LPG-based Ontologies as Schemas for Graph DBs

Stefano Ferilli[1], Domenico Redavid[1] and Davide Di Pierro[1]

[1]University of Bari – Department of Computer Science, Via E. Orabona, 4, Bari, 70125, Italy

Abstract

Graph DBs are an emerging NoSQL technology that is boosting the opportunity of data handling based on interconnection and processing of single instances, rather than batch processing as usual in traditional relational DBs. Differently from relational DBs, the most prominent graph DB, Neo4j, is schema-less and based on the LPG graph model. We propose the definition and uses of ontologies as schemas, which would also enable high-level (logical) automated reasoning on the data. The graph model adopted by standard approaches to ontologies in Computer Science is incompatible with the LPG model. So, we propose a technology, called GraphBRAIN, specifically designed to exploit the full representational power of LPGs, still having a mapping to standard ontological approaches. GraphBRAIN also allows to apply different schemas on one underlying graph, representing different but inter-related views on the same data, and to combine schemas. This paper describes the formalism and outlines its possible applications. Development and implementation of the technology is ongoing, and a prototype is available and running.

Keywords

Graph Databases, Ontologies, Labeled Property Graphs

1. Introduction

New opportunities in data storage and handling have been brought about by the recent development of graph databases, a kind of NoSQL DBs. There are major differences between traditional relational DBs and graph DBS. E.g., the latter aim at optimizing element-driven data browsing rather than batch processing as in the former. Also, the former are based on the 'table' metaphor, while the latter are based on the 'network' metaphor. The relevance of the graph-based approach to DB technology nowadays is witnessed by many big players in the industry developing their own, proprietary and special-purpose, solutions: e.g., Google's 'Knowledge Graph', Facebook's 'Social Graph' and Twitter's 'Interest Graph'. A more general-purpose solution is Microsoft Research's 'Graph Engine' (previously known as 'Trinity') [1]. The most popular graph DB according to DB-Engines[1] is Neo4j [2]. It is being adopted by many big companies and governmental organizations for several different and relevant use cases, including Recommendation, Biology, Artificial Intelligence and Data Analytics, Social Networks, Data Science and Knowledge Graphs[2]. Neo4j adopts the Labeled Property Graphs (LPGs) model [3]. In LPGs, both nodes and arcs may have names (called *labels* for nodes and

SEBD 2022: The 30th Italian Symposium on Advanced Database Systems, June 19-22, 2022, Tirrenia (PI), Italy
stefano.ferilli@uniba.it (S. Ferilli); domenico.redavid1@uniba.it (D. Redavid); davide.dipierro@uniba.it (D. Di Pierro)
0000-0003-1118-0601 (S. Ferilli)

[1]https://db-engines.com/en/ranking
[2]See https://neo4j.com/use-cases/

types for arcs) and can store *properties* represented as key/value maps. Labels usually represent classes, nodes represent class instances, types represent relationships, and arcs represent relationship instances. Each node may have many labels, while each arc may have at most one type. Many arcs, possibly labeled with the same type, may exist between the same pair of nodes. In Neo4j, nodes and arcs are associated with unique identifiers. Neo4j comes with a powerful query language (Cypher) and extensive libraries for advanced data manipulation (APOC).

Neo4j is schema-less: the user may apply any label/type or property to each single node or arc (only simple 'constraints' may be defined to bias the DB content). While ensuring great flexibility, this causes the lack of a clear semantics for the data, and obviously tampers their interpretability and interoperability. This motivated us to develop a formalism for expressing data schemas for LPG-based graph DBs (and specifically Neo4j), so that only data compliant with the schemas can be stored, as in traditional DBs. In particular, we propose to provide schemas in the form of formal ontologies. In fact, according to [4], an *ontology* is "a formal, explicit specification of a shared conceptualization", and thus building an ontology and designing the conceptual model of a DB (e.g., in the form of an E-R diagram) are basically the same activity. Using formal ontologies as schemas we may enjoy the advantages of DBMSs (scalability, storage optimization, efficient handling, mining and browsing of the data, etc.) and LPGs (flexibility, expressive power), while also allowing to carry out high-level logical reasoning on the data. As emphasized in [5], this is an important feature because formal, automated reasoning is much more powerful than the DB's query language. E.g., ontological reasoning tasks include consistency/completeness/correctness checks, instance retrieval, classification, and query answering [6]; rule-based reasoning enables multiple inference strategies (e.g., deduction, abduction, argumentation, etc.), and combinations thereof. In fact, several ready-to-use reasoners are available. This means upgrading from the 'Data Base' (DB) perspective to the 'Knowledge Base' (KB) perspective, investigated in Artificial Intelligence (AI). More specifically, applying an ontology as the data model to the data one obtains a so-called *Knowledge Graph* (KG, a kind of KB) [7]: **ontology + data = knowledge graph** [8]. Since the graph model adopted by standard AI approaches to formal ontologies is partially incompatible with LPGs, in [9] we proposed the GraphBRAIN technology, specifically designed to fully exploit the representational power of LPGs. The core of this technology is a formalism for expressing schemas for LPGs (Neo4j) as ontologies. While the structure of the formalism, and its correspondence to the standard ontological model adopted in the literature, have been described in [9], here we will describe it from a more practical viewpoint and from a more DB-oriented perspective, referring to a new, slightly modified version of the formalism.

