SPARQLing Neo4J

Ezequiel José Veloso Ferreira Moreira
University of Minho, Braga, Portugal
pg38413@alunos.uminho.pt

José Carlos Ramalho
Department of Informatics, University of Minho, Braga, Portugal
jcr@di.uminho.pt

Abstract

The growth experienced by the internet in the past few years as lead to an increased amount of available data and knowledge obtained from said data. However most of this knowledge is lost due to the lack of associated semantics making the task of interpreting data very hard to computers.

To counter this, ontologies provide a extremely solid way to represent data and automatically derive knowledge from it.

In this article we'll present the work being developed with the aim to store and explore ontologies in Neo4J. In order to achieve this a web frontend was developed, integrating a SPARQL to CYPHER translator to allow users to query stored ontologies using SPARQL. This translator and its code generation is the main subject of this paper.

2012 ACM Subject Classification Information systems → Web Ontology Language (OWL); Information systems → Ontologies; Computing methodologies → Ontology engineering; Information systems → Graph-based database models

Keywords and phrases SPARQL, CYPHER, Graph Databases, RDF, OWL, Neo4J, GraphDB

Digital Object Identifier 10.4230/OASIcs.SLATE.2020.17

Category Short Paper

1 Introduction

With the rise of the internet of things and the growing interconnectivity of our world the necessity to store data in an organised and related faction as well as associating semantic metadata to gather useful knowledge from it becomes paramount.

As such ontologies, formal specification of knowledge understandable by both machines and humans, become more relevant and worthy of study. In particular web ontologies based in RDF, RDFS and OWL.

Neo4J is a well known graph database with an attached query language to explore them: CYPHER. While a graph is not an ontology, an ontology can be represented in a graph: RDF graphs are an example of this. The challenge in this work is to use CYPHER to query not just a graph but a knowledge graph. We will describe the work made in order to enable Neo4J to use SPARQL, a query language for ontologies, explaining the steps necessary to translate queries from SPARQL to CYPHER.

In the following sections we give an introduction to graph databases and graph/ontology query languages. We present the work being carried out and we finish with some conclusions and ideas for future work.

2 Graph Databases

Graph databases are purpose-built to store and navigate relationships. Relationships are first-class citizens in graph databases, and most of the value of graph databases is derived from these relationships.

Graph databases use nodes to store data entities, and edges to store relationships between entities. An edge always has a start node, end node, type, and direction, and an edge can describe parent-child relationships, actions, ownership, and the like. There is no limit to the number and kind of relationships a node can have.

A graph in a graph database can be traversed along specific edge types or across the entire graph. In graph databases, traversing the joins or relationships is very fast because the relationships between nodes are not calculated at query times but are persisted in the database. Graph databases have advantages for use cases such as social networking, recommendation engines, and fraud detection, when you need to create relationships between data and quickly query these relationships.

In this work we are using a graph database, Neo4J, to store OWL ontologies.

An OWL ontology can be serialized as a list of triples. Each triple having the form $(subject, predicate, object)$ in which *subject* is an element of the ontology, *predicate* represents either a property of the element or a relationship with another element(dependant on whether the object is a value or an element of the ontology), and *object* stands for a scalar value or another element's identifier.

We can map subjects and objects to nodes and predicates to edges, thus allowing us to use a graph database to store an ontology.

2.1 Neo4J

Neo4j[6] is an open-source NoSQL native graph database that utilises a generic graph structure to store it's data.

Some of its core features are constant time on depth and breadth traversal and a flexible schema that can be changed at any time. Furthermore it's easy to learn and use thanks to its simple and intuitive UI and language used.

Neo4J stores its data using a generic graph structure: each element is a node that can have properties associated and be related to other elements. These relations can have properties associated with them besides their type and, as such, be considered a special kind of node in the database

To explore stored data, Neo4J uses CYPHER, a declarative query language similar to SQL but optimised for graphs. Unlike SPARQL, CYPHER deals with a generic graph structure(as opposed to an ontology), and as such is a lower level query language if we try to query an ontology.

Furthermore, Neo4J does not support inference (since a graph does not have axioms), and while there are attempts at creating one(like GraphScale[3]) there is currently no fully realised project to resolve this lack.

Up to the version 3 (the latest release was version 4) Neo4J only permitted the creation of one database per installation,a handicap that was solved in the latest version.

