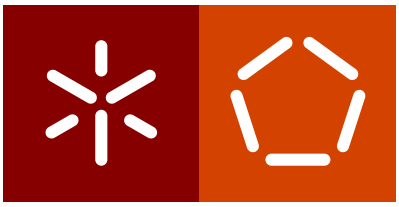


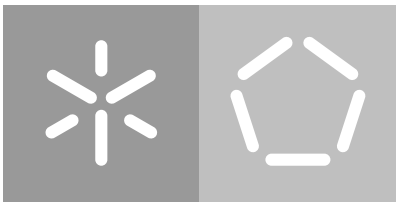
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ezequiel José Veloso Ferreira Moreira

SPARQL versus CYPHER

um estudo comparativo

Setembro 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ezequiel José Veloso Ferreira Moreira

SPARQL versus CYPHER

um estudo comparativo

Dissertação de Mestrado
Dissertação de Mestrado em Engenharia Informática

Trabalho realizado sob a orientação do Professor
José Carlos Leite Ramalho

Setembro 2020

LICENÇA

Atribuição
CC BY
https://creativecommons.org/licenses/by/4.0/

AGRADECIMENTOS

Quero primeiro agradecer ao meu orientador José Carlos Leite Ramalho por todo o esforço e apoio dados não só neste trabalho mas nos anos em que fui estudante dele, por permitir o meu desenvolvimento como estudante e por proporcionar a oportunidade de fazer este trabalho onde aprendi sobre um tópico extremamente interresante e pratiquei numerosas capacidades aprendidas durante a minha estadia na universidade.

Desejo ainda agradecer à Universidade do Minho pelos anos de educação e pelo ambiente extremamente agradável que providenciaram durante toda a minha estadia.

Quero agradecer aos meus avós maternos, por todo o suporte que me deram e por me ensinarem tanto sobre a vida, e a meus pais, que tanto esforço e sacrifício fizeram para permitir que o meu crescimento e desenvolvimento fossem o melhor possível.

Finalmente expresso a minha gratidão aos meus familiares e amigos mais próximos cujo apoio ajudou a continuar motivado.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

RESUMO

Com a crescente necessidade de armazenar dados sobre forma digital as ontologias tornam-se cada vez mais relevantes como maneira simples de expressar conhecimento. Assim bases de dados capazes de guardar este tipo de estruturas de dados de modo eficaz, nomeadamente bases de dados orientadas a grafos, têm visto a sua utilização aumentar.

Nesta dissertação foram estudados dois motores de base de dados deste tipo: o GraphDB (Ontotext (2020a)) e o Neo4j (Neo4j (2020)). GraphDB foi criado para armazenamento de ontologias Web: OWL (Group (2012)), SKOS (Group (2009)) e RDF (Group (2014)), podendo estas ser interrogadas através de SPARQL (W3C (2013a)), enquanto Neo4J foi desenhado para armazenar informação fortemente relacionada: grafos de informação dos quais as ontologias fazem parte, sendo CYPHER (Neo4J (2020b)) a linguagem de *query* utilizada para a sua exploração.

Nesta dissertação estudou-se a viabilidade de armazenar ontologias em Neo4J e explorá-las utilizando CYPHER.

Ao longo do processo da resolução deste problema foi determinado um segundo objectivo: criar uma camada tecnológica que permite o uso de SPARQL para interrogar o Neo4J.

Nesse sentido foi realizado um estudo comparativo das duas linguagens, implementou-se um compilador capaz de traduzir um subconjunto de queries SPARQL em CYPHER e foi desenvolvida uma bateria de testes que permitem fazer o *benchmarking* da tecnologia criada.

Finalmente foi construído um protótipo Web que implementa uma frontend sobre o Neo4J de modo a permitir não só armazenar ontologias como interrogá-las através de SPARQL.

Keywords: bases de dados orientadas a grafos, CYPHER, OWL, RDF, SPARQL

ABSTRACT

With the growing need of storing data in the Web, ontologies are becoming more and more relevant as a simple way of expressing knowledge. As such, databases capable of storing such structures effectively, namely Graph Databases, have seen their use grown.

In this dissertation we studied two Graph Database Engines: GraphDB (Ontotext (2020a)) and Neo4J (Neo4j (2020)). GraphDB was created to store Web based ontologies: OWL (Group (2012)), SKOS (Group (2009)) and RDF (Group (2014)), allowing for their querying through SPARQL (W3C (2013a)), while Neo4J was designed to store strongly linked data: information graphs of which ontologies are an example, with CYPHER (Neo4J (2020b)) as the query language for their exploration.

In this dissertation we studied the viability to store ontologies using Neo4J and explore them through CYPHER.

Exploring this second problem led to a second objective: the creation of a technological layer that allows the use of SPARQL queries in a Neo4J database.

To do this a study was done to compare both languages, a compiler to translate a subset of SPARQL queries into CYPHER was implemented and a series of tests were made to allow the benchmarking of the developed tech.

Finally a Web app prototype that implements a frontend over Neo4J and allows the storage of ontologies and their interrogation by SPARQL was created.

Keywords: bases de dados orientadas a grafos, CYPHER, OWL, RDF, SPARQL

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGO

LISTA DE ABREVIAÇÕES

API Application Programming Interface. 19, 20

BD Base de Dados. 34

HTTP HyperText Transfer Protocol. 19, 74

IRI Internationalized Resource Identifier. 3, 17

JS Javascript. 48, 51, 74, 75

NoSQL Not Only SQL. 19

OWL Web Ontology Language. iv, v, 2–4, 10–12, 14, 15, 18–21, 27, 39, 65, 96

RDF Resource Description Framework. iv, v, 2–5, 10, 14, 17, 19, 27, 30, 96
RDFS RDF Schema. 2–6, 10, 17, 18, 96

SAIL Storage And Inference Layer. 19, 20
SKOS Simple Knowledge Organization System. iv, v, 2, 3, 6–11, 17, 18, 96

TRREE Triple Reasoning and Rule Entailment Engine. 21

URI Uniform Resource Identifier. ix, 3–6, 17, 21, 28, 29, 38, 40, 41, 43, 52, 54–56, 64–67, 88, 90, 91, 95, 97, 98, 114, 115

1

INTRODUÇÃO

O crescimento da industria electrónica criou um aumento significativo à quantidade e complexidade de dados a serem produzidos, tendo estes origem em todo o género de equipamentos electrónicos desde sensores de clima a telemóveis, levando a uma crescente dificuldade no seu processamento.

Para lidar com este problema de salvaguardar dados e obter conhecimento útil deles foram desenvolvidas numerosa tecnologias entre as quais destacamos as ontologias WEB, que definem uma forma simples e bem estruturada de guardar dados e obter conhecimento deles.

Utilizar estas estruturas de modo eficaz requer um esquema sólido para as guardar e manipular, o que nos leva a perguntar: que tipo de bases de dados se adaptam melhor a ontologias?

Esta pergunta tem uma solução simples que advém de um programa que já existe e tem uso generalizado: GraphDB, um motor de bases de dados orientadas a grafos que utiliza ontologias Web para guardar dados e permite questiona-las através do uso de SPARQL.

No entanto GraphDB não é o único motor destas base de dados de uso generalizado, com Neo4J, um motor de bases de dados criado para guardar informação fortemente relacionada que utiliza CYPHER para obter dados, a ser outra alternativa para este tipo de bases de dados que utiliza um esquema mais geral para guardar dados (e como tal a sua capacidade de trabalhar com ontologias é muito mais reduzida).

Assim, considerando as vantagens do uso de ontologias para guardar dados e a sua estrutura similar a grafo, pretendemos adicionar a Neo4J capacidades de lidar com ontologias.

Para isso precisamos de estudar os dois motores de bases de dados mencionados (GraphDB e Neo4J) de modo a perceber as diferenças e semelhanças em como processam pedidos feitos às suas bases de dados (*queries*) e as duas linguagens de *query* associadas a estas, SPARQL e CYPHER respetivamente, de modo a compreender as diferenças entre a escrita (sintaxe) das *queries* e as capacidades (semântica) que as mesmas providenciam para obter dados.

Com este estudo pretendemos não só determinar a viabilidade do uso de ontologias em Neo4J como se é possível explora-las através do uso de SPARQL, sendo que deste segundo objectivo e de o Neo4J não tem suporte para mais nenhuma linguagem de *query* temos a necessidade de desenvolver uma tradução de SPARQL para CYPHER e de criar uma camada tecnológica que permita o uso desta tradução para a exploração de ontologias em Neo4J.

1.1 QUESTÃO DE INVESTIGAÇÃO

As questões principais a que pretendemos dar resposta no final desta tese são:

1. É possível através de algum método interrogar o motor de bases de dados Neo4J através do uso da linguagem de *query* SPARQL?
2. É possível criar um mapeamento entre as linguagens de *query* SPARQL e CYPHER tal que se consiga exprimir uma *query* numa linguagem e através de alguma transformação obter uma *query* equivalente na outra linguagem?
3. É possível armazenar ontologias no Neo4J e explorá-las de uma forma simples?

1.2 ESTRUTURA DA TESE

Para podermos resolver estas questões precisamos de tomar numerosos passos começando pela pesquisa teórica e seguindo pela sua aplicação de modo a obter resultados tangíveis.

Esta tese começa por descrever o conceito teórico mais básico deste trabalho: a ontologia, explorando a sua definição e as linguagens que são usadas para as exprimir no contexto da WEB, nomeadamente RDF/RDFS, SKOS e OWL.

Após esta introdução à teoria passamos à segunda componente que é necessária para compreender o trabalho realizado: bases de dados orientadas a grafos e motores para estas, que no nosso caso se limitaram a GraphDB e Neo4J. Em seguida expusemos as linguagens de *query* usadas para permitir a exploração das bases de dados dentro destes motores: SPARQL e CYPHER, apresentando cada uma separadamente e depois comparando-as directamente.

Com a teoria de base explícita podemos explorar o problema que pretendemos resolver e como o pretendemos fazer.

Finalmente passamos ao trabalho feito para resolver o problema desta tese, começando pela metade mais teórica da solução (especificação de uma gramática de tradução) e seguindo pela metade mais prática desta solução (implementação de uma aplicação WEB).

Tendo mostrado a solução desenvolvida a dissertação termina com um capítulo final onde se apresentam as conclusões obtidas e se apontam ideias para trabalho futuro.

2

ONTOLOGIA

No contexto deste trabalho uma ontologia é uma especificação formal de uma área ou domínio de conhecimento humano (W3C (2015)), ou seja, uma ontologia é uma especificação de uma área de conhecimento humano que é inteligível quer para máquinas quer para humanos.

Existem numerosas maneiras de representar e trabalhar com ontologias, mas neste trabalho apenas estamos interessados em ontologias como listas de triplos (sujeito, predicado, objecto), i.e., ontologias como bases de dados de triplos (sendo que ao longo deste trabalho assumimos sempre que ontologias estão sobre este formato).

Há várias linguagens para especificar ontologias, das quais destacaremos RDF (Group (2014)), RDFS (Group (2004)), SKOS (Group (2009)) e OWL (Group (2012)), pela relevância que têm para o problema que o trabalho pretende resolver.

Deve-se notar que estas linguagens têm uma relação directa entre elas, visto que cada uma delas é construída com base na anterior: OWL em cima de SKOS, SKOS em cima de RDFS e RDFS em cima de RDF, o que leva a que conceitos usados num destes tipos de linguagem seja válido para as acima e que uma ontologia OWL seja também ontologia SKOS, RDFS e RDF.

Existem várias sintaxes alternativas para a descrição ou serialização de ontologias, pelo que é importante realçar que neste documento utilizaremos a sintaxe TURTLE (W3C (2014c)) para representar ontologias sempre que tal for necessário.

Nas secções seguintes apresenta-se um resumo de cada uma das linguagens acima enumeradas.

2.1 RDF/RDFS

Um grafo RDF (W3C (2014a)) consiste num conjunto de entidades que são representadas através de triplos (sujeito, predicado, objecto) onde todos os seus elementos são URI's. A cada um destes triplos chamamos triplos RDF, mas serão referidos apenas como triplos ao longo desta tese para simplificar a leitura e porque não existe ambiguidade no que estamos a referir.

Antes de continuarmos temos de notar duas coisas:

1. RDF usa IRI em vez de URI, pois IRI é uma generalização do URI que permite maior liberdade de trabalhar com *unicode*.

   Mas ao longo do documento usaremos URI para referir IRI por não ser relevante ao problema que estamos a tratar.

2. Para as secções deste capitulo consideramos que uma entidade e um elemento são a mesma coisa. Do mesmo modo chamaremos grafos RDF de especificações RDF e vice-versa.

A gramática RDF tem como objectivo criar uma representação de conhecimento na Web que seja simples e estruturada, identificando entidades por um URI único e associando a cada uma delas diferentes propriedades com valores distintos.

A única propriedade a realçar introduzida pelo RDF é `rdf:type`, que representa o tipo ou classe da entidade na gramática, sendo possível relacionar uma entidade da gramática a vários tipos ou classes.

Um tipo/classe é, em RDF, um agregado de entidades que têm algo em comum (por exemplo, "Carro", "Pessoa", "Guitarra", ...), sendo que este conceito é expandido em OWL para permitir à classe dar maior contexto a seus elementos.

No entanto RDF é extremamente básico e como tal limitado no conhecimento que consegue representar, pelo que foi criado o RDFS para aumentar as capacidades de expressão do RDF.

Este esquema introduz dois conceitos importantes: propriedades de entidades (que são não só propriedades associadas a uma entidade como também relações entre duas delas) e estabelecimento dos tipos de dados a usar nos distintos membros do triplo:

- sujeito: URI ou nodo vazio;
- predicado: URI;
- objecto: URI, literal ou nodo vazio.

Onde

- URI representa um elemento de uma gramática;
- literal é um valor concreto associado a um tipo de dados (como um valor inteiro, uma string, uma data, ..., sendo que a lista completa de tipos de literais pode ser consultada em W3C (2014b))
- nodo vazio é um elemento da gramática que não representa nenhuma entidade/valor (similar a `null` em linguagens de programação).

Com isto podemos falar de propriedades de uma entidade, que são triplos na base de dados que têm essa entidade como sujeito e que descrevem algum conhecimento sobre ela. Por exemplo: se tivermos uma entidade que representa a pessoa Maria e esta estiver num triplo onde o predicado seja `idade` e o objecto seja vinte então podemos concluir que "a Maria tem vinte anos de idade".

Estas propriedades também podem ter URI's como objecto, e nesse caso estamos no que chamaremos de uma relação entre entidades. Por exemplo se tivermos dois elementos a representar as pessoas João e Maria podemos representar a frase "Maria é mãe de João" usando a relação `hasSon` com a Maria no sujeito e o João no objecto.

Assim com o uso de RDF podemos representar a frase "A Maria tem vinte anos e um filho chamado João" da seguinte forma:

```
myGrammar:Joao rdf:type person:Person

myGrammar:Maria rdf:type person:Person
                myGrammar:age 20
                myGrammar:hasSon  myGrammar:Joao
```

Código 2.1: Exemplo simples de RDF

Note-se que o exemplo acima usa o conceito de prefixos (um prefixo funciona como uma abreviatura do *namespace* URI) para simplificar os URI's apresentados.

Estes prefixos permitem utilizar palavras simples em vez de URI's extensos, sendo `rdf:` equivalente a `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, `person:` equivalente a `http://www.w3.org/ns/person#`, com `myGrammar:` a representar o URI associado à gramática RDF que estamos a escrever.

Finalmente RDFS introduz a hierarquia entre classes e entre propriedades através de `rdfs:subClassOf` e `rdfs:subPropertyOf`, respectivamente, que permitem definir uma propriedade ou uma classe como sendo parte de outra propriedade ou classe respectivamente.

2.1.1 *Propriedades lógicas de especificações RDF*

Com as definições acima dadas podemos observar que um triplo RDF codifica uma expressão lógica simples (podendo ela ser verdadeira ou falsa), o que leva a que uma gramática RDF tenha como equivalente lógico a conjunção das expressões que fazem parte dela (i.e., um `AND` lógico de todos os seus triplos).

Com esta definição em mente podemos observar algumas propriedades das especificações RDF:

- **Entailment (Implicação)**: Duas especificações RDF, A e B, estão numa relação de implicação se qualquer combinação de estados que leve a que a gramática A seja verdadeira leva a que a gramática B também o seja (i.e., A $\Rightarrow$ B);
- **Equivalence (Equivalência)**: Duas especificações RDF, A e B, são equivalentes se existe uma relação de implicação entre ambas as gramáticas em ambos os sentidos (i.e., A $\Rightarrow$ B && B $\Rightarrow$ A);
- **Inconsistency (Inconsistência)**: uma especificação RDF diz-se inconsistente se não existe qualquer estado possível onde seja verdadeira, i.e., tem uma contradição interna.

2.2 SKOS

Tendo por base RDFS, o SKOS (W3C (2009)) define uma sintaxe simples para modelar a estrutura básica e conteúdo de esquemas conceptuais como thesaurus, taxonomias e esquemas de classificação.

Para isso introduz múltiplas construções novas, como *Concepts* (Conceitos) e *Concept Schemes* (Esquemas de Conceitos, similares a classes de entidades mas com algumas propriedades extra), introduzindo ainda hierarquia entre estas novas estruturas.

Um conceito é uma unidade de pensamento, i.e., uma ideia, um significado, um objecto/evento ou uma classe de objectos/eventos, existindo assim como entidade abstracta que é independente de qualquer entidade real.

Para representar conceitos são necessários apenas dois passos:

1. definir/reutilizar um URI e
2. colocar a propriedade `rdf:type` associada a esse URI com o valor `skos:Concept`.

Estes conceitos são mais abstractos que elementos de uma especificação RDFS, necessitando assim de várias propriedades e relações adicionais que são adicionadas pelo SKOS, nomeadamente:

Labels (`skos:prefLabel`, `skos:altLabel` e `skos:hiddenLabel`)

São etiquetas que adicionam contexto extra a um Conceito, permitindo especificar palavras que lhe estão associadas de forma directa. Por exemplo:

```
ex:animals rdf:type skos:Concept;
  skos:prefLabel "animals"@en;
  skos:altLabel "creatures"@en;
  skos:prefLabel "animaux"@fr;
  skos:altLabel "creatures"@fr.
```

Código 2.2: *Labels* SKOS

Relações semânticas (`skos:broader`, `skos:narrower` e `skos:related`)

São relações que ocorrem entre conceitos e que permitem estabelecer uma hierarquia lógica ou uma relação não hierárquica.

Por exemplo, podemos utilizar o triplo `ex:Mammals skos:narrower ex:Animals` para representar que apenas uma parte dos animais são mamíferos ou
`ex:Stars skos:related ex:Astronomy` para expressar que a astronomia e as estrelas estão relacionados (mas não têm hierarquia entre eles).

Notas de documentação (`skos:note`, `skos:scopeNote`, `skos:definition`, `skos:example`, `skos:historyNote`, `skos:editorialNote` , `skos:changeNote`)

São propriedades associadas a conceitos que permitem documentar várias características dele, desde a sua definição (`skos:definition`) até mudanças históricas que tenham ocorrido com o conceito (`skos:historyNote`), passando por notas sobre quando se deve usar dito conceito (`skos:scopeNote`).

Com as propriedades e relações acima podemos já definir inúmeros conceitos do mundo real, mas não temos ainda maneira eficaz de agrupar várias conceitos. Para permitir esta união o SKOS define *Concept Schemas* (esquemas de conceitos), que permitem fazer ligação rápida entre conceitos.

Por exemplo, se definirmos um tal esquema da seguinte maneira:

```
ex:animalThesaurus rdf:type skos:ConceptScheme;
  dct:title "Simple animal thesaurus";
  dct:creator ex:antoineIsaac.
```

Código 2.3: Definição de esquema de conceitos SKOS

Podemos ligar-lhe conceitos usando `skos:inScheme` (que permite ligar um conceito directamente a um esquema de conceitos):

```
ex:mammals rdf:type skos:Concept;
  skos:inScheme ex:animalThesaurus.

ex:cows rdf:type skos:Concept;
  skos:broader ex:mammals;
  skos:inScheme ex:animalThesaurus.

ex:fish rdf:type skos:Concept;
  skos:inScheme ex:animalThesaurus.
```

Código 2.4: Uso do esquema de conceitos para relacionar conceitos SKOS

Ou usando `skos:hasTopConcept` (que permite a um esquema de conceitos utilizar conceitos que tenham sido já definidos):

```
ex:animalThesaurus rdf:type skos:ConceptScheme;
  skos:hasTopConcept ex:mammals;
  skos:hasTopConcept ex:fish.
```

Código 2.5: Uso de conceitos SKOS dentro da definição de um esquema de conceitos SKOS

Com estes esquemas podemos agrupar numerosos conceitos, permitindo-nos maior flexibilidade na escrita da especificação e dando-nos capacidade de expressar relações entre numerosos conceitos ao mesmo tempo.

Para estas relações o SKOS define `skos:exactMatch`, `skos:closeMatch`, `skos:broadMatch`, `skos:narrowMatch` e `skos:relatedMatch`, que são usados para conceitos em esquemas distintos mas que têm significados com graus de similaridade variados (dependendo da relação escolhida).

Como exemplo consideremos os seguintes esquemas, que representam maneiras distintas de olhar para animais:

```
ex1:referenceAnimalScheme rdf:type skos:ConceptScheme;
   dct:title "Extensive list of animals"@en.

ex1:animal rdf:type skos:Concept;
   skos:prefLabel "animal"@en;
   skos:inScheme ex1:referenceAnimalScheme.

ex1:platypus rdf:type skos:Concept;
   skos:prefLabel "platypus"@en;
   skos:inScheme ex1:referenceAnimalScheme.


ex2:eggSellerScheme rdf:type skos:ConceptScheme;
   dct:title "Obsessed egg-seller's vocabulary"@en.

ex2:eggLayingAnimals rdf:type skos:Concept;
   skos:prefLabel "animals that lay eggs"@en;
   skos:inScheme ex2:eggSellerScheme.

ex2:animals rdf:type skos:Concept;
   skos:prefLabel "animals"@en;
   skos:inScheme ex2:eggSellerScheme.

ex2:eggs rdf:type skos:Concept;
   skos:prefLabel "eggs"@en;
   skos:inScheme ex2:eggSellerScheme.
```

Código 2.6: Representações de esquemas diferentes em SKOS.

Podemos relacionar conceitos destes esquemas da seguinte forma:

```
 ex1:platypus skos:broadMatch ex2:eggLayingAnimals.

 ex1:platypus skos:relatedMatch ex2:eggs.
```

```
ex1:animal skos:exactMatch ex2:animals.
```

Código 2.7: Relações entre esquemas diferentes em SKOS

Com tanto conceitos como esquemas de conceitos temos capacidade de representar a vasta maioria dos sistemas que referimos na introdução do SKOS, mas existem ainda casos mais complexos que temos dificuldade em expressar.

Para resolver isto o SKOS permite duas capacidades adicionais: colecções de conceitos e hierarquias de conceitos transitivas.

Colecções de conceitos

Por vezes queremos agregar conceitos completamente diferentes e sem relação aparente, como por exemplo leite e um animal (para representar que animal produziu o leite). Para estes casos SKOS disponibiliza a classe `skos:Collection` (e `skos:OrderedCollection`, que é similar mas contém noção de ordem), que permitem definir colecções de conceitos da seguinte forma:

```
ex:milk rdf:type skos:Concept;
  skos:prefLabel "milk"@en.

ex:cowMilk rdf:type skos:Concept;
  skos:prefLabel "cow milk"@en;
  skos:broader ex:milk.

ex:goatMilk rdf:type skos:Concept;
  skos:prefLabel "goat milk"@en;
  skos:broader ex:milk.

ex:buffaloMilk rdf:type skos:Concept;
  skos:prefLabel "buffalo milk"@en;
  skos:broader ex:milk.

_:b0 rdf:type skos:Collection;
   skos:prefLabel "milk by source animal"@en;
   skos:member ex:cowMilk;
   skos:member ex:goatMilk;
   skos:member ex:buffaloMilk.
```

Código 2.8: Colecções de conceitos SKOS

Note-se que no exemplo não houve qualquer alteração dos conceitos de base, apenas a criação de uma colecção e definição dos seus membros.

Hierarquias de conceitos transitivas

A hierarquia definida por `skos:broader` e `skos:narrower` não é, por definição que lhe é atribuída por SKOS, transitiva (apesar de poder ser definida como tal numa gramática).

Para casos onde se pretende ter transitividade explícita o SKOS oferece `skos:broaderTransitive` e `skos:narrowerTransitive`, que se comportam do mesmo modo que as suas versões não transitivas excepto que apresentam transitividade, i.e., se A e B são conceitos relacionados por `skos:broaderTransitive` e B e C também o são então podemos concluir que A e C têm essa mesma relação (i.e., $(A \rightarrow B)$ && $(B \rightarrow C) \Rightarrow A \rightarrow C$).

2.3 OWL

Tendo por base o RDFS (e SKOS), OWL (W3C (2012)) é uma linguagem para a especificação de ontologias, i.e., é uma linguagem criada para exprimir ontologias na Web, fazendo tal através da expansão de numerosos conceitos do RDF/RDFS e da introdução de novos conceitos e estruturas.

Devido à complexidade do OWL, precisamos de primeiro expor alguns conceitos básicos:

AXIOMAS – Um axioma pode ser considerado como um pedaço de conhecimento, i.e., algo que se refere a algo no mundo real e que pode ser verdadeiro ou falso dependendo do contexto (por exemplo, "está a chover", "ontem fui ao parque", "todo homem é mortal", ...). Devido ao facto que um axioma é um booleano, podemos desde já definir consistência e inconsistência de um axioma (que ocorre nas condições que são espectáveis: inconsistente se é uma contradição, consistente caso contrário) e uma relação de implicação entre axiomas (se A axioma é tal que B axioma é verdadeiro sempre que A é verdadeiro estamos neste caso);

ENTIDADES – Elementos usados para referir a coisas do mundo real. Uma entidade é simplesmente um objecto abstracto da vida real como por exemplo "cão", "chuva", "bola", ... Também será chamado de elemento, instância ou objecto ao longo desta secção;

EXPRESSÕES – Combinações de entidades para gerar descrições complexas a partir de elementos mais simples. Em geral expressões são descrições de entidades como "A Maria é alta" ou "A chuva é forte", ou são relações entre diferentes entidades como "A Maria e o José são irmãos";

CONSTRUTORES – Uma das características principais do OWL é a existência de construtores que permitem combinar entidades para formar expressões. Por exemplo, é possível utilizar um construtor para combinar as entidades "professor" e "informática" e criar a expressão "professor de informática".

Com estes conceitos definidos podemos passar à primeira grande expansão feita por OWL: mudanças feitas a classes, indivíduos dessas classes e a suas propriedades.

2.3.1 *Classes*

Uma classe OWL representa uma série de axiomas que descrevem as condições necessárias (ou em numerosos casos as condições necessárias e suficientes) para se poder concluir que determinado indivíduo pertence à classe, pelo que qualquer instância ou indivíduo de uma classe tem de ser tal que verifica todos os axiomas definidos pela classe.

Deve-se notar que para indicar que um elemento é de uma classe OWL usa-se, à semelhança do que se viu anteriormente, o predicado `rdf:type` e do mesmo modo que em SKOS existem relações entre classes tanto hierárquicas como não hierárquicas. Alguns exemplos de definição de classes e de relações entre estas são apresentados em seguida:

OWL:EQUIVALENTCLASS - relação que representa a equivalência entre duas classes de elementos

```
[]  rdf:type     owl:equivalentClass ;
    owl:members  ( :Car  :Automobile ) .
```

Código 2.9: Equivalência de classes em OWL

OWL:ALLDISJOINTCLASSES - relação que representa a disjunção de classes, i.e., duas classes que não compartilham qualquer elemento

```
[]  rdf:type     owl:AllDisjointClasses ;
    owl:members  ( :Woman  :Man ) .
```

Código 2.10: Dijunção de classes em OWL

OWL:INTERSECTIONOF - representa a intercepção dos elementos de duas classes

```
 :Mother  owl:equivalentClass  [
  rdf:type            owl:Class ;
  owl:intersectionOf  ( :Woman :Parent )
] .
```

Código 2.11: Intersecção de classes em OWL

OWL:UNIONOF - representa a união entre os elementos de duas classes

```
:Parent  owl:equivalentClass  [
  rdf:type     owl:Class ;
  owl:unionOf  ( :Mother :Father )
] .
```

Código 2.12: União de classes em OWL

OWL:COMPLEMENTOF - representa o complemento de uma classe, i.e., todos os elementos da ontologia que não são parte da classe especificada

```
:ChildlessPerson  owl:equivalentClass  [
  rdf:type  owl:Class ;
  owl:intersectionOf
       ( :Person
           [ rdf:type owl:Class ;
           owl:complementOf :Parent ]
       )
 ] .
```

Código 2.13: Complemento de classe em OWL

ENUMERAÇÃO DE ELEMENTOS - pode-se definir uma classe por simples enumeração dos seus elementos

```
:MyBirthdayGuests  owl:equivalentClass  [
  rdf:type   owl:Class ;
  owl:oneOf  ( :Bill  :John  :Mary )
] .
```

Código 2.14: Definição de classe por enumeração em OWL

Mas a maneira mais importante que foi adicionada para criar classes OWL é a do uso de restrições de propriedades das suas entidades. Para fazermos esta criação usamos as seguintes propriedades de classe:

OWL:SOMEVALUESFROM - representa a quantificação existencial, onde todos os indivíduos pertencentes à classe têm que ter pelo menos uma relação com outro indivíduo por uma certa relação.

```
:Parent  owl:equivalentClass  [
  rdf:type            owl:Restriction ;
  owl:onProperty      :hasChild ;
  owl:someValuesFrom  :Person
] .
```

Código 2.15: Quantificação existencial em OWL

OWL:ALLVALUESFROM - representa a quantificação universal, onde todos os indivíduos pertencentes à classe estão relacionados exclusivamente com indivíduos de uma certa classe por uma certa relação.

```
:HappyPerson  rdf:type    owl:Class ;
              owl:equivalentClass  [
      rdf:type owl:Class ;
      owl:intersectionOf  ( [ rdf:type owl:Restriction ;
                              owl:onProperty :hasChild ;
                              owl:allValuesFrom :Happy ]
                            [ rdf:type owl:Restriction ;
                              owl:onProperty      :hasChild ;
                              owl:someValuesFrom  :Happy ]
                          )
    ] .
```

Código 2.16: Quantificação universal em OWL

Pode-se ainda usar valores específicos para descrever classes de indivíduos. Por exemplo, podemos definir a classe dos filhos do John da seguinte forma:

```
:JohnsChildren  owl:equivalentClass  [
  rdf:type        owl:Restriction ;
  owl:onProperty  :hasParent ;
  owl:hasValue    :John
] .
```

Código 2.17: Descrição de classes por enumeração em OWL

Ou seja, todos os indivíduos desta classe são sujeitos de um triplo que tem como predicado `:hasParent` e como objeto `:John`.

Um caso específico disto é a relação ser feita do indivíduo para ele próprio, como no exemplo seguinte:

```
:NarcisticPerson owl:equivalentClass  [
  rdf:type        owl:Restriction ;
  owl:onProperty  :loves ;
  owl:hasSelf     true .
] .
```

Código 2.18: Relação de um individuo com ele próprio em OWL

2.3.2 *Elementos de classes e suas propriedades*

Todos os elementos de uma ontologia são, por omissão, considerados elementos únicos no mundo.

Para se poder dizer numa ontologia que duas entidades representam a mesma coisa usamos `owl:sameAs` :

```
:James owl:sameAs :Jim.
```

Código 2.19: Igualdade entre entidades numa ontologia OWL

Podemos também dizer explicitamente que são entidades diferentes usando `owl:differentFrom`:

```
:John owl:differentFrom :Bill .
```

Código 2.20: Diferença explicita entre entidades em OWL

Consideremos a frase "Mary é noiva de John". Em RDF exprimimos isto usando uma propriedade:

```
:John :hasWife :Mary .
```

Código 2.21: Exemplo simples de uma relação em RDF

OWL introduz o complemento para propriedades, onde é possível dizer que duas entidades não estão relacionadas por uma certa relação numa ontologia através do uso de `owl:NegativePropertyAssertion` :

```
[]  rdf:type               owl:NegativePropertyAssertion ;
    owl:sourceIndividual   :Bill ;
    owl:assertionProperty  :hasWife ;
    owl:targetIndividual   :Mary .
```

Código 2.22: Expressão de não relação entre duas entidades em OWL

Para além disto podemos ainda definir hierarquias entre propriedades usando `rdfs:subPropertyOf` e impor restrições de domínio (*domain*) e alcance (*range*):

```
:hasWife rdfs:domain :Man ;
         rdfs:range  :Woman .
```

Código 2.23: Restrições a propriedades de elementos de um ontologia OWL

Para além destas OWL permite adicionar restrições à cardinalidade das propriedades com uso de `owl:maxQualifiedCardinality` (valor máximo que a propriedade pode estar associada a uma entidade com certa classe), `owl:minQualifiedCardinality` (similar à anterior, mas para o valor mínimo), `owl:qualifiedCardinality` (mesmo que as anteriores, mas valor exacto) e `owl:cardinality` (similar aos anteriores, mas as entidades não têm de ter a classe especificada).

Com todas as restrições tratadas passamos agora ao que OWL acrescenta aos diferentes tipos de propriedades e relações.

Os tipos de propriedades que podemos ter em OWL são:

OWL:SYMMETRICPROPERTY E OWL:ASYMMETRICPROPERTY - denotam que uma propriedade é simétrica (quando usada numa relação também se verifica a mesma no sentido inverso) ou assimétrica, respectivamente;

OWL:REFLEXIVEPROPERTY E OWL:IRREFLEXIVEPROPERTY - denotam que uma propriedade é reflexiva (permite relacionar a entidade a si própria) ou irreflexiva, respectivamente;

OWL:FUNCTIONALPROPERTY E OWL:INVERSEFUNCTIONALPROPERTY - ambas denotam uma propriedade que pode ser associada uma única vez a um individuo da ontologia sendo ele o objecto ou o sujeito do triplo, respectivamente.