In the following, after introducing in Section 2 the basics and related works about formal ontologies and graph DBs, Section 3 describes our formalism for interfacing the two technologies, whose opportunities and current status of implementation are discussed in Section 4.

2. Basics and Related Work

A standard formalism for expressing ontologies and KGs is the Web Ontology Language, OWL[3], for which a number of reasoners is available[4]. Operational uses of OWL are typically based on

[3] `https://www.w3.org/OWL/`

[4] `http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/`

the Resource Definition Framework (RDF)[5]. An RDF Graph is a collection of RDF Triples of the form *(Subject, Predicate, Object)* representing directed arcs where the Subject, Predicate, and Object are atomic values (Uniform Resource Identifiers –URIs– or, in the case of the Object, also a literal value). Triplestores (or 'Semantic Graph Databases') are DBMSs specifically focusing on RDF Data, and thus not optimized for generic data handling, like standard DBMSs. Among them, GraphDB may work schema-less or using an RDF ontology as schema. LPGs (and Neo4j) provide more general structure than RDF graphs [5][6]. Most notably, in LPGs nodes and arcs may carry information, which ensures a much more compact structure than RDF graphs (the estimated decrease in number of nodes is of up to one order of magnitude), and improves efficiency (especially in browsing-intensive tasks such as Social Network Analysis or Graph Mining algorithms) and readability[7]. In particular, RDF cannot attach properties to instances of relationships (this may be solved by *reification*, i.e., turning relationships into entities, at the cost of an increased number of nodes/arcs and less readability, or by using annotations, that are not visible to reasoners). Also, RDF cannot distinguish different occurrences of the same relationship between the same pair of nodes; Neo4j can, by assigning unique identifiers to arcs.

The need, but limited adoption, of logic-based Knowledge Representation for the development of KGs is pointed out in [10]. E.g., [11] stores the Freebase KG in Neo4j, but it does not use ontologies as DB schemas, and focuses on simple querying, not on reasoning. Also SciGraph[8] clearly states that creating ontologies and supporting reasoning are not its goals. In contrast, ontologies are the core of our approach. In investigating the mapping of ontologies or KGs to LPGs (or to Neo4j specifically), most works in the literature adopt an 'OWL-centric' perspective. While LPGs can obviously store atomic values in their nodes and arcs, we aim at fully exploiting their representational power and flexibility, and thus our approach is 'LPG-centric'. E.g., [12] studies the expressiveness and complexity of the Shape Expression Schema (ShEx) formalism for RDF. [13] discusses how ontological schemas can be applied to Neo4j graphs whose labeled edges belong to a number of predetermined classes. Some works limit the portion that can be mapped. OWL2LPG[9] identifies specific kinds of queries expressible in Neo4j that should be more performant than using reasoners, and translates the ontology, not the data. VirtualFlyBrain[10] translates only "a well defined subset" of OWL 2 EL ontologies into Neo4j and back that preserve entailments and annotations (not the syntactic structure). Other approaches change the LPGs or RDF models to allow a mapping. E.g., [14] redefines the PG model, while OWLStar[11] and the formal mapping proposed in [15] are based on RDF*, an extension of RDF (and thus not compliant with standard reasoners). We are interested in solutions that fully exploit the LPG model, and in what and how can be mapped onto standard RDF, so as to reuse available reasoners.

