UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Thiago Sant' Helena da Silva

SWITCH: EXECUTANDO SPARQL SOBRE NEO4J

Florianópolis, Santa Catarina – Brasil
2022

Thiago Sant' Helena da Silva

SWITCH: EXECUTANDO SPARQL SOBRE NEO4J

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador(a): Profº Ronaldo dos Santos Mello, Dr.

Florianópolis, Santa Catarina – Brasil
2022

Notas legais:

Não há garantia para qualquer parte do software documentado. Os autores tomaram cuidado na preparação desta tese, mas não fazem nenhuma garantia expressa ou implícita de qualquer tipo e não assumem qualquer responsabilidade por erros ou omissões. Não se assume qualquer responsabilidade por danos incidentais ou consequentes em conexão ou decorrentes do uso das informações ou programas aqui contidos.

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 23:21h do dia 4 de agosto de 2022.

Thiago Sant' Helena da Silva
Switch: executando SPARQL sobre Neo4j / Thiago Sant' Helena da Silva; Orientador(a), Profº Ronaldo dos Santos Mello, Dr. – Florianópolis, Santa Catarina – Brasil, .
148 p.

Trabalho de Conclusão de Curso – Universidade Federal de Santa Catarina, INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico, Curso de Graduação em Ciências da Computação.

Inclui referências

1. SPARQL, 2. Cypher, 3. Neo4j, 4. Triplestore, I. Profº Ronaldo dos Santos Mello, Dr. II. Curso de Graduação em Ciências da Computação III. Switch: executando SPARQL sobre Neo4j

CDU 02:141:005.7

Thiago Sant' Helena da Silva

SWITCH: EXECUTANDO SPARQL SOBRE NEO4J

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Grau de Bacharel em Ciências da Computação, e foi aprovado em sua forma final pelo Curso de Graduação em Ciências da Computação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, .

Profº Jean Everson Martina, Dr.
Coordenador(a) do Curso de Graduação
em Ciências da Computação

Banca Examinadora:

Profº Ronaldo dos Santos Mello, Dr.
Orientador(a)
Universidade Federal de Santa
Catarina – UFSC

Profa. Jerusa Marchi, Dra.
Universidade Federal de Santa Catarina –
UFSC

Profa. Carina Friedrich Dorneles, Dra.
Universidade Federal de Santa Catarina –
UFSC

RESUMO

O armazenamento de dados é um tópico antigo e crucial para o desenvolvimento humano. Os estudos ao longo do tempo sobre diferentes formas de se armazenar, processar e relacionar informação foram importantes motores dos avanços tecnológicos de nossa sociedade. Este projeto trás o desenvolvimento da Switch, ferramenta que visa facilitar a construção de bancos de dados semânticos (*triplestores*) que implementem consultas utilizando SPARQL e armazenem dados em Neo4j. Para isso, são utilizados conceitos de processamento de linguagens e geração de código através de ações semânticas. Além do desenvolvimento da ferramenta, esse trabalho também trás uma revisão bibliográfica a respeito do Estado da Arte do uso e desenvolvimento de *triplestores*. A ferramenta foi desenvolvida com sucesso e foi capaz de traduzir diversas formas de consulta. No entanto, existem melhorias a serem feitas citadas como trabalhos futuros.

Palavras-chaves: SPARQL. Cypher. Neo4j. Triplestore.

ABSTRACT

Data storage is an old and crucial topic for the development of humanity. Studies over time on different ways of storing, processing, and relating information were important drivers of technological advances in our society. This project brings the development of Switch, a tool that aims to facilitate the construction of semantic databases (triplestores) that query using SPARQL and store data in Neo4j. To this do so, concepts of language processing and code generation are used through semantic actions. In addition to the development of the tool, this work also brings a literature review on the State of the Art in the use and development of triplestores. The tool was successfully developed and was able to translate various forms of consultation. However, there are improvements to be made cited as future work.

Keywords: SPARQL. Cypher. Neo4j. Triplestore.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

WWW	*World Wide Web*
W3	*World Wide Web Foundation*
W3C	*World Wide Web Consortium*
RDF	*Resource Description Framework*
OWL	*Ontology Web Language*
SKOS	*Simple Knowledge Organization System*
TCC	Trabalho de Conclusão de Curso
LPG	Labeled Property Graphs
URI	Uniform Resource Identifier
DNS	Domain Name System
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A partir da proposição da World Wide Web (WWW) no início dos anos 90, por Tim Berners-Lee, dados começaram a ser transmitidos através de continentes, criando uma rede quase unificada de informação. A estrutura e protocolos propostos trouxeram ao mundo um novo paradigma de geração e compartilhamento de conhecimento (BERNERS-LEE; CAILLIAU; GROFF, 1992).

Desde a idealização das conexões entre computadores até os dias atuais, os sistemas desenvolvidos geram informação digital em volumes cada vez maiores. O armazenamento dessa grande quantidade de dados é um desafio em si, além do problema de relacionar as informações armazenadas. Este segundo problema motivou a criação de conceitos de padronização e associação de dados conhecidos como *Linked Data* (BIZER; HEATH; BERNERS-LEE, 2009).

Baseado em princípios propostos por Berners-Lee (1998), sistemas de informação podem armazenar e publicar dados em formatos que facilitem e incentivem a interconexão. Dessa forma, bases de dados podem ser explicitamente relacionadas com outras fontes de informação (BIZER; HEATH; BERNERS-LEE, 2009).

Nesse contexto, o padrão para descrição de dados em *Resource Description Framework* (RDF[1]), proposto por um grupo de pesquisa da W3C[2] em 1997, se populariza como forma de descrever e publicar dados online. O RDF serviu de base para a criação de diversas outras especificações utilizadas para enriquecer a descrição de informações, como a *Ontology Web Language* (OWL[3]) e a *Simple Knowledge Organization System* (SKOS[4]). Essas especificações são utilizadas para criar vocabulários (ou ontologias[5]) abertos que aplicam semântica em conjuntos de dados, dando origem a ideia de *Web Semântica* (BERNERS-LEE; LASSILA; HENDLER, 2001).

Os dados da Web Semântica, embora costumem respeitar vocabulários pré-definidos, não formam dados fáceis de se representar em tabelas. Dessa forma, a utilização da bancos de dados relacionais se torna menos interessante, e surge a ideia de sistemas de armazenamento de dados em RDF, chamados de *triplestores*. Esse paradigma traz consigo diversas características, incluindo uma linguagem de consulta própria, a SPARQL[6].

O uso de SPARQL se dá sobre dados em RDF, e a linguagem traz uma sintaxe pouco verbosa, onde diversas estruturas da álgebra relacional podem ser simplificadas

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

[2] https://www.w3c.br/

[3] https://www.w3.org/2001/sw/wiki/OWL

[4] https://www.w3.org/2001/sw/wiki/SKOS

[5] https://www.w3.org/standards/semanticweb/ontology

[6] https://www.w3.org/TR/sparql11-query/

para relações entre triplas. É possível armazenar dados em RDF de diversas formas, a medida que se defina um padrão de mapeamento entre o dado com o sistema de armazenamento utilizado. Existem mapeamentos propostos para esquemas relacionais, por Berners-lee *et al.* (1998) e esquemas de grafos no banco de dados Neo4j[7] com a extensão *Neosemantics*[8], por exemplo.

1.1 MOTIVAÇÃO

Triplestores é um tema em exploração na literatura e diversos autores sugerem estruturas a serem utilizadas para a implementação deste conceito. A exemplo disso, trabalhos como o de Santana e Santos Mello (2017) propõem combinação de diversas tecnologias na camada de armazenamento de dados. Um desafio nessas implementações é a execução de consultas SPARQL, independentemente da estrutura utilizada na camada de armazenamento.

Dessa constatação surge a motivação principal para a Switch, que é a criação de um mecanismo de tradução de consultas SPARQL para outras linguagens de consulta, visando facilitar a conexão entre a interface de consulta aos dados e a camada de armazenamento. Para este trabalho, a linguagem alvo escolhida foi Cypher[9], linguagem de consulta aberta e utilizada no banco de dados Neo4j, que é um dos principais sistemas de gerência de bancos de dados orientados a grafos. Esse mecanismo de tradução pode viabilizar bancos que implementem Cypher como uma possibilidade sólida para a camada de armazenamento de *triplestores*. No escopo deste trabalho, a linguagem de origem será um subconjunto de SPARQL definido com base em alterações na definição formal da gramática original da linguagem. Essas modificações são descritas no capítulo 4.2.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Esse trabalho tem como objetivo o desenvolvimento de um mecanismo de tradução de SPARQL para Cypher, facilitando a criação de *triplestores* baseadas em Neo4j.

1.2.2 Objetivos Específicos

Os objetivos específicos para que este trabalho atinja seu objetivo geral são:

[7] https://neo4j.com/

[8] https://neo4j.com/labs/neosemantics/

[9] https://www.w3.org/wiki/Cypher

- Definir uma representação computacional baseada em estruturas de dados convencionais para uma consulta SPARQL qualquer;
- Criar um conjunto de regras semânticas associadas às produções da gramática de SPARQL para construção da estrutura definida;
- Criar um algoritmo de tradução desta estrutura para uma consulta na linguagem Cypher;
- Validar a ferramenta criada a partir da comparação do resultado da execução das consultas de entrada e saída sobre um mesmo conjunto de dados armazenados em Neo4j e RDF.

1.3 JUSTIFICATIVA

As vantagens de se utilizar semântica no armazenamento e compartilhamento de dados são bem exploradas por Berners-lee, Lassila e Hendler (2001) e Shadbolt, Berners-Lee e Hall (2006). *Triplestores* eficientes são necessárias para se trabalhar com grandes volumes de dados, e como observado por Santana e Santos Mello (2020), existem poucas *triplestores* comerciais que utilizam bancos de dados orientados a grafos para armazenamento, por mais que as semelhanças entre *Labeled Property Graphs* (LPG) e RDF sejam expressivas (THORSTEN, 2018).

Nesse sentido, armazenar dados RDF em bancos de dados orientados a grafos é vantajoso pois permite a criação de diversos mapeamentos diferentes, como analisado por Santana e Santos Mello (2020) a depender da carga de trabalho da aplicação em termos de consultas que se desejam otimizar.

Dessa forma, a principal justificativa para a presente proposta vem do baixo volume de trabalhos que exploram a conexão entre SPARQL e Cypher (capítulo 3.1), ainda que haja procura por mecanismos de consultas com SPARQL sobre o Neo4j nos fóruns da comunidade do banco de dados. A ferramenta aqui proposta será uma das primeiras dedicadas ao banco de dados Neo4j.

1.4 METODOLOGIA

A primeira etapa do trabalho foi o estudo de *triplestores* e como eles são desenvolvidos, buscando entender os tipos de tecnologia empregados. Para isso, foram utilizados artigos científicos e outras publicações pertinentes. Nessa etapa, foi observado que existem poucos trabalhos sobre a utilização de bancos de dados orientados a grafos nesse tema. Dado a falta de produções diretamente ligadas ao tema, foram selecionadas as mais próximas aos objetivos desta proposta para serem utilizados como base para trazer contribuições ao estado da arte do tema.

A segunda etapa foi um estudo sobre a linguagem SPARQL em sua versão 1.1[10]. Esse estudo teve por objetivo mapear e entender as funcionalidades oferecidas pela linguagem visando realizar modificações na gramática para limitação de escopo do trabalho. As modificações feitas sobre a gramática são apresentadas no capítulo 4.1. Essas modificações removeram diversas capacidades da linguagem, e aqui destacamos a que consideramos mais importante: o uso de dados distribuídos. A linguagem SPARQL foi projetada de modo que pudesse acessar diversas bases de dados abertas e expostas na internet no formato de páginas de texto ou . No entanto, ao contarmos com os dados armazenados em um banco de dados, esse tipo de operação ficaria mais complexa e causaria problemas as limitações de tempo deste trabalho. Além disso, diversas funções nativas de SPARQL foram removidas, o que também causa perdas nas possibilidades de aplicação da ferramenta.

A terceira etapa foi a definição de um método para a tradução entre as linguagens, onde foi definida uma abordagem baseada na construção de uma estrutura de dados intermediária através de um conjunto de ações semânticas executadas no *parsing* de uma consulta SPARQL. Essa estrutura é posteriormente utilizada na construção da consulta Cypher. A estrutura definida também é apresentada no capítulo 4.3.

A quarta etapa foi a criação e implementação das ações semânticas que construirão a estrutura a partir de uma consulta de entrada, assim como a implementação do código que transformará tal estrutura em uma consulta em Cypher.