Por exemplo: `hasHusband` é funcional porque todas as pessoas terão apenas um marido no máximo:

```
:hasHusband  rdf:type  owl:FunctionalProperty .
```

Código 2.24: Propriedade funcional em OWL

E é inversamente funcional porque todas as pessoas serão, no máximo, referidas como maridos numa única outra entidade:

```
:hasHusband  rdf:type  owl:InverseFunctionalProperty .
```

Código 2.25: Propriedade inversamente funcional em OWL

OWL:TRANSITIVEPROPERTY - indica uma propriedade é transitiva (se a propriedade X é transitiva então se A e B estão relacionados pela propriedade X e B e C também concluímos que A e C estão relacionados por X):

```
    :hasParent  rdf:type  owl:TransitiveProperty .
```

Código 2.26: Propriedade transitiva em OWL

OWL:INVERSEOF - denota duas propriedades que são o inverso uma da outra, i.e., X e Y serem relações inversas leva a que se A estiver relacionado com B por X então B está relacionado com A por Y;

```
:hasParent owl:inverseOf :hasChild
```

Código 2.27: Propriedades inversas em OWL

OWL:PROPERTYDISJOINTWITH - denota duas propriedades disjuntas, i.e., propriedades que nunca podem aparecer relacionadas com o mesmo elemento;

```
:isHot  owl:propertyDisjointWith  :isCold
```

Código 2.28: Propriedades disjuntas em OWL

OWL:PROPERTYCHAINAXIOM - representa uma cadeia de propriedades, i.e., define uma relação como sendo a conjunção de várias relações distintas numa certa ordem.

```
:hasGrandparent  owl:propertyChainAxiom  ( :hasParent  :hasParent ) .
```

Código 2.29: Cadeia de propriedades em OWL

2.4 REIFICAÇÃO

Um conceito que não faz parte de nenhuma das linguagens para ontologias mas que é bastante importante quando estamos a especificar uma dada ontologia é o de reificação de uma relação.

Para compreender este conceito temos de lembrar que uma relação entre entidades é uma simples propriedade, como `:hasParent`, ou `:isFrom`. Como tal, não temos maneira nenhuma de adicionar contexto a essa relação sem alterar como a relação é feita.

Por exemplo, se houver três cidades (Porto, Braga e Guimarães) e tivermos uma relação `:temEstradaPara` que liga umas às outras, não temos maneira de adicionar dados, por exemplo, a distância que é preciso percorrer para ir de uma cidade para a outra.

Para adicionarmos esta informação extra temos de alterar a relação para ser uma nova classe (vamos chamar-lhe `estradaEntreCidades`) que contenha em si os dados extra que queremos adicionar.

Por exemplo, se antes tínhamos

```
:Braga :temEstradaPara :Porto.
:Braga :temEstradaPara :Guimaraes.
```

```
:Porto :temEstradaPara :Braga.
:Porto :temEstradaPara :Guimaraes.

:Guimaraes :temEstradaPara :Braga.
:Guimaraes :temEstradaPara :Porto.
```

Código 2.30: Pseudo-ontologia antes de reificação

Podemos passar agora a ter:

```
:estrada1 rdf:type :estradaEntreCidades.
:estrada1 :distancia 20;
          :origem :Braga;
          :destino :Porto .

:estrada2 rdf:type :estradaEntreCidades
:estrada2 :distancia 35;
          :origem :Braga;
          :destino :Guimaraes .

:estrada3 rdf:type :estradaEntreCidades
:estrada3 :distancia 55;
          :origem :Porto;
          :destino :Guimaraes .
```

Código 2.31: Pseudo-ontologia após reificação

A este processo chamamos reificação de uma relação e corresponde a criar um nível extra sobre a relação promovendo esta a classe de forma a instanciarmos a relação como indivíduo da classe ao qual podemos adicionar as propriedades que entendermos.

2.5 Resumo

Neste capítulo introduzimos a estrutura de dados mais importante para este trabalho, a ontologia, bem como várias linguagens relevantes para especificar as serializações delas num formato conciso e compreensível.

Falamos de RDF/RDFS, representação base que todas as outras faladas aqui expandem, explicitando conceitos muito importantes para compreender o trabalho como identificadores de elementos de ontologia (URI/IRI) e composição de triplos neste tipo de escrita de ontologias, bem como algumas propriedades lógicas a estes associadas.

Referimos SKOS, expansão de RDFS, que introduz conceitos e hierarquia entre eles, etiquetas, relações semânticas e documentação a ontologias.

Finalmente descrevemos o OWL, linguagem baseada em RDFS e SKOS e que adiciona numerosas capacidades para as ontologias conseguirem exprimir conhecimento, como classes e suas relações bem como elementos de classes e suas propriedades.

Adicionalmente explicamos o conceito de reificação de uma relação, processo pelo qual uma relação passa de propriedade a elemento da ontologia de modo a adicionar contexto.

3

BASES DE DADOS ORIENTADAS A GRAFOS

Uma base de dados orientada a grafos (Angles and Gutierrez (2008)) é uma base de dados NoSQL (Archive (2020)) que utiliza estruturas similares a grafos para guardar informação.

Este tipo de bases de dados utiliza nodos para representar elementos na base de dados e arestas para representar relações entre esses nodos, podendo ambos ter propriedades associadas (dependendo do tipo de motor usado).

Para o nosso caso de estudo consideraremos apenas bases de dados deste tipo onde as relações têm uma direcção associada, visto que esta propriedade é fundamental para representar ontologias do tipo OWL.

Nesta secção vamos introduzir os dois motores de bases de dados orientadas a grafos relevantes para esta tese: Neo4J e GraphDB.

3.1 UMA NOTA SOBRE RDF4J

RDF4 (Eclipse (2020)) é uma *framework* escrita em Java para trabalhar com RDF com suporte para a linguagem de *query* SPARQL e com a seguinte arquitectura:

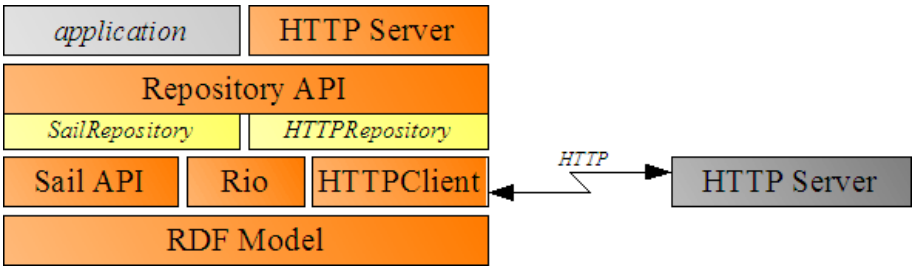


Figura 1: Arquitectura RDF4J

Existem duas maneiras principais de usar RDF4J: como um servidor separado ou como uma biblioteca Java dentro de uma aplicação, e é composto de uma série de componentes modulares que podem facilmente ser substituídas por implementações alternativas e uma série de implementações de SAIL.

SAIL (Ontotext (2020b)) é um conjunto de interfaces Java que permitem abstrair o mecanismo de guardar dados RDF de modo a simplificar o processo de trabalhar com estes.

Geralmente a comunicação com RDF4J é feita através da API que pode ser implementada de numerosas maneiras (incluindo uma através de HTTP).

A razão para estarmos a mencionar isto deve-se ao facto que tanto GraphDB como Neo4j farão referência a esta ferramenta, sendo assim importante mencionar separadamente para oferecer o contexto apropriado.

3.2 GRAPHDB

GraphDB (Ontotext (2020a)) é um motor de bases de dados orientadas a grafos que usa ontologias OWL para guardar dados permitindo a exploração eficaz dos mesmos através das várias capacidades que disponibiliza, desde uma visualização gráfica da base de dados até uma caixa para fazer *queries* directamente, bem como uma listagem dos elementos das várias ontologias nela presentes.

A linguagem usada para fazer as *queries* às várias bases de dados é SPARQL, fazendo uso da API SAIL (Ontotext (2020b)) para RDF4J e tendo a seguinte arquitectura:

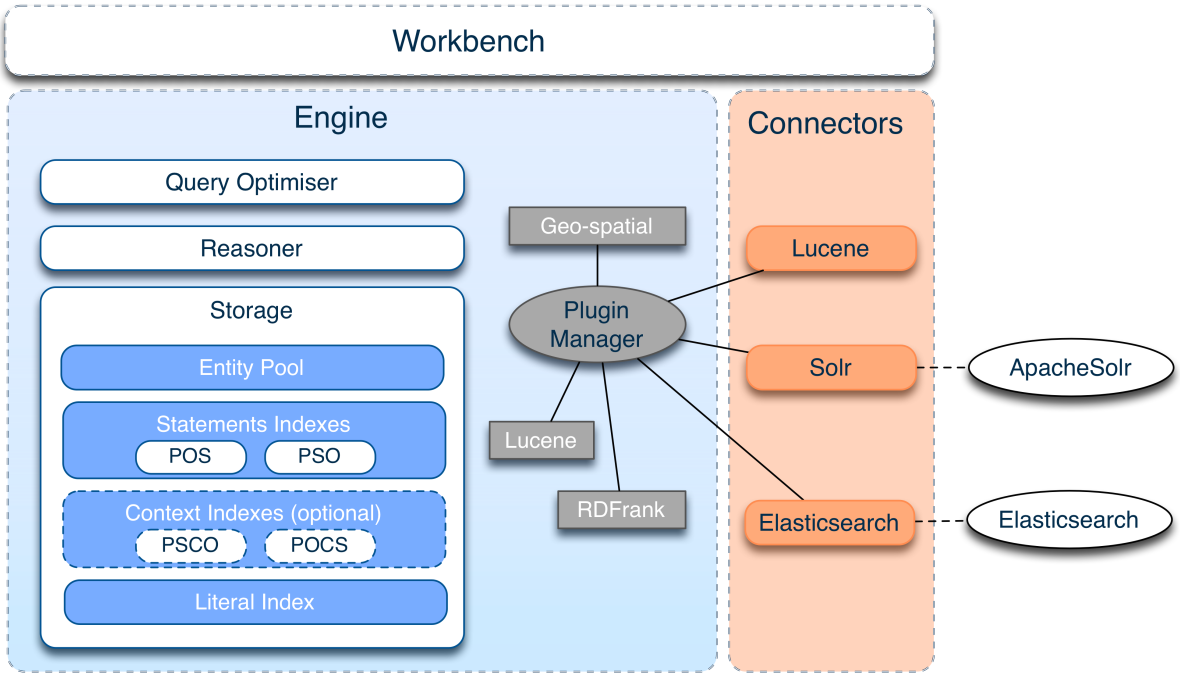


Figura 2: Arquitetura GraphDB

Como se pode verificar na figura acima, existem três partes principais que criam o GraphDB: *Engine*, *Connectors* e *Workbench*.

3.2.1 *Engine*

O *Engine* do GraphDB é feito de numerosas partes díspares, servindo de motor que permite o funcionamento da aplicação WEB.

As partes principais deste motor são:

QUERY OPTIMIZER - componente que tenta optimizar qualquer query que seja feita ao motor, procurando a maneira mais eficiente de obter os dados da *query* que foi feita a uma das suas ontologias através do uso de *query plans* (GraphDB (2020b));

REASONER - componente que faz interpretação das *queries* de modo a determinar como executar a *query* pedida.

GraphDB utiliza TRREE, *Forward-chaining* (GraphDB (2020c)) sobre os triplos da ontologia e *Total materialization* (estratégia onde as regras de inferência são repetidamente aplicadas até não se conseguir inferir mais nada sobre uma *query*, GraphDB (2020d));

STORAGE - todos os dados do GraphDB são guardados nesta directoria que é definida dentro das configurações da ferramenta.

Dentro da *storage* existem numerosos subcomponentes, dos quais realçamos a *Entity Pool*, cujo trabalho é converter entidades OWL (URI's, nodos vazios e literais) em id's internos (números inteiros com trinta e dois a quarenta dígitos em comprimento).

3.2.2 *Connectors*

Connectors permitem fazer procuras e agregações de dados extremamente eficazes e são geralmente associadas a serviços e componentes externos ao GraphDB.

O GraphDB vem com o conector Lucene (GraphDB (2020a)) já implementado, permite optimizar as *queries*, sendo todos os outros desenvolvidos e disponibilizados para serviços externos.

3.2.3 *Workbench*

A *Workbench* é uma ferramenta Web de administração de bases de dados baseada que permite comunicar com as bases de dados do GraphDB.

Esta ferramenta oferece numerosas capacidades, incluindo:

1. Capacidade de gestão de repositórios, permitindo o seu importe, uso, edição e remoção

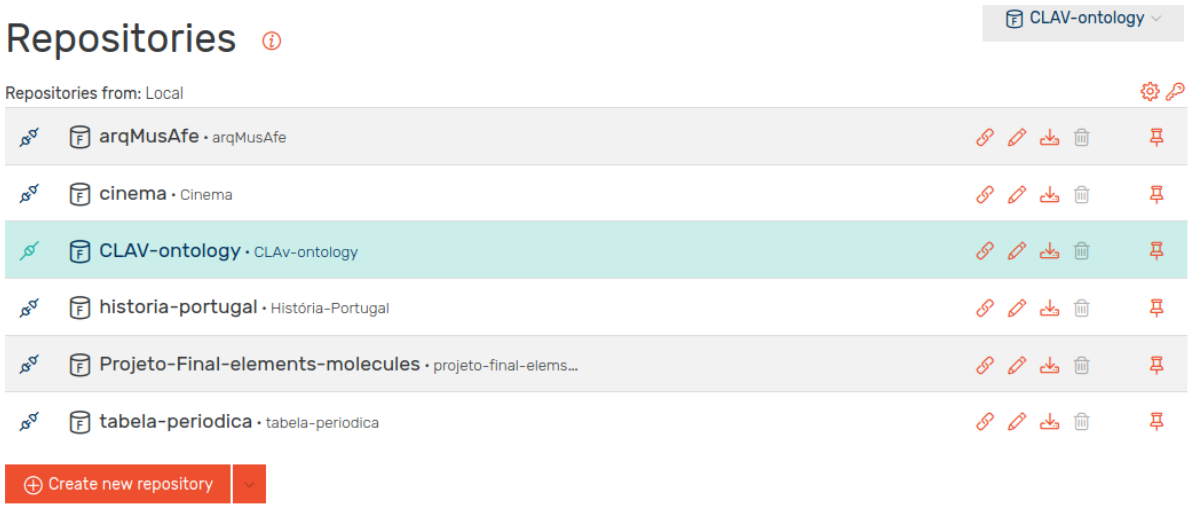


Figura 3: Gestão de ontologias em GraphDB

2. Capacidade de fazer *queries* às ontologias nela carregadas, permitindo a sua interrogação direta

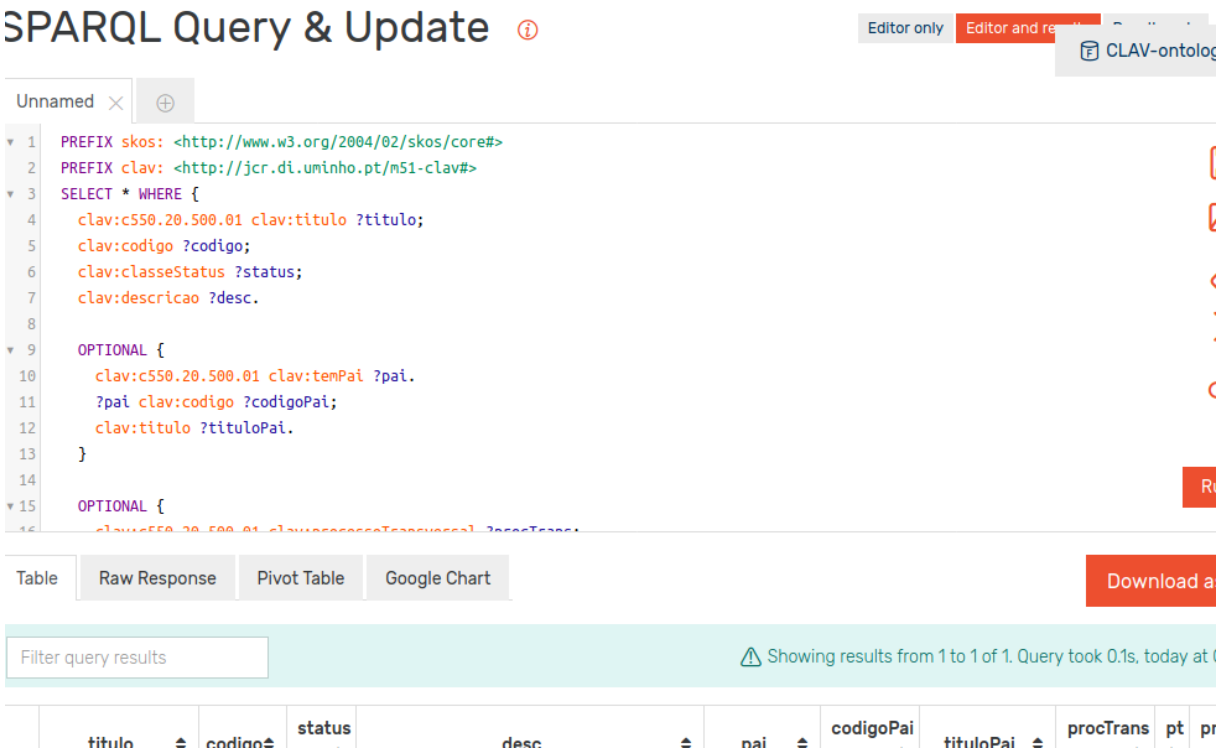


Figura 4: Fazer *queries* a ontologias em GraphDB

3. Capacidade de explorar elementos e relações da ontologia a ser acedida, permitindo observar toda a informação sobre algum elemento dela

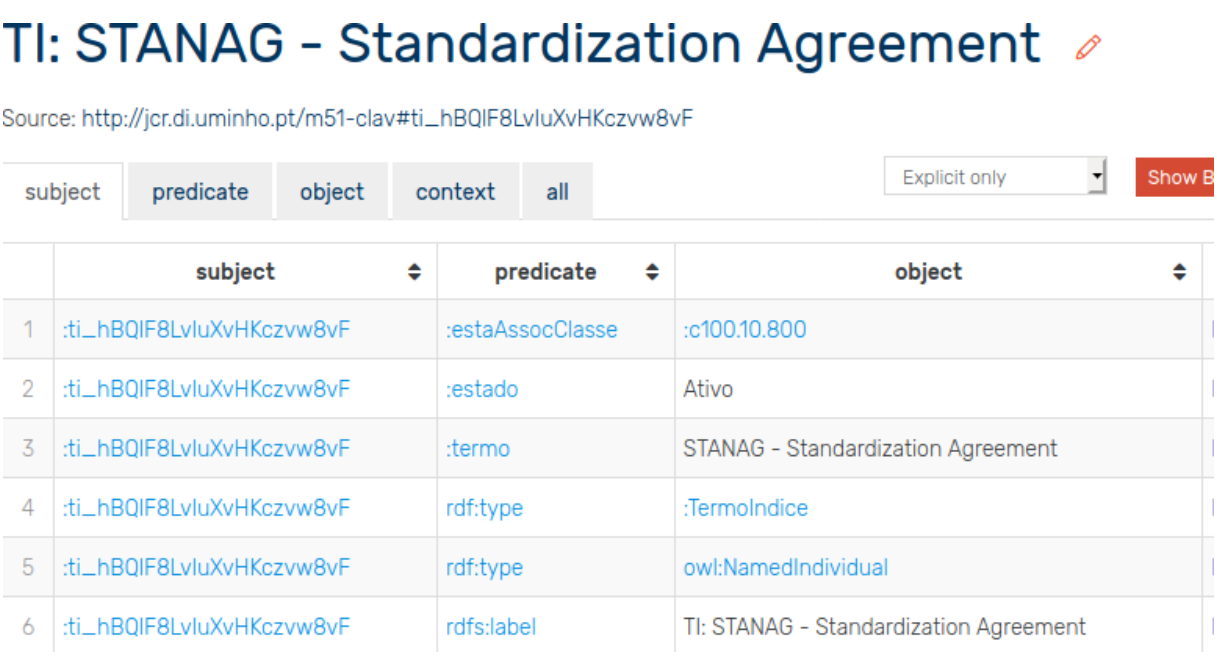


Figura 5: Explorar elementos de ontologias em GraphDB

4. Capacidade de visualizar um grafo de um elemento da ontologia, permitindo explorar a mesma informação mostrada na capacidade anterior de uma forma diferente

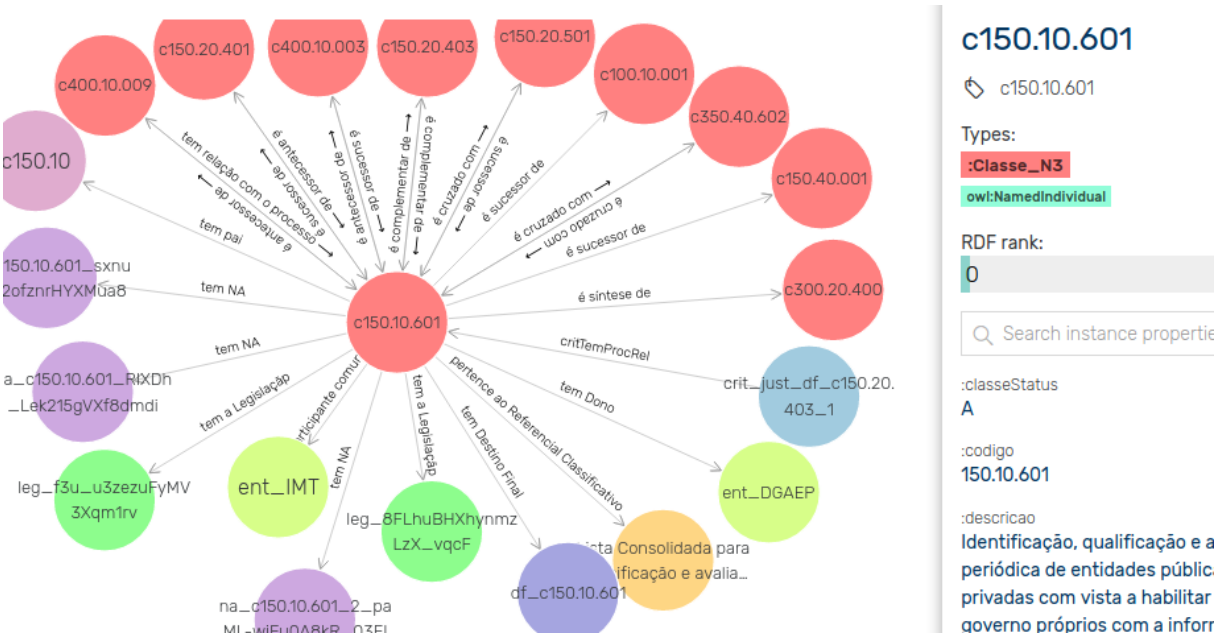


Figura 6: Visualizar o grafo de ditos elementos em GraphDB

3.3 NEO4J

Neo4j (Neo4j (2020)) é um motor de bases de dados orientadas a grafos que utiliza listas ligadas (Neo4J (2020e)) para representar os dados, administrado a partir de uma interface Web que

pode ser instalada localmente (Neo4j Browser) ou acedida online em Neo4J (2020g) (Neo4J Sandbox) e que utiliza a linguagem de *query* CYPHER.

Nas base de dados utilizadas por Neo4J cada elemento é representado por um nodo que contém uma série de `labels` (tipos associados ao nodo), propriedades (campos de dados associado ao nodo) e relações com outros nodos (que são, elas próprias, nodos especiais que têm associados tipos e propriedades especificas).

O Neo4J tem vários componentes descritos com detalhe em Neo4J (2020c). Destes apenas dois são relevantes para este trabalho: **Neo4j Graph Database** e **Neo4j Browser**.

3.3.1 *Neo4j Graph Database*

Em Neo4J a parte mais importante é a base de dados usada para guardar os grafos, estando optimizada para trabalhar com estes.

Esta base de dados oferece numerosas capacidades (Neo4J (2020a)), nomeadamente:

MÚLTIPLAS BASES DE DADOS É possível operar múltiplas bases de dados numa única instalação;

FABRIC Neo4J Fabric permite a separação de dados pelos utilizadores, permitindo o acesso de fragmentos da base de dados quando necessário;

CONTROLO DE ACESSO A DADOS Neo4J implementa um sistema de utilizadores e permissões a estes associadas, permitindo maior segurança e controlo dos dados guardados;

DRIVERS Neo4J tem opção de acesso por uso de drivers intermédios que permitem controlar acessos entre utilizadores e bases de dados de acordo com as necessidades observadas.

3.3.2 *Neo4j Browser*

Contrariamente ao GraphDB, o Neo4J tem uma interface de *browser* bastante simples, contendo apenas um ecrã com a linha para fazer *queries* à base de dados e numerosos menus de lado que contêm informação e opções extra:

Figura 7: Página inicial de Neo4J Enterprize Browser

Estes menus contêm:

1. Informação sobre as bases de dados a serem exploradas, contendo a informação sobre os tipos de elementos presentes na base de dados atual e a listagem de todas as bases de dados que estão atualmente disponíveis, permitindo mudar diretamente para qualquer uma delas

2. Listagem de *queries*, que não só tem numerosos exemplos de *queries* básicas como permite guardar *queries* do utilizador

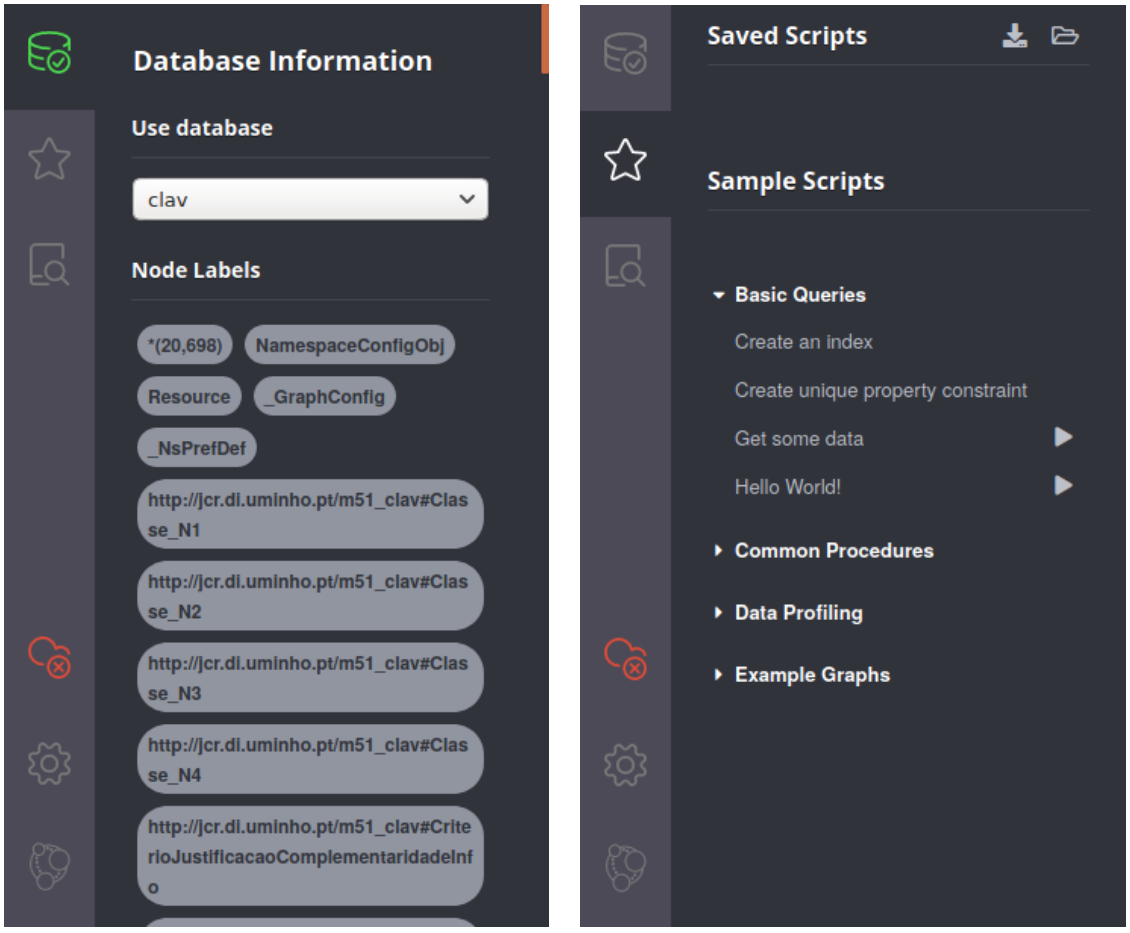


Figura 8: Informação sobre a base de dados (esquerda) e capacidade de guardar *queries* e exemplos destas (direita) de Neo4J Enterprize Browser

3. Uma série de links para as páginas da documentação do programa, detalhando a funcionalidade das várias partes da aplicação

4. Definições da aplicação, permitindo alterar numerosos parâmetros de modo a melhorar a experiência de utilização (tema, número máximo de *queries* guardadas em memória, tempo máximo de espera por resultados, ...)

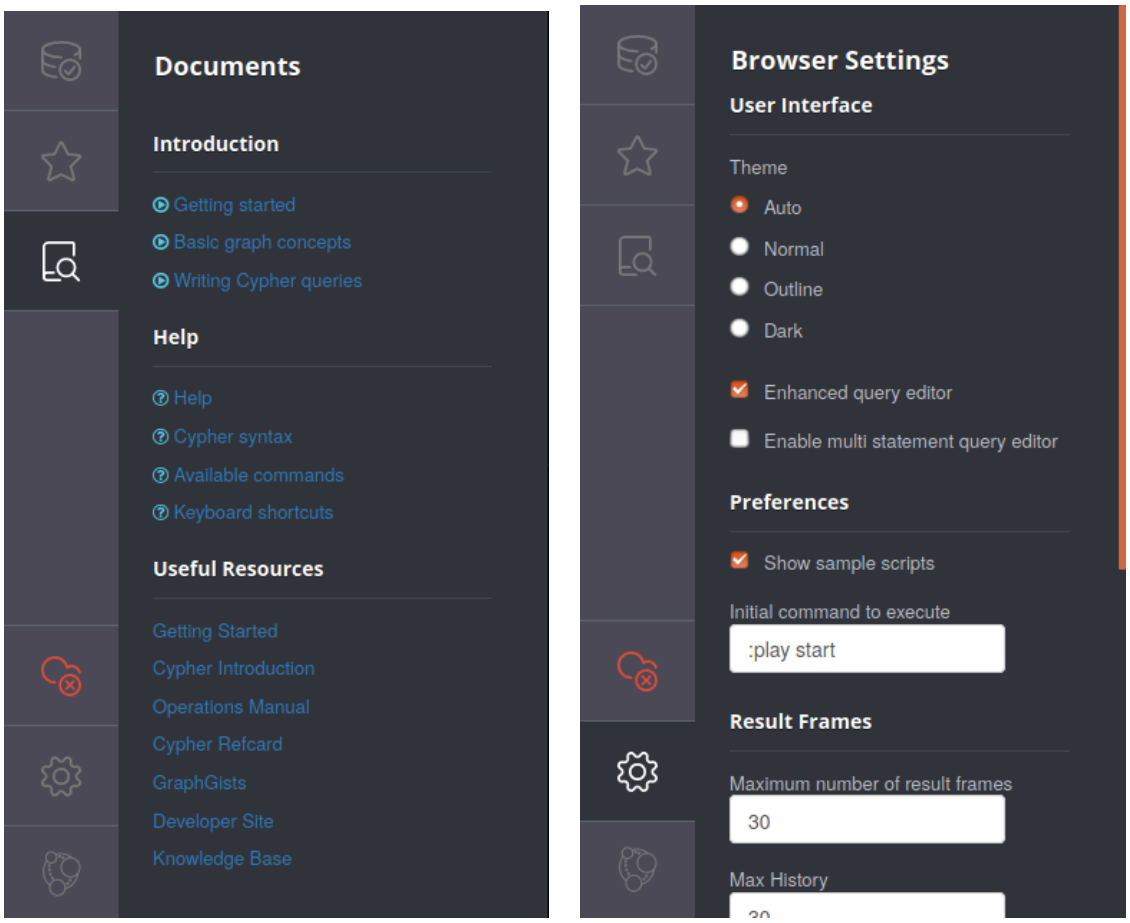


Figura 9: Links para documentação (esquerda) e definições da aplicação (direita) de Neo4J Enterprize Browser

3.3.3 *Neosemantics*

Para esta tese estamos a tentar criar um mapeamento entre CYPHER e SPARQL, usando para isso uma ontologia OWL. No entanto, a plataforma Neo4J só disponibiliza importação de dados em formato CSV, o que não é suficiente para descrever uma ontologia.

Para ultrapassar isto estamos a expandir as capacidades do Neo4J usando Neosemantics (neo4j labs (2020)), que permite importação dos numerosos tipos de ontologias, mencionadas anteriormente neste documento, para poderem ser usadas dentro do Neo4J.

Note-se que esta extensão faz parte do Neo4j Labs (Neo4j (2020)), uma colecção de bibliotecas cujo objectivo é de aumentar as capacidades do Neo4J que são desenvolvidas pela comunidade e reconhecidas no site oficial de Neo4J.

Esta extensão utiliza RDF4J para fazer análise e serialização de ontologias OWL (descritas na documentação como sendo RDF) de modo similar ao GraphDB.

O processo de importação de dados é descrito em Labs (2020) e Barrasa (2016) e é extremamente relevante para compreender o trabalho feito na tradução entre linguagens, pelo que será agora descrita em detalhe[1].