The main approach adopted in the Neo4j community[12] consists in importing into Neo4j the RDF triples specifying the ontology as they are. So, the ontology and the data, albeit disjoint,

[5] https://www.w3.org/RDF/

[6] https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/

[7] Albeit not directly related to data storage and management, readability may be important when a portion of the graph is to be graphically displayed for humans.

[8] https://github.com/SciGraph/SciGraph/wiki/Neo4jMapping

[9] https://protegeproject.github.io/owl2lpg

[10] https://github.com/VirtualFlyBrain/neo4j2owl

[11] https://github.com/cmungall/owlstar

[12] https://neo4j.com/blog/ontologies-in-neo4j-semantics-and-knowledge-graphs/

coexist in the same graph, almost like in relational DBs schemas are stored within the DBMS itself. Since the schema and the data are to be handled in totally different ways, we keep them apart, as in relational DBMSs, where they are stored in different DBs. Ontological reasoning on the imported ontology is carried out using Cypher queries. However, no formal discussion is provided about what kinds of reasoning can be mapped onto graph DB queries. Usually it is quite simple (e.g., navigation of the subclass hierachy), and hardly comparable to those provided by state-of-the-art ontological reasoners. In any case, implementation of these reasoning facilities is still in charge of the applications accessing the DB. Also, this solution does not prevent data that are not compliant with the intended ontology to be inserted into the DB.

Since data representation constrained to using RDF triples does not necessarily make sense out of the Automated Reasoning field, we start from the 'standard' DB perspective based on LPGs, and aim at providing the DB with a schema that may also enable formal reasoning on its contents. In our vision, KB designers must design the data schemas, expressed in the form of ontologies for LPGs, that will drive all subsequent accesses to the DB. By referring to a schema, the applications will commit to be compliant with it, as in traditional databases. Like in Triplestores, this will ensure a tight integration between the data and the schema. As opposed to Triplestores and most of the cited works, the data/instances (stored in the graph DB) are kept apart from the schema/ontology (specified in a file external to the DB, using an ontology representation format). We leverage this separation between the data repository and the data schema to obtain the additional opportunity of applying different (but compatible) schemas to the same DB. Indeed, each schema may represent a different, partial view on the same data, allowing to limit or expand the possible interactions depending on specific needs, and adding flexibility to our solution. Again, this is not even thinkable in Triplestores. [16, 17] deal with automated inference of graph schemas, pointing out difficulties of such a task. We don't aim at inferring the DB schema, but at providing graph DB users a formalism that allows to specify them and to apply high-level reasoning on the DB data.

3. GraphBRAIN DB Scheme Format

As said, the objective of this work is to propose a formalism to endow LPG-based graph DBs with a schema that ensures a clear semantics to the information they contain and drives its management and interpretation. In our approach the schemas are expressed as formal ontologies whose elements are expressed by elements of the LPG model according to Table 1. We call this technology *GraphBRAIN*, and its schemas GBSs (for GraphBRAIN Schemas). GBSs are expressed as XML files, whose syntax is specified by an associated DTD file. In the GBS formalism, all names of domains, entities, relationships and attributes may consist of letters or decimal digits only. We suggest to avoid digits and build multi-word names by juxtaposing their constituent words, using an uppercase letter for the first letter of each subsequent word.

In the following we will describe the XML tags and structure GBS formalism through a 'general' schema running example, including general classes and relationships that are likely to be reused by most other schemas. An excerpt of the GBS describing the 'general' schema is reported in Figures 1 and 2. Let us first describe the general structure of the files. After the standard XML file heading, the main tag, enclosing the whole schema, is **domain**, expressing

Table 1
Correspondence between ontology, relational DB and LPG elements

OWL*	Relational DB	LPG
owl:Ontology	schema	label
owl:Class	entity table name	label
owl:ObjectProperty	relationship table name	type
owl:Individual	entity table tuple	node
object property URI	relationship table tuple	arc
owl:DatatypeProperty	entity table attribute	node property
	relationship table attribute	arc property
	entity table key	node id
	relationship table key	arc id

* (OWL namespace `http://www.w3.org/2002/07/owl`)

in attribute *name* the name of the domain that the schema is describing (which must start with a lowercase letter) and in attribute *author* the author of the schema. Immediately inside this tag, two sections are specified, introduced by tags **entities** and **relationships** and defining the portion of schema specifying nodes and arcs in the graph, respectively.