A quinta etapa foi a validação da ferramenta desenvolvida através de testes. Os testes foram feitos sobre uma mesma base de dados armazenada em memória (consultada diretamente com SPARQL) e em uma instância do banco de dados Neo4j (consultada com Cypher gerado a partir da mesma consulta SPARQL). Espera-se que as capacidades de consulta da linguagem sejam mantidas e que os resultados sejam equivalentes em termos de dados retornados.

1.5 ESTRUTURA DO TRABALHO

Este trabalho se divide em cinco capítulos. Este capítulo apresenta os objetivos do trabalho, assim como a metodologia e a motivação. O segundo capítulo descreve os principais conceitos envolvidos na construção da ferramenta proposta para dar base ao entendimento do processo. No terceiro capítulo são analisados os principais trabalhos relacionados, usados como base e inspiração para este. Por fim, os capítulos quatro e cinco apresentam a solução implementada e os testes executados para validação.

[10] https://www.w3.org/TR/sparql11-query/

2 FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo são apresentados os seguintes conceitos importantes para este trabalho: Web Semântica, RDF, *Triplestore* e bancos de dados orientados a grafos grafos. Além disso, são apresentadas as linguagens e as ferramentas pertinentes para a proposta e que serão utilizadas durante o desenvolvimento e testes da ferramenta criada: SPARQL, Cypher, Neo4j e Neosemantics.

2.1 WEB SEMÂNTICA

Mesmo antes da proposição da Web Semântica, já havia uma preocupação dos especialistas da época com a integração dos dados gerados. O volume de dados que viria a ser gerado pela tecnologia humana ainda não era previsível, porém entendia-se a complexidade de cruzar dados de fontes distintas como um desafio (BERNERS-LEE; LASSILA; HENDLER, 2001).

A Web Semântica, proposta por Tim Berners-Lee, é uma extensão da WWW criada pelo mesmo autor anos antes, pensada para que os dados compartilhados pelas páginas tenham formato mais amigável para máquinas. Dessa forma, agentes automatizados conseguiriam fazer interpretações sobre os dados encontrados por estes em páginas da web de modo a inferir significados ou encontrar informações solicitadas por um usuário.

O grande valor apontado pelos autores da Web Semântica é que essa otimização no compartilhamento de informação com significado mais facilmente interpretado por computadores é uma contribuição para o avanço do conhecimento humano como um todo. Esse conceito é importante para este trabalho a medida que, para se operar com dados carregados de significado, tecnologias que possibilitem o armazenamento dessa semântica sejam aprimoradas e popularizadas.

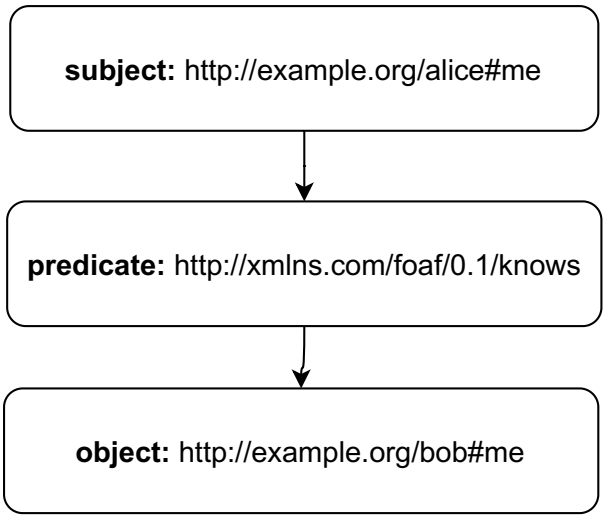
2.1.1 RDF

Junto com a XML[1], o RDF foi apontado como uma das principais tecnologias para a Web Semântica por Berners-lee, Lassila e Hendler (2001). Uma formalização dos conceitos estruturais do RDF é dada por Ladwig e Harth (2011) (tradução livre):

> (Tripla RDF, Termo RDF, Grafo RDF) Dado um conjunto de URIs $\mathcal{I}$, um conjunto de nodos vazios $\mathcal{B}$ e um conjunto de valores literais $\mathcal{L}$:
>
> $$(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$$
>
> é chamada de Tripla RDF. Nós chamamos elementos de $\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ termos RDF. Conjuntos de Triplas RDF são chamados Grafos RDF.

[1] https://www.w3.org/TR/REC-xml/

Figura 1 – Exemplo de tripla RDF



Fonte: produção própria

Um exemplo de tripla RDF é mostrado na Figura 1. Um *Uniform Resource Identifier* (URI) é um identificador semelhante ao conhecido *Universal Resource Locator* (URL), porém com uma codificação mais ampla, possibilitando o uso de caracteres especiais. Esse identificador normalmente é atrelado a um endereço DNS, e por consequência tende a ser único na *Web*. Dessa forma, evita-se ambiguidade entre entidades, por mais que se utilize um mesmo nome para se identificar uma entidade específica dentro de um sistema.

O formato RDF herda o sistema de tipagem do *Extensible Markup Language* (XML), onde estão definidas os tipos primitivos entendidos pelo padrão como valores literais possíveis. Existem algumas formas de serialização de dados RDF, como o próprio XML e o formato *Turtle*[2].

2.1.2 SPARQL

SPARQL é uma sigla recursiva para *SPARQL Protocol and RDF Query Language*[3]. É uma linguagem declarativa criada especificamente para executar consultas em dados no formato RDF. De maneira semelhante ao SQL para dados relacionais, a linguagem oferece recursos para consulta, inserção, remoção e atualização para dados no formato RDF em páginas web ou *triplestores*, padronizada pela W3C.[4].

Tomando como exemplo o conjunto de dados no Código 1 e uma consulta SPARQL como a do Código 2 sobre ele, alguns aspectos fundamentais da linguagem SPARQL ficam visíveis:

[2] https://www.w3.org/TR/turtle/

[3] https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

[4] https://www.w3.org/TR/sparql11-overview/

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3           xmlns:foaf="http://xmlns.com/foaf/0.1/">
4
5    <foaf:Person rdf:about="http://example.org/alice#me">
6      <foaf:name>Alice</foaf:name>
7      <foaf:mbox rdf:resource="mailto:alice@example.org"/>
8      <foaf:knows>
9        <rdf:Description rdf:about="http://example.org/bob#me">
10         <foaf:knows rdf:resource="http://example.org/alice#me"/>
11         <foaf:name>Bob</foaf:name>
12       </rdf:Description>
13     </foaf:knows>
14
15     <foaf:knows>
16       <rdf:Description rdf:about="http://example.org/charlie#me">
17         <foaf:knows rdf:resource="http://example.org/alice#me"/>
18         <foaf:name>Charlie</foaf:name>
19       </rdf:Description>
20     </foaf:knows>
21
22     <foaf:knows>
23       <rdf:Description rdf:about="http://example.org/snoopy">
24         <foaf:name xml:lang="en">Snoopy</foaf:name>
25       </rdf:Description>
26     </foaf:knows>
27
28   </foaf:Person>
29
30 </rdf:RDF>
```

Código 1 – Exemplo de dados RDF

Fonte: Adaptado de https://www.w3.org/TR/sparql11-overview/

- Cláusula `PREFIX`

  Pelo fato de que os elementos são identificados por URIs, que por vezes podem se tornar longas, a definição de um prefixo dentro de uma consulta é semelhante a definição de *namespaces* dentro de XML. A declaração da linha 1 faz com que a referência a `foaf:name` da linha 4 no Código 2 seja equivalente a `http://xmlns.com/foaf/0.1/name`.

- Cláusula `SELECT`

  Assim como na linguagem SQL, a saída de uma consulta SPARQL é composta por uma lista de tuplas. A cláusula `SELECT` é usada para definir a forma dessa tupla através da referência para as variáveis utilizadas na especificação seguinte. As variáveis são denotadas por palavras com o prefixo "?", como pode ser observado

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name (COUNT(?friend) AS ?count)
3 WHERE {
4     ?person foaf:name ?name .
5     ?person foaf:knows ?friend .
6 } GROUP BY ?person ?name
```

Código 2 – Exemplo de consulta SPARQL

Fonte: https://www.w3.org/TR/sparql11-overview/

na linha 2 do Código 2. Também é importante notar a possibilidade de se chamar funções especificadas pela linguagem para modificação do resultado, como a chamada para `COUNT` na linha 2 para criação de uma coluna chamada `?count` nas tuplas de saída a partir da agregação dos dados resultantes da pesquisa.

- Cláusula `WHERE` e padrões de tripla

  A cláusula `WHERE` é uma das partes fundamentais de uma consulta SPARQL. As especificações de padrões de triplas são feitas por meio da especificação de estruturas possíveis aceitas. O resultado da consulta é definido pela combinação de elementos dos dados que se relacionem da maneira especificada pelos padrões de tripla dentro da cláusula `WHERE`.

  Por exemplo, o padrão especificado nas linhas 4 e 5 do Código 2 define que para todo elemento `?person`, que tenha uma propriedade do tipo `foaf:name` com um valor `?name` qualquer e uma propriedade do tipo `foaf:knows` com um valor qualquer `?friend`, satisfaz a condição da consulta. O nome escolhido para as variáveis nessa consulta é de grande ajuda para interpretação dela. No entanto, as propriedades utilizadas são importadas de um vocabulário específico (`foaf`), e nesse vocabulário há descrições de o que significa relacionar dois elementos com essa propriedade. Dessa forma, há uma documentação completa para o significado de uma relação, por mais que a pessoa criadora da base de dados em questão não tenha criado esta explicitamente, ou seja, esteja apenas reutilizado.

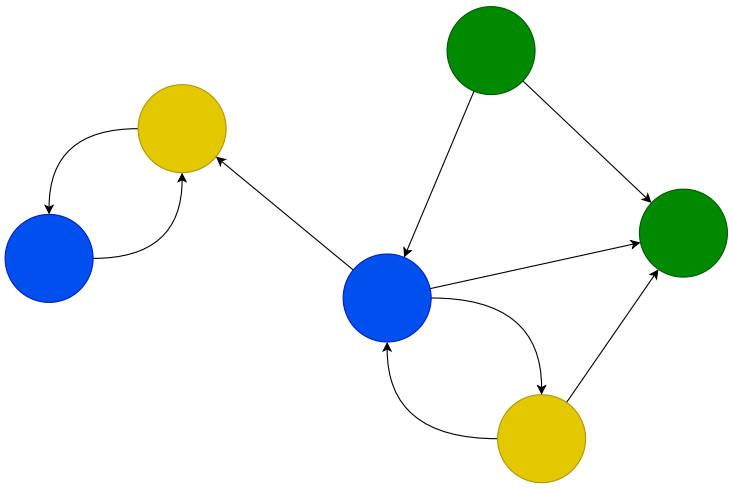
- Modificador de resultados

  SPARQL também dispõe de recursos para modificação de resultados, utilizando as mesmas cláusulas que a linguagem SQL: `LIMIT`, `OFFSET`, `GROUP BY` e `HAVING`, exemplificados pela cláusula `GROUP BY` na linha 6 do Código 2.

2.1.3 Triplestore

Triplestores ou *RDF Stores* são sistemas de armazenamento de triplas RDF. Implementações de tais sistemas são feitas com base em outros sistemas de armaze-

Figura 2 – Estrutura de um grafo



Fonte: produção própria

namento de dados, como bancos de dados relacionais ou de documentos, ou criados especificamente para o padrão.

Um exemplo de implementação de *triplestore* utilizado como referência deste trabalho (ver capítulo 3.5) é o WA-RDF. Ele foi projetado e desenvolvido por Santana e Santos Mello (2017) como um sistema de armazenamento de triplas categorizado como *polystore*, onde os dados são armazenados em múltiplos bancos de dados com objetivo de otimizar determinados tipos de buscas.

Outro exemplo notável de *triplestore* é o *Openlink Virtuoso*[5], capaz de armazenar bilhões de triplas em suas versões mais recentes e que serve de base para projetos abertos de catalogação de informação, como o *WikiData*[6].