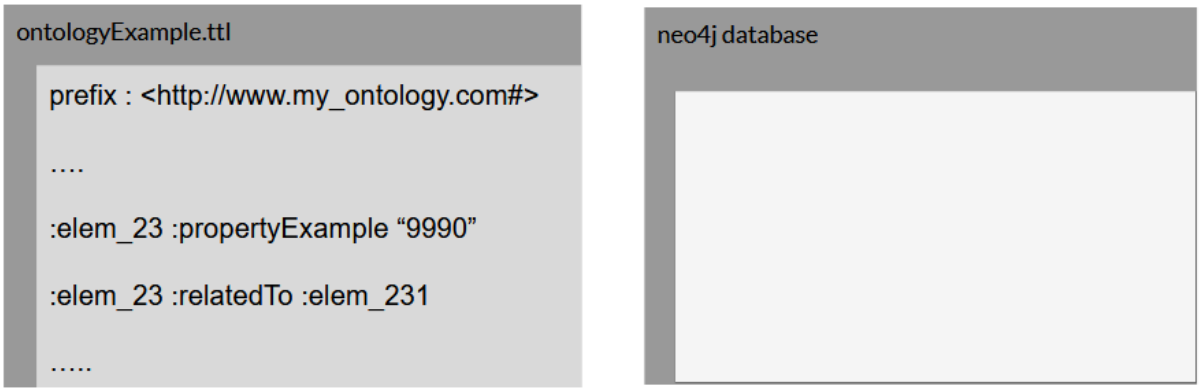


Figura 10: Diagrama representante do processo de importe de dados: à esquerda vemos parte de uma ontologia escrita utilizando sintaxe TURTLE e à direita vemos parte da base de dados em Neo4J para onde estamos a importa-la

Este processo é aplicado a cada triplo da ontologia a ser importada da seguinte maneira:

1. Primeiro é criado um elemento na base de dados, caso não exista, que corresponderá ao sujeito do triplo (note-se que os elementos são identificados no Neo4J através de uma propriedade URI única que lhes é associada);

2. Em seguida é testado se o objecto do triplo é um valor literal ou um URI. Se for um valor literal estamos perante uma propriedade do sujeito, pelo que é adicionada ao elemento que foi criado anteriormente a propriedade cujo nome é o predicado do triplo e cujo valor é o valor literal que está presente no objecto;

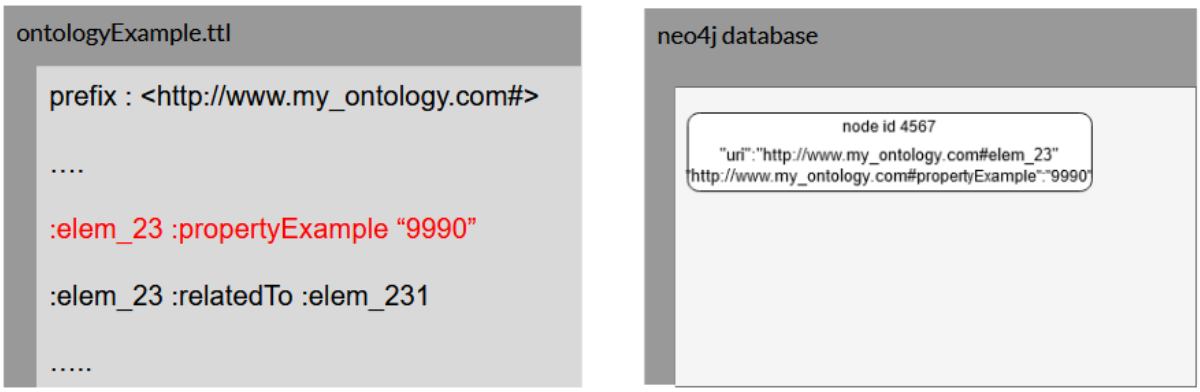


Figura 11: Diagrama representante do que é descrito acima: a linha a vermelho está a ser processada e o seu resultado na base de dados pode ser observado à direita

Se for um URI estamos na presença de uma relação entre dois elementos da ontologia. Logo é adicionado um elemento à ontologia caso não exista que corresponde ao objecto

1 Note-se que esta informação assume que o processo de importação foi feito usando as opções de importação descritas em 7.2.4

do triplo e os dois elementos envolvidos no triplo são ligados através de uma relação cujo tipo é o predicado;

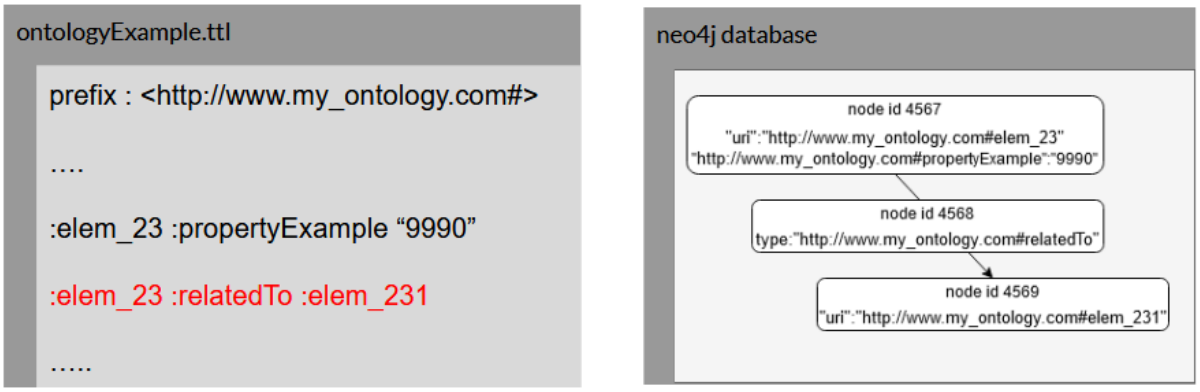


Figura 12: Diagrama representante do que é descrito acima: a linha a vermelho está a ser processada e o seu resultado na base de dados pode ser observado à direita

A partir disto podemos já concluir que a distinção entre o objecto literal e URI vai ser muito importante para a tradução que será feita.

Note-se adicionalmente que, como consequência das relações em Neo4J poderem ter propriedades associadas a elas, todas as relações da ontologia que são importadas para Neo4J são reitificadas automaticamente.

3.4 Resumo

Neste capítulo introduzimos o tipo de bases de dados que serão utilizadas ao longo do trabalho: bases de dados orientadas a grafos.

Para além de explicar o que são falamos de dois tipos de motores que usam este tipo de estruturas para guardar dados: GraphDB e Neo4J, explicando o seu funcionamento base e alguns detalhes técnicos.

Adicionalmente falamos da extensão Neosemantics, usada para permitir a importação e manipulação de dados de ontologias para Neo4J, explicando em detalhe como os dados são processados devido à relevância que este processo tem para a gramática que será desenvolvida.

4

LINGUAGENS DE QUERY

Uma linguagem de *query* corresponde a uma linguagem utilizada para interrogar uma base de dados, i.e., uma linguagem usada para obter dados mediante condições específicas.

Neste documento apenas usamos ontologias como base de dados.

4.1 SPARQL

SPARQL (W3C (2013a)) é uma linguagem de *query* criada para explorar ontologias RDF, utilizada em numerosos contextos e como linguagem para explorar ontologias em GraphDB.

4.1.1 *Estratégia de resposta a uma query*

Uma *query* que seja feita em SPARQL tem avaliação muito simples devido à similaridade entre a sintaxe da linguagem (que utiliza triplos) e o meio de guardar dados (ontologia):

Para cada triplo da *query* faz-se *pattern matching* com a base de dados, retornando isto um conjunto de elementos que verificam o que é pedido pelo triplo. Em seguida faz-se intersecção entre estes conjuntos individuais para cada triplo e obtêm-se os resultados finais da *query*.

Este algoritmo básico tem algumas coisas que queremos realçar, como o facto que é possível filtrar estes conjuntos para apenas obter certos elementos (`FILTER`) e que é possível alterar a disposição dos dados finais em certas maneiras (`ORDER BY`, `LIMIT`, ...).

4.1.2 *Palavras chave*

Na sintaxe de SPARQL existem várias palavras chave, das quais destacaremos:

PREFIX - permite-nos utilizar *namespaces* na nossa *query* SPARQL, simplificando tremendamente a escrita e leitura desta.

Se nenhum for escrito explicitamente são assumidos os seguintes prefixos em qualquer *query* SPARQL:

```
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
```

```
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
PREFIX fn: http://www.w3.org/2005/xpath-functions#
```

Código 4.1: Prefixos pré-definidos em SPARQL.

SELECT - seleciona os valores a retornar dos que foram capturados pela *query*. Nota-se o caso especial `SELECT` $*$ onde todos os dados da *query* são retornados. Em geral aparece no topo da *query* (imediatamente após as definições dos prefixos) e é seguida de um `WHERE` onde a *query* é especificada.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?nameX ?nameY
WHERE
  { ?x foaf:knows ?y ;
       foaf:name ?nameX .
    ?y foaf:name ?nameY .
  }
```

Código 4.2: Palavra chave `SELECT` em SPARQL

DISTINCT - modificador de `SELECT`, usado para remover linhas duplicadas dos resultados finais.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name WHERE { ?x foaf:name ?name }
```

Código 4.3: Palavra chave `DISTINCT` em SPARQL

ORDER BY - ordena os resultados da *query* com base em certos campos de dados dessa mesma *query*, tendo as selecções mais à esquerda prioridade na ordenação.

Note-se que os elementos que podem ser colocados após o `ORDER BY` têm de estar presentes nos resultados da *query* ou de uma operação feita sobre os mesmos.

Note-se ainda outra palavra chave, `DESC`, que inverte a ordem normal de ordenação para a selecção a que está associada.

Exemplo (retorna uma lista de nomes ordenada primeiro alfabeticamente e depois modo descendente com base na sua empresa):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
```

```
WHERE {
    ?x foaf:name ?name ;
       :empId ?emp
}
ORDER BY ?name DESC(?emp)
```

Código 4.4: Palavra chave `ORDER BY` em SPARQL

OPTIONAL - uma *query* que seja precedida de `OPTIONAL` retornará uma coluna vazia em todos os dados que não encontrar resposta ao que está a pedir, sendo normalmente utilizado no meio de outra *query* para especificar variáveis opcionais de outro elementos.

Exemplo (retornará uma lista de nome com os valores *mbox* que lhe são correspondentes caso eles existam):

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }
```

Código 4.5: Palavra chave `OPTIONAL` em SPARQL

UNION - faz a união de duas queries distintas que contêm colunas de resultado similares, juntando os seus resultados.

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?title
WHERE  { { ?book dc10:title  ?title } UNION { ?book dc11:title  ?title } }
```

Código 4.6: Palavra chave UNION em SPARQL

LIMIT - limita o número de linhas do resultado da *query*, que pode ser no máximo o valor indicado após o `LIMIT`.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE { ?x foaf:name ?name }
LIMIT 20
```

Código 4.7: Palavra chave `LIMIT` em SPARQL

FILTER - restringe todas as linhas do resultado a linhas que verificam a expressão, removendo todas as que não a verifiquem.

Exemplo (retorna todas as anotações que tenham sido feitas após 1/1/2015):

```
PREFIX a: <http://www.w3.org/2000/10/annotation-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?annot
WHERE { ?annot  a:annotates  <http://www.w3.org/TR/rdf-sparql-query/> .
        ?annot  dc:date      ?date .
        FILTER ( ?date > "2005-01-01T00:00:00Z"^^xsd:dateTime ) }
```

Código 4.8: Palavra chave `FILTER` em SPARQL

4.1.3 *Queries de exemplo*

Para finalizar a introdução da linguagem colocaremos algumas queries de exemplo extra:

1. Esta *query* retornará os elementos `?id` que contém uma relação `:temPai` para `?pai`, bem como esses mesmos elementos:

```
select * where{
    ?id :temPai ?pai
}
```

Código 4.9: Primeira *query* exemplo SPARQL

2. Esta *query* retornará todos os elementos `?id` e respectivo atributo `:codigo` onde o seu atributo `:numero` é igual a 200:

```
select ?id ?codigo where{
    ?id :numero 200
        :codigo ?codigo
}
order by ?id
```

Código 4.10: Segunda *query* exemplo SPARQL

4.2 CYPHER

CYPHER (Neo4J (2020b)) é uma linguagem de *query* usada primariamente por Neo4J como forma de fazer procuras nas suas bases de dados.

4.2.1 *Estratégia de resposta a uma query*

Uma *query* que seja feita em CYPHER é decomposta em operadores (Neo4J (2020)) que implementam uma parte do trabalho necessário a fazer. Estes operadores são depois combinados numa estrutura similar a uma árvore que é chamada de plano de execução.

Nesta estrutura um operador é representado por um nodo da árvore que recebe zero ou mais colunas e retorna zero ou mais colunas, servindo o que retorna como entrada de outro operador na árvore. Existem ainda operadores que combinam ramos da árvore a receberem *inputs* de dois lados mas retornam apenas um *output*.

O modelo de avaliação do Neo4J começa a avaliar as folhas da árvore (que geralmente não têm *input* e são operações de obter colunas de dados da BD), sendo que os resultados da operação dos nodos é dado aos nodos pais para avaliar, repetindo-se este processo até chegar ao topo da árvore, onde o nodo raiz produzirá os resultados finais da *query*.

Note-se que em geral a avaliação da *query* é *lazy*, com os dados a serem mandados dos filhos para os pais sem ter terminado completamente a sua operação. Mas existem alguns em particular (associados a operações de ordenação e agregação) que têm execução *eager*, estando a listagem de todas as operações e detalhes da sua prioridade disponível em Neo4j (2020).

4.2.2 *Palavras chave*

Na sintaxe do CYPHER existem numerosas palavras reservadas, das quais destacaremos:

MATCH - especifica que padrões devem ser procurados na base de dados.

Os padrões que podem ser procurados podem ser visualizados em Neo4J (2020c);

RETURN - especifica que dados devem ser retornados como resposta à *query* e encontra-se localizado perto do final desta.

Todas as *queries* necessitam de tanto `MATCH` como `RETURN` para serem *queries* válidas em CYPHER.

Exemplos:

1. Retorna todos os nodos da base de dados:

```
MATCH (n)
RETURN n
```

Código 4.11: Primeiro exemplo das palavras chave `MATCH` e `RETURN` em CYPHER

2. Retorna todos os nodos conectados ao nodo com nome `Oliver Stone` por qualquer relação:

```
MATCH (:Person { name: 'Oliver Stone' })-->(movie)
RETURN movie.title
```

Código 4.12: Segundo exemplo das palavras chave `MATCH` e `RETURN` em CYPHER

3. Retorna todos os nodos relacionados com o filme `Wall Street` pela relação **ACTED_IN**:

```
MATCH (wallstreet:Movie { title: 'Wall Street' })<-[:ACTED_IN]-(actor)
RETURN actor.name
```

Código 4.13: Terceiro exemplo das palavras chave `MATCH` e `RETURN` em CYPHER

OPTIONAL MATCH - faz o mesmo que um `MATCH` normal, mas substitui dados em falta por `null`.

Exemplo (neste caso retornará o titulo do filme `Wall Street` e, como não existe nenhuma relação `ACTS_IN` para esse nodo, retornará `null` na coluna `r`):

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-[r:ACTS_IN]->()
RETURN a.title, r
```

Código 4.14: Palavra chave `OPTIONAL MATCH` em CYPHER

ORDER BY - ordena os resultados finais com base em certas variáveis apresentadas, usando ordem ascendente nas variáveis por omissão (*default*) e invertendo essa ordem com a palavra chave `DESC` antes.

Note-se que as variáveis à esquerda têm maior prioridade.

```
MATCH (n)
RETURN n.name, n.age
ORDER BY n.name
```

Código 4.15: Palavra chave `ORDER BY` em CYPHER

WITH ... AS - altera os dados da *query*, restringindo, manipulando e renomeando dados antes de serem processados pelo resto dela.

Exemplos:

1. Obter o nome da pessoa conectada com o nodo cujo nome é `David` que tem pelo menos mais do que uma relação com outros nodos:

```
MATCH (david { name: 'David' })--(otherPerson)-->()
WITH otherPerson, count(*) AS foaf
WHERE foaf > 1
RETURN otherPerson.name
```

Código 4.16: Primeiro exemplo de uso da palavra chave `WITH` em CYPHER

2. Ordenar resultados antes de fazer operação de colecção:

```
MATCH (n)
WITH n
ORDER BY n.name DESC LIMIT 3
RETURN collect(n.name)
```

Código 4.17: Segundo exemplo de uso da palavra chave `WITH` em CYPHER

3. Limitar número de `matches` futuros ao restringir o número de nodos que uma certa variável pode ter (no caso apresentado o último `match` só pode ser feito uma única vez por causa do `LIMIT`)

```
MATCH (n { name: 'Anders' })--(m)
WITH m
ORDER BY m.name DESC LIMIT 1
MATCH (m)--(o)
RETURN o.name
```

Código 4.18: Terceiro exemplo de uso da palavra chave `WITH` em CYPHER

UNION - faz a união dos resultados de duas ou mais *queries* distintas que tenham colunas similares como resultado, juntando-as e eliminando elementos duplicados caso existam (para manter os duplicados usa-se a palavra chave `UNION ALL`).

```
MATCH (n:Actor)
RETURN n.name AS name
UNION
MATCH (n:Movie)
RETURN n.title AS name
```

Código 4.19: Palavra chave `UNION` em CYPHER

LIMIT - limita o número de colunas do resultado, podendo ter, no máximo, o valor indicado após o `LIMIT` de linhas a serem retornadas.

```
MATCH (n)
RETURN n.name
ORDER BY n.name
LIMIT 3
```

Código 4.20: Palavra chave `LIMIT` em CYPHER

WHERE - restringe os resultados obtidos pelos `MATCH` para verificarem a condição desejada, adicionalmente funcionando como filtro no caso de ser usado com o `WITH`.

```
MATCH (n:Person)
WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = 'Timothy')
OR NOT (n.name = 'Timothy' OR n.name = 'Peter')
RETURN n.name, n.age
```

Código 4.21: Palavra chave `WHERE` em CYPHER

CALL{SUBQUERY} - permite a execução de *subqueries* dentro de uma *query*, permitindo obter resultados de outras *queries* e utilizar esses resultados na *query* que a chamou. Isto é particularmente útil para, por exemplo, chamar `UNION` dentro de uma *query* e processar os seus resultados.

Esta funcionalidade foi lançada durante as fases finais deste trabalho, sendo que anteriormente a palavra `CALL` apenas servia para chamar funções em Neo4J.

Exemplo (obter a pessoa mais nova e a pessoa velha na base de dados):

```
CALL {
  MATCH (p:Person) RETURN p ORDER BY p.age ASC LIMIT 1
  UNION
  MATCH (p:Person) RETURN p ORDER BY p.age DESC LIMIT 1
}
RETURN p.name, p.age ORDER BY p.name
```

Código 4.22: Palavra chave `CALL{subquery}` em CYPHER

4.2.3 *Queries de exemplo*

Para finalizar a introdução da linguagem colocaremos algumas *queries* de exemplo extra:

1. Esta *query* retornará todos os elementos `n` onde o seu atributo titulo tenha valor de `Doutor`

```
match(n)
where n.titulo='Doutor'
return n
```

Código 4.23: Primeira *query* exemplo CYPHER

2. Esta *query* retornará todos os nodos `m` e `q` onde exista uma relação de `m` para `q` de tipo `pertenceA` e onde o atributo nome de q tenha valor de `Veiculo`.

```
match(m)-[w:pertenceA]->(q)
where q.nome='Veiculo'
return m,1
```

Código 4.24: Segunda *query* exemplo CYPHER

4.3 SPARQL versus CYPHER

Tendo introduzido tanto as linguagens como os motores destas passamos agora a um passo muito importante para obtermos um mapeamento: comparar ambas as linguagens e observar similaridades/diferenças entre ambas.

Antes de mais nada temos de compreender o impacto que as diferenças na maneira como os dados estão guardados têm neste mapeamento, bem como as diferenças entre os conceitos definidos em *queries* SPARQL e em *queries* CYPHER.

Como o GraphDB foi criado de raiz para lidar com ontologias (usando-as para guardar dados e uma linguagem especificamente criada para obter dados dessas estruturas) certos conceitos que estão definidos através do SPARQL não existem no Neo4J. Nomeadamente Neo4J não tem definido o conceito de triplo, algo que pode ser visto directamente durante o processo de importe de dados mencionado em 3.3.3 e que torna o processo de mapeamento extremamente difícil.

Isto é adicionalmente complicado pelo facto que em SPARQL queremos sempre obter informações em relação a triplos, estando a linguagem definida para facilitar isto, enquanto que em CYPHER a definição da linguagem torna-a apta para obter informação de um grafo genérico mas torna bastante difícil de obter triplos como estão a ser importados pelo Neosemantics.

A dificuldade advém do facto que o processo de importe de dados separa triplos em duas categorias: triplos que correspondem a propriedades de elementos e triplos que correspondem a relações entre elementos. Como tal teremos situações onde não conseguimos distinguir apenas pela *query* que queremos traduzir se um objecto é literal ou URI, o que torna a definição deste mapeamento complexa.

Apesar destas diferenças temos também similaridades importantes entre ambas as linguagens, das quais a mais importante tem a ver com o facto que a vasta maioria das palavras

reservadas e funcionalidades de *queries* feitas são extremamente similares e permitem-nos criar padrões para traduzir entre as linguagens. Por exemplo:

SELECT E RETURN Numa *query* `SELECT` do SPARQL aquilo que queremos seleccionar será traduzido em CYPHER para o `RETURN` final da *query*.

FILTER E WHERE Apesar de não ser uma tradução completa, muitas das filtragens mais simples (i.e., que não usam palavras reservadas) no SPARQL podem ser implementadas quase directamente em CYPHER através da palavra reservada `WHERE`.

LIMIT E ORDER BY As palavras reservadas `LIMIT` e `ORDER BY` estão implementadas de modo equivalente em ambas as linguagens, pelo que a sua tradução é directa.

4.3.1 *Exemplos de queries*

Para dar uma ideia mais completa das similaridades/diferenças entre as linguagens vamos agora apresentar algumas *queries* e traduções entre SPARQL e CYPHER.

Caso de estudo

Antes de mostrarmos os exemplos devemos explicar a ontologia que estamos a fazer as *queries* sobre.

Como o nosso objectivo é de obter um mapeamento genérico entre linguagens deve-mos escolher uma ontologia com tamanho apreciável e com alguma maturidade, de modo a dar um ambiente completo para demonstrar as similaridades e diferenças entre as linguagens em estudo.

Para estes exemplos estamos a usar a ontologia CLAV (Ramalho et al. (2020)), uma ontologia tipo OWL que serve de "base à Plataforma CLAV", cujo propósito é permitir a consulta da "Lista Consolidada para a classificação e avaliação da informação pública", de modo a facilitar a "elaboração dos planos de classificação e tabelas de selecção da Administração Pública, de empresas públicas e de outras entidades".

Esta ontologia consiste em duas partes:

- Catálogo de entidades e de tipologias de entidades produtoras ou gestoras de informação
- Catálogo de legislação que regula e enquadra os processos de negócio e a avaliação da respectiva informação

Com esta ontologia podemos comparar ambas as linguagens de modo directo e relevante, permitindo-nos obter melhor qualidade na comparação das *queries*.

Queries e suas traduções

Para compreender bem esta secção é importante lembrar o modo como os dados são importados em Neo4J, descrito em 3.3.3.

Para dar uma ideia boa de como as diferenças entre as linguagens nos dificultam o processo de tradução vamos traduzir uma *query* SPARQL extremamente simples: obter todos os triplos da ontologia.

```
SELECT * WHERE {
    ?s ?p ?o
}
```

Código 4.25: *Query* exemplo: obter todos os triplos associados à ontologia

A *query* que obtém os mesmos dados que esta em CYPHER é a seguinte (note-se que para obter o URI associado a uma relação em Neo4J tem de se utilizar a função *type* do Neo4J):

```
MATCH (n)-[r]->()
RETURN n,type(r)
```

Código 4.26: *Query* que obtém os mesmos dados que a *query* em 4.25

E é aqui que um dos problemas associados às diferenças estruturais entre as duas linguagens ocorre. Como o conceito de triplo não existe em Neo4J os resultados das duas *queries* estão dispostos de maneira muito diferente:

s	p	o
rdf:type	rdf:type	rdf:Property
rdfs:subPropertyOf	rdf:type	rdf:Property
rdfs:subPropertyOf	rdf:type	owl:TransitiveProperty
rdfs:subClassOf	rdf:type	rdf:Property
rdfs:subClassOf	rdf:type	owl:TransitiveProperty
rdfs:domain	rdf:type	rdf:Property
rdfs:range	rdf:type	rdf:Property
owl:equivalentProperty	rdf:type	owl:SymmetricProperty
owl:equivalentProperty	rdf:type	owl:TransitiveProperty
owl:equivalentClass	rdf:type	owl:SymmetricProperty

Figura 13: Resultados da *query* 4.25 em GraphDB

Figura 14: Resultados da *query* 4.26 em Neo4J

Da imagem pode-se ver que enquanto que GraphDB retorna resultados sempre linha a linha (i.e., cada célula de cada linha tem um e um só URI ou valor literal) o mesmo não acontece em Neo4J, onde cada célula pode em vez de um valor conter um objecto ou uma lista de valores.

Como o nosso objectivo é traduzir de SPARQL para CYPHER é importante que os resultados do CYPHER estejam correctamente dispostos (i.e., venham na mesma forma que o SPARQL). Assim para obter resultados equiparáveis em ambos os lados temos de utilizar a *query* seguinte:

```
match(n)

unwind
[key in keys(n) | [key,n[key]]] + [(n)-[r]->(o) | [type(r),o.uri]]
as listTriples

with n.uri as s, listTriples[0] as p,listTriples[1] as o
RETURN *
```

Código 4.27: *Query* equiparável tanto em resultados como na maneira em que estes são apresentados a 4.25

Esta *query* faz com que os resultados obtidos no final tenham formatos similares e permite a sua comparação direta. No entanto estas *queries* têm o problema de não terem uma ordem explicita nelas, o que tornar o processo de comparar os resultados bastante tedioso. Para resolver isto é apenas necessário adicionar um `order by` no final das *queries* que queremos traduzir para forçar a ordenação.

Com isto dito, e ordenando ambas as *queries*, obtemos um conjunto de resultados comparável entre as duas plataformas, como se pode ver nas imagens seguintes:

s	p	o
:c100	:classeStatus	A
:c100.10	:classeStatus	A
:c100.10.001	:classeStatus	A
:c100.10.002	:classeStatus	A
:c100.10.003	:classeStatus	A
:c100.10.200	:classeStatus	A
:c100.10.400	:classeStatus	A

Figura 15: Resultados da *query* 4.25 em GraphDB ordenados pela segunda coluna

o	p	s
"A"	"http://jcr.di.uminho.pt/m51_clav#classeStatus"	"http://jcr.di.uminho.pt/m51_clav#c250.20.200"
"A"	"http://jcr.di.uminho.pt/m51_clav#classeStatus"	"http://jcr.di.uminho.pt/m51_clav#c250.20.201"
"A"	"http://jcr.di.uminho.pt/m51_clav#classeStatus"	"http://jcr.di.uminho.pt/m51_clav#c250.10.800"
"A"	"http://jcr.di.uminho.pt/m51_clav#classeStatus"	"http://jcr.di.uminho.pt/m51_clav#c200.10.100"

Figura 16: Resultado da *query* 4.27em Neo4J ordenados pela segunda coluna

Note-se que apesar da ordenação equivalente (pela segunda coluna, ou seja, o predicado), os resultados são distintos. Isto é porque SPARQL e CYPHER têm ordenações não explicitas distintas, com SPARQL a ordenar pela primeira coluna, depois pela segunda coluna, ... Em contraste CYPHER não ordena nada explicitamente, mostrando os resultados na mesma ordem que os obteve durante a sua avaliação.

Assim mesmo que se diga explicitamente que a ordenação é por uma coluna ainda pode levar a que tenham um conjunto de resultados ordenados de forma diferente, sendo que para resolver isto tem de se explicitar em *queries* com múltiplas colunas de resultados a ordenação para todas as colunas. Para este trabalho a ordenação geralmente será apenas numa coluna, pois os resultados com múltiplas colunas não são tão grandes a justificar a ordenação em múltiplas colunas.

Para compreender as mudanças que são necessárias fazer para colocar os dados em Neo4J a terem a mesma estrutura que GraphDB vamos analisar outro exemplo. Consideremos a *query* seguinte:

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select * where {
  :c100.10.001 :temPai ?pai .
}
```

Código 4.28: *Query* exemplo 2: obter os elementos da ontologia que são "pais" de um certo elemento

Nesta *query* pretendemos obter todos os elementos `?pai` da ontologia que são pais do elemento `:c100.10.001`, i.e., todos os elementos da ontologia onde `:c100.10.001` tem uma relação `:temPai` com eles. Considerando o contexto do caso de estudo, esta *query* pretende obter a classe que é pai de `:c100.10.001`, sendo que o resultado desta *query* é singular: `:c100.10`

Traduzir para CYPHER, tendo em conta a estrutura dos resultados, leva à *query* seguinte:

```
match(gV1) where gV1.uri = "http://jcr.di.uminho.pt/m51-clav#c100.10.001”

unwind
[(gV1)-[gV2]->(nodoPai) where type(gV2) = "http://jcr.di.uminho.pt/m51-clav#temPai"  |
    nodoPai.uri]
as pai

RETURN *
```

Código 4.29: *Query* equivalente à *query* 4.28

Nesta tradução temos três expressões a considerar: a expressão `match`, a expressão `unwind` e a expressão `return`.

Na expressão `match` da tradução pretendemos obter o nodo que corresponde ao sujeito do triplo (`:c100.10.001`). Para isso, e tendo em conta como o dados são importados para Neo4J, temos que fazer `match` com o nodo da ontologia cujo campo URI seja o do nosso sujeito, fazendo-se isso com um `where` após o `match`.

Na expressão `unwind` pretendemos obter os dados do objecto que está relacionado com o sujeito a partir da relação `:temPai`.

Para isso usamos sintaxe de compreensão de padrões (Neosemantics (2020e)) para obter todos os elementos da ontologia que estão relacionados com o sujeito pela relação pretendida (correspondendo à relação cujo URI é `http://jcr.di.uminho.pt/m51-clav#temPai`), e como para identificar um elemento na ontologia (e como tal nos resultados) apenas é necessário o URI retornamos apenas o URI associado aos elementos capturados.

Como esta expressão `unwind` apenas tem um dado a ser retornado podemos renomear directamente para `pai` o que obtivermos desta procura.

Na expressão `return` especificamos o que pretendemos retornar no final da `query`, sendo que como temos uma *query* `select *` traduzimos para `return *`.

Com esta tradução obtemos os mesmos dados que na *query* original no mesmo formato que GraphDB (apesar que a ordem dos resultados poder não ser a mesma devido à diferença entre as estratégias de avaliação).

Note-se que a *query* acima assume algo: que `:temPai` é uma relação entre dois elementos da ontologia, o que se verifica neste caso pela estrutura da ontologia que já é conhecida.

No entanto se não soubermos nada sobre a estrutura da ontologia que estamos a fazer *query* a (algo que a nossa gramática de tradução não saberá) deparamo-nos com o segundo problema mais relevante das diferenças estruturais entre Neo4J e GraphDB: o facto que triplos podem estar em dois lugares distintos no Neo4J.

Para tornar isto mais óbvio, vamos considerar uma *query* sobre a mesma ontologia onde o predicado do triplo representa uma propriedade do elemento na ontologia:

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select * where {
  :c100.10.001 :codigo ?codigo .
}
```

Código 4.30: *Query* exemplo 3: obter o valor código associado a :c100.10.001

Para esta *query* temos a tradução:

```
match(gV1) where gV1.uri = "http://jcr.di.uminho.pt/m51-clav#c100.10.001”

unwind
[key in keys(gV1) where key = "http://jcr.di.uminho.pt/m51-clav#codigo" | gV1[key]]
as codigo

RETURN *
```

Código 4.31: *Query* equivalente à *query* 4.30

Como podemos ver esta *query* é extremamente similar a 4.29, com excepção da expressão `unwind`, sendo que a diferença deve-se ao facto que queremos agora obter propriedades do elemento (neste caso, a propriedade `http://jcr.di.uminho.pt/m51-clav#codigo`) em vez de relações, usando compreensão de listas (Neosemantics (2020c)).

No entanto em ambos os exemplos anteriores tivemos conhecimento de que o predicado representava uma relação ou uma propriedade. Mas a nossa gramática de tradução, que deverá funcionar para qualquer ontologia em qualquer contexto, nunca terá acesso a esta informação, podendo apenas deduzir informação sobre a estrutura da *query* que está a traduzir atualmente.

Assim a nossa *query* traduzida original caso não saibamos a estrutura da ontologia ficará como:

```
match(gV1) where gV1.uri = "http://jcr.di.uminho.pt/m51-clav#c100.10.001”

unwind
[key in keys(gV1) where key = "http://jcr.di.uminho.pt/m51-clav#temPai"  | gV1[key]]
+
[(gV1)-[gV2]->(nodoPai) where type(gV2) = "http://jcr.di.uminho.pt/m51-clav#temPai"  |
    nodoPai.uri] as pai
```

```
RETURN *
```

Código 4.32: *Query* equivalente à *query* 4.28 para o caso de não sabermos o que o predicado representa

Note-se que como um predicado numa ontologia será sempre ou uma relação entre indivíduos ou uma propriedade de um elemento temos garantia que a junção dos dois *arrays* feita na *query* acima terá toda a informação que pretendemos (sendo que os *arrays* são junção de ambas as partes das duas *queries* apresentadas antes desta), pelo que um dos *arrays* que estamos a juntar será sempre vazio e o outro terá a informação desejada.

Com as *queries* mais gerais observadas passamos a algumas traduções para palavras reservadas de SPARQL.

Comecemos pelas mais simples: `LIMIT` e `ORBER BY`. Estas duas palavras reservadas existem em ambas as linguagens, tornando a sua tradução relativamente simples:

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select distinct ?classe ?dono where {
  ?classe :temDono ?dono .
}
ORDER BY DESC(?dono) ?classe
LIMIT 20
```

Código 4.33: *Query* exemplo 3: obter vinte elementos da ontologia que têm um dono e dito dono da classe tal que os resultados sejam ordenados pelo valor do dono de forma descendente primeiro e pelo valor da classe de forma ascendente depois

Traduzir esta *query* leva a:

```
match(gV1)

unwind
[(gV1)-[gV2]->(nodoDono) where type(gV2) = "http://jcr.di.uminho.pt/m51-clav#temDono"  |
    nodoDono.uri]
as dono

with dono, gV1.uri as classe
RETURN distinct classe,dono
ORDER BY dono DESC,classe
```

Código 4.34: *Query* equivalente à *query* 4.33

Como se pode ver a tradução destas duas palavras reservadas é quase direta: o `order by` teve uma pequena alteração na sintaxe e o `limit` não foi mudado.