Figure 1 focuses on the specification of nodes in the graph DB. The schema assumes an implicit universal entity, and starts describing the hierarchy of entities starting from classes at level 1 (we will call these *top entities*). Top entities in the schema will be used to label nodes in the graph[13]. So, all nodes with the same label will be considered as instances of the same top entity, much like tuples in the same entity table in a relational DB. Top entities should be selected by the DB designer so as to be described by sufficiently different properties. Each entity is enclosed in an **entity** tag, specifying its name in the *name* attribute. In GBS formalism Entity names must start with an uppercase letter. In the 'general' schema, the following top entities are provided for: 'Artifact' (any physical product of human labour), 'Collection' (any collection of any kind of entities), 'ContentDescription', 'Document' (any kind of multimedia document), 'Event', 'IntellectualWork', 'Item' (specific instances of Artifacts), 'Organization', 'Person', 'Place', and 'User'. In Figure 1, the description of entity 'Artifact' has been partially exploded as an example of the XML sub-structure associated to entities. We note two sections.

The section enclosed by tag **attributes** is mandatory, and specifies the attributes that are applicable to the top entity (at least one attribute must be specified for each top entity). This constrains nodes labeled with that top entity to take only these attributes. Each attribute is described by attributes *name* (which must start with a lowercase letter in GBS formalism), *mandatory* (saying whether it must be specified for each node), *datatype* (specifying the type of the attribute), and *distinguishing* (meaning that the attribute is useful to distinguish different items having the same values for mandatory attributes). The set of mandatory and distinguishing attributes acts as a 'logical' key for the entity instances, useful for human users to distinguish them (syntactically, the node id automatically assigned by Neo4j is sufficient to acts as a unique

[13]This is for compliance with arcs, allowing only one type. On nodes, additional labels can be added to specify domains for which the node is relevant. This does not generate ambiguity, since domain names start with a lowercase letter, entity names with an uppercase one.

```
<?xml version="1.0"?>
<domain name="general" author="graphbrain" version="1.0">
   <entities>
      <entity name="Artifact">
         <attributes>
            <attribute name="name" mandatory="true" datatype="string"/>
          <attribute name="description" distinguishing="true" mandatory="false" datatype="string"/>
         </attributes>
         <taxonomy>
            <value name="Artwork"> ... </value>
            <value name="Handicraft"> ... </value>
            <value name="IndustrialWork">
               <taxonomy>
                  <value name="Component"> ... </value>
                  <value name="Device">
                     <attributes>
                        <attribute name="partNumber" mandatory="false" datatype="string"/>
                     </attributes>
                  </value>
                  ...
               </taxonomy>
            </value>
         </taxonomy>
      </entity>
      <entity name="Collection"> ... </entity>
      <entity name="ContentDescription"> ... </entity>
      <entity name="Document"> ... </entity>
      <entity name="Event"> ... </entity>
      <entity name="IntellectualWork"> ... </entity>
      <entity name="Item"> ... </entity>
      <entity name="Organization"> ... </entity>
      <entity name="Person"> ... </entity>
      <entity name="Place"> ... </entity>
      <entity name="User"> ... </entity>
   </entities>
   <relationships> ... </relationships>
</domain>
```

Figure 1: Excerpt of sample GBS file (focus on entities)

key). Attribute names ‘specialization’ and ‘notes’ are reserved, since they are automatically added by GraphBRAIN to all top entities. In the ‘general’ schema, the top entity ‘Artifact’ is described by two attributes, both of type string: ‘name’ (mandatory) and ‘description’ (optional, but distinguishing), plus ‘specialization’ and ‘notes’ (an optional string).