2.2 BANCO DE DADOS ORIENTADOS A GRAFOS

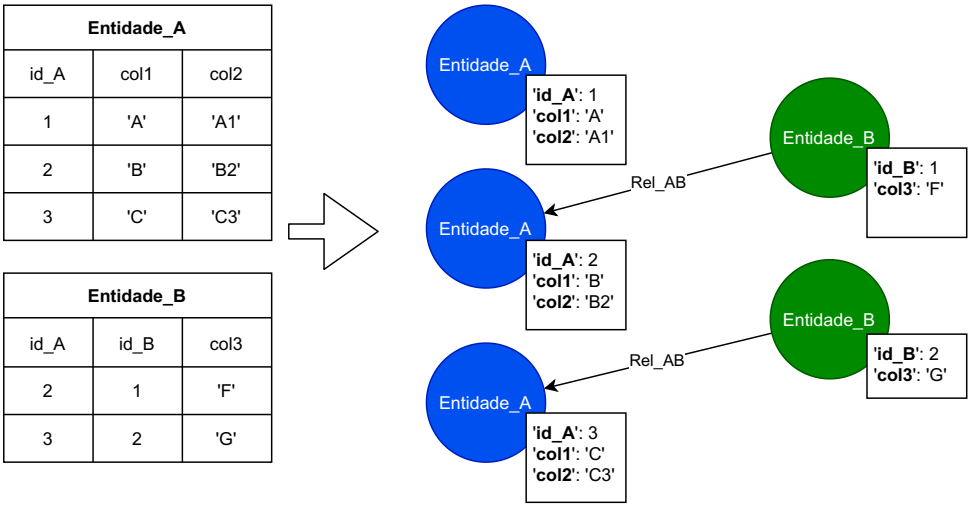
Bancos de dados orientados a grafos são, de modo geral, a implementação de estruturas de armazenamento fortemente baseadas na teoria de grafos. Dessa forma, os conceitos centrais desse tipo de banco são objetos e as relações entre eles, como demonstrado na Figura 2. São especializados para o armazenamento de entidades na forma de vértices (ou nodos), e as relações entre essas entidades são feitas através de arestas. Toda aresta tem, necessariamente, um nodo de partida e um de chegada, um tipo e uma direção.

Estabelecendo uma relação entre a estrutura de bancos de dados relacionais e os de grafos, pode-se assumir cada linha de uma tabela como um nodo, tendo as colunas como propriedades, e as chaves estrangeiras que relacionam os registros

[5] https://virtuoso.openlinksw.com/

[6] https://www.wikidata.org/wiki

Figura 3 – Transformação relacional para grafo



Fonte: produção própria

como as arestas entre os nodos. Um exemplo pode ser visto na Figura 3, onde se pode notar que em bancos de grafos não são necessárias chaves estrangeiras, mas sim arestas nomeadas. Os registros da *Entidade_B* tiveram a coluna *id_A* substituída pela aresta nomeada *Rel_AB*.

A busca através de relações entre entidades é uma operação muito eficiente em bancos de dados orientados a grafos, por conta da arestas serem armazenadas em disco e não calculadas em tempo de consulta como em bancos relacionais. Assim, esse tipo de banco é frequente usado em casos como redes sociais e ferramentas que façam considerações sobre as relações entre indivíduos, como sistemas de recomendação.

2.2.1 Neo4j

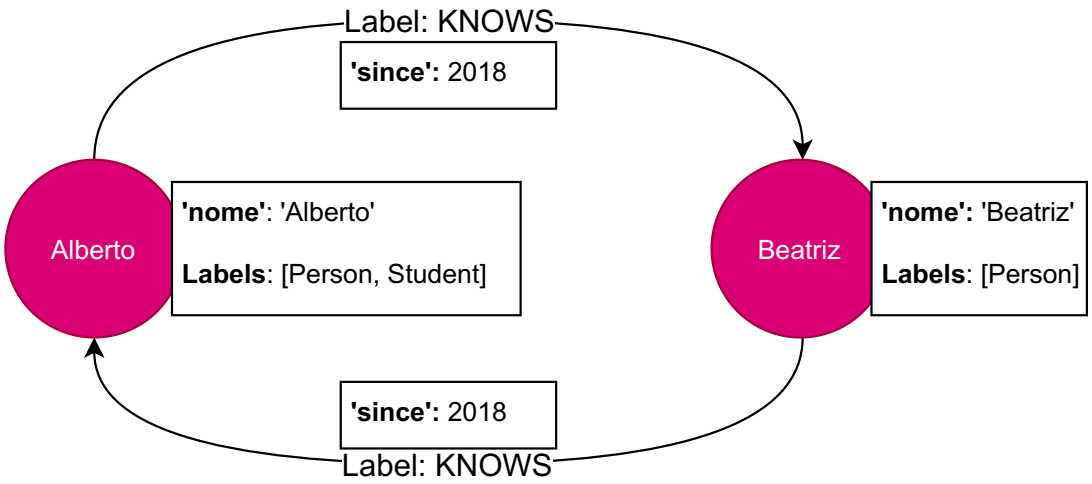
O Neo4j é um dos bancos de dados orientados a grafos mais populares atualmente. Cada elemento no banco é representado por um vértice (ou nodo) que tem relações com outros elementos através de arestas. Tanto vértices quanto arestas podem ter um conjunto de propriedades, como mostra a Figura 4.

Tanto nodos quanto arestas podem ter *labels*, que são usados para associar tipos, facilitando o processo de consulta e aumentando a consistência das estruturas armazenadas. Arestas devem ter um *label*, enquanto nodos podem ter um número qualquer de *labels*, criando a possibilidade de um mesmo nodo representar mais de um tipo, como mostrado na Figura 4 onde os nodos tem os *labels Person* e/ou *Student*.

2.2.1.1 Neosemantics

A extensão *Neosemantics* adiciona um conjunto de funções ao Neo4j para manipulação de dados em RDF, porém esta não conta com suporte para pesquisa sobre

Figura 4 – Exemplo de dados em grafos



Fonte: produção própria

os dados armazenados utilizando SPARQL, sendo esse o espaço que este trabalho busca preencher.

Essas funções serão usadas como base do desenvolvimento da ferramenta proposta, uma vez que elas já implementam um mecanismo de carregamento de dados em RDF para o esquema de grafos utilizado através de um mapeamento específico.

O mapeamento dos dados em RDF para a estrutura de grafos do Neo4j é feita com base no conjunto de regras a seguir, aplicadas a cada tripla. As regras foram traduzidas e reescritas a partir do artigo de Barrasa (2021) e descritas abaixo, com exemplo de aplicação de cada regra utilizando os dados apresentados no Código 1:

1. *Namespaces* são transformados em um nodo com a *label* `_NsPrefDef` e uma propriedade cuja chave é igual a abreviação definida e valor igual a URL base do *namespace* (Figura 5). Todos os *namespaces* são salvos em um único nodo.

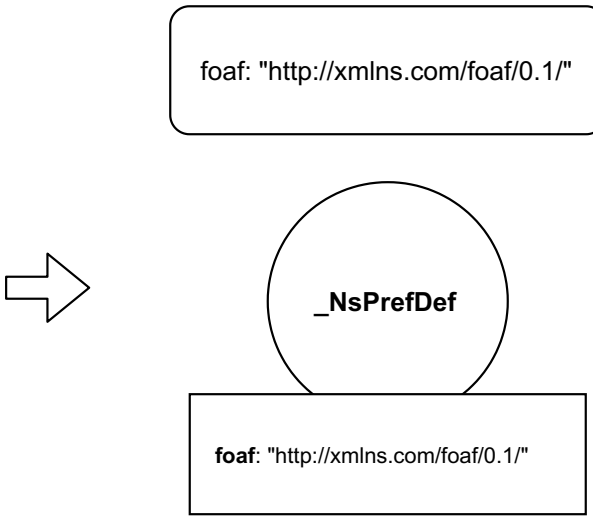
2. O *subject* da tripla é mapeado para um nodo e recebe a *label* `Resource`. Uma propriedade é associada ao nodo com o valor da URI do elemento da tripla (Figura 6). Caso já exista um nodo com a propriedade `uri` igual a URI do elemento, essa regra não se aplica.

3. Se o *object* da tripla não for um valor literal, este é mapeado para um nodo da mesma maneira que o *subject* na regra anterior (Figura 7).

4. O *predicate* da tripla é mapeado para uma propriedade do nodo *subject* com valor igual ao *object* se o *object* for um valor literal (Figura 6). Caso contrário, é mapeado para uma aresta entre os nodos criados nas regras anteriores (Figura 7).

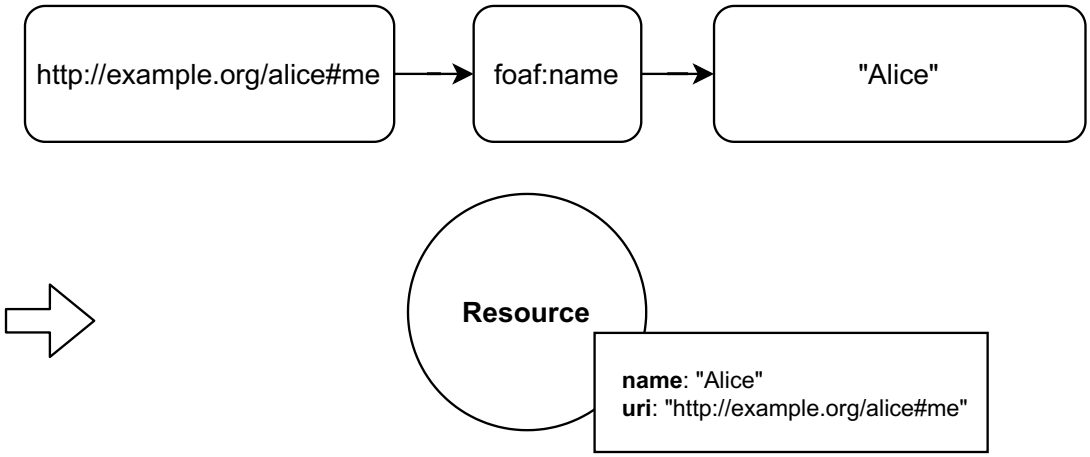
5. Triplas que contenham o predicado nativo do vocabulário do padrão RDF

Figura 5 – Exemplo de mapeamento RDF para grafo: *namespace* para nodo



Fonte: produção própria

Figura 6 – Exemplo de mapeamento RDF para grafo: *object* literal



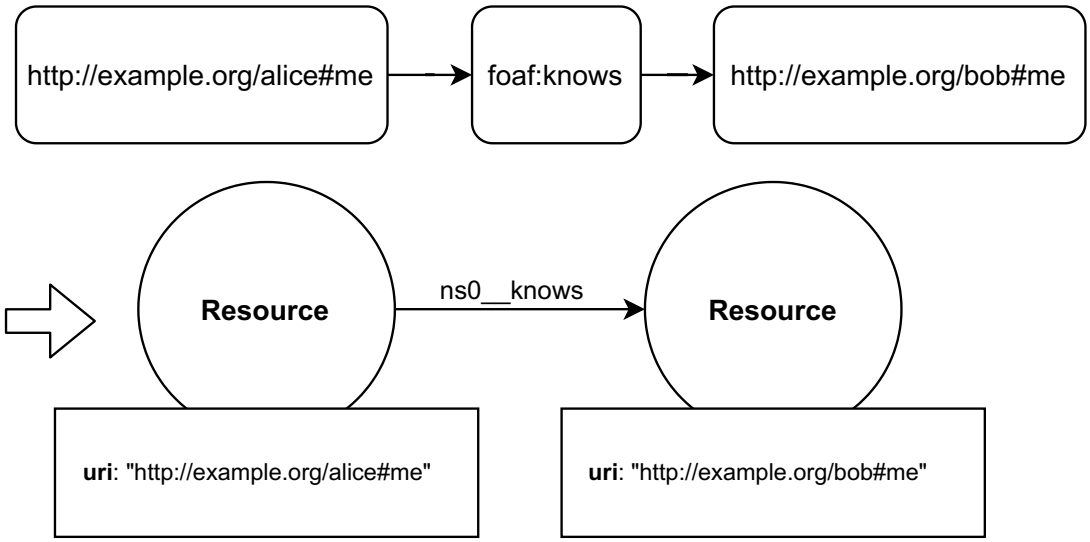
Fonte: produção própria

`rdf:type`, utilizado para denotar elementos que representam tipos, são mapeadas como um único nodo. Esse nodo recebe a *label* `Category` e uma propriedade `uri` com o valor da URI do *subject*.

Ao realizar o carregamento dos dados mostrados em RDF no Código 1 utilizando as funções da extensão, obtém-se um grafo como mostrado na Figura 8. As cores dos nodos são referentes as *labels* que estes recebem. Os nodos em vermelho tem apenas a label “Resource”. O nodo *Alice* recebeu duas *labels* (“Resource” e “ns0__Person”), ficando em azul escuro, e o nodo separado dos demais é o do *namespace*, com a *label* “_NsPrefDef”.