Do mesmo modo a palavra reservada `distinct` é absolutamente trivial de traduzir: se houver um `distinct` após o `select` então deve existir um `distinct` após a `return` em CYPHER.

No entanto nem todas as palavras reservadas são tão simples de traduzir. Notavelmente, o `FILTER` é tão complexo que tentar traduzir todas as partes dela seria um processo extremamente penoso. A razão para esta complexidade está assente nas numerosas palavras reservadas e funções que podem existir associadas ao `filter`, de entre as quais se destacam `EXISTS` e `NOT EXISTS` por permitirem exprimir uma *query* dentro de outra *query* que tem de ser avaliada e cujos resultados afetaram a *query* "acima".

Apesar disto, certas partes do filtro podem ser facilmente traduzidas:

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select distinct ?classe where {
  ?classe :codigo ?codigo .
  filter ((?codigo > "200") && (?codigo < "500")) .
}
```

Código 4.35: *Query* exemplo 4: obter todas as classes da ontologia cujo código seja maior que duzentos e menor que quinhentos (relembrar que o valor de código é uma string pelo que a comparação é feita com valores em strings)

Traduzir esta *query* leva a:

```
match(gV1)

unwind
[key in keys(gV1) where key = "http://jcr.di.uminho.pt/m51-clav#codigo" | gV1[key]]
as codigo

with gV1.uri as classe,codigo
where (codigo > "200") and (codigo < "500")

RETURN classe
```

Código 4.36: *Query* equivalente à *query* 4.35

A tradução desta parte do `filter` é extremamente simples: para comparações entre valores (<, >, =, ...) a tradução é directa enquanto que para operações lógicas deve-se usar uma sintaxe ligeiramente diferente. Toda esta tradução tem depois de ser usada após um `WITH ... WHERE` de modo a permitir a filtragem dos dados.

Existem muitas mais traduções que poderíamos aqui apresentar, mas a única coisa que faria é continuar a expor algo que por agora já é óbvio: uma enorme quantidade de *queries* SPARQL podem ser traduzidas para CYPHER desde que se tenha cuidado com as diferenças entre as linguagens e como os dados estão guardados.

Assim apesar de serem linguagens diferentes temos boas expectativas que uma tradução pelo menos parcial de SPARQL para CYPHER é definitivamente possível.

4.4 RESUMO

Neste capítulo introduzimos as linguagens de *query* entre as quais pretendemos obter uma correspondência: SPARQL e CYPHER.

Para ambas explicamos o processo de avaliação de *queries* e palavras chave usadas em ambas.

Adicionalmente comparamos de forma mais directa através da equivalência de *queries* que mostramos, bem como padrões observados entre as duas linguagens.

5

O PROBLEMA E OS SEUS DESAFIOS

Com os capítulos anteriores temos agora toda a informação necessária para compreender o trabalho feito nesta tese. Mas antes de o podermos mostrar temos de estabelecer que problema estamos a tentar resolver.

Esta tese tem dois problemas principais.

1. Criação de um mapeamento entre SPARQL e CYPHER. Para podermos resolver este problema temos de descobrir como transformar as numerosas *queries* que podemos escrever em SPARQL em *queries* que não só são válidas (i.e., retornam a informação esperada) como têm a mesma estrutura que GraphDB quando feitas a Neo4J (tendo em conta tanto o formato em que os dados são guardados como o processo de importe feito pelo Neosemantics)

2. Criação de uma aplicação Web que permita estender as capacidades de Neo4J para trabalhar com ontologias. Para resolver isto temos de determinar não só os meios pelos quais iremos criar a aplicação (que linguagem a usar, que *frameworks* dentro dessa linguagem são apropriados ao problema, ...) como que funcionalidades queremos oferecer e como conseguir utilizar a gramática que criamos dentro da aplicação.

5.1 SOLUÇÃO PROPOSTA

Após um estudo de várias opções determinamos uma solução para ambos os problemas que permite não só obter um mapeamento entre SPARQL e CYPHER como implementar dita gramática na nossa aplicação: criar uma gramática de tradução usando a ferramenta *PEG.js* (za Ryuu (2020)) e criar uma aplicação Web escrita em Javascript que permita o seu uso.

A gramática de tradução consistirá de duas partes: sintaxe equivalente à da linguagem SPARQL (para poder fazer `match` com qualquer *query* SPARQL e assim permitir o seu reconhecimento) e semântica associada (que consiste de acções que traduzem diferentes partes das *queries* para equivalentes lógicos em CYPHER).

O uso da ferramenta *PEG.js* advém da capacidade de converter uma gramática num *parser* em JS, facilitando a sua implementação na aplicação WEB. Esta aplicação consistirá em dois servidores escritos em JS (`backend` e `frontend`) que nos permitirão não só expor a tradução feita através do *parser* como adicionar capacidades adicionais sobre Neo4J.

5.2 CASO DE ESTUDO

Antes de mostrarmos a implementação e resultados da solução proposta devemos explicar o caso de estudo que usamos nesta fase.

Durante o desenvolvimento da gramática de tradução acima mencionada foi utilizada a ontologia CLAV devido à sua completude, permitindo ter um ambiente onde os resultados de quaisquer *queries* traduzidas pela gramática podem ser comparadas de modo directo às *queries* feitas em SPARQL no GraphDB.

No entanto, na fase da validação da gramática foram escolhidas duas ontologias distintas desta de modo a demonstrar que a tradução funciona independentemente da ontologia à qual estamos a fazer *queries* e que funciona tanto para ontologias complexas como para casos mais simples.

Estas ontologias são: *Pókedex* e *Tabela Periódica*

5.2.1 *Pókedex*

Pókedex é uma ontologia descrita por si própria como "uma ferramenta para encontrar informação sobre *Pokemons* dos três jogos da primeira geração para a consola *Game Boy*: *Pokémon Yellow Version*, *Pokémon Blue Version* e *Pokémon Red Version*".

Nela temos informação sobre os *Pokemons* dos vários jogos, incluindo *Moves* (habilidades associadas a *Pokemons*), *TM's* e *HM's* e suas localizações (itens para permitir a um *Pokemon* aprender novos *Moves*), evoluções e habilidades obtidas por aumento de nível, resistências e fraquezas e como capturar cada *Pokemon* nestes três jogos.

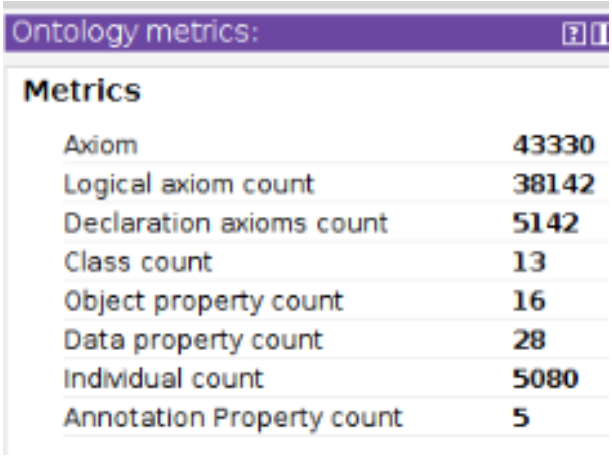


Figura 17: Métricas da ontologia *Pókedex*

5.2.2 *Tabela Periódica*

Em contraste com as ontologias mostradas anteriormente a ontologia da Tabela Periódica escolhida é simples em estrutura e pequena por comparação, servindo para mostrar que os

resultados da tradução criados funcionam para mais do que só ontologias complexas e são úteis em qualquer caso. Devido à sua simplicidade podemos explicar a sua composição em muito mais detalhe que as outras ontologias.

Esta ontologia consiste de três classes principais: *Group*, que representa um grupo da tabela periódica, *Period*, que representa um período da tabela periódica e *Element*, que representa um elemento especifico da tabela periódica.

De notar que a ontologia tem mais classes, mas estas estão ou sem elementos (*Molecule* está vazia) associados ou são agrupamentos de propriedades a serem usadas nos elementos (*Block* para blocos d, f, p e s da tabela periódica, *Classification* para denotar se um elemento é metálico, semi-metálico ou não metálico e *Standard State* para determinar o estado do elemento sobre condições de temperatura e atmosfera normais (sólido, liquido, gás ou estado indeterminado)).

Cada grupo tem associado os elementos que pertencem a esse grupo e valores de número e de nome apropriados a cada um, com o mesmo a acontecer para os períodos da tabela periódica.

Em contraste cada elemento contém numerosos campos de dados associados: grupo, período, estado, bloco, classificação, nome, cor, símbolo atómico, peso atómico, número atómico e número de registo CAS (Society (2020)).

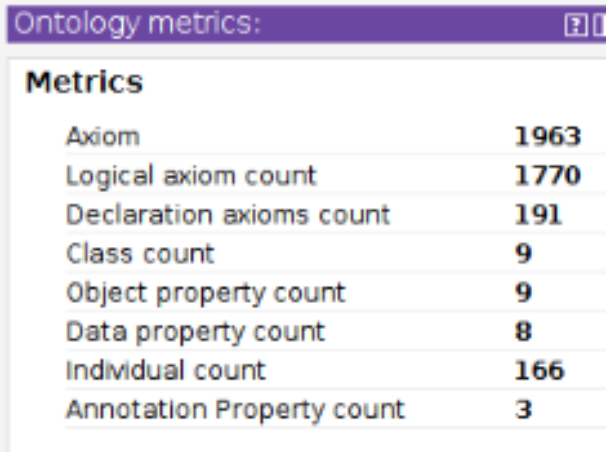


Figura 18: Métricas da ontologia Tabela Periódica

6

GRAMÁTICA DE TRADUÇÃO

Estamos agora em condições para apresentar a gramática de tradução criada para este trabalho.

Ao longo deste capitulo vamos explorar o trabalho feito na parte mais teórica deste trabalho, falando das decisões tomadas e da implementação dela e mostrando depois os resultados obtidos e sua validação.

6.1 DECISÕES

A primeira coisa que devemos falar são as decisões tomadas para como a gramática deve ser implementada.

Uma das decisões tomadas de inicio foi que a aplicação que também está a ser desenvolvida neste trabalho e esta gramática devem poder comunicar, de modo a permitir o uso desta gramática na aplicação.

Como já foi referido acima esta gramática foi implementada usando a ferramenta *PEG.js* (za Ryuu (2020)), algo que permitiu resolver este problema de forma simples.

As razões para esta escolha são:

- É possível criar uma gramática com sintaxe e semântica definida, dando-nos as ferramentas necessárias para criar uma transcrição.
- A ferramenta cria um *parser* escrito em Javascript, permitindo uma integração na aplicação simples.
- A ferramenta está disponível online, facilitando o processo de desenvolver a gramática por não necessitar de instalar software.

Com esta decisão feita o processo de criação da gramática começou, mas após algum tempo um problema foi descoberto: criar uma tradução completa entre SPARQL e CYPHER (e vice-versa) é um trabalho muito para além do que é plausível numa única tese de mestrado.

Tendo em conta que um dos objectivos do trabalho é expandir a capacidade do Neo4J para lidar com ontologias foi decidido que apenas seria desenvolvida uma tradução de *queries* SPARQL para *queries* CYPHER (sendo que se essa tradução for completa seria decidido se fazer a tradução no outro sentido no tempo que resta faz algum sentido).

6.2 Implementação e resultados obtidos

Com as decisões tomadas podemos agora discutir a implementação das várias partes da gramática criada, de modo a compreender o seu funcionamento e capacidades.

Para podermos criar esta gramática precisamos de duas coisas: reconhecer *queries* SPARQL e traduzi-las para CYPHER.

Para podermos reconhecer SPARQL precisamos de ter a sintaxe da gramática que estamos a criar a ser equivalente à sintaxe da linguagem SPARQL. Deste modo a nossa gramática será capaz de reconhecer as *queries* que queremos traduzir.

Felizmente a gramática da linguagem SPARQL está definida formalmente em W3C (2013e), pelo que o nosso trabalho consiste apenas em copiar esta definição e adaptar para que funcione com a ferramenta *PEG.js*.

Este processo foi relativamente simples, tendo apenas duas coisas a notar:

1. A representação de caracteres *unicode* em *PEG.js* é feito usando `\uXXXX`, onde `XXXX` corresponde a um código *unicode*, tendo este código de ter todos os quatro números para funcionar.

2. *PEG.js* avalia a gramática da esquerda para a direita e a definição em W3C não está adequada a este tipo de avaliação.

   Por exemplo em W3C (2013i), assumindo que existem, os caracteres após `PN_CHARS_BASE` devem ser ou `PN_CHARS` ou `-`, excluindo o último dos caracteres dele. Isto é difícil de recriar em *PEG.js*, visto que se tentarmos utilizar directamente a regra a gramática continuará a capturar os caracteres na primeira parte da expressão e nunca chegará ao fim, levando a que o *parsing* falhe. A solução para este exemplo consiste em permitir que todos os caracteres sejam ou `PN_CHARS` ou `-` e depois semanticamente causar com que as sequências onde o último caractere seja '-' levantem erro.

Com a sintaxe definida focamo-nos na segunda parte: traduzir o que reconhecemos para CYPHER. Para isso adicionamos acções semânticas à sintaxe que já definimos de modo a processar as numerosas componentes da *query* em equivalentes CYPHER ou a processar certas partes que são usadas nessa tradução (por exemplo, processar os prefixos para poderem ser usados nas traduções do corpo da *query*).

Inicialmente foram criadas *queries* extremamente simples que foram traduzidas manualmente entre as duas linguagens para permitir dar uma ideia do que é necessário fazer. Durante este período inicial foi determinado que a melhor maneira de lidar com os triplos de uma *query* é fazer o processamento deles triplo por triplo, ou seja, traduzir um triplo e só depois continuar para o triplo seguinte.

Em seguida foram implementadas sequencialmente capacidades à tradução feita, começando por traduzir triplos singulares (com URI's e variáveis), depois composição destes (através de ponto, vírgula e ponto e vírgula), depois algumas palavras reservadas simples (`PREFIX`, `LIMIT`, `ORDER BY`) e finalmente começar com palavras reservadas mais complexas (`FILTER`).

Finalmente foi criada a capacidade de lidar com *queries* `ASK` e `DESCRIBE` para além das *queries* `SELECT` que estavam a ser desenvolvidas até agora.

Para compreender decisões feitas ao longo deste processo vamos usar a gramática criada para traduzir algumas *queries* simples.

Antes de mostrarmos os resultados temos de considerar que a gramática não retorna apenas a *query* transcrita mas sim um objecto JSON com três chaves: *prologue*, um objecto JSON que contém informação sobre prefixos declarados na *query* SPARQL, *query*, contendo uma string que corresponde à tradução da *query* para CYPHER e *type*, uma string que tem três valores possíveis (*select*, *ask* ou *describe*) que tem como propósito dar à aplicação WEB contexto para que tipo de *query* está a ser traduzida (isto afecta como os resultados são disponibilizados quando se faz a *query* e se obtém os resultados), sendo que os exemplos abaixo apenas mostraremos a *query* transcrita.

Adicionalmente a *query* traduzida mostrada neste documento tem duas pequenas alterações feitas à *query* gerada pela gramática original: os valores \" foram substituídos por " e linhas brancas foram removidas. A razão para isto tem a ver que o *PEG* coloca sempre expressamente as barras antes dos " na string, algo que tem de ser removido para que a *query* funcione (a aplicação WEB faz isto automaticamente), e a transcrição gera algumas linhas brancas desnecessárias mas que não afectam a transcrição em nenhuma maneira.

```
SELECT * WHERE {
    ?s ?p ?o
}
```

Código 6.1: *Query* para obter todos os triplos da ontologia

```
match(customVar_s)

unwind [key in keys(customVar_s) | [customVar_s[key],null,key]] + [(customVar_s)-[
    inner_P131869015]->(customVar_o) | [customVar_o.uri,customVar_o,type(inner_P131869015
    )]] as unwindVar_1587005400
with *,unwindVar_1587005400[0] as o,unwindVar_1587005400[1] as customVar_o,
    unwindVar_1587005400[2] as p
with *,customVar_s.uri as s

with s,p,o
where (s is not null) and (p is not null) and (o is not null)

RETURN *
```

Código 6.2: Transcrição da *query* 6.1 pela gramática

Como podemos ver pela tradução desta primeira *query* muitos aspectos discutidos em 4.3.1 encontram-se presentes, com algumas diferenças notáveis:

- As variáveis `gVX` das traduções acima mostradas foram transformados em variáveis com um nome próprio baseado na sua função com um valor aleatório como seu sufixo. Isto é porque essas variáveis são internas da tradução, i.e., não foram pedidas pelo utilizador e são criadas no processo de traduzir a gramática, pelo que o seu nome tem de ser distinto de qualquer nome de variável que o utilizador possa criar em SPARQL (para evitar conflito entre as variáveis internas e o que o utilizador quer obter).

- Todos os nodos que estão associados a variáveis definidas pelo utilizador estão escritas sobre a forma de `customVar_X`, onde `X` é o nome da variável que está associada ao nodo (lembrar que as variáveis são sempre URI se estão associadas a um nodo).

- A expressão `unwind` tem mais elementos a serem retornados e existem muitas mais expressões `with` na tradução que são sempre acompanhados de * (sem contar com a última expressão deste tipo). As expressões `with` extras têm a ver com o facto que triplos bastante distintos em SPARQL têm traduções muito similares e dados que querem obter extremamente próximos, pelo que faz sentido criar um `unwind` que capture mais elementos, nomea-lo com uma variável interna e depois renomear os seus dados para nomes que nos serão úteis no futuro. A razão pela qual todos os `with` são acompanhados de * no inicio tem a ver com legibilidade da *query* traduzida, visto que caso contrário teríamos de colocar em cada `with` todas as varáveis que nos são úteis seguidas das que estamos a tentar obter agora, o que para uma *query* com alguma complexidade é extremamente verboso.

- A expressão `return` é acompanhada de um `with` anterior com todas as variáveis capturadas onde o seu valor não é nulo. A razão para este `with` tem a ver como o `return` e o `select` se relacionam. Devido à sua proximidade, um `select *` é traduzido em `return *` o que leva a que todas as variáveis declaradas até agora na *query* sejam retornadas. Como estamos sempre a usar `with *` todas as variáveis usadas ao longo da *query* estariam declaradas no final se este `with` não existisse, incluindo as variáveis internas que nunca serão úteis ao utilizador. Com este `with` garantimos que no resultado final apenas podemos obter variáveis que foram pedidas pelo utilizador.

Para além destas mudanças temos ainda que discutir um parágrafo que existe neste documento em 4.3.1, nomeadamente:

Mo entanto em ambos os exemplos anteriores tivemos conhecimento de que o predicado representava uma relação ou uma propriedade. Mas a nossa gramática de tradução, que deverá funcionar para qualquer ontologia em qualquer contexto, nunca terá acesso a esta informação, podendo apenas deduzir informação sobre a estrutura da query que está a traduzir atualmente.

É verdade que a nossa gramática não tem disponível qualquer dado sobre a estrutura da ontologia para a qual está a traduzir a *query* actual, mas isso não implica que temos sempre de utilizar a captura por dois *arrays* discutida imediatamente após esse parágrafo.

Para compreender como utilizamos a *query* para evitar essa captura voltemos a considerar como os dados são importados para Neo4J (7.2.4). Podemos ver que na nossa tradução a

procura que temos de fazer não depende de se o predicado é propriedade ou relação mas sim de que tipo de valor o objecto é, tendo assim que:

- Se o objecto for um URI temos garantia que o predicado é uma relação, pelo que podemos utilizar o *array* que captura dados da relação no nosso `unwind`.
- Se o objecto for um literal temos garantia que o predicado é uma propriedade, pelo que podemos usar o *array* que captura dados das propriedades.

Com esta revelação um poderia pensar que poderíamos sempre utilizar apenas um *array* dentro do `unwind` mas existe um problema: é possível ter que membros de um triplo numa *query* SPARQL sejam variáveis em vez de valor fixos de um dos dois tipos (literais ou URI's).

Com isto em mente, o processo de escolha da captura de dados dentro do `unwind` parece simples:

1. se o objecto de um triplo não for uma variável e for um literal, procurar dentro das propriedades do nodo;
2. se o objecto de um triplo não for uma variável e for um URI, procurar dentro das relações do nodo;
3. se o objecto de um triplo for uma variável, procurar em ambos os lados;

   Mas existe algo mais que podemos determinar apenas dos triplos que estamos a analisar, e tem a ver com o facto que para um triplo numa ontologia importada o sujeito e o predicado são sempre URI's:
4. se o objecto for uma variável que apareça como sujeito ou predicado noutro triplo da *query*, procuramos nas relações do nodo.

Com este processo podemos com apenas informação que temos disponível da *query* melhorar a qualidade da tradução ao evitar usar *arrays* desnecessários, levando não só a que a leitura da tradução seja mais simples mas também a optimizar o tempo de execução (pois evita que o Neo4J itere sobre as propriedades ou relações de numerosos nodos quando tal não altera o resultado da tradução da *query*).

Apesar de tudo o que falamos sobre este primeiro exemplo a *query* SPARQL em si é muito básico, não demonstrando propriamente todas as capacidades da gramática.

Para melhor mostrar essas capacidades vamos agora mostrar várias *queries* mais complexas que a gramática consegue traduzir (todas do tipo `select`) para explorar com maior detalhe o que foi feito.

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select ?avo where {
  :c100.10.001 :temPai ?pai .
    ?pai :temPai ?avo .
}
```

Código 6.3: *Query* para obter a classe avô de `:c100.10.001`

```
match(customVar_fixed_S1180356723)
where customVar_fixed_S1180356723.uri = "http://jcr.di.uminho.pt/m51-clav#c100.10.001"

unwind [(customVar_fixed_S1180356723)-[inner_P1096641676]->(customVar_pai) where type(
    inner_P1096641676) = "http://jcr.di.uminho.pt/m51-clav#temPai"  | [customVar_pai.uri,
    customVar_pai,type(inner_P1096641676)]] as unwindURI_1921693601
with *,unwindURI_1921693601[0] as pai,unwindURI_1921693601[1] as customVar_pai

match(customVar_pai)

unwind [key in keys(customVar_pai) where key = "http://jcr.di.uminho.pt/m51-clav#temPai"
     | [customVar_pai[key],null,key]] + [(customVar_pai)-[inner_P287240114]->(
    customVar_avo) where type(inner_P287240114) =
    "http://jcr.di.uminho.pt/m51-clav#temPai"  | [customVar_avo.uri,customVar_avo,type(
    inner_P287240114)]] as unwindVar_1288287199
with *,unwindVar_1288287199[0] as avo,unwindVar_1288287199[1] as customVar_avo

with pai,avo
where (pai is not null) and (avo is not null)

RETURN avo
```

Código 6.4: Transcrição da *query* 6.3 pela gramática

Neste exemplo podemos observar três coisas: como a gramática trata composição ponto, a optimização que falamos acima (ponto 4 da enumeração) e como o retorno de dados é feito para quando uma parte das variáveis são retornadas.

A composição ponto é a mais simples de se tratar: basta fazer o mesmo processo que para um triplo singular para cada um dos triplos na composição, de forma independentemente à parte de variáveis que possam ter sido capturadas. Neste caso como estamos a usar o `?pai` fazemos `match` do sujeito do segundo triplo com um nodo que chamamos `customVar_pai` sem URI associado (pois é variável). Ao fazermos isto temos o efeito secundário de se a *query* tiver já determinado este nodo (por processamento de triplo anterior) ele pode ser usado directamente, removendo uma grande quantidade de esforço na execução da *query* (algo demonstrado no exemplo acima).

A optimização usada encontra-se presente na primeira expressão *unwind* do exemplo: note-se que estamos a usar apenas um dos *arrays*: o que retêm relações com outros nodos. Isto acontece porque sabemos da *query* que `?pai` é sujeito de outro triplo, pelo que é garantidamente um URI, levando à optimização discutida anteriormente.

Quando ao retorno pode-se ver que como apenas queremos retornar o `?avo` da *query* original especificamos essa variável no `return` final.

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select ?classe where {
  ?classe a :Classe_N2;
          :codigo ?codigo .
    FILTER (?codigo > "100" && ?codigo < "200")
}
```

Código 6.5: *Query* para obter todos as classes de nível dois cujo código está entre "100" e "200"

```
match(customVar_classe)

unwind [(customVar_classe)-[inner_P1921693601]->(customVar_o1498700598) where type(
    inner_P1921693601) = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" and
    customVar_o1498700598.uri = "http://jcr.di.uminho.pt/m51-clav#Classe_N2" | type(
    inner_P1921693601)] as unwindURI_1687467722

with *,customVar_classe.uri as classe

unwind [key in keys(customVar_classe) where key =
    "http://jcr.di.uminho.pt/m51-clav#codigo"  | [customVar_classe[key],null,key]] + [(
    customVar_classe)-[inner_P1288287199]->(customVar_codigo) where type(
    inner_P1288287199) = "http://jcr.di.uminho.pt/m51-clav#codigo"  | [customVar_codigo.
    uri,customVar_codigo,type(inner_P1288287199)]] as unwindVar_939664000
with *,unwindVar_939664000[0] as codigo,unwindVar_939664000[1] as customVar_codigo

with * where (codigo > "100") and (codigo < "200")

with classe,codigo
where (classe is not null) and (codigo is not null)

RETURN classe
```

Código 6.6: Transcrição da *query* 6.5 pela gramática

Neste exemplo podemos ver como a filtragem de elementos em SPARQL está a ser parcialmente traduzida para CYPHER, bem como a composição por ponto e vírgula do SPARQL (composição de triplos distintos onde está fixo o sujeito do triplo).

Em particular a parte da filtragem demonstrada é que lida com operações lógicas (&&, ||, ...) e comparações entre valores (<, >, =, ...). Para compreender como funciona temos de compreender como o `FILTER` funciona.

A palavra reservada `FILTER` é extremamente simples: apenas os triplos da ontologia para os quais a condição dada se verifica podem continuar a ser usados na *query* (ou seja, filtra todos os resultados para os quais a condição não se verifica). A dificuldade de implementar `FILTER` não tem a sim a ver com a sua funcionalidade mas sim com que tipo de condições podem ser expressas em SPARQL.

Devido a isto apenas parte da funcionalidade do `FILTER` está a ser traduzida, nomeadamente operações lógicas e comparações entre valores (exemplo acima) e a palavra reservada `EXISTS` (exemplo que será detalhado abaixo).

Para as operações lógicas e comparações a sua implementação é muito simples: `with * where [inserir condições aqui]`, tendo em conta as diferenças entre como operadores lógicos são escritos (`and` para `&&` e `or` para `||`).

A razão pela qual isto funciona deve-se a dois factores.

O primeiro tem a ver com a capacidade do CYPHER para lidar com estas condições lógicas e comparações nativamente, tendo suporte completo para elas (assumindo que se tem em conta a diferença em escrita das condições lógicas).

O segundo deve-se a como o `FILTER` funciona e como as variáveis estão descritas na *query* traduzida, onde os mesmos nomes contêm a mesma informação em tanto a *query* original como a traduzida, o que leva a que se possa fazer a condição a variáveis do mesmo modo que em SPARQL.

A razão para a presença do `with *` antes do `where` é simples: a palavra reservada `where` em CYPHER que serve para filtrar elementos com base numa certa condição tem sempre de vir precedida de uma fonte de dados qualquer (tipicamente um `match` ou um `with`), e como não queremos alterar qualquer nome ou dado na *query* fazemos um `match` com o * para termos acesso às variáveis (e assim poder filtrar resultados).

Quanto à composição ponto e vírgula a sua implementação é similar à da composição ponto onde tratamos os dois triplos separadamente, notando-se uma pequena diferença: é feito apenas um `match` do sujeito.

A razão para isto é simples: como sabemos sempre como chamar o nosso sujeito e sabemos que os triplos têm o mesmo sujeito não faz qualquer sentido fazer `match` dele múltiplas vezes em seguida.

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select distinct ?tipo where {
  ?classe a ?tipo .
    FILTER exists { ?classe :temPCA ?x }
}
```

Código 6.7: *Query* para obter o conjunto das classes da ontologia que podem ter um PCA associado (no caso de estudo são as classes de nível três e quatro)

```
match(customVar_classe)
```

```
unwind [key in keys(customVar_classe) where key = "http://www.w3.org/1999/02/22-rdf-
    syntax-ns#type" | [customVar_classe[key],null,key]] + [(customVar_classe)-[
    inner_P939664000]->(customVar_tipo) where type(inner_P939664000) = "http://www.w3.org
    /1999/02/22-rdf-syntax-ns#type" | [customVar_tipo.uri,customVar_tipo,type(
    inner_P939664000)]] as unwindVar_727096421
with *,unwindVar_727096421[0] as tipo,unwindVar_727096421[1] as customVar_tipo

match(customVar_classe)

unwind [key in keys(customVar_classe) where key = "http://jcr.di.uminho.pt/m51-clav#
    temPCA" | [customVar_classe[key],null,key]] + [(customVar_classe)-[inner_P1687467722
    ]->(customVar_x) where type(inner_P1687467722) = "http://jcr.di.uminho.pt/m51-clav#
    temPCA" | [customVar_x.uri,customVar_x,type(inner_P1687467722)]] as
    unwindVar_131869015
with *,unwindVar_131869015[0] as x,unwindVar_131869015[1] as customVar_x
with *,customVar_classe.uri as classe

with classe,x,tipo
where (classe is not null) and (x is not null) and (tipo is not null)

RETURN distinct tipo
```

Código 6.8: Transcrição da *query* 6.7 pela gramática

Neste exemplo acima podemos ver a palavra reservada `EXISTS` (W3C (2013d)) a ser utilizada dentro de um filtro e a sua tradução para CYPHER, bem como o uso da palavra reservada `DISTINCT` após o `SELECT`.

Ao olhar para esta tradução deve-se notar algo: a tradução da *query* dentro do `EXISTS` é exactamente igual a como se fosse a *query* fora dele e é feita sequencialmente após os triplos que existem antes (e seguida dos que são avaliados depois). À várias razões para esta escolha, entre as quais destacamos:

- A implementação completa de `EXISTS` (e sua negação, `NOT EXISTS`) requeria o uso de chamada de *subqueries* em Neo4J (Neo4J (2020h)), para permitir isolar todas as variáveis e operações correctamente. No entanto esta capacidade apenas está disponível na versão principal mais recente à criação deste documento, 4.1.0 (Neo4J (2020d)), lançada a menos de três meses da data de entrega desde trabalho, pelo que implementar correctamente requereria uma grande quantidade de esforço que nessa fase não faria qualquer sentido.
- Esta implementação funciona como esperado para o que foi implementado até agora: se a *query* dentro do `EXISTS` falhar (avaliar para falso, i.e., a expressão que está a ser avaliada não existe na ontologia) então o filtro falhava (avaliava para falso) pelo que a *query* falhava (retornava um conjunto vazio de resultados), e caso contrário verifica-se que a expressão existe pelo que se deve continuar a avaliação da *query*. Do mesmo modo,

as variáveis declaradas dentro do `EXISTS` são declaradas e podem ser usadas, que é o comportamento esperado desta palavra reservada

Esta implementação tem uma desvantagem principal: é impossível utilizar qualquer operação lógica em paralelo com o `EXISTS` no `FILTER` (i.e., não se pode avaliar a existência e outras condições no mesmo filtro).

Isto leva a que a uma das poucas restrições feitas à sintaxe do SPARQL da gramática original que não são meras adaptações à ferramenta usada: escrever uma *query* que contenha `FILTER` com um `EXISTS` tem de ser feito usando `FILTER EXISTS X`, com X a ser a expressão a avaliar dentro do `EXISTS` (que seja também ela traduzível pela gramática), não sendo possível utilizar `( EXISTS X )`. Caso se tente fazer isto a tradução retornará com um resultado que não avaliará para nada e dá erro se tentar ser corrido no Neo4J.

Outra desvantagem que se pode ver se se focar nos `match` é a de que tratamos da composição do `filter` do mesmo modo que a composição ponto: independentemente. No entanto há situações onde este tratamento leva a duplicação de linhas de modo desnecessário (em particular os `match` da mesma variável múltiplas vezes), como no exemplo acima (podendo também acontecer se por alguma razão o utilizador utilizar o mesmo sujeito em triplos separados por ponto). Uma solução melhor seria sempre que se tenta-se escrever o `match` de uma variável verificar se não existe um `match` já feita a ela e se existir não o escrever (pois é desnecessário).

Em relação à palavra reservada `DISTINCT` pouco à a dizer: a palavra está implementada em CYPHER do mesmo modo que em SPARQL, apenas em posição diferente na *query*. Assim apenas tem de se adicionar um `distinct` após o `return` na *query* traduzida se a *query* original tiver `distinct` após o `select`.