The section enclosed by tag **taxonomy** is optional, and describes the hierarchy of subclasses for the top class. Each immediate sub-class is enclosed by tag **value**, reporting its name in the *name* attribute. The behavior of this tag is just like the **entity** tag: it may specify additional attributes for the subclass (enclosed in an **attributes** tag), to be added to the attributes of all its superclasses in the hierarchy, and further immediate sub-classes (enclosed in a **taxonomy** tag), recursively. In the ‘general’ schema, the top entity ‘Artifact’ has 3 sub-classes (‘Artwork’, ‘Handicraft’ and ‘IndustrialWork’), the latter of which has in turn sub-classes, one of which (‘Device’) provides for an additional attribute (‘partNumber’, an optional string) with respect to the top entity. Instances may belong to any top or intermediate class in the hierarchy, and their most specific class is specified in the *specialization* attribute (for instances of the top class, it

```
<?xml version="1.0"?>
<domain name="general" author="graphbrain" version="1.0">
   <entities> ... </entities>
   <relationships>
      <relationship name="aliasOf" inverse="aliasOf">
         <references>
            <reference subject="Category" object="Category"/>
            <reference subject="Document" object="Document"/>
            <reference subject="User" object="Person"/>
            <reference subject="Person" object="Person"/>
            <reference subject="Place" object="Place"/>
         </references>
      </relationship>
      <relationship name="attended" inverse="attendedBy">
         <references>
            <reference subject="Organization" object="Event"/>
            <reference subject="Person" object="Event"/>
         </references>
         <attributes>
            <attribute name="startDate" mandatory="true" datatype="date"/>
            <attribute name="endDate" mandatory="false" datatype="date"/>
            <attribute name="role" mandatory="false" datatype="string"/>
         </attributes>
      </relationship>
      <relationship name="belongsTo" inverse="includes"> ... </relationship>
      <relationship name="developed" inverse="developedBy"> ... </relationship>
      <relationship name="evolves" inverse="evolvedBy"> ... </relationship>
      <relationship name="expresses" inverse="expressedBy"> ... </relationship>
      <relationship name="instanceOf" inverse="hasInstance"> ... </relationship>
      <relationship name="interactedWith" inverse="interactedWith"> ... </relationship>
      <relationship name="isA" inverse="hasSubclass"> ... </relationship>
      <relationship name="knows" inverse="knownBy"> ... </relationship>
      <relationship name="owned" inverse="ownedBy"> ... </relationship>
      <relationship name="partOf" inverse="hasPart"> ... </relationship>
      <relationship name="produced" inverse="producedBy"> ... </relationship>
      <relationship name="requires" inverse="requiredBy"> ... </relationship>
      <relationship name="wasIn" inverse="hosted"> ... </relationship>
      ...
   </relationships>
</domain>
```

Figure 2: Excerpt of sample GBS file (focus on relationships)

reports the top class itself). Top class and subclass names must be unique in the whole schema.

Figure 2 focuses on the specification of arcs in the graph DB. The schema assumes an implicit universal relationship, and starts describing the hierarchy of relationships starting from level 1 (we will call these *top relationships*). Top relationships in the schema will be used to label arcs in the Neo4j graph (which admits only one type label per arc). So, all arcs with the same type will be considered as instances of the same relationships, much like tuples in the same relational table in a relational DB. As for top entities, top relationships should be selected by the DB designer so as to refer to relationships described by sufficiently different properties. Each relationship is enclosed in a **relationship** tag, specifying its name in the *name* attribute and the name of its inverse relationship in the *inverse* attribute. Relationship (and inverse relationship) names must start with a lowercase letter in GBS formalism. Some of the top relationships provided for by the 'general' schema are: 'aliasOf', 'attended', 'belongsTo', 'developed', etc. In

```
<entity name="Person">
  <attributes>
      <attribute name="surname" mandatory="true" display="true" datatype="string"/>
      <attribute name="name" mandatory="true" display="true" datatype="string"/>
      <attribute name="gender" mandatory="false" datatype="select">
         <values>
            <value name="M"/>
            <value name="F"/>
         </values>
      </attribute>
      <attribute name="bornIn" mandatory="false" datatype="entity" target="Place" />
      <attribute name="birthDate" mandatory="false" datatype="date"/>
      <attribute name="diedIn" mandatory="false" datatype="entity" target="Place" />
      <attribute name="deathDate" mandatory="false" datatype="date"/>
   </attributes>
</entity>
```

Figure 3: Sample entity with attributes of different types

Figure 2, the description of relationships 'aliasOf' and 'attended' has been partially exploded as an example of the XML sub-structure associated to relationships. We note two sections.

The section enclosed by tag **references** is mandatory, and specifies the subject and object entity of the relationship (i.e., the admitted pairs of classes for source and sink nodes of the arcs with that relationship type), using attributes *subject* and *object*, respectively. The section enclosed by tag **attributes** is as for entities, but it is not mandatory, because the relationship itself is a kind of property of the two entities it connects. A third (optional) section, enclosed by tag **taxonomy**, would allow to describe the hierarchy of subrelationships for the top relationships. As for the hierarchy of sub-entities, it is recursive, and allows to express for each sub-relationship its references, attributes and sub-taxonomy (if any), using the same tags and nested structure.

