Formalisation of the rdf2neo Mapping Procedure

Introduction

This document is a supplemental material to the manuscript "Getting the best of Linked Data and Property Graphs: rdf2neo and the KnetMiner Use Case", formalising the approach we use in the rdf2neo tool to produce a Cypher-compatible property graph from an RDF graph.

Definition 1, RDF graph: similar to [1], let Str be the universe of strings, $I \subseteq Str$ the universe of IRIs, V the universe of values. An RDF graph G is a ternary relation $G \subseteq S \times P \times O$, where $S \subseteq I$[1] is a set of subjects, $P \subseteq I$ is a set of properties, $O \subseteq I \cup V$ is a set of objects, members of G are named triples.

Definition 2, property graph: similar to [1–3], we define a property graph PG as a tuple *PG := <N, nl, np, R, rt, src, dst, rp>*, where:

- $N \subseteq N$ is a subset of the universe of nodes N
- $nl := N \to 2^{Str}$, is a labelling function, defining the labels of a node
- $np := N \times Str \to \mathcal{V}$, is a property function, associating each node n and key *k* to a value *v*
- $R \subseteq R$ is a subset of the universe of relations R
- $rt := R \to Str$, is a function yielding the type of each relation
- $src := R \to N$ and $dst := R \to N$ are functions defining the source and destination nodes of relations
- $rp := R \times Str \to \mathcal{V}$ is a function defining the relation properties for each relation r.

Mapping RDF graphs to property graphs

RDF Graphs can be transformed to property graphs according to a defined mapping, which relates sets of RDF triples to coarser grained entities belonging in property graphs.

Definition 3, RDF mapping: given an RDF graph G, a mapping is a tuple $\mathcal{M} := \langle ni_{map}, nl_{map}, np_{map}, ri_{map}, rp_{map} \rangle$, where:

- $ni_{map} := \mathcal{G} \to \mathcal{I}$ is a node IRI selecting function, having *NI* as codomain
- $nl_{map} : \mathcal{G} \times NI \to 2^{Str}$ is a node label mapping function
- $np_{map} : \mathcal{G} \times NI \to (Str \to \mathcal{V})$ is a node property mapping function
- $ri_{map} : \mathcal{G} \to \mathcal{I} \times Str \times \mathcal{I} \times \mathcal{I}$ is a relation IRI selecting function, returning tuples like (ri, t, si, di) (that is, the relation IRI, its type, the IRI of the source and destination nodes), having *RI* as the range of the first element of the returned relation, and having subsets of *NI* as the ranges of the third and fourth elements
- $rp_{map} : \mathcal{G} \times RI \to (Str \to \mathcal{V})$ is a relation property mapping function

Definition 4, RDF/property graph transformation: given a mapping *M*, defined as above, a transformation T_M produced by *M* is a function from G to a property graph *PG := <N, nl, np, R, rt, src, dst, rp>* such that:

- $N := \{n_i : i \in NI\}$ is the set of nodes corresponding to the selected IRIs *NI* (we use the n_i in the

---

[1] We do not consider blank nodes, both for sake of simplicity and because in many real projects it is made the choice of not using blank nodes. For similar simplification reasons, we assume data types for values can be extracted from the values automatically. The rdf2neo tool does that by making reasonable assumptions and we are adding support to allow for data type specification and mapping.

follow with the same meaning)

- *nl* is such that: $nl(n_i) := nl_{map}(\mathcal{G}, i)$
- *np* is such that: $np(n_i, 'iri') := i$, $np(n_i, k) := v$ for all: $np_{map}(\mathcal{G}, i)(k) = v$
- $R := \{r_i : i \in RI\}$ is the set of relations corresponding to the selected IRIs RI (again, r_i is used in the follow with this same meaning)
- *rt*, *src*, *dst* are such that: $rt(r_i) := t, src(r_i) := n_s, dst(r_i) := n_d$ for $(i, t, s, d) \in ri_{map}(i)$ (which implies: $np(n_s, 'iri') = s$ and $np(n_d, 'iri') = d$, i.e., the relation mapping function links source/destination IRIs to the corresponding PG nodes)
- *rp* is such that: $rp(r_i, 'iri') := i$, $rp(r_i, k) := v$ for all: $rp_{map}(\mathcal{G}, i)(k) = v$

In practical terms, a mapping extracts elements to build a property graph from an RDF graph. Let us see an example. Suppose to have the RDF graph[2]:

G := { (kbr:tob1, rdf:type kb:Protein),
(kbr:tob1, rdfs:label, 'TOB1 HUMAN'),

(kbr:pub1, rdf:type, kb:Publication),
(kbr:pub1, kbr:title 'A Publication Title'),