É importante observar a forma como o *namespace* `foaf` foi mapeado. Como detalhado nas regras, é criado um nodo separado para este, e nas propriedades

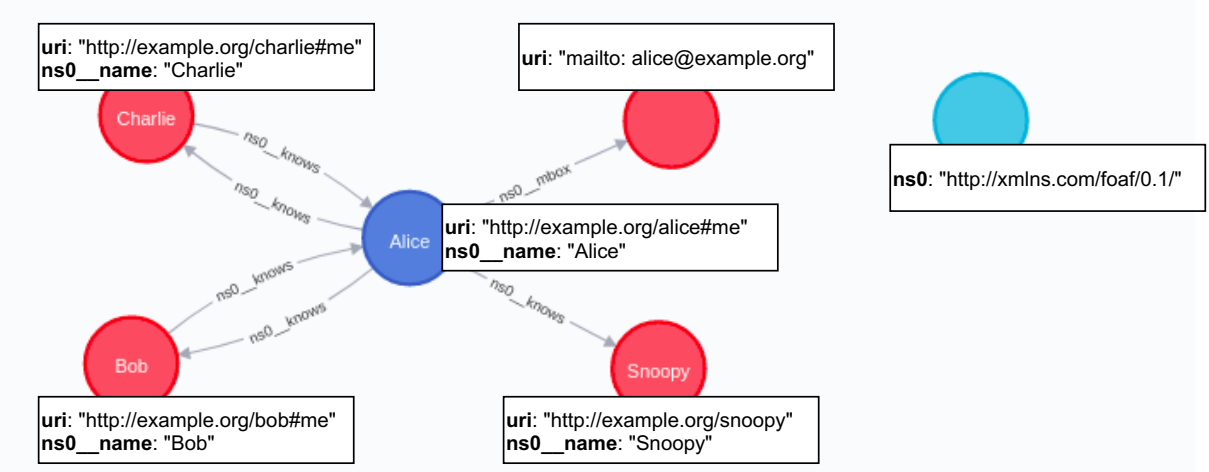
Figura 7 – Exemplo mapeamento RDF para grafo: *object* não literal



Fonte: produção própria

e *labels* do grafo onde ele é necessário, a abreviação do *namespace* é substituída por um valor dado a propriedade deste no nodo com label "_NsPrefDef" (no caso, "`ns0`") e os dois pontos entre a abreviação e o nome do predicado se torna "__". Por exemplo, `foaf:name` se tornou `ns0__name`. Essa característica da importação de dados pelo Neosemantics será relevante para a tradução das consultas pela ferramenta, uma vez que a administração dos `namespaces` em SPARQL é muito utilizada para melhorar a legibilidade em qualquer consulta, inclusive das mais simples.

Figura 8 – Carregamento dos dados exemplo no Neo4j



Fonte: produção própria, utilizando a interface gráfica do Neo4j

```
1 MATCH (person)-[:ns0__knows]->(friend)
2 WHERE person.ns0__name IS NOT NULL
3 RETURN person.ns0__name, COUNT(friend) AS count
```

Código 3 – Tradução manual da consulta nos dados para Cypher

Fonte: produção própria

2.2.1.2 Cypher

Cypher é uma linguagem declarativa, criada inicialmente para execução de consultas no banco de dados Neo4j. Dessa forma, ela é especializada para a descrição das relações dentro de um conjunto de dados estruturado baseado em grafos. Em 2015, a linguagem foi tornada um projeto de código aberto independente pela iniciativa *openCypher*[7].

A sintaxe da linguagem é de fácil leitura, contendo elementos que lembram a escrita em inglês e compartilhando palavras-chave com outras linguagens de consulta, em especial SQL, e até compartilhando algumas estruturas com a linguagem de programação Python. Isso faz com que seus criadores a descrevam como uma linguagem de fácil aprendizado.

O Código 3 exemplifica uma consulta em Cypher. Esse código é a tradução manual da consulta no Código 2 para um banco de dados Neo4j populado com os dados no Código 1 utilizando o Neosemantics. Nesse exemplo podemos observar como a cláusula `SELECT` da SPARQL é aproximadamente representada em Cypher com a cláusula `RETURN` na linha 3, incluindo a função de agregação `COUNT`, que em Cypher conta com uma operação de `GROUP BY` implícita. A cláusula `WHERE` em SPARQL precisa de uma combinação de duas cláusulas (`MATH` e `WHERE`) em Cypher.

O exemplo, no entanto, faz uma suposição forte sobre a estrutura armazenada para funcionar corretamente: trata-se o predicado `foaf:name` diretamente como uma propriedade. Porém, tendo como informação de entrada apenas consultas no Código 2, não é possível afirmar que a variável `?name` assume apenas valores literais. Se o domínio de `foaf:name` fosse outro tipo complexo, essa consulta não retornaria o esperado. Esse e outros detalhes sobre a tradução são discutidos pelos autores em Lombardot *et al.* (2019), e são abordados no capítulo 3.3.

[7] http://opencypher.org/

3 TRABALHOS RELACIONADOS

3.1 REVISÃO SISTEMÁTICA

Para buscar referências relacionadas ao tema deste trabalho, a metodologia de Revisão Sistemática proposta por Kitchenham (2004) foi utilizada. Inicialmente, foi definida a pergunta principal a ser respondida: "*Como SPARQL pode ser utilizado para consultar bancos de dados que implementem consultas com Cypher?*". Como perguntas secundárias, foram formuladas "*Como os dados em RDF podem ser convertidos para bancos de dados orientados a grafos?*" e "*Quais as limitações das ferramentas existentes?*".

A string de busca definida foi gerada pela combinação de três palavras-chave sob uma lógica booleana, aplicada ao texto completo das publicações: `SPARQL AND (Cypher OR Neo4j)`. A utilização dessa lógica nas pesquisas teve por objetivo encontrar trabalhos que relacionassem de alguma maneira SPARQL com Cypher, diretamente, ou com o Neo4j que, por ser consultado apenas com Cypher, uma relação entre SPARQL e Neo4j poderia trazer informações úteis para este trabalho.

Os principais repositórios de publicações científicas, em especial para a área da computação, foram utilizados para efetuar a pesquisa. Na Tabela 1 podem ser vistos os textos de busca utilizados. Para a biblioteca digital Springer Link, a consulta foi feita diretamente pelos parâmetros da URL.

Como critérios para inclusão, foram utilizadas restrições de idioma e da publicação necessariamente relacionar SPARQL com Cypher ou Neo4j. Relacionar as duas linguagens seria o caso ideal, no entanto o Neo4j é consultado apenas com Cypher. Logo, alguma tentativa de consultá-lo usando SPARQL provavelmente remeteria a uma tradução ou processo semelhante.

Como critérios de exclusão, foram definidas a limitação de acesso e excluídos os capítulos de livros, uma vez que a finalidade didática desse tipo de material dificilmente

Repositório	**Texto de busca**
DBLP	SPARQL Cypher|Neo4j
IEEE Xplore	("All Metadata":"SPARQL" AND ("All Metadata":"Cypher"OR "All Metadata":"Neo4j"))
Springer Link	search?facet-content-type="Article" &query=SPARQL+AND+%28Cypher+OR+Neo4j%29 &facet-discipline="Computer+Science"
ACM	[[Keywords: "sparql"] OR [Keywords: "neo4j"] OR [Keywords: "cypher"]] AND [Full Text: "sparql"] AND [[Full Text: "neo4j"] OR [Full Text: "cypher"]]

Tabela 1 – Relação de biblioteca digital e texto de busca

Repositório	Resultados	Leitura do título	Leitura do resumo	Leitura completa	Final
DBLP	1	1	1	1	1
IEEE Xplore	5	3	3	0	0
Springer Link	46	21	1	0	0
ACM Library	20	4	2	2	2

Tabela 2 – Resultado dos filtros sobre as buscas

discorreria sobre conexões entre as linguagens.

Em suma, os critérios usados foram:

Inclusão:

- Publicação em Inglês ou Português;
- Que faça relação direta entre as duas linguagens ou entre SPARQL e Neo4j.

Exclusão:

- Não encontrado para *download*;
- Ser capítulo de livro.

Sobre as publicações encontradas foi ainda aplicada uma filtragem através da leitura do título, resumo e, finalmente, pela leitura completa do material. Os resultados das buscas, em termos de número de resultados, podem ser vistos na Tabela 2, na coluna Resultados. Pela leitura dos títulos, muitas publicações já foram eliminadas, em especial no repositório da Springer, onde muitos artigos não eram a respeito de bancos de dados, mesmo citando SPARQL e Neo4j.

Após a leitura dos resumos também percebemos que o Neo4j é utilizado como parâmetro para comparação de desempenho entre propostas de sistemas de armazenamento de dados, e esse tipo de publicação foge ao escopo deste trabalho. Antes da leitura completa, foi descartado um artigo que não tivemos acesso. Outros três foram descartados por não fazerem as relações definidas, tendo apenas utilizado as palavras-chave em exemplos ou comparações.

Os três artigos selecionados foram Thakkar *et al.* (2018), Lombardot *et al.* (2019) e Fathy *et al.* (2020), e foi adicionado manualmente a tese de doutorado Santana e Santos Mello (2017). A relação destes com o presente trabalho é discutida nas seções a seguir.

3.2 GREMLINATOR

Gremlin[1] é uma linguagem de consulta em grafos, desenvolvida pelo projeto Apache Tinker Pop[2]. A linguagem possibilita consultas de maneira imperativa (semelhante

[1] https://tinkerpop.apache.org/gremlin.html

[2] https://tinkerpop.apache.org/

a um algoritmo) ou declarativa (semelhante a Cypher) sobre diversos sistemas de armazenamento, entre eles o Neo4j.

Neste trabalho, os autores discorrem sobre o uso de SPARQL sobre Gremlin, o que possibilitaria a utilização de SPARQL sobre Neo4j, indiretamente (THAKKAR *et al.*, 2018). Apesar do processo não relacionar SPARQL com Cypher, ele possibilita o uso de SPARQL em Neo4j através da uma tradução entre SPARQL e Gremlin (linguagem utilizada pelo Apache Tinkerpop[3], um *framework* de processamento de grafos).

O processo de tradução dessa ferramenta leva em consideração os padrões de tripla da linguagem SPARQL individualmente, transformando cada uma delas em uma parte da estrutura declarativa de consulta Gremlin. A proximidade sintática entre essas duas linguagens torna a tradução relativamente simples.

Em RDF as arestas são equivalentes aos predicados, que em SPARQL podem ser consideradas variáveis durante uma consulta. Essa possibilidade de considerar um predicado qualquer que conecte dois elementos em RDF é um recurso relevante dentro de SPARQL, e é apresentada como uma limitação para o Gremlinator, onde as propriedades da consulta SPARQL devem ser totalmente especificadas. Essa limitação, de acordo com os autores, se dá pelo próprio funcionamento da linguagem Gremlin, que se baseia em arestas antes de vértices para executar as consultas.

3.3 SPARQLING NEO4J

Os autores propõem uma ferramenta nos mesmos moldes da pretendida no presente trabalho (LOMBARDOT *et al.*, 2019). A abordagem assumida por eles foi de construir um *parser* para a linguagem SPARQL utilizando o *framework* PEG.js[4] e associar as ações semânticas que gerariam a consulta equivalente em Cypher, também supondo uma base de dados RDF carregada em Neo4j utilizando o Neosemantics.

O processo de tradução sugerido no artigo é exemplificado com a consulta em SPARQL do Código 4, uma consulta simples do ponto de vista da linguagem. O retorno dessa consulta seria todos os elementos `?father` (o asterísco na cláusular `SELECT` implica em retornar todas as variáveis utilizadas) que são *object* de uma tripla com o *subject* sendo `http://www.my_ontology.com#elem` e predicado `http://www.my_ontology.com#hasFather`.

A tradução proposta pelos autores para a consulta no Código 4 é a apresentada no Código 5, refatorada para fins de análise aqui feita. As observações feitas pelos autores para esta tradução são:

1. Na linha 1, são selecionados todos os nodos onde a propriedade `uri` é igual a URI do `subject` do padrão de tripla. Essa seleção ampla faz sentido uma vez que

[3] https://tinkerpop.apache.org/

[4] https://pegjs.org

```
1 PREFIX : <http://www.my_ontology.com#>
2 SELECT * WHERE {
3     :elem :hasFather ?father
4 }
```

Código 4 – Consulta exemplo de tradução em SPARQL

Fonte: (LOMBARDOT *et al.*, 2019)

o `subject` de uma tripla é com certeza um não-literal e, portanto, na importação com o Neosemantics é transformado em nodo. O nodo com essa `uri` é nomeado de `gV1` pelo restante da consulta, sendo possível afirmar que um único nodo é selecionado com essa condicional, uma vez que o Neosemantics importa os dados tratando a `uri` como um identificador único.

2. Na linha 3, o operador `UNWIND` é utilizado sobre o resultado da concatenação de duas listas geradas a partir de operações sobre o nodo `gV1`, resultando em listas concatenadas pelo operador + da linha 8. Esse operador executa um desempacotamento sobre uma lista, transformando um único registro com $N \in \mathbb{N}$ elementos em N registros[5].

