An API for Ontology-driven LPG Graph DB Management

Davide Di Pierro[1,†], Stefano Ferilli[1,*,†]

[1]Università degli Studi di Bari Aldo Moro, Via Edoardo Orabona, 4, 70125 Bari BA

Abstract

Graph databases are nowadays in widespread use for both scientific and industrial purposes. Graph models are also exploited in Artificial Intelligence to represent Knowledge Bases, in which the mere storage of data is paired with formal ontologies that allow us to interpret them, express constraints on them, and reason about them. While the standard practice for Knowledge Graphs is based on the RDF graph model, recently a new framework was proposed, named GraphBRAIN, based on the Labelled Property Graph model and more oriented to the DB perspective. Here we describe an API for enacting the new framework and make it available to serve the needs of independent applications interested in using knowledge bases. The API will allow ontology-compliant access to the data.

Keywords

Graph Databases, Ontologies, Knowledge Representation, Knowledge Graph, API,

CEUR-WS.org/Vol-3478/paper49.pdf

1. Introduction

Currently, there is extensive work to enhance the software with technologies and methods for handling the growing, diverse data. Although graph DBs offer interpretability and scalability, their scheme-less nature hinders access to data semantics, crucial in this landscape. Attempts to address this issue have involved introducing constraints or limitations. However, proper schemes would have greater power in determining data representation in the DB. An even stronger solution would involve utilizing formal ontologies as DB schemes, enabling the application of advanced AI solutions and technologies. Unfortunately, the standard graph model used in research on ontologies is different than the Labelled Property Graph (LPG) model adopted by current leading graph DBs such as Neo4j. The GraphBRAIN framework [1] has been proposed as a solution: it can define ontologies specifically suited for the LPG model, and use them as schemes for Neo4j DBs [2], obtaining a fully-fledged knowledge base.

In this paper, we propose the first prototype of an API by which applications can use GraphBRAIN technology to consult or manipulate a knowledge base and reason with it. The API allows the users to define, modify or reuse ontologies, access and manipulate the DB in compliance with a given ontology and run advanced functions on the data, such as automated reasoning

SEBD 2023: 31st Symposium on Advanced Database System, July 02–05, 2023, Galzignano Terme, Padua, Italy

*Corresponding author.

†These authors contributed equally.

davide.dipierro@uniba.it (D. Di Pierro); stefano.ferilli@uniba.it (S. Ferilli)

http://ara.di.uniba.it/ (S. Ferilli)

0000-0002-8081-3292 (D. Di Pierro); 0000-0003-1118-0601 (S. Ferilli)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

and mining. Specifically concerning data access and manipulation, the API allows the users to issue queries using the standard Cypher language of the Neo4j DB, but automatically checking their compliance to the specified ontology/scheme[1] in a way that is fully transparent to the user. We consider this a relevant added value to standard graph DB management since it allows users to create their own schemes and query their data as in traditional DBMS settings. Other features are also available in the API. GraphBRAIN ontologies and instances can be translated into Semantic Web (SW) standards, mapped onto existing SW resources, and passed to SW reasoners to perform classical tasks such as consistency checking and instance checking, or infer new knowledge. Still, being GraphBRAIN ontologies independent of the SW approaches, they can be mapped also onto other formalisms, enabling further kinds of automated reasoning (e.g., rule-based MultiStrategy reasoning).

The rest of the paper is organized as follows. After reporting related work in Sec. 2 and some background in Sec. 3, in Sec. 4 we recall the GraphBRAIN formalism to define ontologies to be used as schemes for LPG-based graph DBs. Then, Sec. 5 describes the API we have developed and shows examples of its use. Sec. 6 concludes the paper and outlines future work issues.

2. Related Works

To the best of our knowledge, there is no previous attempt to integrate graph DBs and ontologies into an API. However, in many applications, the possibility of providing an API interfacing with a DB for supporting integration, management and sharing of knowledge is given.

A great interest exists in the field of medicine, in order to bring common knowledge to presumably work together for a certain objective. [3] provided a web-based portal and an API to connect to a relational database for the virology community. the applications Jmol and STRAP were implemented to visualize and interact with the virus molecular structures and provide sequence–structure alignment capabilities. After extended curation practices that maintain data uniformity, a relational database based on a scheme for macromolecular structures and the APIs provided enhance the ability to perform structural bioinformatics experiments and studies on virus capsids. [4] published different versions of APIs and datasets to enrich the medics community to study bacteria. [5] built a high-performance database of proteins. They provided an API to query simplifying the SQL statements by means of object-oriented structures. [6] developed an ontology of chemical entities of biological interest, named ChEBI, which contains a wealth of chemical data. Unlike many other resources, ChEBI is human-curated, providing a reliable, non-redundant collection of chemical entities and related data.

