen, H. Y. Yip, H. Thakkar, Q. Li, E. Bolton, and O. Bodenreider, "Singleton property graph: Adding a semantic Web abstraction layer to graph databases," in *Proc. Workshop Contextualised Knowl. Graphs (CKG) (ISWC)*, 2019, pp. 1–13.

[30] C. Parent and S. Spaccapietra, "Database integration: The key to data interoperability," in *Advances in Object-Oriented Data Modeling*. Cambridge, MA, USA: MIT Press, 2000, pp. 221–253.

[31] M. Pham and A. P. Boncz, "Exploiting emergent schemas to make RDF systems more efficient," in *Proc. Int. Semantic Web Conf. (ISWC)* Cham, Switzerland: Springer, 2016, pp. 463–479.

[32] S. Ceri, L. Tanca, and R. Zicari, "Supporting interoperability between new database languages," in *Proc. Adv. Comput. Technol., Reliable Syst. Appl.*, 1991, pp. 273–281.
[33] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen, "S2X: Graph-parallel querying of RDF with GraphX," in *Biomedical Data Management and Graph Online Querying*. Cham, Switzerland: Springer, 2015.
[34] F. J. Sequeda, M. Arenas, and P. D. Miranker, "On directly mapping relational databases to RDF and OWL," in *Proc. Int. Conf. World Wide Web (WWW)*, Apr. 2012, pp. 649–658.
[35] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, IEEE Standard 610, Jan. 1991.
[36] H. Thakkar, R. Angles, M. Rodriguez, S. Mallette, and J. Lehmann, "Let's build bridges, not walls: SPARQL querying of TinkerPop graph databases with Sparql-Gremlin," in *Proc. IEEE 14th Int. Conf. Semantic Comput. (ICSC)*, Feb. 2020, pp. 408–415.
[37] H. Thakkar, Y. Keswani, M. Dubey, J. Lehmann, and S. Auer, "Trying not to die benchmarking: Orchestrating RDF and graph data management solution benchmarks using LITMUS," in *Proc. SEMANTICS Conf.*, 2017, pp. 120–127.
[38] H. Thakkar, D. Punjani, J. Lehmann, and S. Auer, "Two for one: Querying property graph databases using SPARQL via Gremlinator," in *Proc. Joint Workshop Graph Data Manage. Exper. Syst. Netw. Data Analytics*, 2018, pp. 1–5.
[39] D. Tomaszuk, "RDF data in property graph model," in *Proc. Res. Conf. Metadata Semantics Res. (MTSR)*. Cham, Switzerland: Springer, 2016.
[40] D. Tomaszuk, R. Angles, L. Szeremeta, K. Litman, and D. Cisterna, "Serialization for property graphs," in *Proc. Int. Conf. Beyond Databases, Archit. Struct.*, 2019.
[41] D. Tomaszuk and D. Hyland-Wood, "RDF 1.1: Knowledge representation and data integration language for the Web," *Symmetry*, vol. 12, no. 1, p. 84, 2020.
[42] D. Tomaszuk and K. Litman, "DRPD: Architecture for intelligent interaction with RDF prefixes," in *Proc. Workshop Decentralizing Semantic Web (ISWC)*, 2018, pp. 1–8.
[43] R. Zhang, P. Liu, X. Guo, S. Li, and X. Wang, "A unified relational storage scheme for RDF and property graphs," in *Proc. Int. Conf. Web Inf. Syst. Appl.* Cham, Switzerland: Springer, 2019, pp. 418–429.

RENZO ANGLES received the bachelor's degree in systems engineering from the Universidad Católica de Santa María, Arequipa, Peru, and the Ph.D. degree in computer science from the Universidad de Chile, in 2009.

In 2013, he carried out Postdoctoral Research at the Department of Computer Science, VU University Amsterdam, as part of his participation in the Linked Data Benchmark Council Project. He is currently an Assistant Professor with the Department of Computer Science, Universidad de Talca, Chile. He participates as a Researcher at the Millennium Institute for Foundational Research on Data, Chile. His research interests lie in the intersection of graph databases and the Semantic Web. Specifically, he works in the theory and design of graph query languages and the interoperability between RDF and graph databases.

HARSH THAKKAR received the B.Eng. degree in computer engineering from the L. D. College of Engineering, Ahmedabad, India, in 2011, and the M.Tech. degree in computer science from NIT Surat, Surat, India, in 2013. He is currently pursuing the Ph.D. degree with the University of Bonn, Germany.

He is a Marie Skłodowska-Curie Alumni with the University of Bonn. He also works as a Subject Matter Expert Consultant with OSTHUS GmbH, Aachen, Germany. He applies semantic technologies in the interdisciplinary field of life sciences for solving real-world data-centric problems. His research interests include graph and RDF data management, benchmarking, graph query languages, and question answering.

DOMINIK TOMASZUK received the M.Sc. degree in computer science from the Bialystok University of Technology, Poland, in 2008, and the Ph.D. degree in computer science from the Warsaw University of Technology, Poland, in 2014. He is currently a Researcher with the Institute of Informatics, University of Białystok, Poland. His current researches focus on Semantic Web, RDF, property graphs, NoSQL databases, and cheminformatics.