(kbr:tob1_pub1_ann, rdf:type, rdf:Statement),
(kbr:tob1_pub1_ann, rdf:subject , kbr:tob1),
(kbr:tob1_pub1_ann, rdf:predicate , kb:isAnnotatedBy),
(kbr:tob1_pub1_ann, rdf:object , kbr:pub1),
(kbr:tob1_pub1_ann, kb:score , 0.95)
}

And the mapping M, defined by:

$$ni_{map}(\mathcal{G}) := \{i : (i, \text{rdf:type}, \text{kb:Protein}) \in \mathcal{G})\} \cup \{i : (i, \text{rdf:type}, \text{kb:Publication}) \in \mathcal{G})\}$$

$$\forall (i, \text{rdf:type}, t) \in \mathcal{G} \Rightarrow nl_{map}(\mathcal{G}, i) = \{iri2id(t), 'Concept'\}$$

$$\forall (i, \text{rdfs:label}, l) \in \mathcal{G} \Rightarrow np_{map}(\mathcal{G}, i)('label') = l$$
$$\forall (i, \text{kb:title}, t) \in \mathcal{G}\} \Rightarrow np_{map}(\mathcal{G}, i)('title') = t$$

$$\begin{aligned} ri_{map}(\mathcal{G}) := \{(ri, iri2id(t), si, di) : \{&(ri, \text{rdf:type}, \text{rdf:Statement}), \\ &(ri, \text{rdf:predicate}, t), \\ &(ri, \text{rdf:subject}, si), \\ &(ri, \text{rdf:object}, oi)\} \subseteq \mathcal{G})\} \end{aligned}$$

$$\forall (ri, \text{kb:score}, w) \in \mathcal{G}\} \Rightarrow rp_{map}(\mathcal{G}, ri)('score') = w$$

Where:

---

[2] Here and in the follow we omit IRI prefix declarations, which might be found either in our ontology file [4], or through resources like https://prefix.cc. Sequences like kbr:tob1 are abbreviations of the IRI obtainable by expanding the corresponding namespace prefix, i.e., http://www.ondex.org/bioknet/resources/tob1. In the following, this applies to strings too.

- iri2id() is a function that converts an IRI into its substring occuring after the last separator character (':' or '#', its implementation is straightforward in common programming languages). So IRIs like 'kb:Protein', 'kb:score' are turned into 'Protein', 'score'.
- The node labelling function assign a default 'Concept' label to each mapped nodes (this might be useful to distinguish between PG nodes obtained from RDF mapping and possible further nodes, computed after the mapping, e.g., provenance or dataset metadata nodes).
- While we use set theory algebra to define the RDF graph patterns needed for the mapping, the SPARQL language can also be used to define such patterns and more complicated ones too. While we do not cover it in this document, this can be demonstrated by means of a SPARQL formalisation (e.g., [5]).

The transformation produced by the property graph above is:

$N := \{n_{kbr:tob1}, n_{kbr:pub1}\}$

$nl(n_{kbr:tob1}) = \{$ 'Protein', 'Concept' $\}$
$np(n_{kbr:tob1},$ 'iri') = 'kbr:tob1'
$np(n_{kbr:tob1},$ 'label') = 'TOB1 HUMAN'

$nl(n_{kbr:pub1}) = \{$ 'Publication', 'Concept' $\}$
$np(n_{kbr:pub1},$ 'iri') = 'kbr:pub1'
$np(n_{kbr:pub1},$ 'title') = 'A Publication Title'

$R := \{r_{kbr:tob1\_pub1\_ann}\}$

$rt (r_{kbr:tob1\_pub1\_ann}) =$ 'isAnnotatedBy'

$src (r_{kbr:tob1\_pub1\_ann}) = n_{kbr:tob1}$
$dst (r_{kbr:tob1\_pub1\_ann}) = n_{kbr:pub1}$

$rp (r_{kbr:tob1\_pub1\_ann, 'score'}) = 0.95$

Which can be represented in a more compact way, using a Cypher-like notation:

(n1: Protein{ iri:'kbr:tob1', label: 'TOB1 HUMAN' })
(n2: Publication{ iri: 'kbr:pub1', title: 'A Publication Title' })
(n1) - [:isAnnotatedBy{ iri:'tob1_pub1_ann', score: 0.95 }] -> (n2)

Cypher-based mapping

Now we can show show that the Cypher instructions we create in the rdf2neo tool actually spawns the transformation described above. Following a notation similar to [5], we first define the semantics of the Cypher instructions we use.