[7] developed the Spectral Physics Environment for Advanced Remote Sensing (SPEARS) application programming interface. It is a local thermal equilibrium (LTE) spectral modelling optimized for synthesizing optical spectra from any combination of fundamental spectroscopic databases. The API facilitates the user who has to specify the following parameters: the thermodynamic state of the species (i.e. temperature, pressure, concentration), the model geometry, and the model resolution.

The added value of APIs is collaboration, that is, the possibility of creating a community which voluntarily contributes to the expansion of some source of knowledge. The technological

[1]In this paper, we will use terms 'scheme' and 'ontology' interchangeably.

leaps moved the discipline of biology into a more information-based one, giving rise to new fields of study like bioinformatics. In parallel to this, algorithms and hardware need to evolve accordingly, to satisfy these growing needs. The success in solving these issues led to the development of some projects like the 1000 Genomes Project for human [8], the 1001 Genomes Project for Arabidopsis [9], and Ensembl [10].

Apart from APIs, there are different works worth to be mentioned when discussing the potential of ontologies and graph DBs.

As said, the use of graph structures is a common solution to store ontological definitions. Elbattah et al [11] compared the use of Neo4j with other DBs and/or structures, analysing the pros and cons of this well-known strategy.

Gong et al. [12] reported the case of an RDF-based ontology that is stored in an LPG-based graph, and the results are prominent when the focus is on querying, storing and scaling without worrying about inference and consistency issues.

Neo4j (and graph DBs in general), differently from ontological settings, are not intended to perform reasoning on the data. As a consequence, a DB will never be enough for keeping peculiarities of ontological purposes. To supply DBs in this task, additional features (plugins) may come in handy. Neo4j is strongly optimised to perform queries and navigate the graph rapidly. Hence, when moving from one perspective to another one will lose advantages. That is the reason why in our previous works we described the separation of concerns strategy so as to distinguish instances from schemes. Here [13] there is a survey of works related to this topic.

A traditional approach to impose constraints on the data, e.g. in Neo4j, is to use specific queries to limit or make mandatory some attributes [14]. However, in our perspective, this may be considered limiting because we cannot express relationships between higher-level concepts, expressed by labels of nodes in the graph.

Taking everything into consideration, with this work we aim to abstract from specific fields of applications, but also to leverage a very flexible database, to easily integrate different sources of knowledge. This API will allow researchers to create their own LPG-based KG, so as to take advantage of the integration of schemes without losing performance in the manipulation and management of instances.

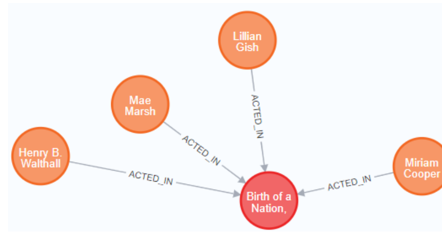
3. Background

Graph DBs may adopt different graph models and various providers offer different implementations [15]. A major difference is in their structure. A standard graph-based model for Knowledge Representation in AI is the Resource Description Framework (RDF), where the graph is described as a set of atomic triples $\langle S, p, O \rangle$, each establishing a relationship (arc) p between a subject resource S and an object resource or literal value O (nodes). A Property Graph [16] is a directed, edge-labelled, attributed, multi-graph [17]:

$$G = (V, I, \omega, D, J, \gamma, P, Q, R, S)$$

where V is a set of vertices, I a set of vertices identifiers, $\omega : V \rightarrow I$ a function from vertices to their identifiers, D a set of directed edges, J a set of edges identifiers, $\gamma : D \rightarrow J$ a function from edges to their identifiers, P (resp., R) the vertices (resp., edges) attributes domain and Q

Figure 1: Actors starring in Birth of a Nation



(resp., S) the domain for allowed vertices (resp., edges) attributes values. It allows representing properties on nodes, in the form of attribute-value pairs. One cannot make any assumption on the attributes available in a node unless additional constraints are introduced. The idea is to ensure flexibility: integration of knowledge is facilitated when there is no need of satisfying strict node or relationship constraints. Indeed, real-world data are often noisy, incorrect or incomplete, likely causing a cumbersome process in the integration of knowledge. Labelled PGs (LPGs) [18] evolve the PG model, so that nodes and arcs may have labels. To give an analogy with the traditional relational model, assigning a label to a node corresponds to stating that it is an instance of the class specified in the label. A node may be associated with multiple labels because in real-world scenarios not all classes are disjoint. One of the most famous graphs DBs, Neo4j[2], adopts the LPG model.