2.1.1 Neosemantics

A very important note to make before we move on is that due to not supporting SPARQL and using a generic graph schema, Neo4j does not contain, by default, any way of importing RDF ontologies into it's databases.

To solve this we are using an extension called Neosemantics[7], under the Neo4J Labs[4] project that aims to extend Neo4J capabilities. This extension provides a way to manipulate ontologies using Neo4J through a simple mapping between the ontology and the graph schema.

2.2 GraphDB

GraphDb [2] is an open-source NoSQL graph database that stores its data using RDF graphs.

Some of its core features are native ontology manipulation tools, multiple ontology management, direct query creation and results using a basic Web text editor and a powerful inference engine capable of fully taking advantage of the knowledge gathering capabilities of ontology stored data.

Unlike Neo4J, GraphDB is built specifically for ontologies , and as such provides far more tools for its manipulation (that could not exist in Neo4J due to the more general data schema it employs) and is far more efficient at processing them.

Furthermore it fully implements SPARQL, a query language designed by W3C and becoming the standard in quering ontologies, giving it a far more robust language to deal with the type of data it stores than the more general CYPHER language.

3 Ontology Query Languages

Both Neo4J and GraphDB allow users to interact directly with their databases using queries, written in a given query language(CYPHER and SPARQL, respectively).

Despite having very distinct syntaxes, the actions that they perform over the database are so similar that a partial mapping has already been created as part of this work.

3.1 CYPHER

CYPHER [1] is a declarative query language inspired by SQL and designed for querying, updating and administering graph databases. It is used almost exclusively in Neo4J.

Many of its keywords are inspired by SQL, with pattern matching functionnalities similar to SPARQL, as well as other minor inspirations(such as Python list semantics and query structure that mimics English prose).

The structure of the language is borrowed from SQL, with queries being built up from various clauses that are chained together and feed intermediate results between themselves.

Two simple CYPHER query examples are provided below.

Query C1 – returns every node in the database that contains a property *code* with value 200:

```
match(n) where n.code = 200 return n
```

Query C2 – returns all nodes in the database that contain a relation $hasFather$ with another node in the database:

```
match(n)-[r:hasFather]->(m) return n
```

3.2 SPARQL

SPARQL [14] is a W3C standard language for the manipulation and querying of RDF graphs.

It fully utilises set theory in it's evaluation strategy[12] and provides an extremely solid standard language, as noted by its generalised use anywhere where RDF is involved.

There are four type of SPARQL queries:

SELECT – Select queries are the most used type of SPARQL queries, they are used to retrieve information from RDF graphs;

ASK – Ask queries are boolean, their only purpose is to check if some triple is in the graph;

CONSTRUCT – Construct queries are a powerful tool and can serve multiple purposes. They are composed by two parts, a SELECT clause and a CONSTRUCT clause. SELECT clause enables us to identify specific situations in the graph and the CONSTRUCT clause enables us to program the generation of new triples;

INSERT and DELETE – INSERT and DELETE queries enable us to implement transactions on the ontology. They were added to SPARQL in the last revision.

In the following examples, the same two queries previously coded in CYPHER are now presented in SPARQL syntax.

Query S1 – returns every node in the database that contains a property *code* with value 200:

```
select ?s where { ?s :code "200" }
```

Query S2 – returns all nodes (URIs) in the database that contain a relation *hasFather* with another node in the database:

```
select ?son where { ?son :hasFather ?o }
```

4 From SPARQL to CYPHER, featuring PEG.js

In order to achieve our goals we must create a way to automatically translate from SPARQL to CYPHER. To do this, we need a SPARQL grammar that we will decorate with semantic actions to translate between the 2 languages.

To specify those semantic actions we must look at how ontology data is stored in Neo4J and see how it can be translated into CYPHER from a SPARQL query.

We can observe that Neosemantics stores data from an ontology using a very simple method[8]: for each triple it creates a node for the subject if it does not exist already(using its uri as an identifier), and then checks if the object is an URI or not. If it is, it creates a node with that URI (if it does not exist already) and then creates a relation between the subject node and the object node, where the relation type equals the URI of the predicate. If it isn't, it creates a node with that URI (if it does not exist already) and then adds a property to that node whose name is the predicate and whose value is the object. This process is repeated for every triple in the ontology.

With this knowledge we can already observe that a query to obtain data from a given element of the ontology will require a match with a node that contains the specific uri in its properties. In the same vein, we can see that due to the way that properties and relations work in Neo4J we will need to find a way to differentiate a property from a relation and gather the data accordingly.