3. Nas linhas 4, 5 e 6 é efetuada uma operação equivalente a um operador "para todo" sobre o retorno da chamada da função `KEYS` sobre o nodo `gV1`. A função `KEYS` retorna uma lista de *strings* contendo o nome de todas as propriedades de um nodo. A cláusula `WHERE` da linha 5 faz um filtro pela propriedade com a chave igual ao valor da propriedade. A linha 6 expressa uma construção a ser feita sobre cada elemento da lista após o filtro, nesse caso cada elemento da lista gerada é transformada em uma tripla, com o valor da chave como primeiro elemento, `null` como segundo e a própria chave como terceiro. O valor nulo é utilizado como segundo elemento nesse caso para garantir compatibilidade com as operações feitas posteriormente. Essa parte da consulta busca extrair o valor para a variável `?father` da consulta original, em caso do domínio do predicado `:hasFather` ser um valor literal e ter sido mapeado para o Neo4j como uma propriedade.

4. Nas linhas 10, 11 e 12, é executada uma operação semelhante a anterior, no entanto, executa uma busca sobre todas as arestas que saem do nodo `gV1`, nomeadas por `gV2`, com um nodo qualquer nomeado `customVar_father`. Na linha 11 é feito um filtro pelo tipo da relação expressa por `gV2`. O operador `TYPE` extrai de uma aresta o seu tipo no formato de *string*[6]. Na linha 12, cada elemento da lista de resultados filtrados é transformado em uma tripla, com o primeiro elemento

[5] https://neo4j.com/docs/cypher-manual/current/clauses/unwind/

[6] https://neo4j.com/docs/cypher-manual/4.3/functions/scalar/#functions-type

```
1  MATCH (gV1) WHERE gV1.uri ="http://www.my_ontology.com#elem"
2
3  UNWIND [
4              key in KEYS(gV1)
5              WHERE key = "http://www.my_ontology.com#hasFather" |
6              [gV1[key],null,key]
7         ]
8         +
9         [
10             (gV1)-[gV2]->(customVar_father)
11             WHERE TYPE(gV2) = "http://www.my_ontology.com#hasFather" |
12             [customVar_father.uri, customVar_father, TYPE(gV2)]
13        ] AS gV3
14
15 WITH *, gV3[0] as father, gV3[1] as customVar_father
16
17 WITH father WHERE (father IS NOT NULL)
18
19 RETURN *
```

Código 5 – Tradução exemplo do Código 4 para Cypher

Fonte: (LOMBARDOT *et al.*, 2019)

sendo a propriedade `uri` do nodo `object`, o próprio nodo `customVar_father` o segundo e a URI do predicado como o terceiro. Essa parte da consulta busca extrair o valor para a variável `?father` da consulta original em caso do domínio do predicado `:hasFather` ser valor não-literal e tiver sido mapeado par ao Neo4j como um nodo.

5. Na linha 13, um nome é associado a coluna de resultados obtidas através do `UNWIND`.

6. Na linha 15, a cláusula `WITH`[7] é utilizada para transformar as triplas da coluna de resultados em três colunas de resultados, nomeando a primeira como "*father*" e a segunda como "*customVar_father*". Essa manipulação serve apenas para melhorar a legibilidade da próxima etapa da consulta. É notável que a terceira coluna, que continha necessariamente apenas a constante `"http://www.my_-ontology.com#hasFather"`, é ignorada a partir desse momento, o que faz sentido já que não é necessária no retorno da consulta. Na linha 17 a cláusula `WITH` é utilizada novamente para remover as linhas de resultado poderiam ser iguais a `null` e na linha 19 o resultado é retornado com a cláusula `RETURN`.

As operações descritas resolvem a observação feita no capítulo 2.2.1.2, uma vez

[7] https://neo4j.com/docs/cypher-manual/4.3/clauses/with/#with-introduction

que, efetuando operações para ambos os casos não implica em falha na consulta a depender do domínio do predicado.

No entanto, os autores não descrevem a tradução de consultas que contêm mais de uma tripla na cláusula `WHERE` e como a tradução lidaria com a forma como os *namespaces* são importados pelo Neosemantics, uma vez que nos exemplos apresentados os *namespaces* estão incluídos em constantes da consulta em Cypher.

3.4 QUERYING HETEROGENEOUS PROPERTY GRAPH DATA SOURCES BASED ON A UNIFIED CONCEPTUAL VIEW

Fathy *et al.* (2020) traz uma forma de traduzir as duas linguagens através de uma álgebra intermediária chamada xR2RML, proposta por Michel *et al.* (2016). A tradução envolve um processo de três passos, envolvendo uma etapa de tradução de SPARQL para expressões em xR2RML, manipulações sobre as expressões geradas para evitar duplicações dos dados e o uso das expressões algébricas reescritas para geração da consulta em Cypher.

Os autores efetuam a tradução de um conjunto de consultas sobre duas populares bases de dados e apresentam os resultados na forma do tempo necessário para as traduções. A tradução para uma álgebra intermediária que é posteriormente otimizada antes da geração da consulta correspondente em Cypher parece ser o gargalo desta abordagem.

3.5 A MIDDLEWARE FOR WORKLOAD-AWARE MANIPULATION OF RDF DATA STORED INTO NOSQL DATABASES

Nesta tese de Doutorado os autores demonstram os resultados de uma série de estudos e publicações passadas a respeito do desenvolvimento de um *triplestore* denominado WA-RDF, capaz de armazenar dados em múltiplos bancos de dados NoSQL (SANTANA; SANTOS MELLO, 2017). A principal motivação por trás da utilização de diferentes bancos de dados NoSQL para a implementação dos autores é tentar utilizar as operações mais eficientes que cada sistema oferece.

Os esquemas de armazenamento escolhidos pelos autores foram banco de dados de documentos[8] e bancos de dados orientados a grafos, sendo o primeiro mais otimizado para consultas em forma de estrela e o segundo para consultas em forma de corrente. Uma consulta em forma de estrela em SPARQL é quando múltiplas relações são feitas sobre um único elemento, como exemplifica o Código 6. A consulta forma uma estrela de três pontas em torno da variável `?x`, fazendo considerações sobre suas outras relações. Esse tipo de consulta para o WA-RDF seria direcionada para o banco de dados de documentos.

[8] Banco de dados baseado no armazenamento de estruturas de chave-valor aninhadas.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x WHERE {
    ?x foaf:name "John" .
    ?x foaf:knows ?y .
    ?x rdf:type foaf:Person
}
```

Código 6 – Consulta SPARQL em estrela

Fonte: produção própria

Uma consulta em forma de corrente, por outro lado, implica na navegação através das relações entre múltiplos nodos, como exemplificado no Código 7. A consulta forma uma corrente através da conexão entre os elementos pelo predicado `foaf:knows`. Esse tipo de consulta para o WA-RDF é então direcionada para o banco de dados orientado a grafos. Os autores utilizam dois bancos de dados como camada de armazenamento: MongoDB[9] e Neo4j.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x, ?n WHERE {
    ?x foaf:knows ?y .
    ?y foaf:knows ?z .
    ?z foaf:knows ?m .
    ?m foaf:knows ?n
}
```

Código 7 – Consulta SPARQL em corrente

Fonte: produção própria

Uma terceira estrutura de consulta ainda é considerada: uma composição de consultas das duas formas anteriores, chamada de consulta complexa. A estratégia para execução desse tipo de busca no WA-RDF é quebrar a consulta em estruturas de estrela e corrente, executá-las nos respectivos bancos de dados e fazer a união dos resultados.

Com base nessas estruturas, os autores criam algoritmos para controlar quais triplas RDF são armazenadas em cada banco de dados em tempo de carregamento, fazendo considerações a respeito dos tipos de consultas esperadas e estrutura dos dados.

Para realizar as traduções necessárias entre SPARQL e Cypher, no entanto, o WA-RDF conta com uma premissa muito forte: os dados foram carregados no Neo4j de uma maneira específica de acordo com o tipo de consulta que seriam feitas no

[9] https://www.mongodb.com/

banco de dados. Dessa forma, as traduções efetuadas pelo sistema são diretas e relativamente simples, uma vez que precisam executar apenas um tipo de consulta sobre dados em uma estrutura conhecida.

A importância do WA-RDF para este trabalho é a demonstração feita pelos autores de que é possível utilizar o Neo4j como camada de armazenamento de dados para construção de *triplestores*. Nesse trabalho, tentaremos abrir ainda mais essa possibilidade, apresentando um método para tradução de consultas SPARQL para Cypher sem a necessidade de que os dados armazenados precisem ter passado por um pré-processamento específico para armazenar os dados de acordo com as estruturas que ele forme.

3.6 CONSIDERAÇÕES FINAIS

Há poucas publicações que relacionem diretamente as linguagens em questão. No entanto, há trabalhos, como o de Fathy *et al.* (2020), que descrevem ferramentas capazes de executar a tradução através de um processo de três etapas bem formalizado, e o de Thakkar *et al.* (2018) que, por mais que conte com uma terceira linguagem intermediária e tenha limitações sobre os tipos de consulta, também possibilita o uso de SPARQL sobre Neo4j.

4 SWITCH: UMA FERRAMENTA PARA EXECUÇÃO DE SPARQL SOBRE NEO4J

Este trabalho de conclusão de curso tem por objetivo principal o desenvolvimento de uma ferramenta para tradução de consultas SPARQL para Cypher levando em conta a importação de dados RDF executada pela extensão Neosemantics do banco de dados Neo4j. Todo código criado está disponível no repositório deste trabalho e a versão aqui apresentada pode ser acessada pela *tag* `v0.1.1-alpha`[1]. Uma visão geral da ferramenta é apresentada na Figura 9. No decorrer deste capítulo, cada parte da figura será explicada em detalhes.

4.1 DELIMITAÇÃO DE ESCOPO

Neste trabalho optou-se por limitar a linguagem SPARQL para somente bconsultas iniciadas com a palavra `SELECT` e comandos que visam obter dados de múltiplas fontes RDF da web, uma vez que as consultas geradas pelo Switch visam ser executadas sobre um único banco de dados Neo4j. As mudanças feitas são detalhadas no capítulo 4.2.

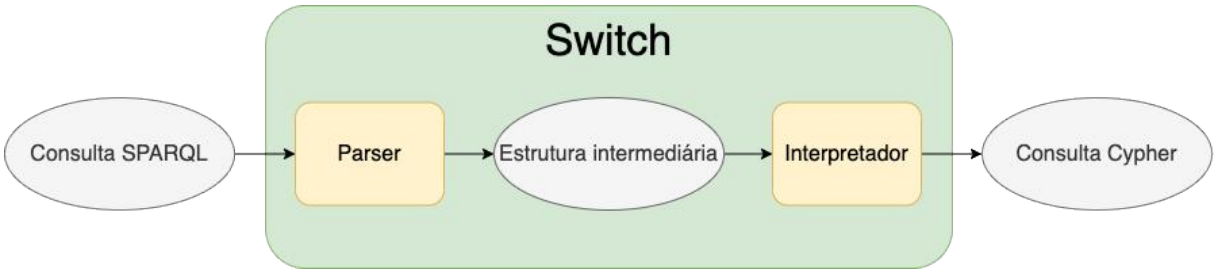
A estratégia de tradução adotada foi um processo de compilação em dois passos, onde utilizamos a entrada para criar uma estrutura intermediária que é interpretada para gerar a saída. Dessa forma, a ordenação dos padrões de tripla na cláusula `WHERE` não altera o resultado final, e a geração de código Cypher pode levar em conta a estrutura completa da consulta de entrada. A descrição da estrutura é apresentada no capítulo 4.3.

A linguagem de programação escolhida para desenvolver o projeto foi Python, utilizando a biblioteca PLY para geração do analisador léxico e sintático sobre a `SELECT SPARQL`. Como premissa para os dados RDF armazenados no Neo4j, assumimos que a carga dos dados foi feita utilizando a função `n10s.rdf.import.fetch`[2] da biblioteca Neosemantics, descrita no capítulo 2.2.1.1.