Among all NoSQL DB, [19] models, graph DBs are gaining momentum both in academia and industry applications thanks to their flexible, intuitive, and interpretable structure. In Graph DBs, the data are stored in nodes, and nodes (or data blocks) may be associated with one another via connections called edges [20], relationships which provide index-free adjacency. They are described by the following main characteristics:

- the data and/or the scheme are represented by graphs, or by data structures generalizing the notion of a graph (hypergraphs or hypernodes).
- manipulation is obtained by transformations on a graph, or by operations whose main primitives are on graph features like paths, neighbourhoods, subgraphs, graph patterns, connectivity, and graph statistics.
- integrity constraints enforce data consistency. These constraints can be grouped into scheme-instance consistency, identity and referential integrity, and functional and inclusion dependencies [21].

An example of data manipulation and constraining is given in Fig. 1, that shows a fragment of the Movie dataset [3] concerning the actors starring in a specific movie.

We can add a constraint about nodes labelled 'Movie', saying that the URLs of movies are unique, using Cypher:

```
CREATE CONSTRAINT unique_rel FOR (n:Movie) REQUIRE n.url IS UNIQUE
```

[2]https://neo4j.com/
[3]https://github.com/neo4j-graph-examples/recommendations

After the activation of this constraint, trying to insert a node having an already present URL would return the following:

```
Node (4987) already exists with label 'Movie' and
property 'url' = 'https://themoviedb.org/movie/618'
```

In Neo4j there are other types of constraints regarding arcs, but not nearly as powerful as ontology definitions.

Compared to previous DB models (e.g., the relational one), graph DBs are more suitable when the relationships matter more than the single data blocks. Also, the way these relationships become visible and interpretable (by names) is relevant.

Graph DBs may provide a solution to a major challenge for DB research, which is to provide a scalable architecture to support Big Data [22] domains and exploit all state-of-the-art approaches in Artificial Intelligence (AI) leveraging graph structures. The Big Data problem refers not only to the volume of data but also to their variety. Indeed, 'volume' and 'variety' are two of the five V's [23] that distinguish problems to be included in big data scenarios from those that are not. Moreover, graph DBs often represent an obvious choice when dealing with data that are inherently organized as graphs, such as networks.

3.1. GraphBRAIN

GraphBRAIN is a knowledge base management system aimed at joining the efficiency in data handling provided LPG graph DBs, specifically Neo4j, with the expressive power of ontologies. Since Neo4j is schemaless, the ontologies would act as schemes and guide all the general CRUD operations available in traditional DB settings. They would also help to keep consistency in the graph as well as to discern the type of information that can be retrieved. Last but not least, using ontologies as schemes provide high-level and formal interpretations of the data, and enables advanced and semantic-aware automated reasoning and mining functions on them.

While standard approaches to ontology description in AI adopt the RDF graph model, GraphBRAIN defined its own formalism specifically built around the more powerful LPG model [2]. It can express the standard ontological concepts: (a hierarchy of) entities (or 'classes' in SW terminology), (a hierarchy of) relationships (or 'object properties' in SW terminology), user-defined types, and entity attributes (or 'datatype properties' in SW terminology). Differently from standard SW approaches, it can express attributes on both entities and relationships, it can label the nodes, and it can set several instances of the same relationship between the same pair of instances (thanks to unique identifiers automatically assigned to every node and arc in the graph). More specifically, the following mapping is established between elements of the graph and elements of the ontologies/schemes:

- entity instances are represented as nodes, labelled with the most specific class (only) to which they belong;
- relationship instances are represented as arcs, labelled with the most specific relationship (only) they express;
- literal-valued attributes of entity *and relationship* instances are represented as node/arc properties.

GraphBRAIN offers many services for handling both schemes and data. Regarding schemes, it allows users to create, modify and merge ontologies. Merging is based on the unique name assumption, that is, concepts sharing the same name must refer to the same concept.

Users may search, browse and modify the database content. They can also explore the available ontologies and define their own ontologies, from scratch or as a variation of existing ones. They can also run several graph mining algorithms to obtain relevant indications on the graph content. Examples of the functionalities provided by GraphBRAIN are:

- assess the relevance of nodes and arcs in the graph, and extract the most relevant ones;
- extract a portion of the graph that is relevant to some specified starting points (nodes and/or arcs);
- extract frequent patterns and associated sub-graphs;
- predict possible links between nodes;
- retrieve complex patterns of nodes and relationships;
- translate portions of the graph into a more understandable form.

Our schemes aim to also provide an abstract middle layer between representations. Storing schemes in an intermediate format (e.g. XML) gives us the possibility of moving towards other semantic directions. As an example, the intermediate format may be exported into other (formal) languages to exploit, for instance, reasoning capabilities. Formal schemes are prone to be exploited in the field of the SW [24]. In [25], a preliminary mapping between our schemes and SW schemes (serialised in the Web Ontology Language (OWL) [26]) is shown.