To do all this, we first need a way to create a parsing grammar that is compatible with Javascript (in order to be easily integrated in our application). We have decided to use PEG.js[9], a parser generator for Javascript that provides fast parsing and good error handling, mainly due to its easiness of integration with other tools or apps.

With this tool, an initial version of a grammar that translates SPARQL into CYPHER has been developed. This translation grammar is based upon the W3C documentation that describes the syntax rules for the SPARQL language[13](modified to work with PEG.js's way of parsing) with added semantic actions that allow it to transform valid SPARQL queries into valid CYPHER queries.

4.1 Query translation example in depth

To give an idea of how translation works, we present the following example.

Let's take the following query in SPARQL and translate it using our grammar:

Query S3 – returns all pairs $(elem, father)$ where $father$ is the URI of every element with a $: hasFather$ relation with the subject with id $http://www.my_ontology.com\#elem$. We can see the declaration of a namespace as default:

```
prefix : <http://www.my_ontology.com#>
select * where{ :elem :hasFather ?father }
```

Before going into the translation note that: the translation showed here had small changes made to them that do not alter the query in any other way then to make it prettier for this paper, namely the fact that all variables that are generated by the grammar that are not directly related to the results we want to get use a pseudo RNG algorithm([11]) in their names to guarantee that will never collide with user set variables.

This does, however, make them quite hard to understand in a paper. As such, they have been altered to gVX, where X is an integer that denotes a unique internal variable.

Query C3 – query **S3** translated to CYPHER by the grammar:

```
match(gV1) where gV1.uri ="http://www.my_ontology.com#elem"

unwind [key in keys(gV1) where key = "http://www.my_ontology.com#hasFather" | [
    gV1[key],null,key]] + [(gV1)-[gV2]->(customVar_father) where type(gV2) = "
    http://www.my_ontology.com#hasFather" | [customVar_father.uri,
    customVar_father,type(gV2)]] as gV3
with *,gV3[0] as father,gV3[1] as customVar_father

with father where (father is not null)
RETURN *
```

The first line of the translation contains a match with a node that contains a specific uri in the ontology(in this example,$< http://www.my\_ontology.com\#elem >$). This is necessary since in order to obtain the triples associated to a given element of the ontology we must first match with it in the database. As such, this first line corresponds to us matching with $: elem$ in the SPARQL query.

After matching with our subject, we have to obtain the desired object that is associated to our subject and the predicate $: hasFather(http://www.my\_ontology.com\#hasFather)$. To do that, we must search for the predicates and objects associated to our subject node and find the ones that have a predicate $: hasFather$, and to do that we must use arrays and unwinds inside the query.

This unwind expression and the corresponding with expression is complex, so we'll break it down in small parts:

(1) **[key in keys(gV1) where key= "http://www.my_ontology.com#hasFather" | [gV1[key],null,key]]** – with this expression we gather all the information about triples from the ontology that are properties of our subject that follow the conditions we want (in this case, the predicate must have a value: $: hasFather$).

The gathering of the properties is done by
key in keys(gV1) where key=
"http://www.my_ontology.com#hasFather"
and the return of the data is done by
[gV1[key],null,key]
The reason we return a null value has to do with the fact we may need to join the results of this search in the properties with the search in the relations, and that search returns 3 results, only one is relevant to the other search.

(2) **[(gV1)-[gV2]->(customVar_father) where type(gV2)="http://www.my_ontology.com#hasFather" | [customVar_father.uri,customVar_father,type(gV2)]]** – with this expression we gather all the information about triples from the ontology that are relations of our subject that follow the conditions we want.
The gathering of the relations is done by
(gV1)-[gV2]->(customVar_father) where
type(gV2)="http://www.my_ontology.com#hasFather"
and the return of the data is done by
[customVar_father.uri,customVar_father,type(gV2)]
Unlike in the properties, we have to search the relations and the nodes that are associated to those relations. Because of these nodes, our return has another value to return: the node reference itself. The reason we must return this is because we might use this variable again down the line in the query, and if we do that then we can access the node directly without searching the database again, saving us a lot of time in processing the query.