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select ?classe ?titulo where {
  ?classe a :Classe_N3 ;
          :titulo ?titulo ;
          :temDono :ent_INFARMED,:ent_IMT .
}
order by ?titulo
limit 5
```

Código 6.9: *Query* para obter as primeiras cinco classes de nível três e seus títulos que têm como seus donos tanto o INFARMED como o IMT ordenadas alfabeticamente pelo seu titulo

```
match(customVar_classe)

unwind [(customVar_classe)-[inner_P1921693601]->(customVar_o1498700598) where type(
    inner_P1921693601) = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" and
    customVar_o1498700598.uri = "http://jcr.di.uminho.pt/m51-clav#Classe_N3" | type(
    inner_P1921693601)] as unwindURI_1687467722
```

```
with *,customVar_classe.uri as classe

unwind [key in keys(customVar_classe) where key = "http://jcr.di.uminho.pt/m51-clav#
    titulo" | [customVar_classe[key],null,key]] + [(customVar_classe)-[inner_P1288287199
    ]->(customVar_titulo) where type(inner_P1288287199) = "http://jcr.di.uminho.pt/m51-
    clav#titulo" | [customVar_titulo.uri,customVar_titulo,type(inner_P1288287199)]] as
    unwindVar_939664000
with *,unwindVar_939664000[0] as titulo,unwindVar_939664000[1] as customVar_titulo

unwind [(customVar_classe)-[inner_P1386205736]->(customVar_o3605269353) where type(
    inner_P1386205736) = "http://jcr.di.uminho.pt/m51-clav#temDono" and
    customVar_o3605269353.uri = "http://jcr.di.uminho.pt/m51-clav#ent_INFARMED" | type(
    inner_P1386205736)] as unwindURI_1041270957

unwind [(customVar_classe)-[inner_P3935305205]->(customVar_o3541583258) where type(
    inner_P3935305205) = "http://jcr.di.uminho.pt/m51-clav#temDono" and
    customVar_o3541583258.uri = "http://jcr.di.uminho.pt/m51-clav#ent_IMT" | type(
    inner_P3935305205)] as unwindURI_1917891950

with classe,titulo
where (classe is not null) and (titulo is not null)

RETURN classe,titulo
ORDER BY titulo
LIMIT 5
```

Código 6.10: Transcrição da *query* 6.9 pela gramática

Vamos agora falar das últimas palavras reservadas que estão traduzidas e funcionais: `LIMIT` e `ORDER BY`, bem como da composição vírgula entre triplos e de um detalhe sobre a tradução que até agora não foi expresso directamente mas que é importante.

A implementação de `LIMIT` em CYPHER é trivial pelo facto que a palavra reservada existe em ambas as linguagens e tem exactamente a mesma sintaxe e funcionalidade, pelo que apenas é preciso copiar o `LIMIT` da *query* SPARQL para a *query* CYPHER.

A implementação do `ORDER BY` também é muito similar pelo facto que em ambas as plataformas ele existe e tem o mesmo comportamento, mas não é cópia exacta porque :

1. as variáveis em SPARQL são sempre precedidas de **?** enquanto que são apenas nomes em CYPHER (tem de se retirar os pontos de interrogação)

2. declarar ordenação ascendente e descendente com base na variável é um pouco diferente (ver Neo4J (2020f) e W3C (2013h) para mais detalhes), sendo que para a nossa tradução apenas está a funcionar ordenação com uma única variável (podendo-se utilizar `ASC` e `DESC` livremente, se bem que `ASC` é desnecessário pois é a ordenação que ocorre naturalmente)

Finalmente falamos da última composição entre triplos do SPARQL: por vírgula. Nesta composição estamos a fixar tanto o sujeito como o predicado do objecto (em contraste com a composição ponto e virgula que apenas fixa o sujeito e a composição ponto que não fixa nada). Devido a isto esta composição é normalmente usada para fazer `match` de triplos onde o mesmo predicado tem vários valores possíveis, como no exemplo acima onde estamos à procura de classes que tenham dois donos específicos. Traduzir este tipo de composição é feito do mesmo modo que a composição vírgula e ponto, tratando os dois triplos de forma sequencial com único `match` do sujeito.

Este exemplo permite-nos ver algo curioso: existem expressões `unwind` que nunca são chamadas, ou seja, nunca usamos os dados delas obtidas no resto da *query*. Isto acontece pela maneira como os `unwinds` nesta tradução estão a funcionar.

Estas expressões obtêm dados de triplos sobre o predicado e objecto que são depois renomeados caso sejam necessários. No entanto para a composição ponto que estamos a exemplificar tanto o predicado como o objecto do triplo são valores literais na *query* feita, i.e., não são variáveis, pelo que os dados obtidos pela expressão `unwind` são inúteis e como tal nunca são mais usados. A razão pela qual fazemos estes `unwinds` tem a ver com o facto que ao procurar dados para triplos também estamos a confirmar a sua existência na ontologia, pelo que ao calcular esta expressão a *query* está a fazer a procura dos triplos que, apesar de terem valor fixo para predicado e objecto, têm de existir na ontologia sobre certas condições para certo sujeito para que o triplos tenha solução válida na ontologia atual.

Uma optimização que foi ainda implementada tem a ver com os nodos que obtemos para elementos numa situação muito especifica: quando são usados como objecto noutro triplo. Consideremos o exemplo seguinte:

```
PREFIX : <http://jcr.di.uminho.pt/m51-clav#>
select ?classe where {
  :c100.10 :temPai ?pai .
    ?classe :temPai ?pai .
}
order by ?classe
```

Código 6.11: *Query* para obter todas as classes que tenham o mesmo pai que `:c100.10`

A sua tradução pela gramática é

```
match(customVar_fixed_S1180356723)
where customVar_fixed_S1180356723.uri ="http://jcr.di.uminho.pt/m51-clav#c100.10"

unwind [key in keys(customVar_fixed_S1180356723) where key =
    "http://jcr.di.uminho.pt/m51-clav#temPai"  | [customVar_fixed_S1180356723[key],null,
    key]] + [(customVar_fixed_S1180356723)-[inner_P1096641676]->(customVar_pai) where
    type(inner_P1096641676) = "http://jcr.di.uminho.pt/m51-clav#temPai"  | [customVar_pai
    .uri,customVar_pai,type(inner_P1096641676)]] as unwindVar_1921693601
```

```
with *,unwindVar_1921693601[0] as pai,unwindVar_1921693601[1] as customVar_pai

match(customVar_classe)

unwind
case customVar_pai is null
when true
then [key in keys(customVar_classe) where key =
    "http://jcr.di.uminho.pt/m51-clav#temPai"  and customVar_classe[key]=pai | [
    customVar_classe[key],null,key]]
else
[(customVar_classe)-[inner_P287240114]->(customVar_pai) where type(inner_P287240114) =
    "http://jcr.di.uminho.pt/m51-clav#temPai"  | [customVar_pai.uri,customVar_pai,type(
    inner_P287240114)]]
end as unwindVar_1288287199

with *,customVar_classe.uri as classe

with pai,classe
where (pai is not null) and (classe is not null)

RETURN classe
ORDER BY classe
```

Código 6.12: Transcrição da *query* 6.11 pela gramática

Podemos ver que o `unwind` desta tradução é bastante diferente de todos os que foram apresentados até agora: em vez de conter *arrays* dentro dele podemos observar que tem um `case` do Neo4J (Neo4J (2020b)), onde estamos a testar se a variável `customVar_pai` (variável que contém os dados da variável `?pai` em Neo4J) tem valor `null` ou não. Se tiver, estamos a procurar apenas nas propriedades do nodo da classe que queremos obter. Caso contrário, procuramos apenas nas relações que este tem com outros nodos da ontologia.

Esta implementação é uma optimização da forma mais geral de procurar, que noutros casos seria a de junção dos dois *arrays*, possibilitada neste caso pelos dados que já obtivemos anteriormente na *query*: o valor do nodo que terá o valor de pai. Na verdade este valor pode ser nulo, algo que acontece sempre que estamos na presença de uma propriedade do nodo e não numa relação com outros nodos (para confirmar isto observe que todos os *arrays* que obtêm dados das propriedades de um elemento retornam um valor nulo que na procura de propriedades corresponde ao nodo capturado).

Assim, ao utilizarmos o `case` procurando ver se o valor é nulo podemos determinar conclusivamente se estamos numa propriedade de um elemento ou numa relação com outro nodo, e como tal podemos utilizar apenas um dos *arrays* para fazer a captura dos dados pretendidos.

6.2.1 *Queries ask e describe*

Finalmente temos de referir como as *queries* `ask` (W3C (2013b)) e `describe` (W3C (2013c)) foram implementadas, visto que até agora apenas discutimos *queries* `select`.

As *queries* `ask` em SPARQL retornam sempre ou `true` (quando a *query* tem solução na ontologia) ou `false` (caso contrário), sendo que as *queries* que se podem fazer à ontologia em `ask` são equivalentes às do `select` em todas as maneiras. Assim, implementar a *query* `ask` não é mais do que alterar o valor de retorno da *query* para ter valor `true`.

Esta solução garante que se a *query* tiver solução o retorno será `true`, mas caso falhe o seu retorno não será `false`: será `null` (nenhum resultado retornado). Para resolvr isto a aplicação Web detecta este tipo de *queries* quando são feitas e se isto acontecer altera o resultado para `false` e apresenta isso ao utilizador.

As *queries* `describe` são um pouco mais complexas, tanto pelo facto que podem ter URI's singulares no seu cabeçalho como que descrevem sempre elementos da ontologia (que estão nas condições ditadas pela *query* a ser feita).

Para isso, a solução passou por duas alterações ao que tinha na `select`: após fazer a *query* que está entre { e } como no `select` são feitos numerosos `matches` com os elementos que obtivemos ao longo da *query*, fazendo `match` a partir tanto dos URI's singulares como dos que foram obtidos na *query* acima, sendo que no final se retorna tudo.

Na aplicação Web estas são feitas usando a função de exporte do Neosemantics (Neosemantics (2020a)) para fazer a colecção dos elementos que lhe pedimos na *query* e os resultados são depois enviados e colocados num ficheiro para o utilizador fazer download de (que terá dentro dele a resposta à *query* feita).

6.3 Validação da transcrição feita pela gramática

Tendo explicado as capacidades da gramática produzida temos agora de validar a transcrição feita para confirmar que o que foi criado está funcional.

A metodologia usada para fazer esta validação é simples: foram seleccionadas duas ontologias que são carregadas para ambos os motores de bases de dados (directamente em GraphDB, através do Neosemantics em Neo4J), sendo em seguida feitas uma série de *queries* em SPARQL a ambas (directamente em GraphDB e após traduzidas em Neo4J). Os resultados das *queries* são depois comparados para verificar a qualidade da tradução feita.

Para esta validação foram escolhidas as ontologias *Pókedex* e Tabela Periódica (ver 5.2 para mais detalhes).

Serão agora apresentadas seis *queries* feitas e seus resultados (sendo que todo o trabalho feito para validação da gramática pode ser consultado em A.1).

Estas *queries* estão divididas em duas categorias: *queries* genéricas (i.e., *queries* que se aplicam a qualquer ontologia) e *queries* especificas da ontologia (i.e., *queries* que só fazem sentido no contexto da ontologia a que estão a ser feitas), com a primeira categoria a mostrar resultados de

queries em ambas as ontologias em tanto GraphDB como Neo4J e a segunda a apenas mostrar resultados para uma ontologia (mas em ambos os motores).

De notar que os resultados do Neo4J estão a ser transformados para terem a mesma visualização que os do GraphDB, pois como foi dito anteriormente o Neo4J contém os URI's completos guardados, o que levaria a que se fossem escritos directamente os resultados seriam extremamente difíceis de comparar.

Assim os resultados do Neo4J foram convertidos usando os prefixos indicados ao inicio de cada subsecção das ontologias. Para além dessas deve-se ter em conta as convenções do SPARQL para *namespaces* pré-definidos (W3C (2013g)), bem como o *namespace* do OWL (que tem valor de `http://www.w3.org/2002/07/owl#`).

6.3.1 *Queries genéricas*

Devido à sua natureza estas *queries* servem mais para determinar estrutura e informações não especificas sobre as ontologias que estão a ser questionadas como determinar classes que fazem parte da ontologia, tipos de relações e propriedades presentes, número de indivíduos presentes, ...

Foram assim escolhidas duas *queries* nesta categoria: a primeira para obter todas as classes de uma ontologia e a segunda para obter a sua lista de elementos.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
select distinct * where {
  ?s a owl:Class .
}
order by ?s
```

Código 6.13: Query SPARQL para obter todas as classes de uma ontologia

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
select distinct * where {
  ?s a owl:NamedIndividual .
}
order by ?s
limit 10
```

Código 6.14: *Query* SPARQL para obter dez elementos de uma ontologia

Estas *queries* foram escolhidas em particular por serem *queries* muito comuns de se fazer a ontologias durante a exploração dela, pelo que servem como bom exemplo para a validação da tradução.

Note-se que ambas as *queries* têm os seus resultados ordenados (para simplificar a leitura dos resultados obtidos nas tabelas) e que a segunda *query* está limitada a apenas dez elementos (pois para ambas as ontologias a serem usadas obter todos os elementos da ontologia geraria tabelas que ocupariam dezenas de páginas neste documento), sendo que em anexo pode-se ver resultados dessa segunda *query* para vinte e quarenta elementos capturados.

Apresentamos em seguida as tabelas de resultados das *queries* para ambas as ontologias em ambos os motores. É importante observar que as tabelas de resultados que estejam lado a lado têm sempre os resultados de GraphDB à esquerda e os resultados do Neo4J à direita

Comecemos por ver os resultados na ontologia *Pókedex*.

Pókedex

Dentro desta ontologia o prefixo *pokemon:* tem o valor URI completo de `<http://www.chalkos.net/ontologies/2015/pokemon#>`.

s
pokemon:AcquisitionMethod
pokemon:Attribute
pokemon:Description
pokemon:DmgMultiplier
pokemon:Evolution
pokemon:Game
pokemon:Location
pokemon:Move
pokemon:MoveLearning
pokemon:PokeType
pokemon:Pokemon
pokemon:PokemonAcquisition
pokemon:PokemonAttr

s
pokemon:AcquisitionMethod
pokemon:Attribute
pokemon:Description
pokemon:DmgMultiplier
pokemon:Evolution
pokemon:Game
pokemon:Location
pokemon:Move
pokemon:MoveLearning
pokemon:PokeType
pokemon:Pokemon
pokemon:PokemonAcquisition
pokemon:PokemonAttr

Tabela 1: Resultados para a *query* 6.13 na ontologia *Pókedex* nos motores GraphDB e Neo4J

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking

Tabela 2: Resultados para a *query* 6.14 na ontologia *Pókedex* no motor GraphDB

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking

Tabela 3: Resultados para a *query* 6.14 na ontologia *Pókedex* no motor Neo4J

Como é fácil de ver os resultados obtidos para ambas as *queries* são exactamente iguais em ambos os motores, sendo que para ambas estas *queries* a tradução feita funciona sem qualquer perda causada pelas diferenças entre como os dados estão guardados em ambos os motores, o que leva a que possamos dizer que para estas duas *queries* a tradução é boa.

Para confirmarmos que está verdadeiramente funcional, testamos agora na outra ontologia que estamos a estudar.

Tabela Periódica

Dentro desta ontologia o prefixo *:* tem o valor URI completo de
`<http://www.daml.org/2003/01/periodictable/PeriodicTable#>`

s
:Block
:Classification
:Element
:Group
:Molecule
:Period
:StandardState
:hasElement

s
:Block
:Classification
:Element
:Group
:Molecule
:Period
:StandardState
:hasElement

Tabela 4: Resultados para a *query* 6.13 na ontologia Tabela Periódica nos motores GraphDB e Neo4J

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba

Tabela 5: Resultados para a *query* 6.14 na ontologia Tabela Periódica nos motores GraphDB e Neo4J

Como podemos observar, os resultados obtidos são perfeitamente iguais, o que nos leva a concluir que a tradução feita automaticamente é válida no caso geral para as *queries* escolhidas.

6.3.2 *Queries especificas*

Com os exemplos mais gerais tratados vamos passar a *queries* mais especificas da ontologia, i.e., *queries* que servem para obter dados que apenas fazem sentido para a ontologia que estamos a fazer a *query* a.

Para cada uma das ontologias faremos o mesmo que nas *queries* gerais, mostrando resultados de duas *queries* de entre as que foram feitas para a validação da gramática, apresentados da mesma forma que anteriormente.

A diferença principal é que apenas mostraremos os resultados referentes à ontologia para a qual a *query* especifica se destina a visto que, pela natureza deste tipo de *queries*, os resultados na outra ontologia de estudo não seriam úteis.

Devido à natureza destas *queries* todas foram feitas para emular o que alguém que queira usar os dados da ontologia e tenha conhecimento da disposição destes poderia questionar a ontologia sobre (i.e., pretendemos obter dados úteis e não só listagens de propriedades e relações entre elementos da ontologia).

Pókedex

Comecemos por ver *queries* especificas ao *Pókedex*. Devido à natureza desta ontologia, a maior parte de *queries* especificas estará centrada em obter dados associados a *Pokemons*.

A primeira *query* escolhida pretende verificar que *Pokemons* existem no jogo *Pokemon Red Version* que podem ser capturados tanto na rota um como na rota dois.

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?s where {
  ?s a pokemon:Pokemon ;
```

```
      pokemon:HasPokemonAcquisition ?aquisitionList .
    ?aquisitionList pokemon:PokeAcquiredInGame pokemon:game_red ;
        pokemon:PokeAcquiredInLocation pokemon:loc_route_1,pokemon:loc_route_2 .
}
order by ?s
limit 10
```

Código 6.15: *Query* SPARQL para obter *Pokemons* que podem ser capturados em tanto a rota um como a rota dois do jogo *Pokemon Red Version*

Esta *query* é mais complexa que outras mostradas até agora, contendo as três maneiras de compor triplos diferentes: ponto, vírgula e vírgula e ponto. A tradução desta *query* e obtenção dos resultados abaixo mostrados em combinação com todos os exemplos mostrados permite-nos concluir com certeza que a tradução automática destas três composições está completamente funcional.

s
pokemon:poke_016
pokemon:poke_019

s
pokemon:poke_016
pokemon:poke_019

Tabela 6: Resultados para a *query* 6.15 nos motores GraphDB e Neo4J

Como segundo exemplo para esta ontologia pretendemos obter *Pokemons* da ontologia nos quais pelo menos um dos seus tipos é *flying*.

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?s where {
  ?s a pokemon:Pokemon .
    filter Exists { ?s pokemon:HasType pokemon:flying }
}
order by ?s
limit 10
```

Código 6.16: *Query* SPARQL para obter *Pokemons* em que um dos seus tipo é *flying*

Esta *query* permite-nos validar uma das palavras reservadas implementadas: o `EXISTS` dentro do `filter` que neste caso filtra os *Pokemons* para que tenha de existir um triplo que indique que o seu tipo é *flying*, servindo assim para fazer com que a *query* apenas retorne os *Pokemons* que queremos obter dela. Os resultados abaixo demonstram que a *query* foi bem traduzida e dão-nos confiança na tradução para este tipo de *queries* pela gramática.

s
pokemon:poke_006
pokemon:poke_012
pokemon:poke_016
pokemon:poke_017
pokemon:poke_018
pokemon:poke_021
pokemon:poke_022
pokemon:poke_041
pokemon:poke_042
pokemon:poke_083

s
pokemon:poke_006
pokemon:poke_012
pokemon:poke_016
pokemon:poke_017
pokemon:poke_018
pokemon:poke_021
pokemon:poke_022
pokemon:poke_041
pokemon:poke_042
pokemon:poke_083

Tabela 7: Resultados para a *query* 6.16 nos motores GraphDB e Neo4J

Tabela Periódica

Para finalizar este capítulo vamos observar as *queries* especificas à Tabela Periódica. Estas *queries* serão completamente focadas em obter dados de elementos da tabela periódica e associações entre estes.

Como primeiro exemplo de *query* escolhemos obter todos os elementos que estão associados ao período sete da tabela periódica.

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select * where{
    ?s a :Element ;
       :period :period_7 .
}
order by ?s
```

Código 6.17: *Query* SPARQL para obter todos os elementos que que são parte do 7º período da tabela periódica

s	s
:Ac	:Ac
:Am	:Am
:Bh	:Bh
:Bk	:Bk
:Cf	:Cf
:Cm	:Cm
:Db	:Db
:Es	:Es
:Fm	:Fm
:Fr	:Fr
:Hs	:Hs
:Lr	:Lr
:Md	:Md
:Mt	:Mt
:No	:No
:Np	:Np
:Pa	:Pa
:Pu	:Pu
:Ra	:Ra
:Rf	:Rf
:Sg	:Sg
:Th	:Th
:U	:U
:Uub	:Uub
:Uuh	:Uuh
:Uun	:Uun
:Uuo	:Uuo
:Uup	:Uup
:Uuq	:Uuq
:Uus	:Uus
:Uut	:Uut
:Uuu	:Uuu

Tabela 8: Resultados para a *query* 6.17 nos motores GraphDB e Neo4J

Podemos ver que os resultados são idênticos, pelo que a tradução é boa. No entanto, a *query* é bastante simples, e nada do que foi pedido não tinha já sido observado anteriormente.

Vamos agora considerar um exemplo mais complexo: queremos obter todos os elementos cujo peso atómico está entre cinco e vinte, correspondendo à *query* seguinte:

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select ?s where{
    ?s a :Element ;
       :atomicNumber ?number .
    FILTER ( (?number > 5) && (?number < 20) )
```