The SW is just one perspective that we can exploit for reasoning purposes. We are able to translate schemes into a first-order logical language (Prolog), or into the OWL language. In this perspective, by adding information about instances in the same formalism, we can implement all reasoning techniques on Knowledge Bases. All approaches come under the umbrella term "multistrategy reasoning" [27].

4. GraphBRAIN Scheme Specification

We have mentioned the capability of our solution to deal with schemes without injecting them into the graph. Schemes are stored in separate files, written in a standardised formalism.

Let us now quickly describe how GraphBRAIN ontologies/schemes are formally represented. More detailed information can be found in [28]. Compared to that description, here we refer to a newer version that allows also us to specify sub-relationships and user-defined types. GraphBRAIN schemes need identifiers for the concepts that are represented. By convention. they consist of uppercase letters, lowercase letters or decimal digits only.

GraphBRAIN schemes are expressed as XML files, organized according to the specifications of a so-called GraphBRAIN Scheme (GBS) format. A scheme consists of an XML file whose tags provide all the kinds of components in GBS ontology. In order to simplify the description of the components, we will write XML tag names in boldface, XML tag attribute names in italics and entity or relationship names in small caps. Text in plain typeface reports comments useful to understand the various elements and their behaviour. For ease of reference, we report in Figure 2 an excerpt of a GBS scheme defining the ontology for the domain of 'computing', and

Figure 2: Excerpt of GBS file

```
<domain name="computing" author="stefano" version="1">
    <imports>
        <import scheme="general" />
    </imports>
    <types>
        <type name="logisticOperator" datatype="select">
            <values>
                <value name="NAND"/>
                <value name="AND"/>
                <value name="XOR"/>
            </values>
        </type>
    </types>
    <entities>
        <entity name="Award">
            <attributes>
                <attribute name="name" mandatory="true" distinguishing="true" datatype="string"/>
                <attribute name="date" mandatory="false" datatype="date"/>
                <attribute name="description"
                mandatory="false" datatype="text"/>
            </attributes>
        </entity>
        <entity name="ElectronicComponent">
        [...]
            <taxonomy>
                    [...]
                    <value name="Chip">
                        <taxonomy>
                            <value name="Logic">
                            [...]
                            <value name="Decoder"/>
                            [...]
                            <value name="Gate">
                                <attributes>
                                    <attribute name="function" mandatory="false" datatype="logisticOperator"/>
                                </attributes>
                            </value>
                            [...]
            </taxonomy>
            [...]
        </entity>
        [...]
    </entities>
    <relationships>
        <relationship name="acquired" inverse="acquiredBy">
            <references>
                <reference subject="Organization" object="Organization"/>
            </references>
            <attributes>
                <attribute name="date" mandatory="false" datatype="date"/>
                <attribute name="originalPrice" mandatory="false" datatype="real"/>
            </attributes>
        </relationship>
        [...]
        <relationship name="mayReplace" inverse="mayBeReplacedBy">
            <references>
                <reference subject="Component" object="Component"/>
            </references>
            <taxonomy>
                <value name="replace" inverse="isReplaced"/>
            </taxonomy>
        </relationship>
        [...]
    </relationships>
```

will use it as a running example in the rest of this section. As visible in Figure 2, the root tag in the XML is **domain** and specifies the name of the scheme in its *name* attribute.

Schema Modularity Each GBS scheme is intended to describe one domain. Still, to enforce modular knowledge representation, some schemes might want to reuse the elements of other, more 'basic' schemes (e.g., a scheme describing the Cultural Heritage domain might want to reuse the scheme describing the Library domain). The first (optional) section of a GBS

file, enclosed in tag **imports**, allows doing this. Each scheme to be imported is specified in the *scheme* attribute of an **import** tag (e.g., in Figure 2 the scheme ‘general’ is imported into ‘computing’). Should some elements of the imported scheme be unnecessary, they might be removed using **delete** tags in this section (a feature not used in Figure 2). Imports are transitive, that is, an imported scheme may in turn import others. Obviously, loops in importing are not allowed. When defining an element (entity or relationship) with the same name as a previously imported one, GraphBRAIN will assume they refer to the same concept, and try to merge them if compatible or raise an error. Merging involves taking the union of their attributes (attributes with the same name must be of the same type) and of their sub-classes or sub-relationships (it is forbidden that a class X is a subclass of a class Y in one scheme and *vice-versa* in the other).