In the 'general' schema, an arc of type 'aliasOf' (to associate nodes that refer to the same thing) can be added only between pairs of nodes labeled Category and Category, Document and Document, User and Person, Person and Person, or Place and Place. It has no attributes. An arc of type 'attended' (expressing event attendance) can be added only between pairs of nodes labeled Organization and Event, or Person and Event. It has 3 attributes, 2 of type date ('startDate', mandatory, and 'endDate', optional) and one of type string (role, optional). Neither of these relationships has a taxonomy of subrelationships.

For attributes, GBS admits the following datatypes: **integer**, **real**, **boolean**, **string**, **text**, **select**, **tree**, **date**, **entity**. Of these, *integer*, *real*, *boolean*, *string*, *text* take an atomic value of the corresponding type, where *text* is intended for free text of any length, differently from *string* which has a limited maximum length that can be specified in the *length* attribute. Values of these types are stored as literal values for the corresponding DB types provided by Neo4j: Integer and Float (both subtypes of an abstract type Number), Boolean, and String.

Attributes of type *select* denote a choice in an enumeration of values, enclosed in a tag **values** with each value enclosed in a tag **value**, specifying the value in the *name* attribute. GraphBRAIN always adds a default value 'Other' to the list of values of any attribute of type *select*. This type is used in Figure 3 for attribute 'gender' of entity 'Person', that may take values 'M' or 'F' or 'Other'. Attributes of type *tree* are similar to attributes of type *select*, but indicate a choice in a tree of values. The tree can be described by allowing nested **values** tags inside **value** tags. For

```
<entities>
   <entity name="Timeline"/>
   <entity name="Year">
      <attributes>
         <attribute name="year" mandatory="true" datatype="integer"/>
      </attributes>
   </entity>
   <entity name="Month">
      <attributes>
         <attribute name="belongsTo" mandatory="true" datatype="entity" target="Year"/>
         <attribute name="month" mandatory="true" datatype="integer"/>
      </attributes>
   </entity>
   <entity name="Day">
      <attributes>
         <attribute name="belongsTo" mandatory="true" datatype="entity" target="Month"/>
         <attribute name="day" mandatory="true" datatype="integer"/>
      </attributes>
   </entity>
</entities>
<relationships>
   <relationship name="belongsTo" inverse="includes">
      <references>
         <reference subject="Year" object="Timeline"/>
      </references>
   </relationship>
   <relationship name="follows" inverse="precedes">
      <references>
         <reference subject="Day" object="Day"/>
         <reference subject="Month" object="Month"/>
         <reference subject="Year" object="Year"/>
      </references>
   </relationship>
</relationships>
```

Figure 4: Implicit entities and relationships for time handling

values of these types, the corresponding string is stored.

Attributes of type *entity* denote 1:1 relationships between an instance of the current entity and an instance of another entity (specified in the *target* attribute of the tag). So, they are stored in the DB as arcs connecting the nodes corresponding to these two instances and having the attribute name as type (note that in our proposed naming policy attribute names start with a lowercase letter, just like relationship names). E.g., in Figure 3 the birthplace of an entity Person would be modeled as a 'bornIn' attribute of type *entity* with entity Place as the target. In the graph, an arc labeled 'bornIn' will be added from the Person node to the Place node.

Finally, albeit Neo4j provides for temporal types, including Date, following [2] GraphBRAIN models attributes of type *date* as 1:1 relationships to entities 'Day' (with *year/month/day* values), 'Month' (with *year/month* values), or 'Year' (with *year* values). This allows to specify dates at different granularity, differently from Neo4j's Date type. Neo4j provides functions for Date truncation to Month or Year, but such truncations actually correspond to the first day of the month or year and thus there is no way to distinguish whether a date like 2020/01/01 actually refers to the specific day or is a truncation for the month (2020/01) or year (2020). Using attribute 'belongsTo' (of type *entity*), 'Day' nodes are connected to the corresponding 'Month' nodes,

```
<?xml version="1.0"?>
<domain name="lam">
   <imports>
      <import schema="general"/>
   </imports>
   <entities> ... </entities>
   <relationships> ... </relationships>
</domain>
```

Figure 5: Excerpt of sample GBS file (focus on imports)

which in turn are connected to the corresponding 'Year' nodes, which are finally all connected to the 'Timeline' nodes. Arcs of type 'follows' may be added and maintained between adjacent days, months or years in the DB, so as to easily extract from the DB time intervals and associated information. Figure 4 shows the portion of ontology defining temporal elements.