[1] https://github.com/shthiago/switch/tree/v0.1.1-alpha

[2] https://neo4j.com/labs/neosemantics/4.2/import/#actual-rdf-import

Figura 9 – Visão geral da Switch



Fonte: produção própria

4.2 LINGUAGEM SELECT SPARQL

A partir da gramática original de SPARQL[3], foram aplicadas as seguintes modificações:

1. Remover os não-terminais `ConstructQuery`, `DescribeQuery` e `AskQuery` da gramática. Dessa forma, a linguagem faz apenas leitura de dados, sem escrita, atualização ou deleção;

2. Remover o não-terminal `DatasetClause`, `GroupGraphPattern` e `ServiceGraphPattern`, pois as consultas em Cypher serão executadas sobre dados previamente carregados utilizando as funções da biblioteca NeoSemantic. Dessa forma, não há a possibilidade de carregar grafos ou serviços externos da internet;

3. Remover os não-terminais `iriOrFunction` e `FunctionCall`, substituindo a ocorrência de `iriOrFunction` como produção de `PrimaryExpression` por apenas `iri`. Esses símbolos servem para a execução de funções definidas pelo usuário e, por conta da complexidade essa possibilidade foi removida;

4. Remover as chamadas para as funções/operadores abaixo: `STRBEFORE`, `STRAFTER`, `ENCODE_FOR_URI`, `LANGMATCHES`, `SAMPLE`, `GROUP_CONCAT`, `EXISTS`, `NOT EXISTS`, `sameTerm`, `isIRI`, `isURI`, `isBLANK`, `isLITERAL`, `isNUMERIC`, `STR`, `LANG`, `DATATYPE`, `IRI`, `URI`,`BNODE`, `STRDT`, `STRLANG`, `UUID`, `STRUUID`, `BOUND`, `IF`, `MD5`, `SHA1`, `SHA256`, `SHA384`, `SHA51` e, consequentemente, remover os não-terminais `ExistsFunc` e `NotExistsFunc`. No estudo feito, essas funções não têm equivalência na linguagem Cypher e, portanto, seriam muito complexas de se implementar;

5. Remover o não-terminal `SubSelect`, por conta da complexidade;

6. Remover não-terminais não utilizados no corpo de alguma produção: `Load`, `DeleteClause`, `GraphOrDefault`, `NamedGraphClause`, `DeleteData`, `Clear`, `Update1`, `Drop`, `ConstructTriples`, `Update`, `TriplesTemplate`, `ConstructTemplate`, `Copy`, `Add`, `InsertClause`, `Quads`, `DefaultGraphClause`, `GraphRefAll`, `TriplesSameSubject`, `ArgList`, `UsingClause`, `GraphRef`, `Move`, `Modify`, `Integer`, `DeleteWhere`, `UpdateUnit`, `QuadsNotTriples`, `QuadPattern`, `Create`, `InsertData`, `QuadData`, `SourceSelector`, `PropertyList`;

7. Remover os não-terminais `InlineData`, `Bind` e `ValuesClause`, uma vez que Cypher não tem suporte para criação de variáveis de maneira explícita, tornando essa capacidade complexa para se reproduzir;

8. Normalizar as produções, substituindo as notações `*`, `+` e `?` por novas produções

[3] Disponível em https://www.w3.org/TR/sparql11-query/

```
    @dataclass
    class Query:
        graph_pattern: Optional[GraphPattern] = None
        variables: List[str] = field(default_factory=list)
        modifiers: ModifiersNode = field(default_factory=ModifiersNode)
        namespaces: List[Namespace] = field(default_factory=list)
        returning: List[SelectedVar] = field(default_factory=list)
```

Código 8 – Raiz da estrutura intermediária

```
PREFIX b:<http://www.geonames.org/ontology#>
PREFIX dct:<http://purl.org/dc/terms/>
SELECT ?country (COUNT(?state) AS ?stateCount) WHERE {
    ?country rdf:type b:Country .
    ?country dct:hasPart ?state
} GROUP BY ?country
LIMIT 10
```

Código 9 – Consulta exemplo para análise

que expressem a intenção, porém apenas com produções simples.

As modificações executadas foram feitas de forma que possam ser revertidas individualmente, a fim de possibilitar a ampliação do escopo do trabalho ou mesmo facilitar a continuação em trabalhos futuros. A gramática resultante está apresentada no Apêndice A, e a descrição dos lexemas no Apêndice B.

4.3 ESTRUTURA INTERMEDIÁRIA PARA TRADUÇÃO

Criar ações semânticas para serem associadas as produções da SELECT SPARQL e gerar diretamente a consulta equivalente em Cypher seria complexo, dado que as estruturas das linguagens são significativamente diferentes. Dessa forma, optamos por gerar uma estrutura de dados intermediária capaz de representar a consulta SPARQL de entrada, para ser processada posteriormente.

A estrutura aqui proposta se assemelha a uma árvore de análise sintática, porém reorganizada para agrupar as características relevantes da consulta de entrada. O código da raiz da estrutura (apresentado no Código 8) contém as referências para o que definimos como as partes principais de uma consulta SPARQL: padrão do grafo pesquisado(`graph_pattern`, instância de `GraphPattern`), variáveis existentes (`variables`, lista de `strings`), modificadores de resultado (`modifiers`, instância de `ModifiersNode`), *namespaces* (`namespaces`, lista de instâncias de `Namespace` e bloco de variáveis retornadas pela consultas (`returning`, lista de instâncias de `SelectedVar`).

O restante desta seção detalha cada parte da estrutura, usando como base a consulta do Código 9.

```
1    @dataclass
2    class SelectedVar:
3        value: Union[str, OrExpression]
4        alias: Optional[str] = None
```

Código 10 – `SelectedVar`

4.3.1 Namespaces

Os *namespaces* são um artifício da linguagem para simplificar a escrita de URIs em uma consulta SPARQL. A definição de um *namespace* se dá pela cláusula `PREFIX` seguida do prefixo a ser usado e o valor completo do *namespace*. No Código 9, nas linhas 1 e 2 são definidos dois *namespace*, o primeiro nomeado como `b` e o segundo como `dct`. Utilizar os prefixos nas triplas da consulta é equivalente a concatenar o valor que segue o prefixo ao valor completo do *namespace*. Por exemplo, na linha 5 do Código 9, o predicado `dct:hasPart` se refere a URI `http://purl.org/dc/terms/hasPart`.

4.3.2 Variáveis e valores de retorno

No contexto da linguagem SPARQL, uma variável é apenas um identificador, sem especificação de tipo. No entanto, as variáveis retornadas (especificadas após a cláusula `SELECT`, como no Código 9 na linha 3) podem ser um apelido para o resultado de uma expressão. No exemplo, a expressão em questão é `COUNT(?state)` e o apelido é `?stateCount`.

A representação para o par expressão e pseudônimo pode ser vista no Código 10. O campo `value` pode ser tanto uma *string* quando não houver um pseudônimo, quanto uma instância de `OrExpression` (detalhada na seção 4.3.5). O campo `alias` pode não existir quando `value` for uma *string* ou ser uma *string* quando o campo `value` for uma expressão.

4.3.3 Padrão do grafo

A estrutura `GraphPattern` (Código 11) descreve o conteúdo da cláusula `WHERE` da consulta SPARQL, exemplificado no Código 9 nas linhas 4 e 5. Na SELECT SPARQL, o conteúdo da cláusula `WHERE` pode conter:

- triplas;
- cláusulas `UNION`;
- cláusulas `FILTER`;
- cláusulas `MINUS`;
- cláusulas `OPTIONAL`.

```python
@dataclass
class Triple:
    subject: str
    predicate: str
    object: str


@dataclass
class GraphPattern:
    and_triples: List[Triple] = field(default_factory=list)
    or_blocks: List[List["GraphPattern"]] = field(default_factory=list)
    filters: List[FilterNode] = field(default_factory=list)
    minus: List["GraphPattern"] = field(default_factory=list)
    optionals: List["GraphPattern"] = field(default_factory=list)
```

Código 11 – `GraphPattern` e `Triple`

```python
GraphPattern(
    and_triples=[
        Triple("?country", "rdf:type", "b:Country"),
        Triple("?country", "dct:hasPart", "?state")
    ],
    or_blocks=[],
    filters=[],
    minus=[],
    optionals=[],
)
```

Código 12 – Exemplo de instância de `GraphPattern`

Supondo um conjunto de dados RDF como representado do Código 13, existiriam dois pares `(?country, ?state)` como resultado para a consulta:

- `(http://ontologi.es/place/BR, http://ontologi.es/place/BR-DF)`;
- `(http://ontologi.es/place/BR, http://ontologi.es/place/BR-MT)`.

As triplas da consulta do Código 9 formam uma operação de conjunção: para satisfazer a consulta, os elementos nas variáveis precisam ter a estrutura de ambas as triplas. A instância de `GraphPattern`, valor do campo `graph_pattern`, para a consulta seria como demonstrado no Código 12.

A cláusula `UNION` é utilizada para criar disjunções. O Código 14, por exemplo, retornaria o valor da propriedade `rdfs:label` de todos os elementos que os elementos `b:BR` ou `b:US` se relacionem através da propriedade `dct:hasPart`. A instância de `GraphPattern` para essa consulta seria como exemplificado no Código 15.

A cláusula `FILTER` é utilizada para filtrar os elementos utilizando uma expressão. O Código 16 apresenta a estrutura dos elementos da lista do campo `filters` de `GraphPattern`. `BuiltInFunction` é definida na seção 4.3.5.

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
   xmlns:dc1="http://purl.org/dc/terms/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
    <rdf:Description rdf:about="http://ontologi.es/place/BR">
        <dc1:title>About: Distrito Federal</dc1:title>
    </rdf:Description>
    <rdf:Description rdf:about="http://ontologi.es/place/BR-MT.rdf">
        <dc1:title>About: Mato Grosso</dc1:title>
    </rdf:Description>
    <rdf:Description rdf:about="http://ontologi.es/place/BR">
        <rdf:type
           rdf:resource="http://www.geonames.org/ontology#Country"/>
        <dc1:hasPart rdf:resource="http://ontologi.es/place/BR-DF"/>
        <dc1:hasPart rdf:resource="http://ontologi.es/place/BR-MT"/>
    </rdf:Description>
</rdf:RDF>
```

Código 13 – Exemplo de dados RDF

```
PREFIX b:<http://ontologi.es/place/>
PREFIX dct:<http://purl.org/dc/terms/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
SELECT ?stateName WHERE {
    {
        b:BR dct:hasPart ?brazilState .
        ?brazilState rdfs:label ?stateName
    } UNION {
        b:US dct:hasPart ?usaState .
        ?usaState rdfs:label ?stateName
    }
}
```

Código 14 – Consulta exemplo com `UNION`

```
GraphPattern(
    and_triples=[],
    or_blocks=[
        GraphPattern(
            and_triples=[
                Triple("b:BR", "dct:hasPart", "?brazilState"),
                Triple("?brazilState", "rdfs:label", "?stateName"),
            ],
            or_blocks=[],
            filters=[],
            minus=[],
            optionals=[],
        ),
        GraphPattern(
            and_triples=[
                Triple("b:US", "dct:hasPart", "?usaState"),
                Triple("?usaState", "rdfs:label", "?stateName"),
            ],
            or_blocks=[],
            filters=[],
            minus=[],
            optionals=[],
        ),
    ],
    filters=[],
    minus=[],
    optionals=[],
)
```

Código 15 – Exemplo de instância de `GraphPattern`

```
@dataclass
class FilterNode:
    constraint: Union[OrExpression, BuiltInFunction]
```

Código 16 – `FilterNode`

As cláusulas `MINUS` e `OPTIONAL`, por sua vez, criam elementos recursivos nos campos `minus` e `optionals`, respectivamente, de `GraphPattern`, e são usadas para adicionar ou remover dados do resultado da consulta baseado em mais padrões de grafo.

4.3.4 Modificadores de resultado

Os modificadores de resultado da linguagem SPARQL são os mesmos utilizados em diversas implementações de SQL (como PostgreSQL[4], por exemplo): `OFFSET`, `LIMIT`, `ORDER BY`, `GROUP BY` e `HAVING`. Essas cláusulas ficam no final da consulta SPARQL, como pode ser observado nas linhas 6 e 7 do Código 9. O Código 17 apresenta a estrutura definida para armazenar os modificadores de resultado processados pelo Switch em

[4] https://www.postgresql.org/docs/14/sql-select.html

```
1     @dataclass
2     class GroupCondition:
3         value: Union[str, OrExpression, BuiltInFunction]
4         alias: Optional[str] = None
5
6
7     @dataclass
8     class GroupClauseNode:
9         conditions: List[GroupCondition]
10
11
12    @dataclass
13    class OrderCondition:
14        value: Optional[Union[str, OrExpression, BuiltInFunction]] = None
15        order: str = "ASC"
16
17
18    @dataclass
19    class OrderNode:
20        conditions: List[OrderCondition]
21
22
23    @dataclass
24    class HavingClauseNode:
25        constraints: List[Union[OrExpression, BuiltInFunction]]
26
27
28    @dataclass
29    class ModifiersNode:
30        group: Optional[GroupClauseNode] = None
31        having: Optional[HavingClauseNode] = None
32        order: Optional[OrderNode] = None
33        limit: Optional[int] = None
34        offset: Optional[int] = None
```

Código 17 – Modificadores de resultado

uma consulta.

A estrutura principal (Figura 17, linha 29) contém as referências para os possíveis modificadores. O modificador `GROUP BY` é armazenado no campo `group`, em um `GroupClauseNode` (Figura 17, linha 8). O `GroupClauseNode` é composto por uma lista de condições de agrupamento (`GroupCondition`, linha 2), uma vez que múltiplas podem ser utilizadas. Cada condição de agrupamento é formada por um par de valor (`value`) e pseudônimo (`alias`). O valor pode ser uma variável (uma *string*), uma expressão (`OrExpression`) ou uma chamada de função (`BuiltInFunction`, também detalhada na seção 4.3.5), enquanto o pseudônimo é opcional e especifica um nome de variável ao qual será associado o resultado da expressão ou da chamada de função.

Filtros sobre dados agregados após um agrupamento são feitos com a palavra-chave `HAVING` e são armazenados na estrutura principal no campo `having` como um `HavingClauseNode` (Figura 17, linha 24). Um `HavingClauseNode` é composto por uma