In the next (again, optional) section of a GBS file, enclosed in the **types** tag, new datatypes can be defined for use in the entity or relationship attributes. Again, this supports reuse and modularity. Each user-defined type is enclosed in a **type** tag. Its name, specified in the *name* attribute, must be unique and will be referenced in the entity or relationship attributes. A user-defined type consists of an enumeration of values of two possible kinds (reported in the *datatype* attribute): ‘select’ (representing a plain list of values) or ‘tree’ (representing a hierarchy of values). The lists of values are enclosed in tag **values**, with each value specified in a tag **value** under attribute *name*. For ‘tree’ datatypes, tag **value** may recursively include nested **values** tags. In Figure 2, a ‘select’ type logisticOperator is defined to represent logistic operators, and used for attribute function of entity ElectronicComponent.

Entities and relationships The last two sections in a GBS file, enclosed in tags **entities** and **relationships**, allow specifying the classes and relationships of the ontology and their generalization/specialization hierarchies. The universal entity Entity and the universal relationship Relationship are the roots of their corresponding hierarchies and are abstract. We will call *top-level* entities and relationships starting from the first level. Top-level entities (resp. relationships) are specified using tag **entity** (resp. **relationship**). Direct specializations of a class (resp. relationship) can be specified in an optional **taxonomy** tag (used only if the entity or relationship has direct specializations), each enclosed in a **value** tag. The **taxonomy** tag is recursive (**value** tags may in turn include it if they have further direct specializations in the hierarchy. In Figure 2, top-class Award has no sub-classes, while top-class ElectronicComponent has a direct sub-class **Chip**, that in turn has several direct sub-classes. All classes and relationships, except abstract ones (for which attribute *abstract*=“true”) may have instances.

Attributes Each entity or relationship may have attributes, enclosed in the **attributes** tag. It is mandatory for non-abstract entities (their instances must be described by some attribute) and optional for relationships (a relationship carries information in its very linking of two instances). Each attribute is described using an **attribute** tag, which reports, among other information, its name (in attribute *name*) and type (in attribute *datatype*). It can take values from a primitive data type (if *datatype* is one of ‘integer’, ‘real’, ‘boolean’, ‘string’, ‘text’, ‘date’), an enumeration (if *datatype* is ‘select’ or ‘tree’, following the same specification as for user-defined data types), a user-defined type (if *datatype* is the name of a user-defined type) or an instance of another class in ontology (if *datatype* is ‘entity’, in which case attribute *target* specifies the

class). Attributes of type 'entity' are represented in the graph as arcs connecting the class instance to the other entity's instance; the same happens for dates, where the target entity is one of the GraphBRAIN pre-defined classes Day, Month or Year. These actually represent 1:1 (i.e., functional) relationships between an instance of the entity and an instance of the target entity. As usual in generalization taxonomies (e.g., in the Object Oriented programming paradigm [29] or in ontologies [30]), sub-classes and sub-relationships inherit the attributes of all their generalizations.

Subject-Object pairs Relationships must also express their subject and object entities [31] (or 'domain' and 'range', using SW terminology). Relationships with the same name may be established between different pairs of entities. The set of pairs is enclosed in a **references** tag, within which each pair is specified by a **reference** tag, whose *subject* and *object* attributes specify the two classes. Note that the pairs are independent of each other, so they actually identify different relationships (in SW approaches they should get different names). Sub-relationships inherit all references from the parent (clearly, subjects and objects of sub-relationships must be the same as the subjects or objects of their parents, or subclasses thereof). In Figure 2, relationship mayReplace may be established between two ElectronicComponents and has a sub-relationship replace.

5. API

While a Web application is available to interactively query, explore and modify a GraphBRAIN knowledge graph[4], our objective is to make the technology available to other applications. If powered by GraphBRAIN, these applications might boost their effectiveness, and provide new and advanced support to their users. To this aim, we propose an API providing all GraphBRAIN functionality, from ontology definition to ontology-based data access and manipulation, from semantic aware data mining to automated human-like reasoning on the available knowledge. The first prototype of the API was developed in Java language, obtaining a jar that can be imported as a library in Java applications and used to obtain the GraphBRAIN functionality.

5.1. Functionality

Development of the GraphBRAIN API is an ongoing project. We will list and describe here the functions that are already fully implemented. A first requirement was that the API must provide at least all the functions that are available in the interface, as a direct query or by a combination of queries. Every function in the API must guarantee consistency in the DB. Hence, creations and updates of nodes, relationships, and attributes are guided by the scheme previously created through the API itself.

We implemented first the most general CRUD operations, common to all DB applications, both for the instances (the content of the DB):

- **login**: login to a specific instance of Neo4j by providing the URL and credentials (user, password).