Definition 5, Cypher 'CREATE' semantics: The Cypher command

CREATE (n: $l_1, l_2, \ldots l_u$)
SET n = { iri: 'α', k_1: v_1, k_2: $v_2 \ldots k_m$: v_m }

where: $\{ l_1, l_2, \ldots l_u \} \subseteq$ Str is a set of labels, $\alpha \in$ I is an IRI, $\{ k_1, k_2, \ldots k_m \} \subseteq$ Str is a set of keys, $\{ v_1, v_2, \ldots v_m \} \subseteq$

V is a set of values, generates the following property graph:

N := { nα }
nl(nα) := { $l_1, l_2, \ldots l_u$ }
np(nα, 'iri') := α
np(nα, k_i) := v_i for i = 1..m

This can be easily extended to the richer syntax:

UNWIND { iri: 'α_1' k_{11}: v_{11}, k_{12}: v_{12} ... k_{1m}: v_{1m} },
{ iri: 'α_2', k_{21}: v_{21}, k_{22}: v_{22} ... k_{2m}: v_{2m} },
...
{ iri: 'α_q', k_{q1}: v_{q1}, k_{q2}: v_{q2} ... k_{qm}: v_{qm} } AS props
CREATE (n: $l_1, l_2, \ldots l_u$)
SET n = props

which allows one to create multiple nodes with shared labels and, semantically, it can be seen as a compact syntax for writing multiple commands in the first form. It is worth to note that the above apply to existing property graphs having no intersection with the graph generated by the described commands. In that case, the generated graph is added to the existing one.

Theorem 1, correct semantics of rdf2neo Cypher commands for node creation: given a mapping *M* on an RDF graph G, which, for the nodes, is defined as:

$$ni_{map}(\mathcal{G}) := \{\alpha\}$$

$$nl_{map}(\mathcal{G}, \alpha) = \{l_i\}, i := 1..u$$

$$np_{map}(\mathcal{G}, \alpha)(\text{'iri'}) := \alpha$$
$$np_{map}(\mathcal{G}, \alpha)(k_i) := v_i, i := 1..m$$

then the property graph generated by the command defined in definition 5 coincides with the T_M defined by the mapping *M* for what concerns the property graph nodes.

The proof is straightforward from definition 4 and definition 5. This can be extended to the case of a command creating multiple nodes, as defined above, and to the general case of nodes instantiating different sets of labels.

The shown node grouping based on labels is done for practical reasons: the Neo4j client does not accept Cypher CREATE commands where the labels to be created are a parameter passed by the (Java) invoker. Parameterised queries are a way to avoid the overhead of query parsing and compilation, so a good approach to create many nodes is to group the latter by set of labels (using an hash table having the lexicographically sorted labels as keys) and issue commands like the ones above.

The same approach can be used to prove the correctness of the syntax used to create relationships.

Definition 6, Cypher relation creation semantics: Let *PG* be a property graph, obtained by an RDF transformation as above, defined by:

N := { n_α, n_β}
$L \in$ nl(n_α), $L \in$ nl(n_β)
np(n_α, ‘iri’) := α
np(n_β, ‘iri’) := β
np(n_α, k_i) := …, np(n_β, k_i) := …

where L is a default label, α and β are IRIs. Then, the command:

MATCH (na:L { iri: ‘α’ }), (nb:L { iri: ‘β’ })
CREATE (na) - [rg: τ] -> (nb)
SET { iri: ‘γ’, k_1: v_1, k_2: v_2 … k_m: v_m }

where τ is a relation type and γ is an IRI, generates a new property graph *PG'* , which is *PG* augmented with the following:

R := { r_γ } (it’s added to the existing relations)
rt (r_γ) := τ
src (r_γ) := n_α, dst (r_γ) := n_β
rp(r_γ, ‘iri’) := γ
rp(r_γ, k_i) := v_i for i = 1..m

from which we can prove the:

Theorem 2, correct semantics of rdf2neo Cypher commands for relation creation: let *M* be a mapping defined as:

$$ni_{map}(\mathcal{G}) := \{\alpha, \beta\}$$

$$\mathcal{L} \in nl_{map}(\mathcal{G}, \alpha), nl_{map}(\mathcal{G}, \beta)$$

$$np_{map}(\mathcal{G}, \alpha)(\text{'iri'}) := \alpha$$
$$np_{map}(\mathcal{G}, \beta)(\text{'iri'}) := \beta$$

$$ri_{map}(\mathcal{G}) := \{(\gamma, \tau, \alpha, \beta)\}$$
$$rp_{map}(\mathcal{G}, \gamma)(\text{'iri'}) := \gamma$$
$$rp_{map}(\mathcal{G}, \gamma)(k_i) := v_i, i := 1..m$$

then the transformation of the mapping M T_M, as defined above coincides with the graph produced by the commands defined in definition 5 for the nodes n_α, n_β, followed by the command defined in definition 6.