```
1    @dataclass
2    class OrExpression:
3        base: AndExpression
4        others: List[AndExpression] = field(default_factory=list)
```

Código 18 – `OrExpression`

lista heterogênea de expressões ou chamadas de função, que semanticamente devem resultar em valores booleanos (verdadeiro ou falso).

O modificador `ORDER BY` fica no campo `order` e é um `OrderNode` (Figura 17, linha 19). Um `OrderNode` é composto por uma lista de condições de ordenamento (`OrderCondition`, linha 13) onde cada condição é dada por um valor (que pode ser uma variável, uma `OrExpression` ou uma `BuiltInFunction`) e uma forma de ordenação (`ASC` ou `DESC`).

Por fim, os modificadores `OFFSET` e `LIMIT` são os mais simples e representados como valores inteiros nos campos `limit` e `offset`, respectivamente.

4.3.5 Expressão

As expressões na gramática de SPARQL são geradas a partir do símbolo não-terminal `Expression`. As produções a partir do não-terminal `Expression`, usando a notação EBNF[5], estão apresentadas no Apêndice C para melhor visualização. A estrutura de dados definida para as expressões é uma aproximação para a árvore sintática. Cada pedaço da estrutura representa um não-terminal, com as propriedades necessárias para representá-lo.

Como o não-terminal `Expression` deriva unicamente para uma `ConditionalOrExpression`, adotamos esta como raíz das expressões da estrutura. A classe `OrExpression` no Código 18 representa as derivações de `ConditionalOrExpression`, onde haverá no mínimo um `ConditionalAndExpression` no campo `base` e um número qualquer de outros do mesmo tipo no campo `others`. Ao processar essa estrutura, para criar uma expressão em Cypher equivalente, saberemos que o usuário espera uma disjunção entre todas as instâncias de `AndExpression`. Da mesma forma, a classe `AndExpression`, no Código 19, representa o equivalente para o não-terminal `ConditionalAndExpression` porém para a conjunção de instâncias de `RelationalExpression`. Para estas classes, caso não hajam valores nas listas em `others`, entende-se que o valor da instância é o valor no campo `base`.

Como o não-terminal `ValueLogical` deriva necessariamente para `RelationalExpression`, esta foi ignorada. Para as produções de `RelationalExpression`, a classe de mesmo nome no Código 20 terá no mínimo uma instância de `AdditiveExpression` no campo `first` e opcionalmente uma tupla no campo `second`, contendo um valor do enu-

[5] https://pt.wikipedia.org/wiki/Formalismo_de_Backus-Naur_Estendido

```
    @dataclass
    class AndExpression:
        base: RelationalExpression
        others: List[RelationalExpression] = field(default_factory=list)
```

Código 19 – `AndExpression`

```
    class LogOperator(Enum):
        EQ = auto()
        NEQ = auto()
        LT = auto()
        GT = auto()
        LTE = auto()
        GTE = auto()
        IN = auto()
        NOT_IN = auto()


    @dataclass
    class RelationalExpression:
        first: AdditiveExpression
        second: Optional[Tuple[LogOperator, AdditiveExpression]] = None
```

Código 20 – `RelationalExpression`

merador `LogOperator` indicando a operação que o usuário deseja executar entre `first` e `second`, e outra instância de `AdditiveExpression`. Quando `second` for nulo, entende-se que a expressão sendo processada não contém uma operação lógica, e o valor da instância é igual ao valor da instância em `first`.

De maneira semelhante a `RelationalExpression`, as classes `AdditiveExpression` e `MultiplicativeExpression` no Código 21 representam as derivações possíveis para os não-terminais de mesmo nome. Ambas apresentam o campo `base`, contendo instâncias da próxima classe representando um não-terminal (`MultiplicativeExpression`, no caso de `AdditiveExpression` e `UnaryExpression`, no caso de `MultiplicativeExpression`), e o campo `others` contendo uma lista de tuplas com o valor do respectivo enumerador indicando a operação e instância do mesmo tipo em `base`. Caso não hajam valores na lista em `others`, o valor da instância será igual ao valor do campo `base`.

A classe `UnaryExpression` no Código 22, representando as produções do não terminal de mesmo nome, contém apenas como valor uma instância de `PrimaryExpression` no campo `value` e opcionalmente um operador unário, valor do enumerador `UnaryOperador`, no campo `op`.

A classe `PrimaryExpression`, no Código 23, contém no campo `value` um valor literal (`string`, número inteiro ou ponto flutuante), uma chamada de função nativa de SPARQL (`BuiltInFunction`) ou uma nova instância de `OrExpression`, tornando a estrutura recursiva. Para facilitar o processamento da estrutura, o campo `type` indica o tipo de valor armazenado baseado no enumerador `PrimaryType`.

```
    class MultiplicativeOperator(Enum):
        MULT = auto()
        DIV = auto()


    @dataclass
    class MultiplicativeExpression:
        base: UnaryExpression
        others: List[Tuple[MultiplicativeOperator, UnaryExpression]] =
                                                    field(
            default_factory=list
        )


    class AdditiveOperator(Enum):
        SUM = auto()
        SUB = auto()


    @dataclass
    class AdditiveExpression:
        base: MultiplicativeExpression
        others: List[Tuple[AdditiveOperator, MultiplicativeExpression]] =
                                                    field(
            default_factory=list
        )
```

Código 21 – `AdditiveExpression` e `MultiplicativeExpression`

```
    class UnaryOperator(Enum):
        PLUS = auto()
        MINUS = auto()
        NOT = auto()


    @dataclass
    class UnaryExpression:
        value: PrimaryExpression
        op: Optional[UnaryOperator] = None
```

Código 22 – `UnaryExpression`

```
    class PrimaryType(Enum):
        EXP = auto()
        IRI = auto()
        NUM_LITERAL = auto()
        BOOL_LITERAL = auto()
        STR_LITERAL = auto()
        VAR = auto()
        FUNC = auto()


    @dataclass
    class BuiltInFunction:
        name: str
        params: List["OrExpression"] = field(default_factory=list)


    @dataclass
    class PrimaryExpression:
        type: PrimaryType
        value: Union[str, int, float, "OrExpression", BuiltInFunction]
```

Código 23 – `PrimaryExpression` e `BuiltInFunction`

As chamadas de função são representadas pela classe `BuiltInFunction` que contém o nome da função sendo invocada no campo `name` e uma lista de parâmetros no campo `params`. O processamento de uma instância de `BuiltInFunction` usa o nome da função para mapear a chamada para uma função ou construção equivalente em Cypher.

O mapeamento das funções nativas de SPARQL para Cypher foi objeto de estudo desse trabalho. Algumas funções foram removidas da gramática (seção 4.2), removendo as funções sem reprodução direta em Cypher. O resultado obtido é apresentado na Tabela 3. A primeira coluna indica o nome da função em SPARQL e a coluna da direita indica a expressão em Cypher equivalente, podendo ser essa uma chamada de função ou operação chamada sobre algum dos argumentos. Os argumentos na coluna da direita são representados por `argX`, onde `X` é índice do argumento na chamada em SPARQL e são resultado do valor de `OrExpression` em Cypher, como pode ser observado na gramática no Apêndice A. Os mapeamentos estudados foram testados manualmente executando consultas em SPARQL e Cypher, porém nem todas as chamadas foram implementadas neste trabalho.

4.4 CONSTRUÇÃO DA ESTRUTURA VIA AÇÕES SEMÂNTICAS

A biblioteca PLY foi utilizada para implementar um analisador léxico (*lexer*) e um analisador sintático (*parser*) utilizando a SELECT SPARQL (Apêndice A). Ao *parser* foram adicionadas as ações semânticas que montam a estrutura descrita na seção 4.3 enquanto a consulta em SPARQL de entrada é processada. O funcionamento da

Função em SPARQL	Equivalente em Cypher
NOW	datetime()
YEAR	arg0.year
MONTH	arg0.month
DAY	arg0.day
HOURS	arg0.hour
MINUTES	arg0.minute
SECONDS	arg0.second
TIMEZONE	arg0.timezone
TZ	arg0.timezone
ABS	abs(arg0)
ROUND	round(arg0)
CEIL	ceil(arg0)
FLOOR	floor(arg0)
RAND	rand()
COALESCE	coalesce(args)
STRLEN	size(arg0)
SUBSTR	substring(arg0, arg1)
UCASE	toUpper(arg0)
LCASE	toLower(arg0)
STRSTARTS	substring(arg0, 0, size(arg1)) == arg1
STRENDS	substring(arg0, size(arg0) - size(arg1)) == arg1
CONTAINS	arg0 =~ ('.*' + arg1 + '.*')
CONCAT	arg0 + ... + argn
REGEX	arg0 =~ arg1
REPLACE	replace(arg0, arg1, arg2)
COUNT	count(args)
SUM	sum(args)
MIN	min(args)
MAX	max(args)
AVG	avg(args)

Tabela 3 – Tabela de mapeamento de funções

biblioteca é baseado na criação de um *lexer*, definindo todos os *tokens* da linguagem, utilizando expressões regulares[6], como exemplificado no Código 24. O código exemplo apresenta abreviações (simbolizadas por `[...]`) por conta da extensão e sua versão completa pode ser encontrada no repositório do projeto. A definição das expressões regulares para os `tokens` é feita através da declaração de variáveis nomeadas com o prefixo `t_` seguidos do nome do `token` no contexto da instanciação do objeto `lex`, disponibilizado na biblioteca. No caso desta implementação, o contexto é a própria instância da classe (em `lex.lex(module=self, **kwargs)` na linha 3). A declaração das palavras-chave em um dicionário a parte foi feita para melhor organização do código e o mesmo foi feito para declaração do nome da funções nativas da linguagem.

[6] https://github.com/shthiago/switch/blob/v0.1.1-alpha/transpiler/lexer.py

```
    class SelectSparqlLexer:
        def __init__(self, **kwargs):
            self.lexer = lex.lex(module=self, **kwargs)
            self._input = ""

        # Keywords
        keywords = {
            "BASE": "KW_BASE",
            "PREFIX": "KW_PREFIX",
            "SELECT": "KW_SELECT",
            [...]
            "IN": "KW_IN",
            "GROUP": "KW_GROUP",
        }

        t_IRIREF = r'<([^<>"{}|^`\\ ])*>'
        [...]
        t_SYMB_TRUE = r"true"
        t_SYMB_FALSE = r"false"

        # Tokens
        tokens = [
            # Keywords
            *keywords.values(),
            [...]
            # Terminals
            "IRI_REF",
            [...]
            "SYMB_TRUE",
            "SYMB_FALSE",
        ]