[4]Available at `http://193.204.187.73:8088/GraphBRAIN/`

- **create node**: create a node by assigning it a label (and optionally properties).
- **connect two nodes with an arc**: create an arc by specifying properties that identify the subject and object of the relationship.
- **create arc(s)**: same functionality as the previous one but not limiting the case to a 1 to 1 relationship.
- **delete node/arc**: delete node/arc by specifying its properties.
- **filter nodes/arcs**: return list of nodes/arcs identified by specific properties.
- **get node/relationship info**: return info about a specific node/relationship by specifying known properties.

and for scheme/ontology definition:

- **load scheme**: load an entire GBS or OWL scheme.
- **create/rename/delete class**: add, rename or remove a class to/in/from the ontology.
- **create/rename/delete relationship**: add, rename or remove a relationship to/in/from the ontology.
- **create attribute**: create an attribute by specifying at least its name and type.
- **add/remove attribute to/from class**: add/remove a previously-generated attribute to a class.
- **add/remove attribute to/from relationship**: add/remove a previously-generated attribute to a relationship.

The possibility of loading an external scheme/ontology is of paramount importance to not create from scratch large state-of-the-art ontologies. The API allows importing ontologies in GBS formalism or in the standard OWL language. OWL may express more complex constraints but in our settings we just import what the GB scheme allows, ignoring the rest. Moreover, the API can export the ontology and (a portion of) the graph to OWL/RDF. This allows us to exploit OWL reasoners to perform common inference operations like instance checking, ontology consistency and so on. For instance, the signature of the methods to load an entire schema is:

```
(i) public DomainData(String domainName, File file)
(ii) public static DomainData readOWL(String filePath)
```

DomainData is the Java class that stores an entire schema. It stores two hierarchies (classes and relationships), and for each class (resp., relationship) information about their attributes. Both methods return a DomainData object: (i) takes the name of the domain and a GBS file, (ii) takes the path of the OWL source.

5.2. Technicalities

The API requires the (local or online) URL of a Neo4j instance to be queried, and associated access credentials. Then, given a GBS schema and a query in Cypher language, it presents the result to the user only if the query is scheme-compliant, just like in traditional DBs. For the sake of simplicity, but without loss of generality, we will show how it works for the following query, which represents the basic building block of all Cypher queries:

```
MATCH (n:lN{pN})-[r:tR{pR}]->(m:lM{pM}) RETURN n.qN, r.qR, m.qM
```

meaning: “Return properties *qN*, *qR*, *qM* of nodes *n*, *m* and arc *r*, respectively, such that *n* and *m* are connected by arc *r* and *n*, *m* and *r* have the specified labels *lN*, *tR*, *lM* and property values *pN*, *pR*, *pM*, respectively”.

The general strategy is as follows: the variables (aliases) used in the query to name the subject, relationship and object are identified, and the labels and properties associated with them are extracted. Then, the ontology is consulted to check if relationship *tR* is valid between classes *lN* and *lM* or their generalizations, in which case the ontology is consulted again to check if the properties are valid for the three components (or for their generalizations). If so, the query is finally issued, and the results are collected and returned.

The API parses the query to extract its building blocks and creates a map structure in which the keys are the aliases and the attributes store all information about them. So, the first objective is identifying the aliases. The API first parses the query to distinguish the pattern-matching clause, introduced by the keyword MATCH, from the results clause, introduced by the keyword RETURN. The aliases are located in the former. So, the former is further split, based on a pair of square brackets that enclose the relationship. In the resulting 3 parts, the aliases are identified because they are the identifiers that precede the colon **:**. The labels of the aliases are recognized because they follow the colon; in turn, they are followed by the properties, recognized because delimited by braces **{** and **}**. Then, additional properties associated with the aliases are located in the results clause. They are associated with their respective aliases, connected to them by a dot **.** and separated by commas **,**. These properties are added to the list of properties of the aliases, if not already present.

5.3. Use Examples

Here we show two examples of applications of our API. These examples were run on the online DB available with the GraphBRAIN Web Application, and the ontology/scheme **retrocomputing**, also available in the Web Application, which concerns the history of computing.

The former example regards the execution of a query on an existing DB according to a given scheme. Using the API we loaded the ontology by specifying the path of its GBS file. This is the straightforward way when we already have a scheme. After loading, we may modify the scheme but we focus now on the graph querying. This is the query example we gave:

```
MATCH (n:Person {name:'Donald', surname:'Michie'})-[r:developed {role
    :'author'}]->(m:Book{language:'ita'}) RETURN n.name, n.surname,r.
    role,m.title
```

asking the DB for all the books in the Italian language written by Donald Michie as the author. The API parses the query and identifies 3 variables: two nodes *n*, *m* and an arc *r*, such that the label of *n* is **Person**, the label of *m* is **Book** and the type of *r* is **developed**. It looks at the ontology for a relationship **Person.developed.Book** and realizes that, while it does not exist, **Person**-**Document** is a valid subject-object pair for relationship **developed**, where **Document** is a superclass of **Book**. So far, so good. Now, for each such element, the ontology is queried to check whether the specified attributes are valid. In fact, the ontology confirms that attributes **name** and **surname** are specified for entity **Person**, attributes **language** and **title** are specified for entity **Document** (and thus inherited by subclass **Book**), and attribute **role** is specified for relationship **developed**. So, the query is run and the following result is returned:

```
Record<{n.name: "Donald", n.surname: "Michie", r.role: "author",
   m.title: "Intelligenza Artificiale e futuro dell'uomo"}>
```