As above, the proof of this can be obtained by applying the definitions 5 to 6.
Again, the above theorem can be extended to the multiple relation creation that we use in our code:

UNWIND
{ iri: ‘γ_1’ fromIri: s_{11}, toIri: d_{11}, k_{11}: v_{11}, k_{12}: v_{12} … k_{1m}: v_{1m} },
{ iri: ‘γ_2’ fromIri: s_{12}, toIri: d_{12}, k_{21}: v_{21}, k_{22}: v_{22} … k_{2m}: v_{2m} },
…
{ iri: ‘γ_q’ fromIri: s_{q1}, toIri: d_{q1}, k_{q1}: v_{q1}, k_{q2}: v_{q2} … k_{qm}: v_{qm} }

```
AS rdef
MATCH ( from:L { iri: rdef.fromIri } ), ( to:`L`{ iri: rdef.toIri } )
CREATE (from)-[r:`τ`]->(to)
SET r = rdef.properties
```

In the definitions above, we assume every node created from mapping RDF has always a default label L (our code automatically add it to the configured mapping) and we group relations by type τ into single commands, for reasons similar to what explained above about node creations: the Neo4j client does not allow for commands with parameterised relation types, so we generate them with many relations within the same type, after having created a working hash table having the relation types as keys.

Computational complexity

Let us consider the computational complexity of the Cypher-based creation of the mapping transformations described above, by starting from the CREATE command given in definition 5 and assuming the number of labels and properties passed to the command is small enough to not being significant (in particular, not varying significantly with the number of nodes/commands of that type issued to the database). The instruction takes an approximately constant time to be executed as it can be verified empirically (and also considering that it implies the addition of a record and the possible update of an index). So, we can say the execution time for creating a single node is $O(1)$. Similar considerations apply to the UNWIND … CREATE command, which creates multiple nodes in a single command, so its computational complexity of it is $O(q)$, where q is the number of nodes sent to the command. Overall the complexity of creating a property graph given the nodes returned by a mapping is $O(|N|)$, where N is the set of nodes.

The Cypher command to create a single relation, given in the definition 6 requires two index lookups, plus the creation of a link between existing nodes. All three can are computed in constant time, so the complexity of the command is $O(3)$, while ti can be shown that the complexity of multi-relation creation is $O(3q)$, where q is the number of relations sent to the corresponding UNWIND … command, and the total complexity for creating all the mapping relations is $O(3|R|)$, where R is the number of relations.

Hence, the overall complexity for creating a property graph from the entities returned by a mapping is in the order of $O(|N|+|R|)$, which can be further reduced with splitting the nodes (or relations) into subsets and sending them to parallel procedures, each running the Cypher commands shown above.
This makes us to conclude that the critical aspect regarding the performance of rdf2neo is the implementation of the mapping. We have chosen to do so by means of configurable SPARQL, which of evaluation is known to be a PSPACE-complete problem [5] in general. However that highly varies depending on the SPARQL queries that are written and the query engine. In particular, there are so-called well-designed and weakly well-designed queries, which are known to be within the tractable LOGSPACE class of problems [5].

References

1. Tomaszuk D. RDF Data in Property Graph Model. In: Garoufallou E, Subirats Coll I, Stellato A, Greenberg J, editors. Metadata and Semantics Research [Internet]. Cham: Springer International Publishing; 2016 [cited 2018 Sep 28]. p. 104–15. Available from: http://link.springer.com/10.1007/978-3-319-49157-8_9

2. Nadime Francis. Formal Specification of Cypher [Internet]. Available from: https://s3.amazonaws.com/artifacts.opencypher.org/website/ocim2/slides/Slides.pdf

3. Gábor Szárnyas, Anna Gujgiczer, Márton Elekes. Learning Timed Automata with Cypher [Internet]. Available from: https://www.slideshare.net/openCypher/learning-timed-automata-with-cypher-99745432

4. BioKNO, The Biological Knowledge Network Ontology [Internet]. Available from: https://github.com/Rothamsted/bioknet-onto

5. Pérez J, Arenas M, Gutierrez C. Semantics and Complexity of SPARQL. In: Cruz I, Decker S, Allemang D, Preist C, Schwabe D, Mika P, et al., editors. The Semantic Web - ISWC 2006 [Internet]. Berlin, Heidelberg: Springer Berlin Heidelberg; 2006 [cited 2018 Sep 28]. p. 30–43. Available from: http://link.springer.com/10.1007/11926078_3