Each GBS schema is meant to describe one domain. However, GraphBRAIN may apply several schemas to one graph DB, each representing a specific perspective on the data in the DB, and providing a partial view of its contents (e.g., in order to limit access to the DB contents for some users or applications). Shared entities and relationships in those schemas act as bridges that allow to connect the data from the various domains/perspectives, and enforce cross-fertilization of the data, which is important for AI applications. GraphBRAIN considers entities and relationships in different schemas as the same (i.e., shared) if they have the same name. Shared entities and relationships may have different attributes in different schemas, reflecting the different perspectives associated with the different domains. However, attributes that are present in different domains must have the same datatype in all of them.

Schemas may also be combined, provided that their elements (entities and relationships) are compatible. By *compatible* we mean that elements having the same name in the different schemas must be in the taxonomy of the same top element, and their attributes having the same name must have the same datatype, too. The other attributes, or non-shared elements, can be freely defined. Moreover, given two elements A and B, it cannot happen that A is a specialization of B in one domain, and *vice versa* in another. GraphBRAIN can integrate the taxonomies even if intermediate sub-elements are present in either taxonomy but not in the other. Schema composition allows to define more complex schemas that describe wider domains (e.g., elements in the 'general' schema might be reused by a schema concerning libraries, archives and museums, or 'lam' domain). In addition to allowing reuse, this would also foster standardization of the definitions. Schemas are combined in GBS using an optional section enclosed by tag **imports**, located in the XML file before the entities and relationships sections. Each schema to be imported is specified using tag **import**, with attribute *schema* for the name of the schema. E.g., in Figure 5 the schema 'lam' imports schema 'general'. Schemas are imported in the order specified by the sequence of **import** tags. For each of them, its elements are progressively added in the suitable places of the taxonomies, provided they are compatible with the existing elements. Finally, elements defined in the **entities** or **relationships** sections of the importing schema are added. Since it may happen that some elements of the imported schemas are not needed in the current domain, **delete** tags (with attribute *element* to specify the element to be deleted) allow to remove them from the overall ontology.

4. Discussion & Conclusions

Graph DBs are a NoSQL kind of DBs which is gaining momentum for research and industrial applications. Neo4j is the most relevant graph DB available nowadays, based on the LPG graph model. Neo4j is schema-less, which ensures great flexibility, but at the cost of a lack of a clear semantics for the graph contents, and of losing interpretability and interoperability of the data. As a solution we propose the use of ontologies as schemas, which as a nice side-effect would also enable high-level (logical) automated reasoning on the data in addition to standard DB data manipulation. Since the standard approach to ontologies in Computer Science is incompatible with the LPG model, we proposed the GraphBRAIN technology, which is specifically LPG-oriented, while still having a mapping to standard ontological approaches. GraphBRAIN wraps the graph DB: it takes as input a GBS schema and controls all interactions, allowing the external applications to manipulate and consult only information items that are compliant with the schema. GraphBRAIN allows many schemas to be applied to the same graph to express different domains or perspectives on its content. Shared entities and relationships among these sthemas enforce cross-fertilization of the knowledge from the corresponding domains.

This paper described the GraphBRAIN schema formalism and the opportunities it provides. Development and implementation of the technology is ongoing, and a prototype Web application is available and running to comfortably build, browse and edit both the schemas expressed in this format and the corresponding data (see http://193.204.187.73:8088/GraphBRAIN/, where also the schema used as a running example in this paper can be seen in full) [18]. Research is ongoing to further expand the expressive power of the GBS formalism, and to implement its features. At the same time, schemas are being built and refined for different domains, and data are being entered for them, so as to support several applications. The current prototype includes schemas for the domains of 'tourism', 'food', 'computing' (concerning computing devices and their history), and 'lam' (concerning libraries, archives and museums). Practical use cases in these domains can be found in [19, 20]. Finally, an investigation of AI algorithms that may leverage the power of the GraphBRAIN framework is being carried out.