In the other example, we show how we can move from the graph DB perspective to the SW one by mapping schemes and instances onto an OWL/RDF-based representation, and executing an SW reasoner. For the latter task, we used the OWL API [5], which is compatible with Java, and makes available various SW reasoners, including Pellet [32]. Since mapping a large number of instances may require a big computational effort, without loss of generality, we exported only a portion of the whole graph. To demonstrate the versatility of our solution, the second use case exploits an external ontology regarding tourism [6]. To this KB, some Italian instances have been added in order to create value from reasoning. The exported portion was obtained starting from a specified node (purposely chosen to belong to the **Hotel** class specified in the ontology) and taking all its neighbouring within a fixed maximum number (k) of hops from it. Here we chose $k = 2$. Then we performed the following reasoning tasks, concerning both the exported instances and the scheme: ontology consistency, subclasses of a class, atoms inferences.

Here is an extract of the results provided by Pellet.

```
Is the ontology consistent? True
Is it consistent? True

Subclasses of Site :
Nodeset[Node( <http://www.semanticweb.org/ontologies/Hotel.owl#Park> ), Node( <http://www.semanticweb.org/ontologies
   /Hotel.owl#Hotel_3_Stars> ), Node( <http://www.semanticweb.org/ontologies/Hotel.owl#Fire_Station> ), Node( <
   http://www.semanticweb.org/ontologies/Hotel.owl#Hotel> ), Node( <http://www.semanticweb.org/ontologies/Hotel.
   owl#Subway_Station> ),
...

Axiom :- SubClassOf(<http://www.semanticweb.org/ontologies/Hotel.owl#WI-FI> <http://www.semanticweb.org/ontologies/
   Hotel.owl#Public_facility>)
Is axiom entailed by reasoner ? :- true
Is axiom contained in ontology ? :- true
No. of Explanations :- 1
Explanation :-
[SubClassOf(<http://www.semanticweb.org/ontologies/Hotel.owl#WI-FI> <http://www.semanticweb.org/ontologies/Hotel.owl
   #Public_facility>)]
```

The reasoner confirmed KB consistency, built using the online ontology and schema-compliant data loaded into GraphBRAIN. It correctly recognized subclasses and provided axioms based on ontological hierarchy since no external rules were introduced.

6. Conclusion and Future Work

Graph DBs are becoming increasingly popular for industrial and research purposes. However, their lack of schemes hampers data interpretability and semantic operations. To address this, the GraphBRAIN framework offers a solution by introducing LPG-compatible ontologies for Neo4j, transforming the DB into a comprehensive knowledge base. The paper presents an API that allows applications to easily construct, refine, and reuse ontologies, enabling seamless access and manipulation of data. Future developments include expanding functionalities, language support (Python), and a full-fledged integration with standard Semantic Web tools.

[5]https://owlapi.sourceforge.net/
[6]http://www.di.uniba.it/ lisi/ontologies/OnTourism.owl

Acknowledgments

This work was partially supported by the projects FAIR – Future AI Research (PE00000013), spoke 6 – Symbiotic AI, and CHANGES – Cultural Heritage Active innovation for Next-GEn Sustainable society (PE00000020), Spoke 3 – Digital Libraries, Archives and Philology, under the NRRP MUR program funded by the NextGenerationEU.

References

[1] S. Ferilli, D. Redavid, The graphbrain system for knowledge graph management and advanced fruition, in: Foundations of Intelligent Systems: 25th International Symposium, ISMIS 2020, Graz, Austria, September 23–25, 2020, Proceedings, Springer, 2020, pp. 308–317.

[2] S. Ferilli, D. Redavid, D. D. Pierro, Lpg-based ontologies as schemas for graph dbs, in: Proceedings of the SEBD 2022: The 30th Italian Symposium on Advanced Database Systems (SEBD 2022), volume 3194 of *Central Europe (CEUR) Workshop Proceedings*, 2022, pp. 256–267.

[3] M. Carrillo-Tripp, C. M. Shepherd, I. A. Borelli, S. Venkataraman, G. Lander, P. Natarajan, J. E. Johnson, C. L. Brooks III, V. S. Reddy, Viperdb2: an enhanced and web api enabled relational database for structural virology, Nucleic acids research 37 (2009) D436–D442.

[4] G. Funke, F. Renaud, J. Freney, P. Riegel, Multicenter evaluation of the updated and extended api (rapid) coryne database 2.0, Journal of Clinical Microbiology 35 (1997) 3122–3126.