```
}
order by ?s
```

Código 6.18: *Query* SPARQL para obter todos os elementos cujo peso atómico é superior a cinco e inferior a vinte

s
:Al
:Ar
:C
:Cl
:F
:K
:Mg
:N
:Na
:Ne
:O
:P
:S
:Si

s
:Al
:Ar
:C
:Cl
:F
:K
:Mg
:N
:Na
:Ne
:O
:P
:S
:Si

Tabela 9: Resultados para a *query* 6.18 nos motores GraphDB e Neo4J

Tal como em todas as *queries* apresentadas até agora, os resultados são equivalentes entre os motores.

Com todas estes exemplos (incluindo os presente em A.1) podemos concluir que a tradução feita é de boa qualidade para as funcionalidades que são delineadas neste capítulo.

6.4 RESUMO

Neste capitulo falamos da parte mais teórica da resolução do trabalho: a gramática desenvolvida para traduzir entre SPARQL e CYPHER.

Falamos de decisões tomadas na sua criação e no processo da sua implementação, seguindo-se os resultados obtidos e a validação da transcrição final.

Com este capitulo estamos finalmente em condição de responder a todas as questões de investigação:

1. É possível armazenar ontologias no Neo4J e explorá-las de uma forma simples?

   O armazenamento de ontologias em Neo4J é possível por uso de Neosemantics e a sua exploração é possível através de CYPHER. No entanto, esta exploração apenas será eficaz se houver o conhecimento de como os dados são guardados pelo aplicação.

2. É possível criar um mapeamento entre as linguagens de *query* SPARQL e CYPHER tal que se consiga exprimir uma *query* numa linguagem e através de alguma transformação obter uma *query* equivalente na outra linguagem?

   O trabalho apresentado acima prova que é possível, pelo menos de modo parcial, mapear uma *query* escrita em SPARQL em CYPHER e obter resultados equivalentes quando feitos à mesma ontologia.

   Uma coisa que não é mencionada explicitamente mas que deve também ser óbvia é que a tradução no outro sentido (CYPHER para SPARQL) não faz qualquer sentido, visto que uma grande quantidade de *queries* em CYPHER não têm qualquer equivalente em SPARQL devido à maior generalidade do esquema utilizado pelo Neo4J.

3. É possível através de algum método interrogar o motor de bases de dados Neo4J através do uso da linguagem de *query* SPARQL?

   Pelo uso das transcrições feitas pela gramática desenvolvida é possível criar traduções válidas e equivalentes a *queries* SPARQL em CYPHER e é possível interrogar Neo4J diretamente com essas traduções, pelo que é possível (de forma indireta e apenas para alguns tipos de *queries*) interrogar Neo4J pelo uso de SPARQL.

7

APLICAÇÃO WEB PARA NEO4J

Com a primeira parte do trabalho explicitada podemos falar do resto do trabalho feito para esta tese: a aplicação WEB.

Como foi referido anteriormente o objectivo desta aplicação Web é de adicionar capacidades ao Neo4J para lidar com ontologias (ou seja, ter capacidades similares às do GraphDB).

Neste capitulo vamos explorar todos os detalhes sobre a aplicação e as capacidades que adiciona a Neo4J.

7.1 ARQUITECTURA DA APLICAÇÃO

A primeira coisa que devemos explicar é a arquitectura da aplicação criada, que consiste em dois servidores: `backend` e `frontend`.

Backend

O servidor de `backend` está escrito em JS usando Express.js (Foundation (2020)), utilizando as capacidades desse `framework` para permitir tanto contactar o Neo4J como a `frontend` através de HTTP.

É neste servidor que quase todas as capacidades da aplicação são implementadas.

Frontend

O servidor de frontend está escrito em JS usando o `framework` Vue (Vue (2020)) + Vuetify (Vuetify (2020)) para permitir ao utilizador interagir com as capacidades disponibilizadas pela backend e assim usar a aplicação.

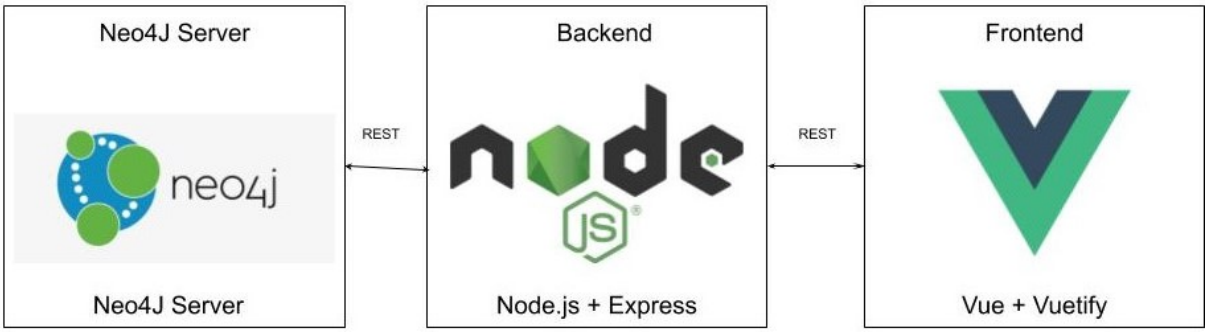


Figura 19: Arquitectura da aplicação

Adicionalmente a aplicação utiliza um *parser* externo, criado da gramática definida ao longo da tese através da ferramenta PEG.js e escrito em JS, podendo assim ser importado e usado de forma directa pela aplicação.

7.2 Funcionalidades e páginas

Ao entrar na aplicação (i.e., conectar ao servidor na rota /) deparamo-nos com a página inicial da aplicação.

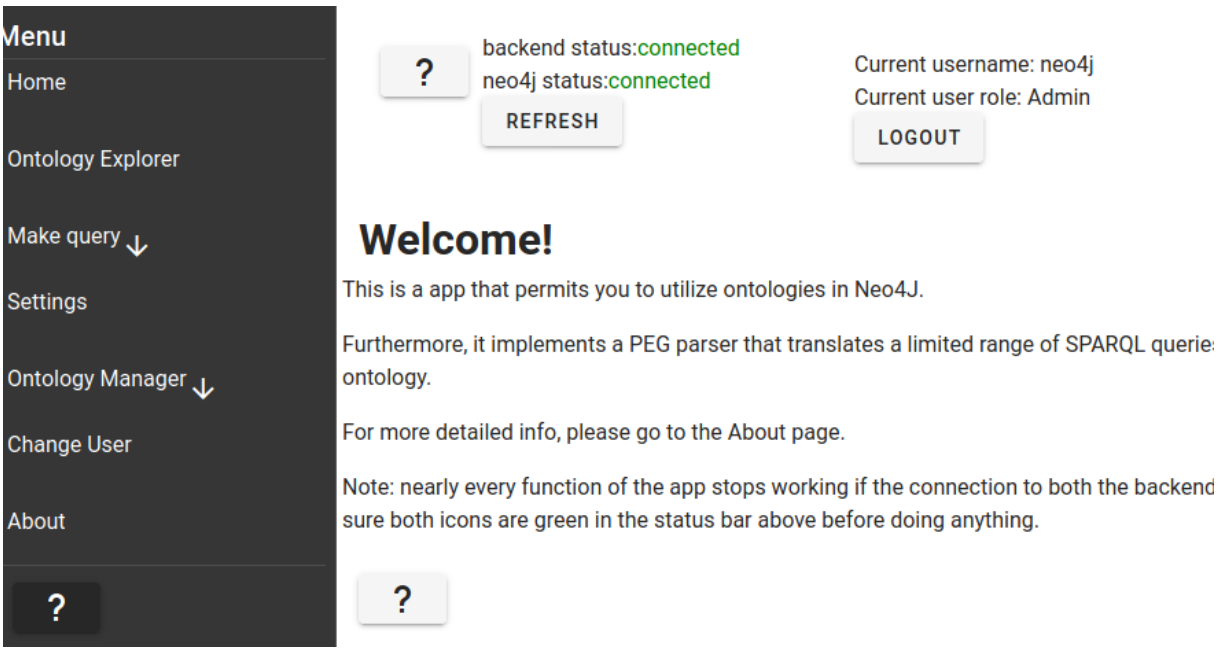


Figura 20: Página inicial da aplicação

Antes de continuarmos a falar do resto das páginas da aplicação, temos de falar sobre como elas estão estruturadas

7.2.1 *Layout da página*

Esta página, bem como todas as outras páginas da aplicação, é constituída de três componentes:

Barra lateral de navegação

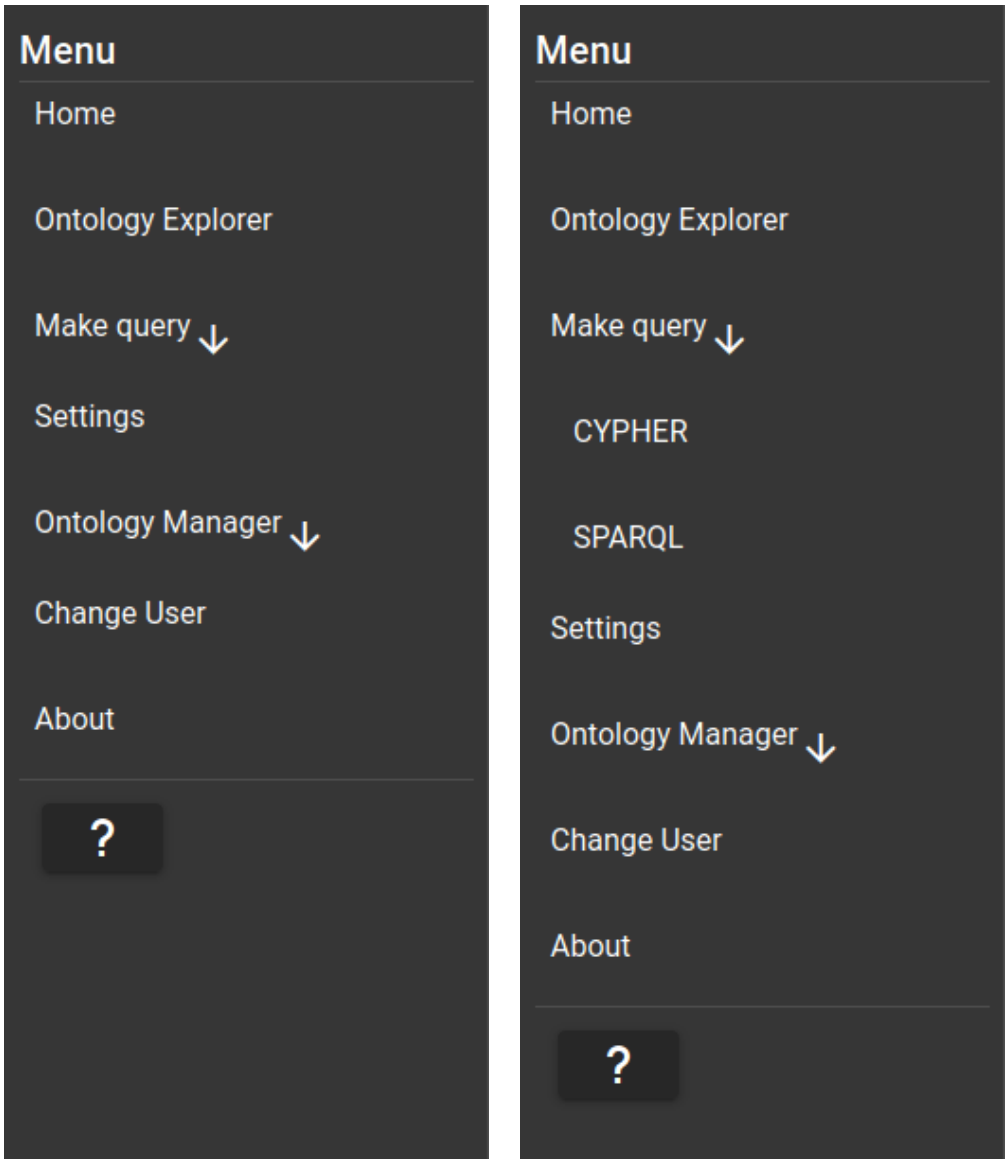


Figura 21: Barra lateral da aplicação compressa (esquerda) e com submenu expandido (direita)

Esta barra lateral contém a navegação pela aplicação. Cada elemento da barra pode ser carregado para alterar a página actual, de modo a expor todas as capacidades da aplicação.

Adicionalmente pode-se expandir certos elementos da barra para mostrar sublistas de navegação agrupadas por funcionalidade.

Esta barra está presente em todas as páginas permitindo navegar para quase todas as rotas da aplicação, como pode-se ver no diagrama de fluxo seguinte:

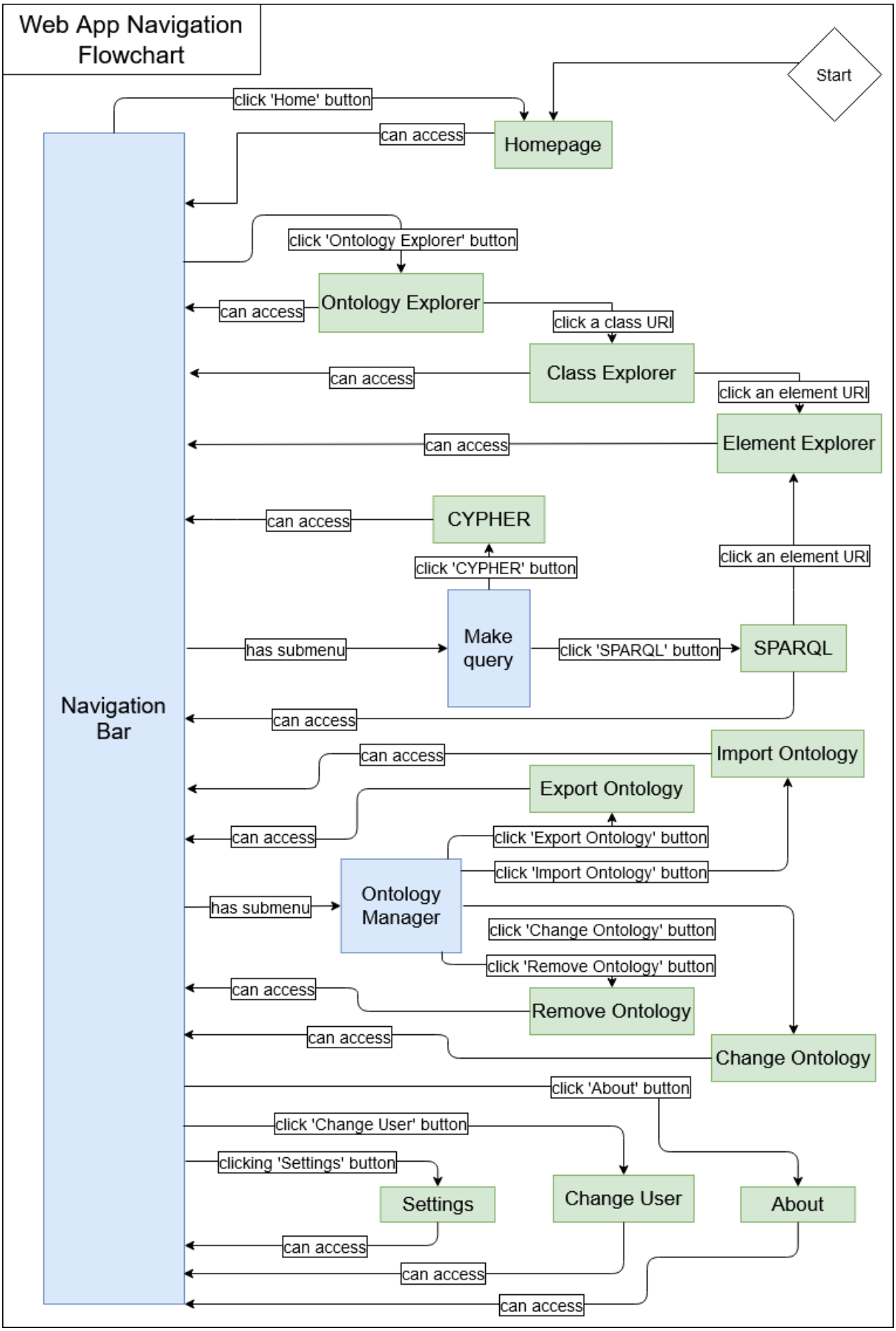


Figura 22: Diagrama de fluxo da navegação dentro da aplicação

Barra de topo de estado

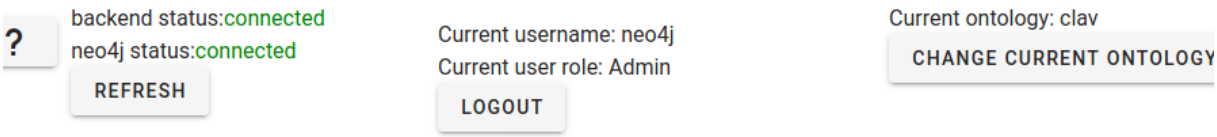


Figura 23: Barra de estado da aplicação

Esta barra de topo contém informação sobre numerosos estados que a aplicação actual têm, nomeadamente:

1. O estado da conexão com os servidores da *backend* e do Neo4J
2. O utilizador atualmente conectado e o seu papel atual (nível de autorização que tem dentro do Neo4J)
3. A ontologia selecionada atualmente, que será aquela à qual todas as *queries* serão feitas e que pode ser explorada pela aplicação

Para todas as funcionalidades da aplicação associadas ao utilizador actual (ver 7.2.3 para mais detalhes) estarem completamente operacionais é necessário que:

- Os dois servidores estão ligados e acessíveis pela aplicação
- Um utilizador deve ter feito *login* do seu utilizador de Neo4J
- Uma ontologia deve estar selecionada para ser explorada

No caso de isto não ser verdade, a barra muda e permite ao utilizador resolver o(s) problema(s) (excepto ligar os servidores, uma operação que é feita externamente):

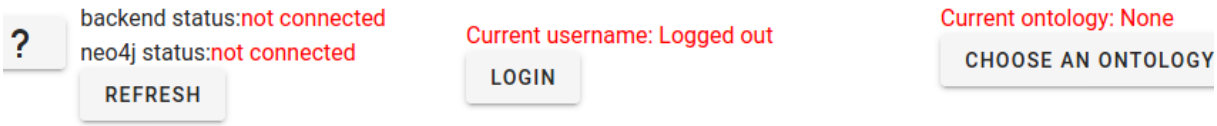


Figura 24: Barra de estado da aplicação caso os servidores não estejam ligados, não há utilizador conectado e não há ontologia selecionada

Assim, o utilizador pode carregar no botão **Login** para ser redirecionado para a página onde pode fazer *login* na aplicação, e pode carregar no botão **Choose an ontology** para escolher a ontologia a explorar na aplicação.

Conteúdo da página actual

Finalmente temos o conteúdo da página que está a ser visualizada, que no caso da página inicial é uma simples introdução à aplicação.

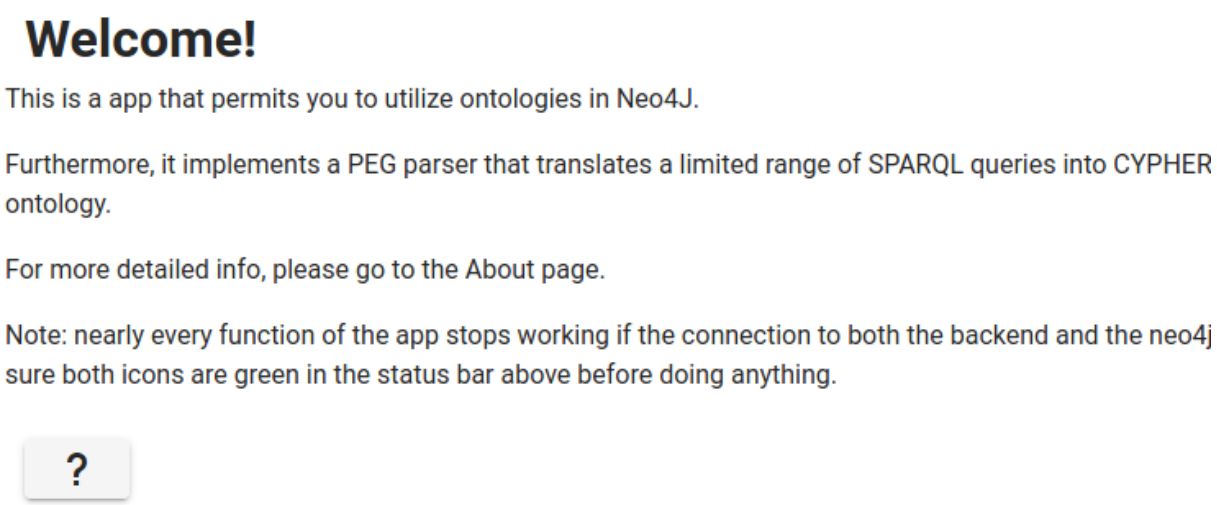


Figura 25: Parte do conteúdo da página principal

Botões de ajuda

Como deve ter notado nas imagens mostradas anteriormente existem vários botões com pontos de interrogação neles.

Figura 26: Botão de ajuda

Estes botões, presentes em todas as páginas e nas diferentes componentes da aplicação, contêm explicações simples para o que cada componente/página faz, de modo a auxiliar o utilizador caso não compreenda algo.

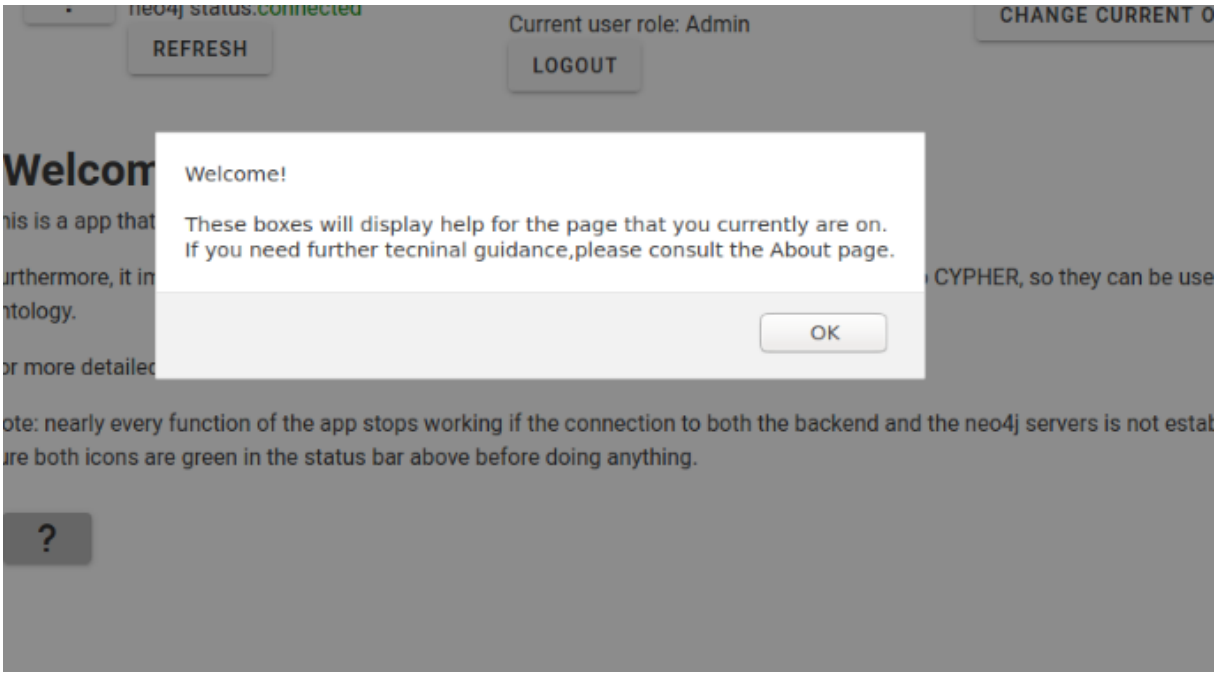


Figura 27: Mensagem associada ao botão de ajuda da página principal

7.2.2 *Páginas da aplicação*

Com o `layout` básico explicado podemos agora explorar as diferentes capacidades da aplicação disponibilizadas em diferentes páginas.

Páginas de informação

A aplicação tem duas páginas que não contêm nenhuma capacidade e servem apenas para mostrar informação ao utilizador: a página de entrada (referida anteriormente) e a página **About**.

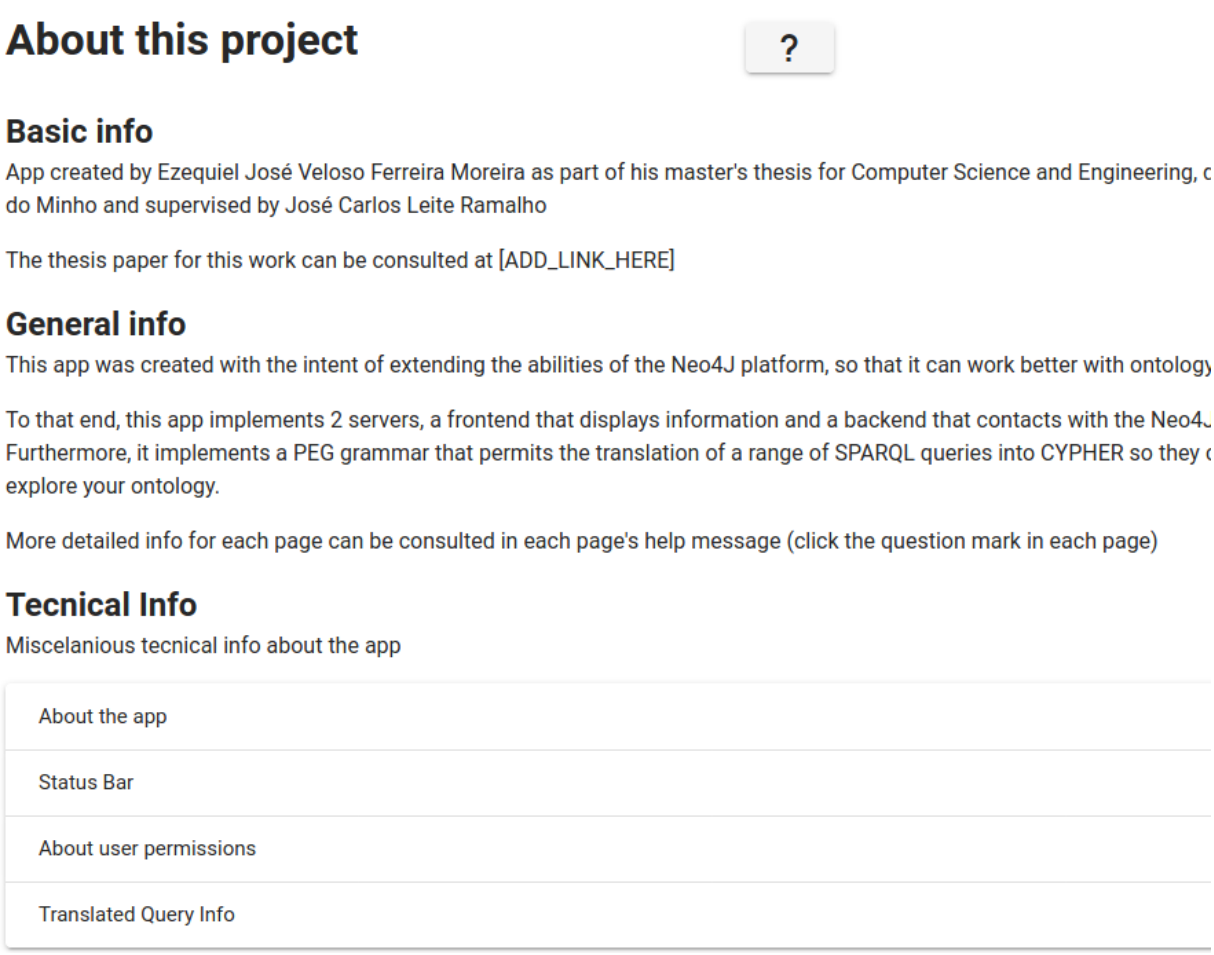


Figura 28: Página **About**, contendo informação detalhada sobre o projecto

Esta página contêm informação detalhada sobre numerosas partes do projecto e da aplicação, explicando em detalhe a barra de estado, permissões de utilizadores (ver 7.2.3) e tipos de *queries* SPARQL que a gramática consegue traduzir (ver 6.2).

Operações sobre o estado da frontend

Existem três páginas na aplicação que afectam directamente o estado do servidor frontend sem alterarem qualquer coisa na base de dados: **Settings**, **Change user** e **Change Ontology**.

As páginas de mudança de utilizador e mudança de ontologia são extremamente simples, consistindo apenas de duas linhas para introduzir informações de *login* e uma lista para selecionar entre as ontologias que estão carregadas em Neo4J respectivamente.

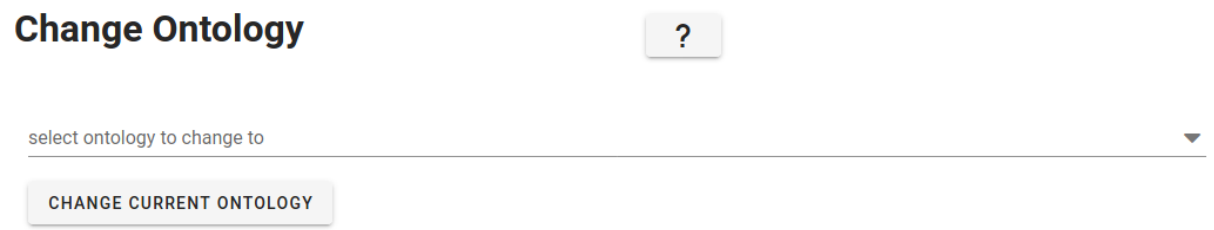


Figura 29: Página para mudar ontologia

Já a página de **Settings** é um pouco mais complicada, consistindo de numerosos menus e submenus que contêm configurações da aplicação (preferências de visualização do utilizador).

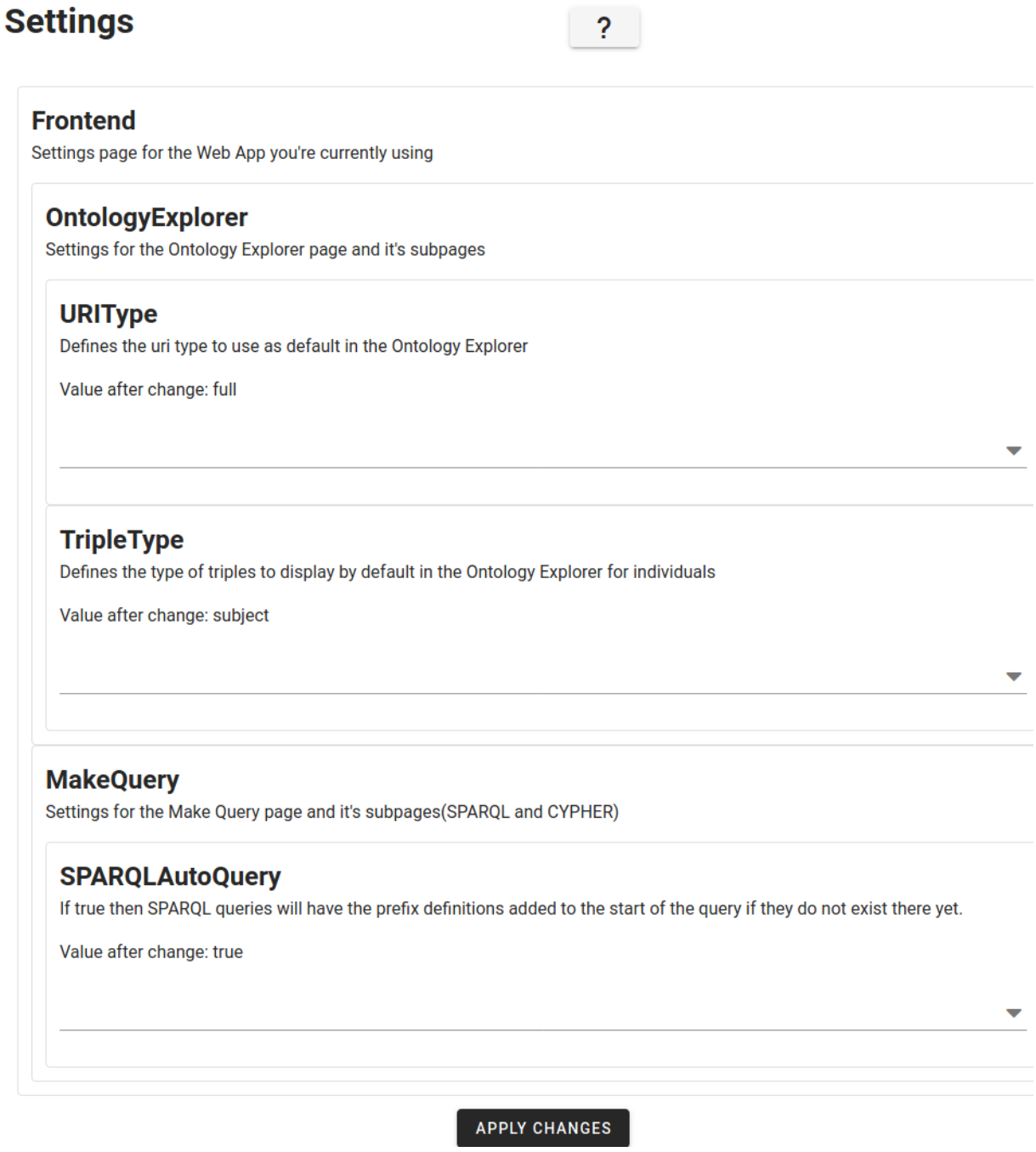


Figura 30: Página de configuração da aplicação

O utilizador pode selecionar qualquer configuração que possa mudar aqui e mudar para valores alternativos, confirmando as mudanças com um botão no fundo da página.

Operações sobre Neo4J

Na aplicação temos três páginas que permitem interagir com o Neo4J mais directamente: **Import Ontology**, **Export Ontology** e **Remove Ontology**.

Das três a que menos impacto tem é a página de exportação de ontologia, que pede para selecionar entre as ontologias do Neo4J que se quer obter e em qual dos formatos oferecidos quer que a ontologia seja exportada.

Após ter selecionado tudo, poderá carregar num botão e após uns segundos aparecerá uma caixa de dialogo para fazer download do ficheiro que contém a ontologia exportada.

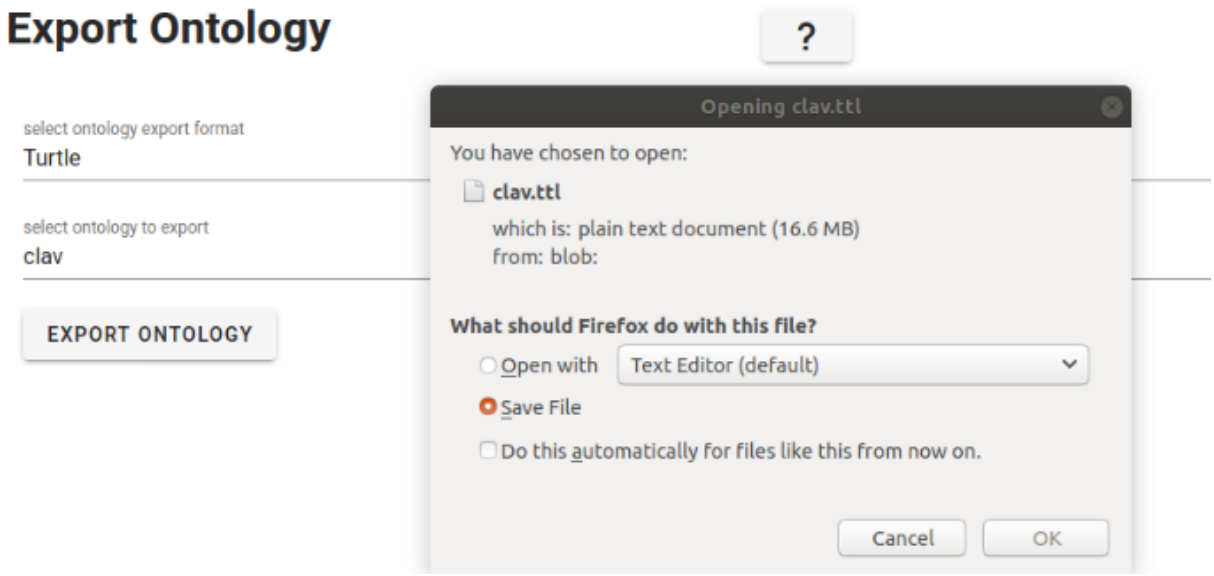


Figura 31: Caixa de dialogo para fazer download do ficheiro com a ontologia exportada

Em contraste com as outras duas páginas desta categoria esta operação não altera nada no Neo4J.

A página para fazer importe de uma ontologia para Neo4J consiste inicialmente numa linha onde se deve escrever o nome para colocar esta ontologia no Neo4J e uma caixa onde após carregar é aberto um explorador de documentos para permitir selecionar o ficheiro que contém a ontologia (em formato TURTLE ou RDF/XML).

Após ter selecionado o ficheiro aparece uma tabela com dados sobre os prefixos detectados nessa ontologia e é pedido que se escolha um deles para ser o principal da ontologia (sendo explicada na caixa de ajuda que aparece o que esse prefixo principal faz).

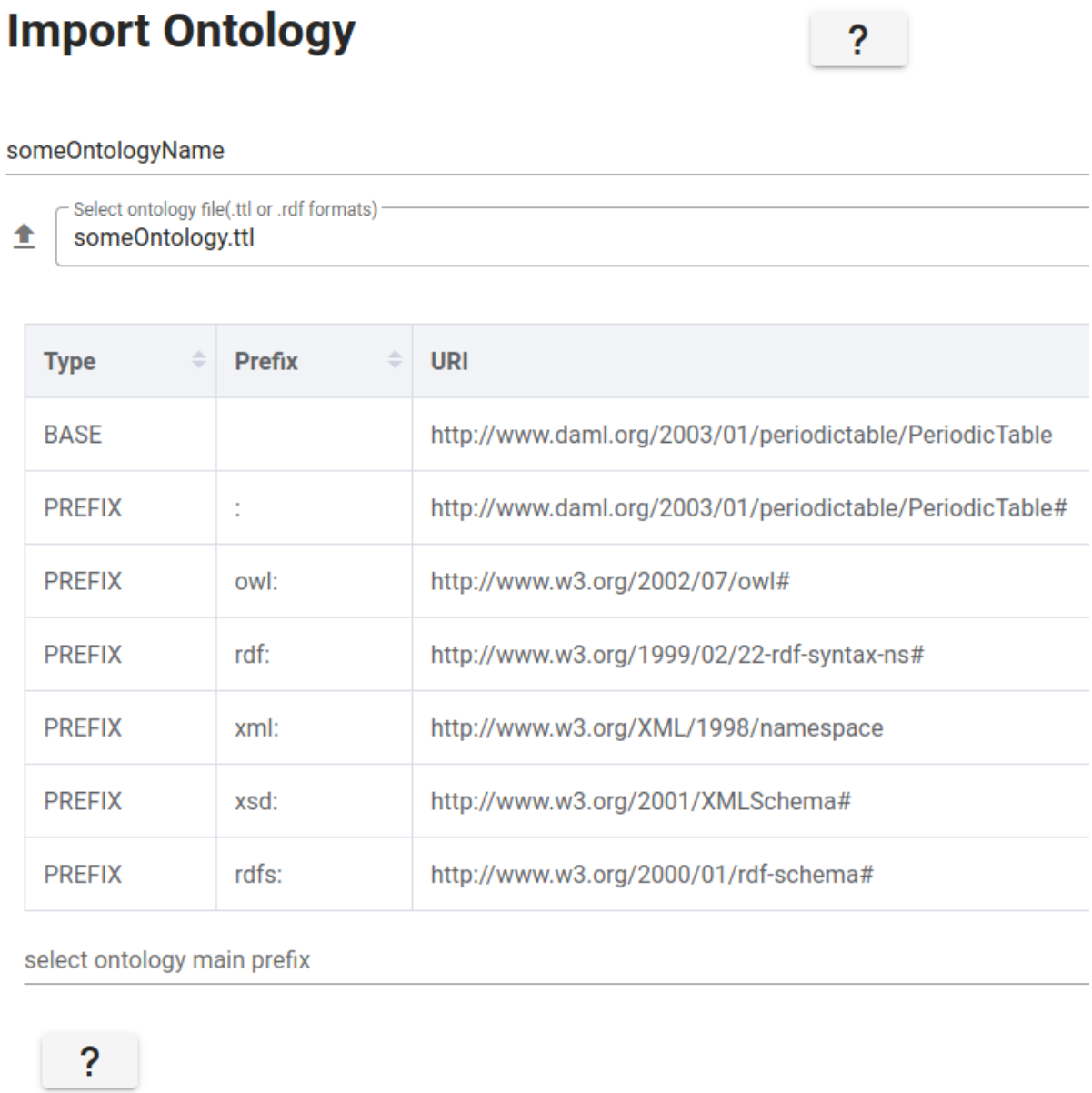


Figura 32: Importe de ontologias após selecção do nome e do ficheiro

Tendo sido tudo preenchido aparece um botão para permitir o importe de dados e, caso o utilizador tenha permissão para tal, começa o processo de importe de dados (onde a aplicação fica bloqueada num ecrã de `loading` até a operação concluir) após a confirmação do utilizador.

Durante este processo é criada uma nova base de dados no Neo4J com o nome especificado e é feito todo o processo de importe de dados da ontologia (ver 7.2.4).

Quando terminado o utilizador é avisado e a aplicação volta a como estava antes.

Em contraste a página de remoção de ontologia é extremamente simples, consistindo numa lista de ontologias que existem no Neo4J que após selecionada deve ser removida.

Após se ter selecionado a ontologia aparece um botão que ao ser carregado abre uma caixa de alerta para confirmar que se quer remover a ontologia.

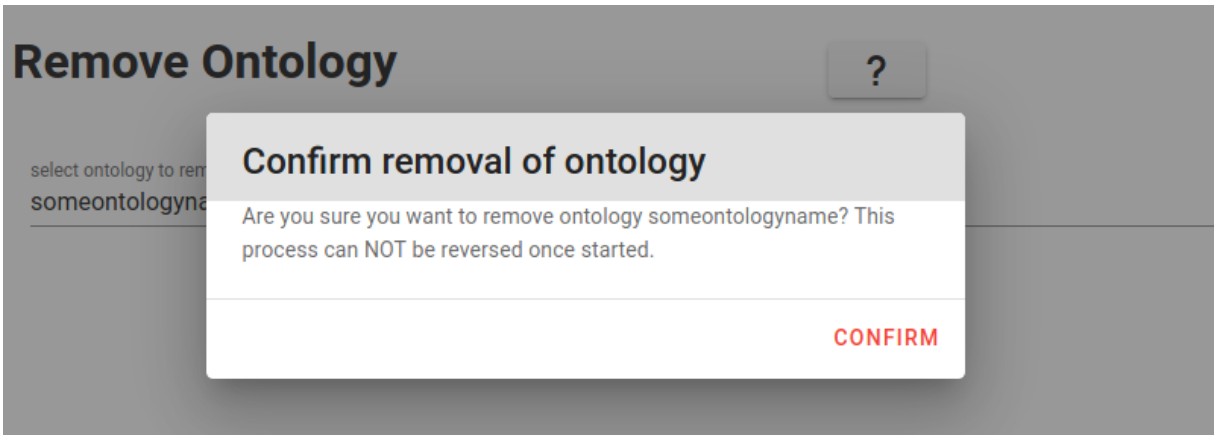


Figura 33: Caixa de confirmação de remoção da ontologia

Após ser confirmado e assumindo que tem permissão para tal o programa bloqueia e é realizada a remoção da ontologia em Neo4J, que consiste em remover a base de dados cujo nome foi selecionado.

No final é dada confirmação ao utilizador e pode continuar a trabalhar na aplicação.

Operações sobre a ontologia

Finalmente a aplicação tem cinco páginas (três das quais com navegação directa para elas) para lidar com exploração de ontologias.

As primeiras duas que vamos referir são as que existem sobre o menu **Make query**, tendo ambas esse objectivo: fazer *queries* sobre a ontologia actual usando CYPHER ou SPARQL.

Uma nota antes de falarmos em maior detalhe sobre as diferentes páginas para fazer *queries*: é impossível fazer *queries* que modifiquem a base de dados em Neo4J por aqui. Isto deve-se ao facto que todas as *queries* feitas a Neo4J são feitas por transações que para terem efeito permanente na base e dados requerem que seja feito `commit`. Como a gramática produzida não consegue transcrever *queries* que modifiquem ontologias permanentemente não faz sentido permitir que a aplicação faça modificações à base de dados. Note-se que o Neo4J pela *query* CYPHER ainda é capaz de dar resultados a *queries* que façam operações de modificação à base de dados e depois os processem, mas estas alterações não terão efeito permanente nos dados a serem guardados.

A página dedicada a fazer *queries* CYPHER é simples: uma caixa de texto para fazer *queries* e um botão para as submeter. Estas *queries* são depois enviadas a Neo4J directamente, feitas à base de dados da ontologia actual e os resultados aparecem depois numa tabela em baixo, mostrados de forma directa (i.e., não como triplos) devido ao facto que CYPHER não tem qualquer equivalente de triplos para mostrar os resultados das suas *queries*.

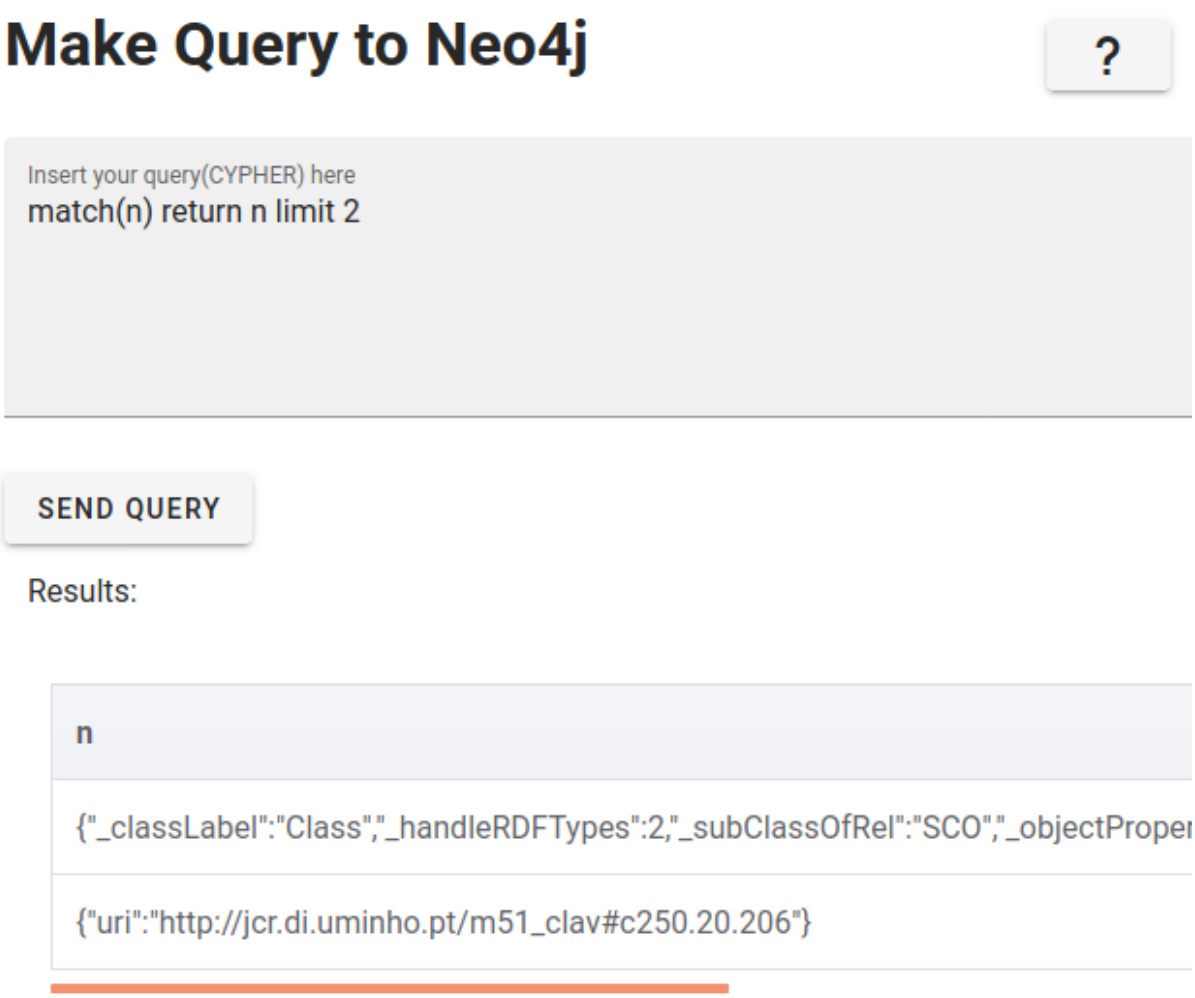


Figura 34: Página de *queries* SPARQL com *query* exemplo e tabela de resultados

Em contraste a página de *queries* SPARQL é mais complexa devido à necessidade de tradução. Ao inicio, é apenas uma caixa de texto que pede por uma *query* SPARQL e tem um botão para fazer tradução para CYPHER. Esta tradução é feita através do *parser* gerado por PEG.js que foi guardado dentro da aplicação e que é usado directamente na *query*.

Antes de passarmos ao resto da página devemos notar que existe prefixação automática com base na ontologia actual nesta caixa de *queries*, mas esta não é imediatamente visível. A razão para isto é técnica: a adição de definições `PREFIX` ao inicio da *query* é feita quando a componente da caixa de texto é actualizada, algo que acontece quando o utilizador deixa de conseguir escrever na caixa, i.e., quando seleciona algo fora da caixa de texto. Assim para poder ter prefixação automática tem de se carregar fora da caixa de texto para funcionar.

Figura 35: *Query* SPARQL de exemplo com prefixos que foram adicionados automaticamente

Após se ter colocado a *query* e carregado no botão **Parse Query** a página ganha dois novos botões e baixo bem como confirmação de que o *parsing* funcionou. Estes botões são **Toggle parsed query visibility**, que permite visualizar a *query* traduzida obtida da *query* SPARQL original (e que está escondida antes de se carregar no botão), e **Send translated Query**, que causa com que a *query* traduzida seja enviada ao Neo4J na base de dados da ontologia actual e seja feita.

Tendo-se enviado a *query* ao Neo4J os resultados são processados e mostrados ao utilizador sobre forma de tabela onde cada resultado é um elemento de um triplo da ontologia (de forma similar à que o GraphDB faz) que correspondem a fazer aquela *query* SPARQL à ontologia actualmente selecionada.

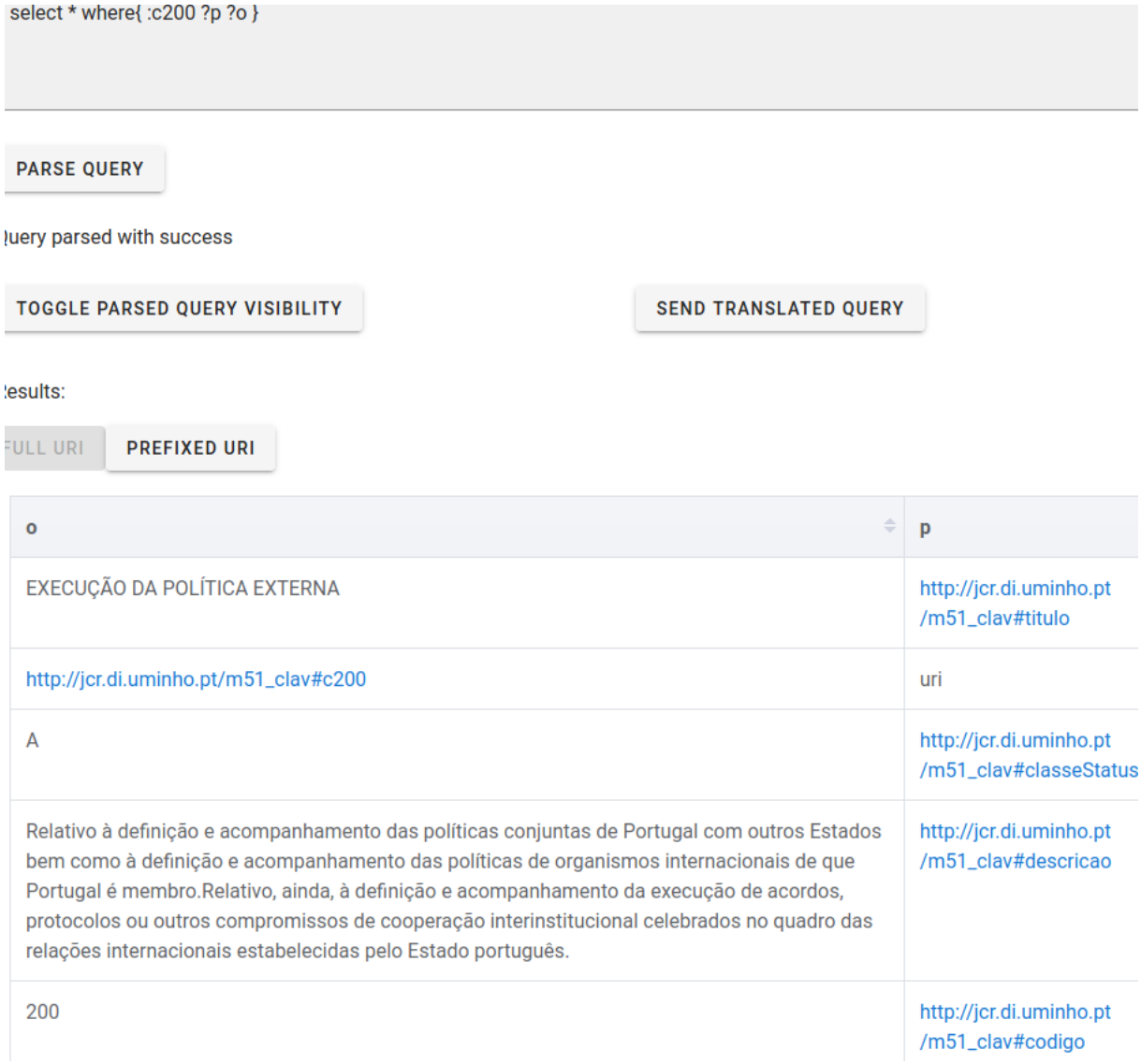
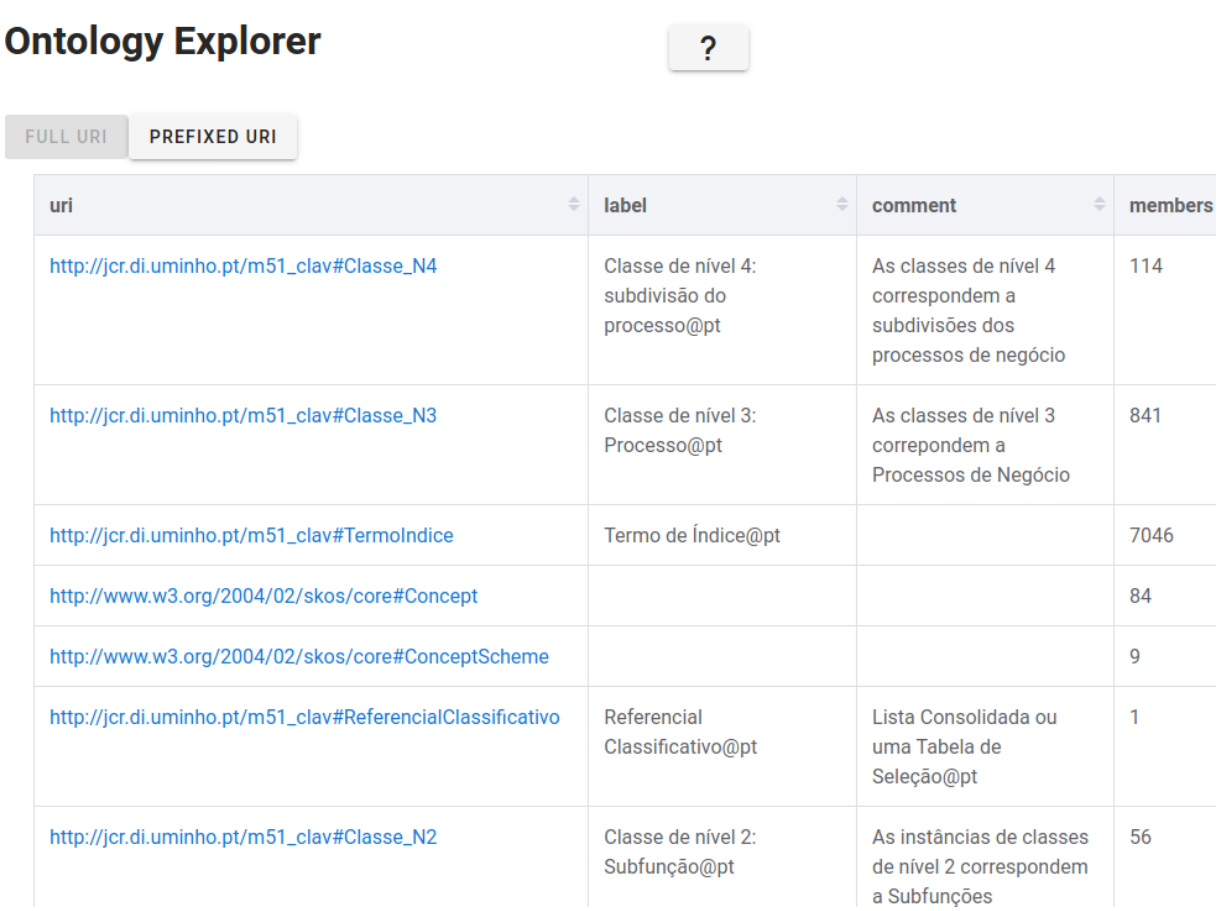


Figura 36: Tabela de resultados da *query* traduzida e enviada a Neo4J

Explorador de ontologia

Finalmente as últimas três páginas que lidam com a operações sobre ontologias estão sobre **Ontology Explorer**.

Quando se entra nesta página e após alguns segundos de espera serão colocadas na tabela da página informações sobre as diferentes classes que existem dentro da ontologia actualmente selecionada na aplicação: URI da classe (URI), etiqueta da classe (`label`), comentários associados à classe (`comment`) e número de elementos na classe (`members`).



uri	label	comment	members
http://jcr.di.uminho.pt/m51_clav#Classe_N4	Classe de nível 4: subdivisão do processo@pt	As classes de nível 4 correspondem a subdivisões dos processos de negócio	114
http://jcr.di.uminho.pt/m51_clav#Classe_N3	Classe de nível 3: Processo@pt	As classes de nível 3 correpondem a Processos de Negócio	841
http://jcr.di.uminho.pt/m51_clav#TermoIndice	Termo de Índice@pt		7046
http://www.w3.org/2004/02/skos/core#Concept			84
http://www.w3.org/2004/02/skos/core#ConceptScheme			9
http://jcr.di.uminho.pt/m51_clav#ReferencialClassificativo	Referencial Classificativo@pt	Lista Consolidada ou uma Tabela de Seleção@pt	1
http://jcr.di.uminho.pt/m51_clav#Classe_N2	Classe de nível 2: Subfunção@pt	As instâncias de classes de nível 2 correspondem a Subfunções	56

Figura 37: Explorador de ontologia, consistindo da listagem de classes dela

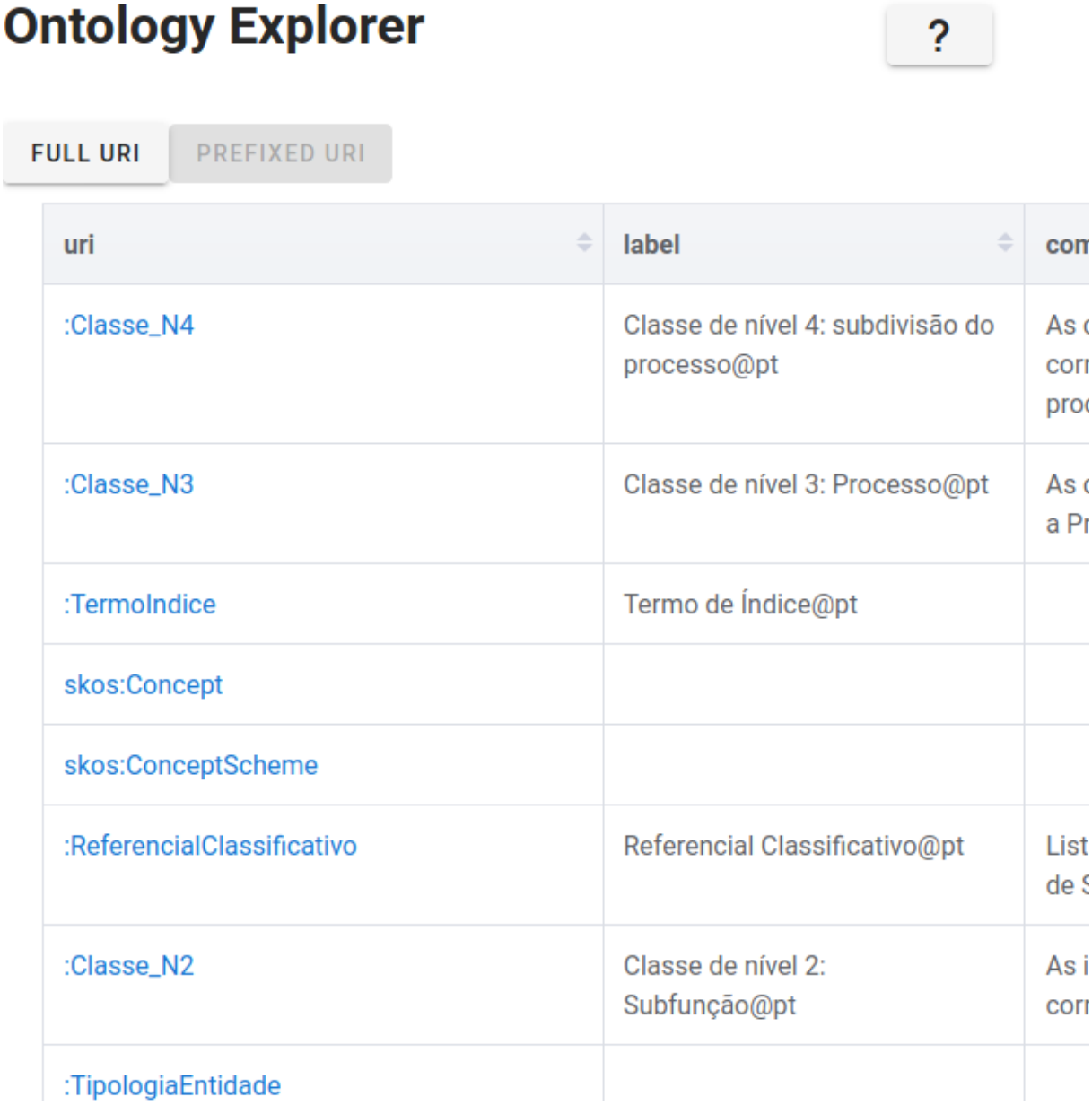


Figura 38: Explorador de ontologia com URI's completos transformados através dos prefixos da ontologia

Ao carregar no URI de uma das classes o utilizador é reencaminhado para a página especifica da classe da ontologia que foi carregada. Esta página contém todos os triplos que são propriedades de elementos da classe selecionada.

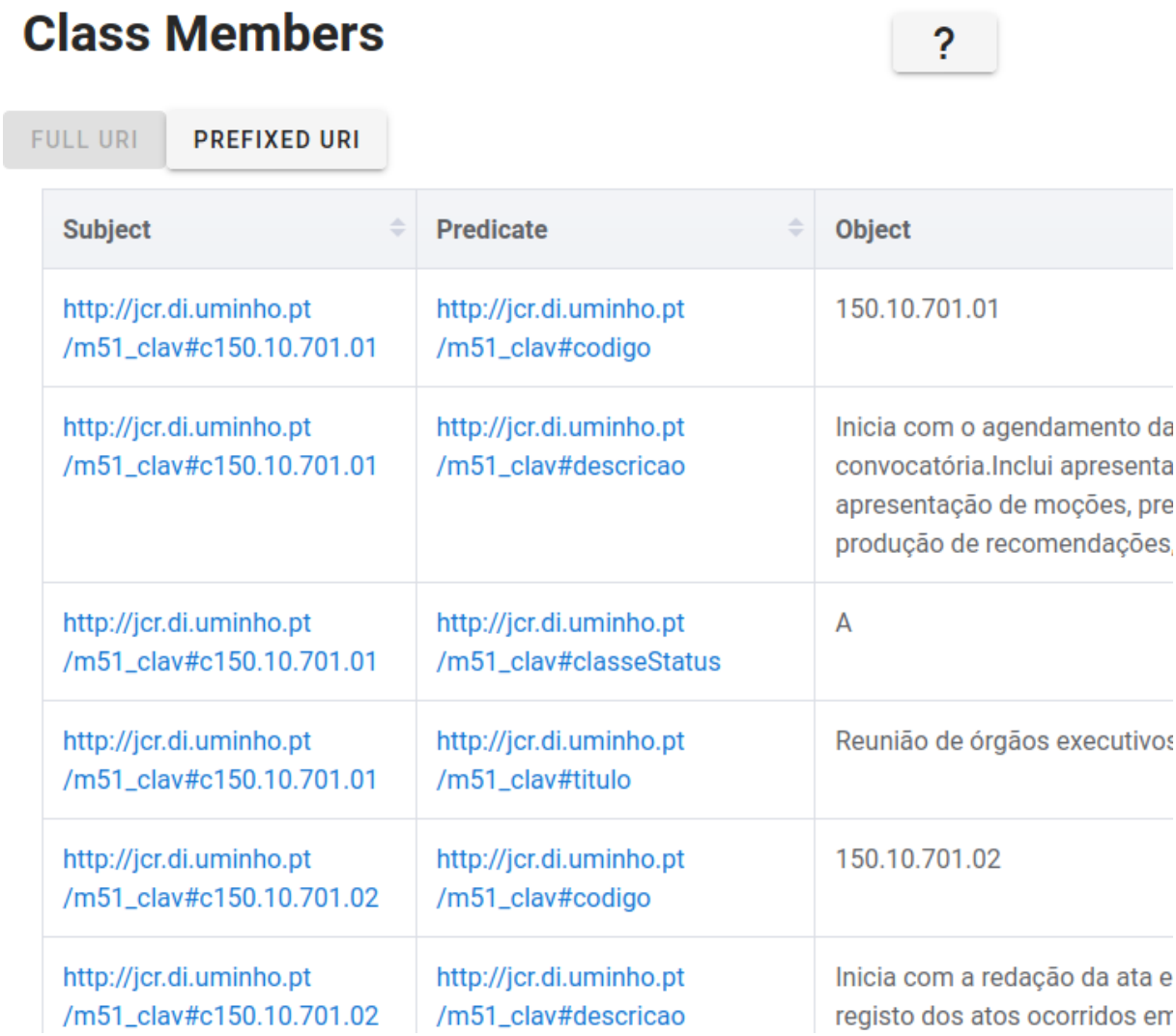


Figura 39: Explorador de classes, especifico a uma classe da ontologia actual

Finalmente ao se carregar num dos URI's dos elementos é-se reencaminhado para a página especifica a esse elemento. Nesta página pode-se observar todos os triplos que fazem parte da ontologia que têm o elemento selecionado como sujeito, predicado ou objecto, sendo que é possível carregar cada URI desses triplos para se ser reencaminhado para as páginas desses elementos.

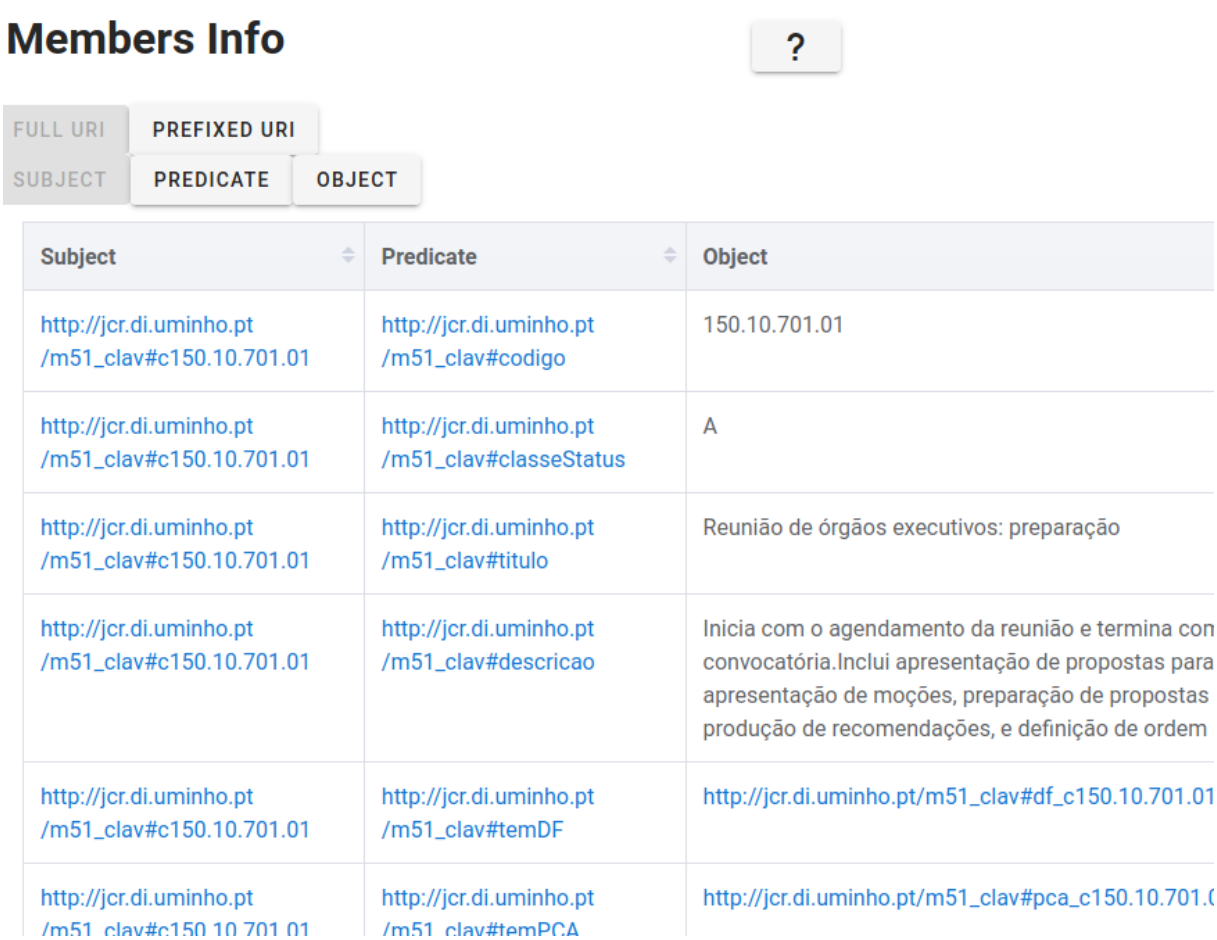


Figura 40: Explorador de elemento, mostrando os triplos da ontologia onde ele é sujeito

Note-se que como estamos numa página de um elemento da ontologia e não de uma propriedade a lista de triplos onde este elemento está como tal é vazia.

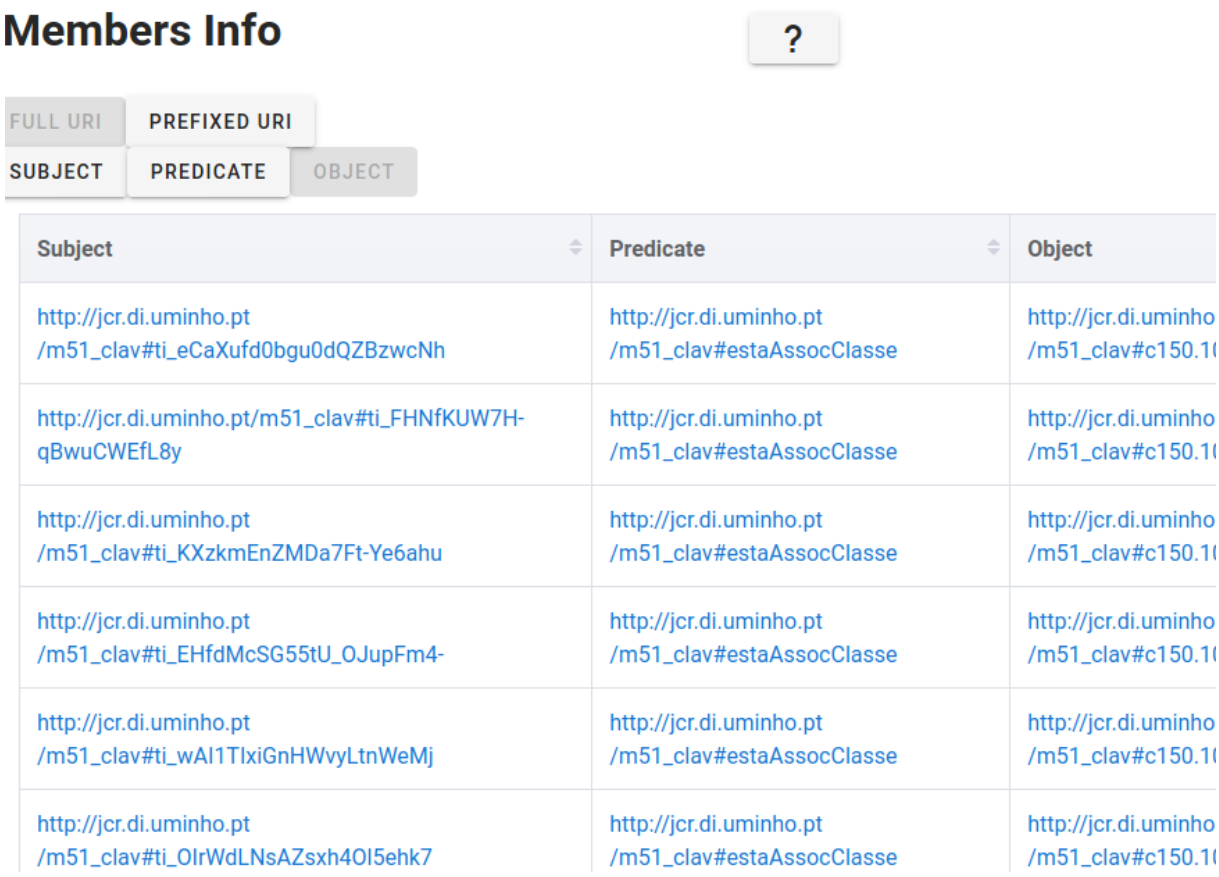


Figura 41: Explorador de elemento, mostrando os triplos da ontologia onde ele é objecto

Com isto todas as funcionalidades da aplicação foram expostas. No entanto, ainda é preciso referir alguns detalhes importantes sobre utilizadores (e permissões associadas a estes) e sobre a formatação do importe de ontologias através do Neosemantics.

7.2.3 *Utilizadores e suas permissões*

Uma nota importante sobre o sistema de *login* são os utilizadores que existem na aplicação.

Apesar dos numerosos níveis de permissões existente em Neo4J e correspondentes tipos de utilizadores (Neo4J (2020a)), a nossa aplicação só tem de distinguir três tipos de utilizadores: sem utilizador (o utilizador actual não consegue fazer qualquer *query* ao Neo4J), User (o utilizador actual tem capacidade de ler dados da base de dados mas não de administração) e Admin (o utilizador tem capacidade de criar/remover/editar bases de dados dentro do Neo4J).

As capacidades disponibilizadas pela aplicação são limitadas pelo tipo de utilizador actualmente conectado, de acordo com o seguinte diagrama:

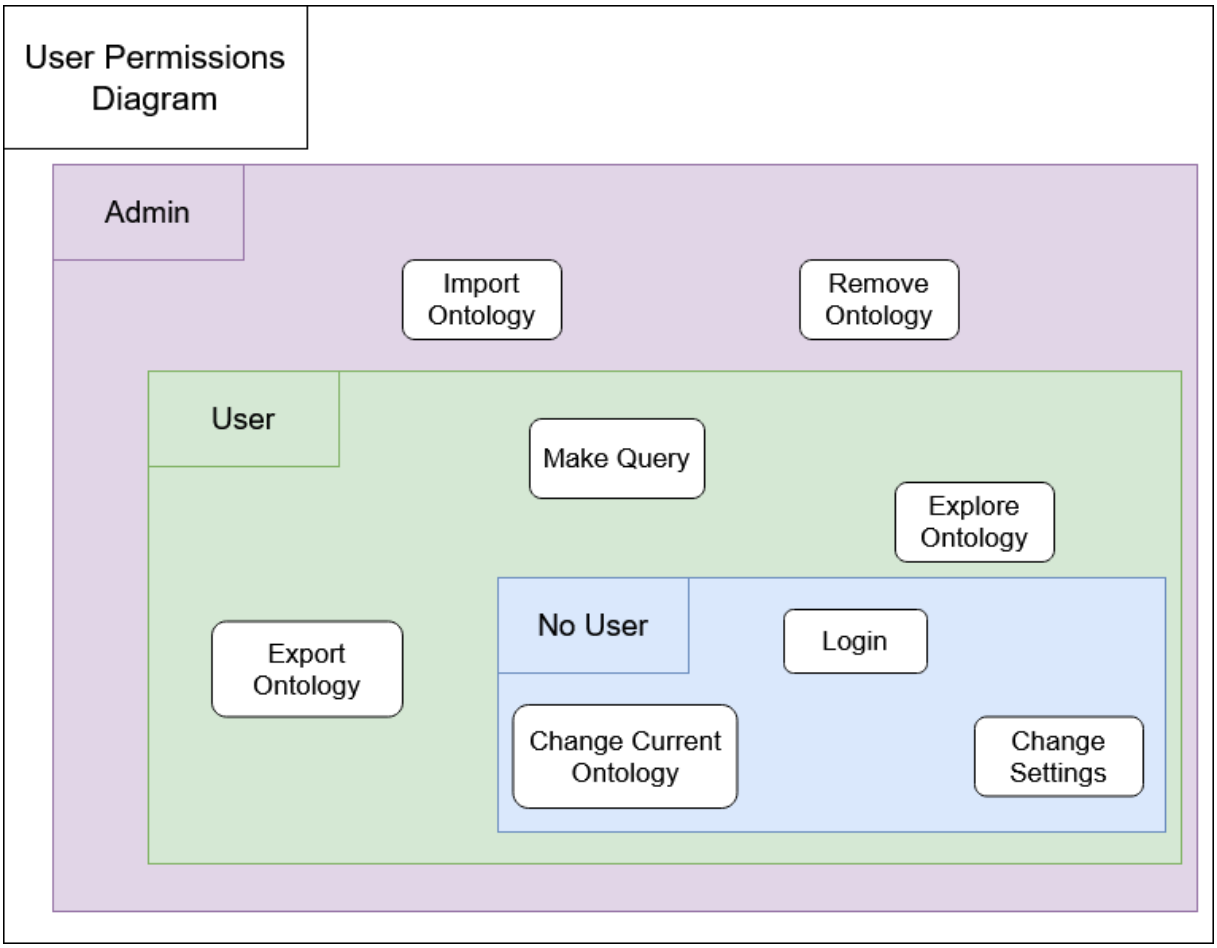


Figura 42: Diagrama de permissões do utilizador na aplicação

, onde podemos ver que:

- se não existir utilizador (`No User`), pode alterar os `settings` da aplicação, alterar a ontologia actual e fazer `login`
- se for um utilizador regular (`User`), tem todas as capacidades anteriormente mencionadas e é ainda capaz de fazer operações de consulta de ontologias, podendo exportar ontologias, fazer *queries* e explora-las.
- se for uma conta administrador do Neo4J (`Admin`) pode para além das capacidades anteriores adicionar (`import`) e remover (`remove`) ontologias da base de dados Neo4J

7.2.4 *Opções de import de dados do Neosematics*

Devido a quão importante é para a funcionalidade da gramática de tradução vamos aqui falar rapidamente das opções de importe que temos de definir para importar a ontologia em Neo4J.

O objecto de configuração usado em Neosemantics (2020b) é