(3) **(1) + (2)** – this expression allows us to gather results from all the triples that our subject has and join them in a single array of results.
The reason we must use this in this case as to do with the fact that since the grammar does not know if : $hasFather$ denotes a relationship or a property name we cannot limit our search to either set of elements, and must search both of them.
This is not always the case. If the object is a literal value, this expression will be **(1)**, and if the object is an uri this expression will be **(2)**.

(4) **unwind (3) as gV3** – with (3) processed we now have an array of data that has the data we want to gather. However, it has an array form, and as such is not easy to manipulate or reference if we later need the data that we have gathered.
To make this data available, we must process element by element of the array, and we use the unwind command to do that.

(5) **with *,gV3[0] as father,gV3[1] as customVar_father** – finally, we return the data we have gathered using appropriate names to easily reference them later.
Since we don't care about the predicate value(since it's a fixed value and thus not returned by the query) we only need to concern ourselves with 2 values: the value that corresponds to the ?$father$ variable and the node that corresponds to the element of that variable(if it exists).
As such, we return the first value as $father$ and the node value as $customVar\_father$.

Finally, since we don't have to process any other triples we reach the query data return, last 3 lines of the query. In this data return we catch only the data we have captured (in this case, the $father$ value), then we check if it is not null (since we don't have the optional keyword working yet in the translation) and finally we return what we have been asked to return(in this case its a **RETURN *** since the SPARQL query is a **select ***).

4.2 Translated types of queries

At this moment, the grammar can translate a SPARQL portion into CYPHER:

prefixes – in SPARQL a PREFIX statement defines an alias for the ontology namespace, it has the form: $<prefix : namespace>$. This prevents the need to write down the ontology full URI whenever we want to reference a subject or a property from that ontology. When the name that precedes the ":" is empty then we are in a special case, we are declaring the default namespace that will be associated with the empty prefix;

same subject or different subject triples – SPARQL uses triples in order to make matches with the graph data. It can make multiple of these matches in a single query, through the use of ";" to fix the subject between triples and the use of "." to denote completely different triples that should be matched. This allows us to chain them together to obtain more refined data, as well as extract more specific knowledge from the database;

filter expressions – SPARQL FILTER statement restricts the query results, only triples that satisfy the predicate inside FILTER clause are returned as result.

Unlike the previous, only part of the filter expression has been dealt with by the grammar due to its complexity. As of now, only basic mathematical and logical expressions are working, as well as the $EXISTS$ keyword;

limit and order by – SPARQL LIMIT expression limits the amount of returned results. SPARQL ORDER BY expression orders the results of the query by one of the columns present in the result;

ask query – a SPARQL ASK query is used to determine if a given triple is present in the ontology;

describe query – a SPARQL DESCRIBE query is used to gather every triple associated with the elements that it matches with.

A note about the translation grammar is that due to the complexity of the operation it does not directly do the describe operation. Instead it captures the elements that the query should capture and then the result from the query must be used with the neosemantics export operation using CYPHER query to obtain the desired result.

5 Grammar validation

In order to validate our grammar we have choosen 2 base ontologies for our tests: one that contains information about the periodic table and another that contains information about the pokedex (an encyclopedia for the Pokemon game franchise).

Both ontologies were imported to GraphDB and Neo4J after being processed through the use of a reasoner in protégé([10]). This was done so that any triples that arise from the rules laid out by the ontology are explicitly stated inside the ontology we are importing.

Next, we created numerous queries in SPARQL testing different types of queries.

After that we had to query the databases. The method used to do this was: directly make the query to the database in GraphDB, use the grammar to translate it into CYPHER and then query Neo4J with the translated query.

While numerous queries were made, we show the results of one of them made to the periodic table dataset, specifically a query that is used to obtain all triples associated to a given subject of the ontology *"give me anything you know about Carbon"*:

Query S4 – returns all predicates and object that have the subject <http://www.daml.org/2003/01/periodictable/PeriodicTable#C>:

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select * where { :C ?p ?o . }
order by ?p
```

Query C4 – S4 translation to CYPHER