[5] E. R. Jefferson, T. P. Walsh, T. J. Roberts, G. J. Barton, Snappi-db: a database and api of structures, interfaces and alignments for protein–protein interactions, Nucleic acids research 35 (2007) D580–D589.

[6] N. Swainston, J. Hastings, A. Dekker, V. Muthukrishnan, J. May, C. Steinbeck, P. Mendes, libchebi: an api for accessing the chebi database, Journal of Cheminformatics 8 (2016) 1–6.

[7] C. Murzyn, E. Jans, M. Clemenson, Spears: A database-invariant spectral modeling api, Journal of Quantitative Spectroscopy and Radiative Transfer 277 (2022) 107958.

[8] N. Siva, 1000 genomes project, Nature biotechnology 26 (2008) 256–257.

[9] D. Weigel, R. Mott, The 1001 genomes project for arabidopsis thaliana, Genome biology 10 (2009) 1–5.

[10] D. Rios, W. M. McLaren, Y. Chen, E. Birney, A. Stabenau, P. Flicek, F. Cunningham, A database and api for variation, dense genotyping and resequencing data, BMC bioinformatics 11 (2010) 1–10.

[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: 2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS), 2015, pp. 39–43. doi:10.1109/IntelCIS.2015.7397191.

[12] F. Gong, Y. Ma, W. Gong, X. Li, C. Li, X. Yuan, Neo4j graph database realizes efficient storage performance of oilfield ontology, PloS one 13 (2018) e0207595.

[13] D. Di Pierro, S. Ferilli, D. Redavid, Lpg-based knowledge graphs: A survey, a proposal and current trends, Information 14 (2023) 154.

[14] J. Pokornỳ, M. Valenta, J. Kovačič, Integrity constraints in graph databases, Procedia Computer Science 109 (2017) 975–981.
[15] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazan, J. L. Larriba-Pey, Survey of graph database performance on the hpc scalable graph analysis benchmark, in: Web-Age Information Management: WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010 Revised Selected Papers 11, Springer, 2010, pp. 37–48.
[16] R. Angles, The property graph database model., in: AMW, 2018, pp. 1–10.
[17] S. Jouili, V. Vansteenberghe, An empirical comparison of graph databases, in: 2013 International Conference on Social Computing, IEEE, 2013, pp. 708–715.
[18] M. Y. Kpiebaareh, W.-P. Wu, S. Bayitaa, C. R. Haruna, L. Tandoh, User-connection behaviour analysis in service management using bipartite labelled property graph, in: Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, 2019, pp. 318–327.
[19] S. Sharma, R. Shandilya, S. Patnaik, A. Mahapatra, Leading nosql models for handling big data: a brief review, International Journal of Business Information Systems 22 (2016) 1–25.
[20] G. ShefaliPatil, A. Bhatia, Graph databases-an overview, 1Student, ME Computers, Terna College of Engg, Navi Mumbai 2 (2014) 657–660.
[21] R. Angles, C. Gutierrez, Survey of graph database models, ACM Computing Surveys (CSUR) 40 (2008) 1–39.
[22] S. Sagiroglu, D. Sinanc, Big data: A review, in: 2013 international conference on collaboration technologies and systems (CTS), IEEE, 2013, pp. 42–47.
[23] T. L. Nguyen, A framework for five big v's of big data and organizational culture in firms, in: 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 5411–5413.
[24] H.-G. Kim, Semantic web, Recuperado de http://semanticweb. org/wiki/Main_Page. html (2003).
[25] D. Di Pierro, D. Redavid, S. Ferilli, Linking graph databases and semantic web for reasoning in library domains, in: Proceedings of the 18th Italian Research Conference on Digital Libraries, volume 3160, 2022, pp. 1–12. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85134261683&partnerID=40&md5=7da94dc9c8902e78b717199709f28991.
[26] D. Allemang, J. Hendler, Semantic web for the working ontologist: effective modeling in RDFS and OWL, Elsevier, 2011.
[27] S. Ferilli, Gear: A general inference engine for automated multistrategy reasoning, Electronics 12 (2023) 256.
[28] S. Ferilli, Integration strategy and tool between formal ontology and graph database technology, Electronics 10 (2021) 2616.
[29] T. Rentsch, Object oriented programming, ACM Sigplan Notices 17 (1982) 51–57.
[30] B. Smith, Ontology, in: The furniture of the world, Brill, 2012, pp. 47–68.
[31] S. Decker, S. Melnik, F. V. Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, I. Horrocks, The semantic web: The roles of xml and rdf, IEEE Internet computing 4 (2000) 63–73.
[32] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical owl-dl reasoner, Journal of Web Semantics 5 (2007) 51–53.