```
{handleVocabUris:"KEEP",keepLangTag:true, handleRDFTypes:"LABELS_AND_NODES",
keepCustomDataTypes:true }
```

Vamos explicar cada parte desta configuração:

`handleVocabUris:"KEEP"` - esta configuração garante que os URI's são mantidos no importe em todos os dados (em vez de usar o sistema de prefixação definido pela aplicação)

`keepLangTag:true` - esta configuração garante que as *tags* de linguagem são mantidas caso existam nos dados importados

`handleRDFTypes:"LABELS_AND_NODES"` - esta configuração faz com que os dados `rdf:type` sejam tanto `labels` associadas a nodos como relações entre os nodos e outros que representam esse tipo (de modo a permitir que a tradução captura correctamente o triplo do tipo do nodo)

`keepCustomDataTypes:true` - esta configuração garante que todos os elementos que tenham um tipo de dados não nativo ao Neo4J seja, guardados como string com o sufixo do URI desse tipo de dados

Com estas configurações a nossa tradução funciona correctamente como descrito ao longo deste documento.

7.3 RESUMO

Neste capitulo falamos da parte mais prática da resolução do trabalho: a aplicação Web criada para estender as capacidades de Neo4J para lidar com ontologias.

Explicamos como funciona, tanto na sua arquitectura como na sua funcionalidade, explicando detalhadamente cada parte e detalhes que lhe são relevantes.

Falamos ainda das opções utilizadas para a importação de dados pelo Neosemantics.

8

CONCLUSÃO

Com tudo o que foi estudado e criado já apresentado temos apenas de resumir tudo o que foi feito e deixar alguns pensamentos sobre trabalho futuro a ser feito para desenvolver mais o que foi obtido.

O objectivo do trabalho é o de permitir a manipulação de ontologias dentro de Neo4J. Para podermos atravessar esta meta precisamos de responder a algumas questões, sendo que para fazermos isso temos de saber como as ontologias estão definidas e que semelhanças/diferenças as plataformas que estão em estudo têm.

Nesse sentido começamos por definir ontologias, primeiro de forma genérica e depois especificamente para o nosso trabalho, e explorar as linguagens que foram definidas para este tipo de estruturas de dados: RDF/RDFS, SKOS e OWL.

Depois exploramos as bases de dados que nos permitem guardar ontologias, bases de dados orientadas a grafos, e os motores que são usados para as guardar em estudo neste trabalho: GraphDB e Neo4J.

Para terminar a exposição das bases teóricas necessárias a este trabalho exploramos as duas linguagens de *query* cujo papel é principal nesta tese: SPARQL e CYPHER, explorando tanto a estratégia para resolver as interrogações que o utilizador faz às bases de dados como as numerosas capacidades disponibilizadas sobre forma de palavras chave/reservadas especificas das linguagens.

Com a teoria explicada podemos finalmente expor os problemas a resolver e as soluções propostas: criar uma tradução entre SPARQL e CYPHER e desenvolver uma aplicação Web capaz de expandir Neo4J para permitir que este lide com ontologias de forma mais prática, sendo que nos dois capítulos seguintes expomos como esta solução foi implementada.

Primeiro mostramos a resolução ao problema da tradução, feita através do uso de `PEG.js` para especificar uma gramática de tradução que é capaz de traduzir um conjunto limitado de *queries* SPARQL em *queries* equivalentes em CYPHER, especificando não só que *queries* podem ser traduzidas mas também uma série de exemplos utilizados para validar essas mesmas traduções.

Finalmente mostramos a solução para o segundo problema de criar uma camada tecnológica sobre Neo4J: a aplicação WEB, expondo numerosos detalhes técnicos relevantes e as funcionalidades que são oferecidas por esta (importação, exportação e remoção de ontologias, fazer *queries* e explorar a ontologias guardadas em Neo4J, ...).

Considerando todo o trabalho teórico necessário e a complexidade dos detalhes práticos para se conseguir tanto a tradução como a aplicação podemos ver que os resultados finais foram bons e estão preparados para uso prático caso seja necessário.

8.1 Trabalho futuro

Apesar do trabalho feito ainda há muito para melhorar como trabalho futuro, maioritariamente na tradução entre linguagens mas também da aplicação criada. Para a aplicação é conveniente expandir as capacidades de visualização de resultados de *queries* e da gramática, de modo a tornar mais intuitiva e simples a exploração dos dados (por exemplo apresentar um grafo para representar dados durante a exploração da ontologia de modo similar a GraphDB). Para a tradução automática existem mil e uma melhorias possíveis, desde a adição de palavras reservadas adicionais até à limpeza de como está escrita (para facilitar o seu desenvolvimento futuro).

Para além destas melhorias mais gerais destacamos ainda algumas de maior importância:

Resolver problema de múltiplos valores para a mesma propriedade não funcionarem

Para compreender o problema primeiro deve ler Neosemantics (2020d) (para compreender o processo de importe do Neosemantics e como as opções de configuração o afectam) e 7.2.4 (para compreender as opções que estamos a usar para importar as ontologias através da aplicação WEB).

Como se pode ver para colocarmos múltiplos valores associados a uma certa propriedade a funcionar correctamente é necessário explicitar todas as propriedades da ontologia que podem ter múltiplos valores na configuração inicial.

A consequência disto é que apenas um dos valores da propriedade é adicionada ao nodo (o último que é lido, para ser exacto), o que leva a uma perda de informação significante (com um exemplo disto disponível em A.1.2).

Para além disto mesmo que tivesse-mos os dados como *arrays* teríamos o problema da gramática actual assumir que todas as propriedades associadas a um nodo de um elemento são valores individuais e não *arrays*.

Assim a solução tem duas possibilidades:

- não declarar que URI's correspondem a propriedades com múltiplos valores e activar a opção para colocar *multi values* como *arrays* (que causa com que **todas** as propriedades sejam tratadas como *arrays*) e alterar a gramática para assumir que todas as propriedades são *arrays*.

- determinar durante o processo de importe (ou por selecção manualmente ou automaticamente) que propriedades podem ter múltiplos (e adicionar à configuração antes do importe) valores e encontrar uma maneira de colocar a gramática a ser capaz de determinar quando uma propriedade tem um único valor ou é um *array* deles.

Utilizar capacidade de subquery de CYPHER para expandir gramática

Como já foi mencionado anteriormente a capacidade de chamar *subqueries* dentro do Neo4J foi implementada demasiado tarde no desenvolvimento da gramática para poder ter sido usada, algo que tornou a implementação de certa palavras reservadas (nomeadamente `UNION`) e da capacidade de chamar *subqueries* em SPARQL impossível.

Assim trabalho futuro no desenvolvimento da gramática deve-se focar em parte em utilizar estas *subqueries* do CYPHER para expandir as capacidades de tradução da gramática.

Formalizar a tradução feita

Apesar de todo o trabalho feito e padrões observados a tradução que foi criada não está formalmente definida, ou seja, não tem regras definidas que permitam provar que a tradução é válida para além de por tentativa e erro em ontologias.

Isto é em contraste com SPARQL, que tem na sua documentação uma definição do comportamento de cada parte da avaliação de uma *query* (W3C (2013f)).

Assim como trabalho futuro a tradução das várias partes desta definição em suas equivalentes funcionais em CYPHER permitiria criar uma tradução formal e robusta entre as linguagens.

Criar um método de dar a conhecer à gramática a estrutura da ontologia que tem de explorar

Finalmente queremos dar atenção a um problema que houve durante todo o desenvolvimento da gramática: esta nunca tem conhecimento sobre a ontologia que está a fazer *queries* a o que leva a *queries* mais verbosas que o que é necessário, bem como alguma redução na optimização de *queries* onde não conseguimos determinar se o objecto é um URI ou um literal e sabemos o predicado que lhe está associado.

Assim encontrar uma maneira de disponibilizar dados sobre a estrutura da ontologia à qual estamos a fazer *queries* é importante para melhorar a qualidade da tradução.

BIBLIOGRAPHY

Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40, 02 2008. doi: 10.1145/1322432.1322433.

NoSQL Archive. Nosql: Your ultimate guide to the non-relational universe! https://hostingdata.co.uk/nosql-database/, 2020.

Jesús Barrasa. Importing rdf data into neo4j. https://jbarrasa.com/2016/06/07/importing-rdf-data-into-neo4j/, 2016.

Eclipse. Eclipse rdf4j. https://rdf4j.org/, 2020.

OpenJS Foundation. Express : Fast, unopinionated, minimalist web framework for node.js. https://expressjs.com/, 2020.

GraphDB. Lucene graphdb connector. http://graphdb.ontotext.com/documentation/free/lucene-graphdb-connector.html, 2020a.

GraphDB. Explain plan. http://graphdb.ontotext.com/documentation/free/explain-plan.html, 2020b.

GraphDB. Reasoning strategies. http://graphdb.ontotext.com/documentation/free/introduction-to-semantic-web.html#introduction-to-semantic-web-reasoning-strategies, 2020c.

GraphDB. Total materialization. http://graphdb.ontotext.com/documentation/free/introduction-to-semantic-web.html#total-materialization, 2020d.

OWL Working Group. Web ontology language (owl). https://www.w3.org/OWL/, 2012.

RDF Working Group. Rdf vocabulary description language 1.0: Rdf schema (rdfs). https://www.w3.org/2001/sw/wiki/RDFS, 2004.

RDF Working Group. Resource description framework (rdf). https://www.w3.org/RDF/, 2014.

Semantic Web Deployment Working Group. Simple knowledge organization system (skos). https://www.w3.org/2001/sw/wiki/SKOS, 2009.

Neo4j Labs. Chapter 3. importing rdf data. https://neo4j.com/docs/labs/nsmntx/4.0/import/, 2020.

Neo4J. Built-in roles. https://neo4j.com/docs/operations-manual/4.1/authentication-authorization/built-in-roles/, 2020a.

Neo4J. The neo4j cypher manual v4.0. https://neo4j.com/docs/cypher-manual/4.1/, 2020b.

Neo4J. 2.9. patterns. https://neo4j.com/docs/cypher-manual/4.1/syntax/patterns/, 2020c.

Neo4j. Chapter 7. execution plans. https://neo4j.com/docs/cypher-manual/4.1/execution-plans/#execution-plan-operators-summary, 2020.

Neo4J. Chapter 7. execution plans. https://neo4j.com/docs/cypher-manual/3.5/execution-plans/#execution-plan-introduction, 2020.

Neo4j. Neo4j graph platform. https://neo4j.com/, 2020.

Neo4J. Neo4j graph database. https://neo4j.com/developer/neo4j-database/, 2020a.

Neo4J. Expression. https://neo4j.com/docs/cypher-manual/4.1/syntax/expressions/#query-syntax-case, 2020b.

Neo4J. Neo4j graph platform. https://neo4j.com/developer/graph-platform/, 2020c.

Neo4J. Neo4j version 4.1.0 release notes. https://neo4j.com/release-notes/neo4j-4-1-0/, 2020d.

Neo4J. What is neo4j? https://neo4j.com/developer/graph-database/#neo4j-overview, 2020e.

Neo4J. 3.7. order by. https://neo4j.com/docs/cypher-manual/4.1/clauses/order-by/#query-order, 2020f.

Neo4J. Neo4j sandbox. https://neo4j.com/sandbox-v2/, 2020g.

Neo4J. 3.16. call {} (subquery). https://neo4j.com/docs/cypher-manual/4.1/clauses/call-subquery/, 2020h.

Neo4j. Neo4j labs. https://neo4j.com/labs/, 2020.

neo4j labs. Nsmntx. https://github.com/neo4j-labs/neosemantics, 2020.

Neosemantics. Chapter 8. exporting rdf data. https://neo4j.com/docs/labs/nsmntx/4.0/export/, 2020a.

Neosemantics. 3.3. setting the configuration of the graph. https://neo4j.com/docs/labs/nsmntx/4.0/config/#_setting_the_configuration_of_the_graph, 2020b.

Neosemantics. 2.11.2. list comprehension. https://neo4j.com/docs/cypher-manual/4.1/syntax/lists/#cypher-list-comprehension, 2020c.

Neosemantics. Chapter 4. importing rdf data - handling multivalued properties. https://neo4j.com/docs/labs/nsmntx/4.0/import/#handling-multivalued-properties, 2020d.

Neosemantics. 2.11.3. pattern comprehension. https://neo4j.com/docs/cypher-manual/4.1/syntax/lists/#cypher-pattern-comprehension, 2020e.

Ontotext. Graphdb. http://graphdb.ontotext.com/, 2020a.

Ontotext. The sail api. https://graphdb.ontotext.com/documentation/8.7/standard/architecture-components.html#the-sail-api, 2020b.

José Carlos Ramalho, Alexandra Lourenço, Pedro Penteado, and Rita Gago. Clav - classificação e avaliação da informação pública. http://clav.dglab.gov.pt/, 2020.

American Chemical Society. Cas registry and cas registry number faqs : What is cas registry? https://www.cas.org/support/documentation/chemical-substances/faqs#1, 2020.

Vue. Vue : The progressive javascript framework. https://vuejs.org/, 2020.

Vuetify. Vue material design component framework. https://vuetifyjs.com/en/, 2020.

W3C. Skos simple knowledge organization system primer. https://www.w3.org/TR/2009/NOTE-skos-primer-20090818/, 2009.

W3C. Owl 2 web ontology language primer (second edition). https://www.w3.org/TR/2012/REC-owl2-primer-20121211/, 2012.

W3C. Sparql 1.1 query language. https://www.w3.org/TR/sparql11-query/, 2013a.

W3C. Sparql grammar definition: Ask query type. https://www.w3.org/TR/rdf-sparql-query/#ask, 2013b.

W3C. Sparql grammar definition: Describe query type. https://www.w3.org/TR/rdf-sparql-query/#describe, 2013c.

W3C. Sparql grammar definition: Not exists and exists. https://www.w3.org/TR/sparql11-query/#func-filter-exists, 2013d.

W3C. Sparql grammar definition. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#sparqlGrammar, 2013e.

W3C. 18 definition of sparql. https://www.w3.org/TR/sparql11-query/#sparqlDefinition, 2013f.

W3C. Sparql grammar definition: Document conventions - namespaces. https://www.w3.org/TR/rdf-sparql-query/#docNamespaces, 2013g.

W3C. Sparql grammar definition: Order by. https://www.w3.org/TR/sparql11-query/#modOrderBy, 2013h.

W3C. Sparql grammar definition: Pn_prefix. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#rPN_PREFIX, 2013i.

W3C. Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/rdf11-primer/, 2014a.

W3C. 5. datatypes. https://www.w3.org/TR/rdf11-concepts/#section-Datatypes, 2014b.

W3C. Rdf 1.1 turtle. https://www.w3.org/TR/turtle/, 2014c.

W3C. W3c standards: What is a vocabulary? https://www.w3.org/standards/semanticweb/ontology#summary, 2015.

Futago za Ryuu. Peg.js - parser generator for javascript. https://pegjs.org/, 2020.

A

DETALHES DE RESULTADOS OBTIDOS

A.1 VALIDAÇÃO DA GRAMÁTICA

Esta secção contém detalhes de todas as queries feitas para validação e seus resultados, colocadas aqui pelo seu tamanho que torna impossível incorporar no documento.

Está dividida em três secções equivalentes ao capitulo da validação da gramática: queries genéricas, queries especificas a Pokedex e queries especificas à tabela periódica.

Neste resultados é importante relembrar as definições dos prefixos dadas no capítulo da validação da gramática, bem como o método que foi aí discutido para como a validação é feita. É importante relembrar ainda que as tabelas de resultados que estejam lado a lado têm sempre os resultados de GraphDB à esquerda e os resultados do Neo4J à direita

A.1.1 *Queries genéricas*

Query genética 1: obter classes da ontologia

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
select distinct * where {
  ?s a owl:Class .
}
order by ?s
```