References

[1] B. Shao, H. Wang, Y. Li, Trinity: A distributed graph engine on a memory cloud, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13), 2013, pp. 505–516.

[2] I. Robinson, J. Webber, E. Eifrem, Graph Databases, 2nd ed., O'Reilly Media, 2015.

[3] M. Rodriguez, P. Neubauer, Constructions from dots and lines, Bul. Am. Soc.Info. Sci. Tech. 36 (2010) 35–41.

[4] R. Studer, R. Benjamins, D. Fensel, Knowledge engineering: Principles and methods, Data & Knowledge Engineering 25 (1998) 161–198.

[5] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al., The Future is Big Graphs: A Community View on Graph Processing Systems, Communications of the ACM 64 (2021) 62–71.

[6] S. Rudolph, Foundations of Description Logics, in: Reasoning Web. Semantic Technologies

for the Web of Data: 7th International Summer School 2011, Tutorial Lectures, Springer, 2011, pp. 76–136.
[7] L. Ehrlinger, W. Wolfram, Towards a definition of knowledge graphs, in: SEMANTICS 2016: Posters and Demos Track, volume 1695 of *CEUR Workshop Proceedings*, 2016.
[8] B. Schrader, What's the Difference Between an Ontology and a Knowledge Graph? (White Paper), Technical Report, Enterprise Knowledge, 2020.
[9] S. Ferilli, Integration Strategy and Tool between Formal Ontology and Graph Database Technology, Electronics 10 (2021).
[10] M. Krötzsch, Ontologies for Knowledge Graphs?, in: Proceedings of the 30th International Workshop on Description Logics, volume 1879 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1879/invited2.pdf.
[11] M. Elbattah, M. Roushdy, M. Aref, A.-B. M. Salem, Large-scale ontology storage and query using graph database-oriented approach: The case of Freebase, in: 7th International Conference on Intelligent Computing and Information Systems (ICICIS), IEEE, 2015, pp. 39–43.
[12] S. Staworko, I. Boneva, J. E. L. Gayo, S. Hym, E. G. Prud'Hommeaux, H. Solbrig, Complexity and Expressiveness of ShEx for RDF, in: 18th International Conference on Database Theory (ICDT 2015), 2015.
[13] G. Drakopoulos, A. Kanavos, P. Mylonas, S. Sioutas, D. Tsolis, Towards a framework for tensor ontologies over Neo4j: Representations and operations, in: 8th International Conference on Information, Intelligence, Systems & Applications, IISA 2017, Larnaca, Cyprus, August 27-30, 2017, IEEE, 2017, pp. 1–6.
[14] H. Chiba, R. Yamanaka, S. Matsumoto, G2GML: Graph to Graph Mapping Language for Bridging RDF and Property Graphs, in: The Semantic Web – ISWC 2020, Springer, Cham, 2020, pp. 160–175.
[15] O. Hartig, Foundations to Query Labeled Property Graphs using SPARQL, in: Joint Proceedings of the 1st International Workshop On Semantics For Transport and the 1st International Workshop on Approaches for Making Data Interoperable co-located with 15th Semantics Conference (SEMANTiCS 2019), volume 2447 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019.
[16] B. Groz, A. Lemay, S. Staworko, P. Wieczorek, Inference of Shape Graphs for Graph Databases, in: 25th International Conference on Database Theory (ICDT 2022), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
[17] H. Lbath, A. Bonifati, R. Harmer, Schema inference for property graphs, in: EDBT 2021-24th International Conference on Extending Database Technology, 2021, pp. 499–504.
[18] S. Ferilli, D. Redavid, The GraphBRAIN System for Knowledge Graph Management and Advanced Fruition, in: Foundations of Intelligent Systems, volume 12117 of *LNAI*, Springer, Berlin, Heidelberg, 2020, pp. 308–317.
[19] S. Ferilli, D. Redavid, An Ontology and a Collaborative Knowledge Base for History of Computing, in: 1st International Workshop on Open Data and Ontologies for Cultural Heritage (ODOCH-2019), volume 2375 of *CEUR Workshop Proceedings*, 2019, pp. 49–60.
[20] S. Ferilli, D. Redavid, An Ontology and Knowledge Graph Infrastructure for Digital Library Knowledge Representation, in: Digital Libraries: The Era of Big Data and Data Science, volume 1177 of *CCIS*, Springer, Berlin, Heidelberg, 2020, pp. 47–61.