```
match(gV1) where gV1.uri ="http://www.daml.org/2003/01/periodictable/
    PeriodicTable#C"
unwind [key in keys(gV1) | [gV1[key],null,key]] + [(gV1)-[gV2]->(customVar_o) |
     [customVar_o.uri,customVar_o,type(gV2)]] as gV3
with *,gV3[0] as o,gV3[1] as customVar_o,gV3[2] as p
with p,o
where (p is not null) and (o is not null)
RETURN *
ORDER BY p
```

After making these queries to the respective platforms, we obtained the result tables shown in Figure 1.

Result tables show that results are similar between both result sets. However, there are a couple differences, namely:

- Cypher's results have an additional row. This row contains the predicate "uri" and the object ":C". The reason this row exists is due to how neosemantics handles uri's. In order to guarantee that every uri is unique inside the ontology being imported, neosemantics requires the addition of a uniqueness constraint([5]). In order for this to work, neosemantics adds an uri field to every element of the ontology it imports, serving as an identifier to what that element is supposed to represent.
- Values in SPARQL that have a type attached to them have that type disappear and are imported as literal values. The reason for this is because of how neosemantics deals with custom data types versus the default data types. If the datatype is default and during the setup of the database we choose to allow custom data types, then they will be appended to the end of the value, working the same way as you'd expect them to in GraphDB. The reason it does not work like that is that neosemantics import only does this for custom data types, i.e., data types that are not already implemented in Neo4J. For those, it simply imports the value into the dataset without the tag at the end. Unfortunately, despite Neo4J fully implements floating point numbers, the import process seems to simplify them into integers, and thus the differences between the results.

6 Conclusion

Translating SPARQL into CYPHER is a complex process due to differences both in the languages syntax and how the data is stored in the associated platforms, making a full correlation between the 2 extremely difficult.

Throughout this article we have exposed part of that process, from explaining the engines that operate with those languages to the specifics of each one, as well as showing the translation capabilities of the parser implemented so far, as well as showing part of the validation process for the developed grammar.

p	o
:atomicNumber	“6”^^xsd:integer
:atomicWeight	“12.0107”^^xsd:float
:block	:p-block
:casRegistryID	7440-44-0
:classification	:Non-metallic
:color	graphite is black, diamond is colourless
:group	:group_14
:name	carbon
:period	:period_2
:standardState	:solid
:symbol	C
rdf:type	:Element
rdf:type	owl:NamedIndividual

(a) Result table for query in GraphDB.

p	o
:atomicNumber	6
:atomicWeight	12
:block	:p-block
:casRegistryID	7440-44-0
:classification	:Non-metallic
:color	graphite is black, diamond is colourless
:group	:group_14
:name	carbon
:period	:period_2
:standardState	:solid
:symbol	C
rdf:type	:Element
rdf:type	owl:NamedIndividual
uri	:C

(b) Result table for query in Neo4J.

Figure 1 Result tables.

References

1 Cypher introduction. `https://neo4j.com/docs/cypher-manual/current/introduction/`. Accessed: 2020-04-30.

2 Graphdb free. `http://graphdb.ontotext.com/documentation/free/`. Accessed: 2020-04-07.

3 Neo4j: A reasonable rdf graph database & reasoning engine [community post]. `https://neo4j.com/blog/neo4j-rdf-graph-database-reasoning-engine/`. Accessed: 2020-04-07.

4 Neo4j labs:incubating the next generation of graph developer tooling. `https://neo4j.com/labs/`. Accessed: 2020-04-30.

5 Neoseantics create uniqueness constraint. `https://neo4j.com/docs/labs/nsmntx/current/config/#create-resource-uniqueness-constraint`. Accessed: 2020-06-30.

6 Neo4j overview. `https://neo4j.com/developer/graph-database/#neo4j-overview`. Accessed: 2020-04-08.

7 Nsmntx – neo4j rdf & semantics toolkit. https://neo4j.com/labs/nsmtx-rdf/. Accessed: 2020-04-30.
8 Importing rdf data into neo4j. https://jbarrasa.com/2016/06/07/importing-rdf-data-into-neo4j/. Accessed: 2020-07-04.
9 Peg.js website. https://pegjs.org/. Accessed: 2020-04-30.
10 protégé:a free, open-source ontology editor and framework for building intelligent systems. https://protege.stanford.edu/. Accessed: 2020-06-30.
11 Linear congruential generator. https://en.wikipedia.org/wiki/Linear_congruential_generator. Accessed: 2020-06-30.
12 Definition of sparql. https://www.w3.org/TR/sparql11-query/#sparqlDefinition. Accessed: 2020-04-30.
13 Sparql: Grammar. https://www.w3.org/TR/sparql11-query/#sparqlGrammar. Accessed: 2020-04-30.
14 Sparql 1.1 query language. https://www.w3.org/TR/sparql11-query/. Accessed: 2020-04-30.