Código A.1: *Query* SPARQL para obter todas as classes de uma ontologia

s
pokemon:AcquisitionMethod
pokemon:Attribute
pokemon:Description
pokemon:DmgMultiplier
pokemon:Evolution
pokemon:Game
pokemon:Location
pokemon:Move
pokemon:MoveLearning
pokemon:PokeType
pokemon:Pokemon
pokemon:PokemonAcquisition
pokemon:PokemonAttr

s
pokemon:AcquisitionMethod
pokemon:Attribute
pokemon:Description
pokemon:DmgMultiplier
pokemon:Evolution
pokemon:Game
pokemon:Location
pokemon:Move
pokemon:MoveLearning
pokemon:PokeType
pokemon:Pokemon
pokemon:PokemonAcquisition
pokemon:PokemonAttr

Tabela 10: Resultados para a *query* A.1 na ontologia *Pókedex* nos motores GraphDB e Neo4J

s
:Block
:Classification
:Element
:Group
:Molecule
:Period
:StandardState
:hasElement

s
:Block
:Classification
:Element
:Group
:Molecule
:Period
:StandardState
:hasElement

Tabela 11: Resultados para a *query* A.1 na ontologia Tabela Periódica nos motores GraphDB e Neo4J

Query genérica 2: obter certa quantidade de elementos da ontologia

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
select distinct * where {
  ?s a owl:NamedIndividual .
}
order by ?s
limit X
```

Código A.2: *Query* SPARQL para obter X elementos de uma ontologia

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking

Tabela 12: Resultados para a *query* A.2 na ontologia *Pókedex* no motor GraphDB para X=10

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking

Tabela 13: Resultados para a *query* A.2 na ontologia *Pókedex* no motor Neo4J para X=10

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba

Tabela 14: Resultados para a *query* A.2 na ontologia Tabela Periódica nos motores GraphDB e Neo4J para X=10

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking
pokemon:am_1
pokemon:am_10
pokemon:am_100
pokemon:am_101
pokemon:am_102
pokemon:am_103
pokemon:am_104
pokemon:am_105
pokemon:am_106
pokemon:am_107

Tabela 15: Resultados para a *query* A.2 na ontologia *Pókedex* no motor GraphDB para X=20

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking
pokemon:am_1
pokemon:am_10
pokemon:am_100
pokemon:am_101
pokemon:am_102
pokemon:am_103
pokemon:am_104
pokemon:am_105
pokemon:am_106
pokemon:am_107

Tabela 16: Resultados para a *query* A.2 na ontologia *Pókedex* no motor Neo4J para X=20

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba
:Be
:Bh
:Bi
:Bk
:Br
:C
:Ca
:CarbonDioxide
:Cd
:Ce

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba
:Be
:Bh
:Bi
:Bk
:Br
:C
:Ca
:CarbonDioxide
:Cd
:Ce

Tabela 17: Resultados para a *query* A.2 na ontologia Tabela Periódica nos motores GraphDB e Neo4J para X=20

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking
pokemon:am_1
pokemon:am_10
pokemon:am_100
pokemon:am_101
pokemon:am_102
pokemon:am_103
pokemon:am_104
pokemon:am_105
pokemon:am_106
pokemon:am_107
pokemon:am_108
pokemon:am_109
pokemon:am_11
pokemon:am_110
pokemon:am_111
pokemon:am_112
pokemon:am_113
pokemon:am_114
pokemon:am_115
pokemon:am_116
pokemon:am_117
pokemon:am_118
pokemon:am_119
pokemon:am_12
pokemon:am_120
pokemon:am_121
pokemon:am_122
pokemon:am_123
pokemon:am_124
pokemon:am_125

Tabela 18: Resultados para a *query* A.2 na ontologia *Pókedex* no motor GraphDB para X=40

s
pokemon:acq_method_evolution
pokemon:acq_method_fishing_good
pokemon:acq_method_fishing_old
pokemon:acq_method_fishing_super
pokemon:acq_method_in_game_trade
pokemon:acq_method_interaction_with_entity
pokemon:acq_method_nintendo_event
pokemon:acq_method_swimming
pokemon:acq_method_trade
pokemon:acq_method_walking
pokemon:am_1
pokemon:am_10
pokemon:am_100
pokemon:am_101
pokemon:am_102
pokemon:am_103
pokemon:am_104
pokemon:am_105
pokemon:am_106
pokemon:am_107
pokemon:am_108
pokemon:am_109
pokemon:am_11
pokemon:am_110
pokemon:am_111
pokemon:am_112
pokemon:am_113
pokemon:am_114
pokemon:am_115
pokemon:am_116
pokemon:am_117
pokemon:am_118
pokemon:am_119
pokemon:am_12
pokemon:am_120
pokemon:am_121
pokemon:am_122
pokemon:am_123
pokemon:am_124
pokemon:am_125

Tabela 19: Resultados para a *query* A.2 na ontologia *Pókedex* no motor Neo4J para X=40

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba
:Be
:Bh
:Bi
:Bk
:Br
:C
:Ca
:CarbonDioxide
:Cd
:Ce
:Cf
:Cl
:Cm
:Co
:Cr
:Cs
:Cu
:Db
:Dy
:Er
:Es
:Eu
:F
:Fe
:Fm
:Fr
:Ga
:Gd
:Ge
:H

s
:Ac
:Ag
:Al
:Am
:Ar
:As
:At
:Au
:B
:Ba
:Be
:Bh
:Bi
:Bk
:Br
:C
:Ca
:CarbonDioxide
:Cd
:Ce
:Cf
:Cl
:Cm
:Co
:Cr
:Cs
:Cu
:Db
:Dy
:Er
:Es
:Eu
:F
:Fe
:Fm
:Fr
:Ga
:Gd
:Ge
:H

Tabela 20: Resultados para a *query* A.2 na ontologia Tabela Periódica nos motores GraphDB e Neo4J para X=40

Query genérica 3: obter todos os elementos de uma certa classe da ontologia

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

select distinct * where {
  ?s a X .
}
order by ?s
```

Código A.3: *Query* SPARQL para obter todos os elementos da ontologia da classe X

Resultados da *query* genérica 3:

s
pokemon:bug
pokemon:dragon
pokemon:electric
pokemon:fighting
pokemon:fire
pokemon:flying
pokemon:ghost
pokemon:grass
pokemon:ground
pokemon:ice
pokemon:normal
pokemon:poison
pokemon:psychic
pokemon:rock
pokemon:water

s
pokemon:bug
pokemon:dragon
pokemon:electric
pokemon:fighting
pokemon:fire
pokemon:flying
pokemon:ghost
pokemon:grass
pokemon:ground
pokemon:ice
pokemon:normal
pokemon:poison
pokemon:psychic
pokemon:rock
pokemon:water

Tabela 21: Resultados para a *query* A.3 na ontologia *Pókedex* nos motores GraphDB e Neo4J para X="pokemon:PokeType"

s
:period_1
:period_2
:period_3
:period_4
:period_5
:period_6
:period_7

s
:period_1
:period_2
:period_3
:period_4
:period_5
:period_6
:period_7

Tabela 22: Resultados para a *query* A.3 na ontologia Tabela Periódica nos motores GraphDB e Neo4J para X=":Period"

Query genérica 4: obter todos os triplos da ontologia associados a um certo elemento da ontologia onde ele é sujeito.

```
select * where {
  X ?p ?o .
}
order by ?p
```

Código A.4: *Query* SPARQL para obter todos os triplos associados a um elemento X da ontologia onde ele é sujeito do triplo

p	o
pokemon:AcquiredBy	pokemon:acq_method_fishing_super
pokemon:PokeAcquiredInGame	pokemon:game_red
pokemon:PokeAcquiredInLocation	pokemon:loc_cerulean_city
pokemon:PokeAcquiredInLocation	pokemon:loc_cerulean_gym
pokemon:PokeAcquiredInLocation	pokemon:loc_route_24
pokemon:PokeAcquiredInLocation	pokemon:loc_route_25
pokemon:PokeAcquiredInLocation	pokemon:loc_safari_zone
pokemon:Rarity	Common
rdf:type	pokemon:Attribute
rdf:type	pokemon:PokemonAcquisition
rdf:type	pokemon:PokemonAttr
rdf:type	owl:NamedIndividual

Tabela 23: Resultados para a *query* A.4 na ontologia *Pókedex* no motor GraphDB para X = "pokemon:am_101"

p	o
pokemon:AcquiredBy	pokemon:acq_method_fishing_super
pokemon:PokeAcquiredInGame	pokemon:game_red
pokemon:PokeAcquiredInLocation	pokemon:loc_route_25
pokemon:PokeAcquiredInLocation	pokemon:loc_cerulean_city
pokemon:PokeAcquiredInLocation	pokemon:loc_route_24
pokemon:PokeAcquiredInLocation	pokemon:loc_cerulean_gym
pokemon:PokeAcquiredInLocation	pokemon:loc_safari_zone
pokemon:Rarity	Common
rdf:type	pokemon:PokemonAttr
rdf:type	pokemon:Attribute
rdf:type	pokemon:PokemonAcquisition
rdf:type	owl:NamedIndividual
uri	:am_101

Tabela 24: Resultados para a *query* A.4 na ontologia *Pókedex* no motor Neo4J para X = "pokemon:am_101"

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

Tabela 25: Resultados para a *query* A.4 na ontologia Tabela Periódica no motor GraphDB para X=":C"

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

Tabela 26: Resultados para a *query* A.4 na ontologia Tabela Periódica no motor Neo4J para X=":C"

Algumas clarificações sobre os resultados mostrados acima:

- A ordenação dos resultados da segunda coluna é diferente por causa de apenas termos ordenado os resultados explicitamente pelo coluna `?p` na *query*, o que leva a que as diferentes avaliações de *query* feitas pelas plataformas mudem a posição dos elementos. No entanto ainda se pode comparar facilmente os resultados mostrados acima pela pequena quantidade de linhas, pelo que não faz diferença para este exemplo a falta de ordenação na segunda coluna.
- Os resultados da plataforma Neo4J têm uma linha adicional em comparação com os do GraphDB: a propriedade URI cujo valor é a identidade do elemento. A razão para isto

tem a ver com o facto que o Neosemantics, para identificar os elementos da ontologia no Neo4J, adiciona a cada nodo uma propriedade URI, o que leva a que quando retornamos dados sobre todas as propriedades de um nodo (como neste exemplo) obteremos sempre um resultado adicional em comparação com o GraphDB

- Nos resultados da ontologia da tabela periódica podemos ver que os tipos dos resultados mudaram, nomeadamente que as marcas xsd:integer e xsd:float desapareceram dos resultados do Neo4J. Isto tem a ver com como o Neosemantics lida com tipos dos triplos durante a importação: se forem tipos não nativos ele faz guarda o valor dentro da string e depois adiciona o tipo como seu prefixo, mas para tipos nativos ele faz o mesmo sem adicionar o sufixo, o que leva a que fiquem apenas strings dos valores nos resultados. É no entanto desconhecida a razão pela qual há perda da precisão float durante o importe de dados.

A.1.2 *Queries especificas à ontologia Pokedex*

Query 0: obter nomes das evoluções de *Bulbasaur* (nome inglês)/*Bulbizarre* (nome francês)

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?evName where {
  ?s a pokemon:Pokemon ;
      pokemon:Name Y ;
      pokemon:EvolvesFrom ?evolutionRel .
    ?evolutionRel pokemon:EvolvesTo ?evolveTo .
    ?evolveTo pokemon:Name ?evName .
}
order by ?evName
limit 10
```

Código A.5: *Query* SPARQL para obter da evolução de Y onde Y

Resultado 1: Y = "Bulbasaur"@en

evName
"Ivysaur"@en
"Herbizarre"@fr

null

Tabela 27: Resultados para a *query* A.5 nos motores GraphDB e Neo4J com Y = "Bulbasaur"@en

Note-se que os dois resultados dados por GraphDB correspondem a um só *Pokemon* (que tem nomes em inglês e em francês).

Resultado 2: Y = *Bulbizarre*@fr

evName
"Ivysaur"@en
"Herbizarre"@fr

evName
Ivysaur@en

Tabela 28: Resultados para a *query* A.5 nos motores GraphDB e Neo4J com Y = "Bulbizarre"@fr

Esta *query* de longe demonstra um dos problemas com o Neosemantics em lidar com múltiplos valores associados a uma propriedade: ou se colocam manualmente que propriedades podem ter múltiplos valores ou fica um único valor associado. Isto leva a que queries que lidem com múltiplos valores em SPARQL falhem pela tradução que estamos a fazer ou levem a resultados indesejados.

Infelizmente como efeito secundário não podemos utilizar os nomes de *Pokemons* para fazer queries, visto que não só temos de determinar que nome ficou na base de dados como os nomes dos *Pokemons* nos resultados só terão metade da informação do GraphDB (que tem os nomes em inglês e francês).

Note-se que devido a como o Neosemantics faz importe de dados esta restrição apenas se verifica para propriedades de elementos, não tendo o mesmo problema para múltiplas relações do mesmo tipo para vários elementos diferentes.

Query 1: obter os níveis de evolução para os primeiros dez *Pokemons* que tenham dois estágios de evolução

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?s ?level ?level2   where {
  ?s a pokemon:Pokemon ;
      pokemon:EvolvesFrom ?evolutionRel .
    ?evolutionRel pokemon:EvolutionLevel ?level ;
                  pokemon:EvolvesTo ?evol2 .
    ?evol2 pokemon:EvolvesFrom ?evolRel2 .
    ?evolRel2 pokemon:EvolutionLevel ?level2 .
}
order by ?s
limit 10
```

Código A.6: *Query* SPARQL para obter os níveis de evolução para os primeiros dez *Pokemons* que tenham dois estágios de evolução

s	level	level2
pokemon:poke_001	"16"^^xsd:integer	"32"^^xsd:integer
pokemon:poke_004	"16"^^xsd:integer	"36"^^xsd:integer
pokemon:poke_007	"16"^^xsd:integer	"36"^^xsd:integer
pokemon:poke_010	"7"^^xsd:integer	"10"^^xsd:integer
pokemon:poke_013	"7"^^xsd:integer	"10"^^xsd:integer
pokemon:poke_016	"18"^^xsd:integer	"32"^^xsd:integer
pokemon:poke_147	"30"^^xsd:integer	"55"^^xsd:integer

Tabela 29: Resultados para a *query* A.6 no motor GraphDB

s	level	level2
pokemon:poke_001	16	32
pokemon:poke_004	16	36
pokemon:poke_007	16	36
pokemon:poke_010	7	10
pokemon:poke_013	7	10
pokemon:poke_016	18	32
pokemon:poke_147	30	55

Tabela 30: Resultados para a *query* A.6 no motor Neo4J

Query 2: determinar todos os movimentos cujo tipo seja *grass* que podem ser aprendidos pelo *Pokemon* com id 001 (*Bulbasaur*) antes da primeira evolução.

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?s   where {
  pokemon:poke_001 pokemon:PokemonLearns ?pokemonMovRel ;
            pokemon:EvolvesFrom ?evolutionRel .
    ?evolutionRel pokemon:EvolutionLevel ?level .
    ?pokemonMovRel pokemon:LearnsTheMove ?pokemonMove ;
                   pokemon:MoveLearnedLevel ?pokemonMoveLevel.
    filter (?pokemonMoveLevel < ?level)
    ?s pokemon:MoveType pokemon:grass .
}
order by ?s
```

Código A.7: *Query* SPARQL para determinar todos os movimentos cujo tipo seja *grass* que podem ser aprendidos pelo *Pokemon* com id 001 antes da primeira evolução

s
pokemon:move_absorb
pokemon:move_leech_seed
pokemon:move_mega_drain
pokemon:move_petal_dance
pokemon:move_razor_leaf
pokemon:move_sleep_powder
pokemon:move_solarbeam
pokemon:move_spore
pokemon:move_stun_spore
pokemon:move_vine_whip

s
pokemon:move_absorb
pokemon:move_leech_seed
pokemon:move_mega_drain
pokemon:move_petal_dance
pokemon:move_razor_leaf
pokemon:move_sleep_powder
pokemon:move_solarbeam
pokemon:move_spore
pokemon:move_stun_spore
pokemon:move_vine_whip

Tabela 31: Resultados para a *query* A.7 nos motores GraphDB e Neo4J

Query 3: obter *Pokemons* que podem ser capturados em tanto a rota um como a rota dois do jogo *Pokemon Red Version*

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?s where {
  ?s a pokemon:Pokemon ;
      pokemon:HasPokemonAcquisition ?aquisitionList .
    ?aquisitionList pokemon:PokeAcquiredInGame pokemon:game_red ;
        pokemon:PokeAcquiredInLocation pokemon:loc_route_1,pokemon:loc_route_2 .
}
order by ?s
limit 10
```

Código A.8: *Query* SPARQL para obter *Pokemons* que podem ser capturados em tanto a rota um como a rota dois do jogo *Pokemon Red Version*

s
pokemon:poke_016
pokemon:poke_019

s
pokemon:poke_016
pokemon:poke_019

Tabela 32: Resultados para a *query* A.8 nos motores GraphDB e Neo4J

Query 4: obter *Pokemons* em que um dos seus tipo é *flying*

```
PREFIX pokemon: <http://www.chalkos.net/ontologies/2015/pokemon#>
select distinct ?s where {
  ?s a pokemon:Pokemon .
    filter Exists { ?s pokemon:HasType pokemon:flying }
}
order by ?s
limit 10
```

Código A.9: *Query* SPARQL para obter pokemons em que um dos seus tipo é *flying*

s
pokemon:poke_006
pokemon:poke_012
pokemon:poke_016
pokemon:poke_017
pokemon:poke_018
pokemon:poke_021
pokemon:poke_022
pokemon:poke_041
pokemon:poke_042
pokemon:poke_083

s
pokemon:poke_006
pokemon:poke_012
pokemon:poke_016
pokemon:poke_017
pokemon:poke_018
pokemon:poke_021
pokemon:poke_022
pokemon:poke_041
pokemon:poke_042
pokemon:poke_083

Tabela 33: Resultados para a *query* A.9 nos motores GraphDB e Neo4J

A.1.3 *Queries especificas à ontologia Tabela Periódica*

Query 1: retornar todos os elementos do 7º período da tabela periódica

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select * where{
    ?s a :Element ;
       :period :period_7 .
}
order by ?s
```

Código A.10: *Query* SPARQL para obter todos os elementos que são partes do 7º período da tabela periódica

s
:Ac
:Am
:Bh
:Bk
:Cf
:Cm
:Db
:Es
:Fm
:Fr
:Hs
:Lr
:Md
:Mt
:No
:Np
:Pa
:Pu
:Ra
:Rf
:Sg
:Th
:U
:Uub
:Uuh
:Uun
:Uuo
:Uup
:Uuq
:Uus
:Uut
:Uuu

s
:Ac
:Am
:Bh
:Bk
:Cf
:Cm
:Db
:Es
:Fm
:Fr
:Hs
:Lr
:Md
:Mt
:No
:Np
:Pa
:Pu
:Ra
:Rf
:Sg
:Th
:U
:Uub
:Uuh
:Uun
:Uuo
:Uup
:Uuq
:Uus
:Uut
:Uuu

Tabela 34: Resultados para a *query* A.10 nos motores GraphDB e Neo4J

Query especifica 2: obter todos os elementos cujos pesos atómicos estejam entre cinco e vinte

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select ?s where{
    ?s a :Element ;
       :atomicNumber ?number .
    FILTER ( (?number > 5) && (?number < 20) )
}
order by ?s
```

Código A.11: *Query* SPARQL para obter todos os elementos cujo peso atómico é superior a cinco e inferior a vinte

s	s
:Al	:Al
:Ar	:Ar
:C	:C
:Cl	:Cl
:F	:F
:K	:K
:Mg	:Mg
:N	:N
:Na	:Na
:Ne	:Ne
:O	:O
:P	:P
:S	:S
:Si	:Si

Tabela 35: Resultados para a *query* A.11 nos motores GraphDB e Neo4J

Query especifica 3: obter todos os elementos metálicos cujo estado sobre condições normais é gasoso.

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select ?s where{
    ?s a :Element ;
       :classification :Metallic ;
       :standardState :liquid .
}
order by ?s
```

Código A.12: *Query* SPARQL para obter todos os elementos metálicos cujo estado sobre condições normais é gasoso

s
:HG
:Uub

s
:HG
:Uub

Tabela 36: Resultados para a *query* A.12 nos motores GraphDB e Neo4J

Query especifica 4: obter todos os elementos que tenham o mesmo grupo do Carbono

```
PREFIX : <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
select ?s where{
    :C :group ?group .
    ?s a :Element ;
       :group ?group .
}
order by ?s
```

Código A.13: *Query* SPARQL para obter todos os elementos que partilham grupo com o carbono (elemento ":C")

s
:C
:Ge
:Pb
:Si
:Sn
:Uuq

s
:C
:Ge
:Pb
:Si
:Sn
:Uuq

Tabela 37: Resultados para a *query* A.13 nos motores GraphDB e Neo4J