        # Caracter ignorado durante o parsing
        t_ignore = " "
        [...]
```

Código 24 – `SelectSparqlLexer`

Com o *lexer* definido e contendo todos os *tokens* da gramática, criamos o *parser*[7]. Para cada produção da gramática, cria-se uma função com o prefixo `p_`, seguido de um texto único arbitrário. A `docstring` da função (`string` na primeira linha dentro da função) deve conter a produção na forma:

$$NT : P$$

onde `NT` é um símbolo não-terminal e `P` a produção. A função deve receber um argumento que será do tipo `ply.yacc.YaccProduction`, que funciona como um vetor sem tipagem, ou seja, cada elemento pode ser um valor de qualquer tipo. O elemento no índice zero, ao fim da produção, será o valor que esse não-terminal terá na produção que o gerou, durante a execução do analisador. O valor nos demais índices é o valor

[7] https://github.com/shthiago/switch/blob/v0.1.1-alpha/transpiler/parser.py

de cada símbolo na produção, onde os não-terminais tem o valor do índice zero de sua produção e os símbolos terminais tem o valor do `token`. A manipulação desse argumento é a parte mais importante para a implementação das ações semânticas.

O *parser* foi implementado como uma classe e algumas produções com ações semânticas são exemplificadas no Código 25. O código completo pode ser encontrado no repositório. Ações mais simples, como `p_production_387`, `p_production_386` e `p_production_402` funcionam como uma ligação, apenas associando o valor do símbolo terminal ao símbolo não-terminal na produção que o gerou, garantindo o tipo correto, sendo `string` o padrão. Ações como `p_production_292`, `p_production_293`, `p_production_294` e `p_production_295` formam estruturas que lidam com as recursões da gramática, lidando com a criação de listas. No caso das produções do exemplo, elas criam a instância de `AdditiveExpression`, com o valor de `base` sendo a instância de `MultiplicativeExpression` gerada por outra cadeia de produções e o valor de `others` uma lista de pares de operação e instâncias de `MultiplicativeExpression`. Produções como `p_production_72` usam do fato que criam estruturas únicas na consulta (que compõem diretamente a classe `Query`) e atribuem valores diretamente a instância de `Query` pertencente a classe do *parser*.

Todas as produções fazem parte da classe `SelectSparqlParser`, que pode ser observada no Código 26. No construtor `__init__` é possível observar o uso do *lexer* para instanciar o objeto *yacc* que criar o *parser*. Uma instância de `Query` é criada para facilitar a lógica da construção, onde os elementos finais apenas preenchem os valores no objeto instanciado, como demonstrado anteriormente. O retorno da função `parse` é a instância da classe `Query` para o código de entrada.

4.5 CONVERSÃO DA ESTRUTURA EM CYPHER

Esta seção está dividida em explicações de como a transformação é feita para as partes da classe `Query`, utilizando trechos de código retirados do repositório do projeto[8]. Os campos que geram código efetivamente são `graph_pattern` e `modifiers`, enquanto `returning`, `variables` e `namespaces` fornecem informações utilizadas na geração.

A função `generate`, implementada dentro da classe `CypherGenerator`, que recebe o código SPARQL e faz as chamadas para construção dos blocos de código em Cypher é apresentada no Código 27. A chamada para `self.parse_query` executa a construção da estrutura intermediária e seu retorno é armazenado em `query`. As chamadas para `self.setup_aliases` e `setup_namespaces` servem para associar valores ao próprio objeto de modo facilitar a construção de outras funções.

[8] https://github.com/shthiago/switch/blob/v0.1.1-alpha/transpiler/cypher_generator.py

```
    def p_production_72(self, p):
        """LimitClause : KW_LIMIT INTEGER"""
        self.query.modifiers.limit = int(p[2])

    def p_production_292(self, p):
        """AdditiveExpression : MultiplicativeExpression
                                              AdditiveExpressionAux1"""
        p[0] = nodes.AdditiveExpression(p[1], p[2])

    def p_production_293(self, p):
        """AdditiveExpressionAux1 : SYMB_PLUS MultiplicativeExpression
                                              AdditiveExpressionAux1"""
        p[0] = [(nodes.AdditiveOperator.SUB, p[2]), *p[3]]

    def p_production_294(self, p):
        """AdditiveExpressionAux1 : SYMB_MINUS MultiplicativeExpression
                                              AdditiveExpressionAux1"""
        p[0] = [(nodes.AdditiveOperator.SUB, p[2]), *p[3]]

    def p_production_295(self, p):
        """AdditiveExpressionAux1 : empty"""
        p[0] = []

    def p_production_363(self, p):
        """Aggregate : FUNC_COUNT SYMB_LP AggregateAux1 AggregateAux2
                                              SYMB_RP"""
        p[0] = nodes.BuiltInFunction("COUNT", [p[4]])

    def p_production_402(self, p):
        """iri : IRIREF"""
        p[0] = p[1]

    def p_production_386(self, p):
        """NumericLiteralPositive : INTEGER_POSITIVE"""
        p[0] = int(p[1])

    def p_production_387(self, p):
        """NumericLiteralPositive : DECIMAL_POSITIVE"""
        p[0] = float(p[1])
```

Código 25 – Exemplos de produções do analisador sintático

```
    class SelectSparqlParser:
        def __init__(self, **kwargs):
            self.lexer = SelectSparqlLexer()
            self.tokens = self.lexer.tokens

            self.yacc = yacc.yacc(module=self, check_recursion=False, **
                                                    kwargs)

            self.query = Query()

        def parse(self, source_code: str) -> Query:
            return self.yacc.parse(source_code, lexer=self.lexer)

        [...]
```

Código 26 – `SelectSparqlParser`

```
    def generate(self, sparql_query: str) -> str:
        """Generate cypher from sparql"""
        query = self.parse_query(sparql_query)

        self.setup_aliases(query.returning)

        self.setup_namespaces(query.namespaces)

        patterns = self.split_pattern(query.graph_pattern)

        code_blocks = [
            self.code_block_for_pattern(p, query)
            for p in patterns
            if len(p.and_triples) > 0
        ]

        united_code = "\nUNION\n".join(code_blocks)

        modifiers = self.result_modifier(query.modifiers)
        having_part = self.having_clause(query.modifiers)

        if modifiers or having_part:
            ret_clause = "RETURN *"
            modified_code = (
                "CALL {\n"
                + united_code
                + "\n}"
                + ("\n" + having_part if having_part else "")
                + "\n"
                + ret_clause
                + ("\n" + modifiers if modifiers else "")
            )

        else:
            modified_code = united_code

        return modified_code
```

Código 27 – Função de entrada para geração de Cypher

```
1  PREFIX b:<http://ontologi.es/place/>
2  PREFIX dct:<http://purl.org/dc/terms/>
3  PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
4  SELECT ?stateName WHERE {
5      ?state rdfs:label ?stateName .
6      {
7          b:BR dct:hasPart ?state
8      } UNION {
9          b:US dct:hasPart ?state
10     }
11 }
```

Código 28 – Consulta com distributiva

4.5.1 Campo `graph_pattern`

Este bloco tem uma característica recursiva, uma vez que no campo `or_blocks` pode haver uma lista de outras instâncias de `GraphPattern`, cada uma com sua própria lista de triplas no campo `and_triples`. Para transformar as consultas para Cypher, essa estrutura tornaria a lógica mais complexa. Portanto, a primeira operação feita sobre a estrutura simplifica a estrutura de `graph_pattern` (função `split_pattern`).

De modo geral, podemos aproximar as agregações de bloco de triplas em SPARQL com uma expressão booleana. Por exemplo, na consulta apresentada no Código 28, temos que a variável `?state`, não importando o que ela for, se relaciona através da URI `rdfs:label` com a variável `?stateName` e a variável `?state` é valor da propriedade `dct:hasPart` das URIs `b:BR` ou `b:US`. A instância de `GraphPattern` para essa consulta, omitindo campos nulos ou vazios, é apresentada no Código 29. Podemos aproximar as triplas como afirmações lógicas, e chamar a tripla do Código 28 da linha 5, 7 e 9 de `P1`, `P2` e `P3`, respectivamente. A expressão booleana que representa essa consulta seria dada por:

$$P1 \wedge (P2 \vee P3)$$

e utilizando a propriedade distributiva da operação de conjunção, podemos reescrever a expressão como:

$$(P1 \wedge P2) \vee (P1 \wedge P3)$$

Essa lógica pode ser extrapolada para qualquer estrutura de triplas associadas com blocos de `UNION` e triplas, e essa estrutura de expressão formada pela disjunção entre conjunções é mais fácil de se representada em Cypher, como será demonstrado nesta seção.

O algoritmo executado pela função `split_patterns` é apresentado no Código 30. O resultado desse algoritmo será sempre uma lista de `GraphPattern` onde os valores no campo `or_blocks` podem ser desconsiderados e logicamente a disjunção das triplas dos campos `and_triples` de cada elemento da lista é equivalente à estrutura expressa

```
    GraphPattern(
        and_triples=[Triple("?state", "rdfs:label", "?stateName")],
        or_blocks=[
            GraphPattern(and_triples=[
                Triple("b:BR", "dct:hasPart", "?state")
            ]),
            GraphPattern(and_triples=[
                Triple("b:US", "dct:hasPart", "?state")
            ]),
        ]
    )
```

Código 29 – `GraphPattern` para o Código 28

```
Entradas:
    GraphPattern grafo;

Se graph.or_blocks for uma lista vazia:
    Retorna [grafo]

Lista[GraphPattern] grafos_a_visitar = [grafo]
Lista[GraphPattern] padroes = []

Enquanto tamanho(grafos_a_visitar) > 0:
    GraphPattern grafo_atual = remove_primeiro(grafos_a_visitar)
    Se tamanho(grafo_atual.or_blocks) > 0:
        Para todo bloco em grafo_atual.or_blocks:
            Para todo grafo_b em bloco:
                Adiciona grafo_b.and_triples ao final de padroes
                Adiciona grafo em grafos_a_visitar
    Senão:
        Adiciona grafo_atual em grafos_a_visitar

Retorna padroes
```

Código 30 – Algoritmo para planificação de `GraphPattern`

pela consulta original.

Após essa transformação, cada instância de `GraphPattern` é processada de modo a criar uma consulta Cypher independente baseada nas triplas do campo `and_triples`. Dada a estrutura de triplas definida no capítulo 2.1.1, para o *subject* e para o *predicate* há duas possibilidades: eles podem ser variáveis ou URIs. O *object* pode ser uma variável, uma URI ou um *literal* (*string*, inteiro, decimal). Para cada combinação, tendo um total de doze combinações, uma estrutura diferente em Cypher reproduzirá o filtro que a tripla tem em SPARQL.

4.5.1.1 Transformação das triplas

Para encontrar nodos e arestas que satisfaçam uma tripla dentro de um banco Neo4j, precisamos primeiro avaliar o *subject* da tripla. Se este for uma variável, verificamos se esta variável já foi usada na consulta resultante. Se não tiver sido usada, adicionamos uma cláusula `MATCH` para o nome daquela variável, do contrário esse passo não é necessário. Se o *subject* for uma URI, adicionamos uma cláusula `MATCH` com um nome gerado para aquela URI seguida de uma cláusula `WHERE` que avalia se a propriedade `uri` daquele nodo é igual a URI esperada. Há um caso específico para a construção desse bloco inicial quando *predicate* for `rdf:type`. Nesse caso, a cláusula `WHERE` deverá filtrar se a URI especificada no *object* está inclusa na lista de *labels* do nodo.

Após a avaliação do *subject*, é necessário avaliar *predicate* e *object*. Para essa etapa, as considerações de Lombardot *et al.* (2019) são importantes. As relações quando carregadas pelo Neosemantics podem ser inseridas como uma propriedade do nodo quando o valor for um *literal*, como uma *label* do nodo quando a relação for através da propriedade `rdf:type` e como uma relação entre dois nodos nos demais casos. A exceção para a propriedade `rdf:type` foi descrita no parágrafo anterior, e para esta não é necessário gerar mais nenhum código, apenas o bloco `MATCH ... WHERE ...` representa a tripla. Para os demais casos, precisamos criar blocos de código contando que a relação pode ser qualquer um dos casos.

Supondo a tripla `(?s ?p ?o)` em uma consulta SPARQL, a sua representação em Cypher (desconsiderando a cláusula `MATCH`) seria como demonstrado no Código 31. São duas construções de lista concatenadas e renomeadas como `triples`. A primeira lista, na linha 1 logo após a cláusula `UNWIND`, trata do caso em que o *object* é valor de uma propriedade `predicate`. Sendo assim, para cada propriedade `key` na lista de propriedades (retornada pela chamada para a função `keys(node)`), cria-se uma sublista de três elementos composta pelo nodo `s`, a propriedade `key` e o valor da propriedade para aquele nodo `s[key]`. Na linha 2, cobre-se o caso de *object* ser um outro nodo ao qual *subject* se relaciona através de uma relação qualquer. Nesse caso, cria-se uma consulta interna `(s)-[_relation]-(o)` onde a variável `s` já foi definida anteriormente e as demais são definidas na consulta. Para cada resultado dessa consulta, ou seja, todas as relações em qualquer direção ligando a variável `s` com outro nodo, forma-se uma lista de três elementos contendo o nodo `s`, a relação `_relation` e outro nodo `o`. As duas listas criadas são concatenadas e contêm todas as relações e propriedades do nodo `s`. A cláusula `UNWIND` transforma uma lista em linhas individuais de resultado, e a renomeação como `triples` permite aplicarmos novas operações sobre essa lista de resultados.

A tripla apresentada é composta apenas por variáveis, portanto essa é a forma mais genérica possível de se buscar os elementos que satisfazem a tripla. Quando

```
UNWIND [key in keys(s) | [s, key, s[key]]] +
       [(s)-[\_relation]-(o) | [s, \_relation, o]] AS triples
```

Código 31 – Filtro de triplas para `(?s, ?p, ?o)`

```
UNWIND [key in keys(country)
            WHERE key = "ns0_hasPart"
        | [country, key, country[key]]] +
       [(country)-[_relation]-(state)
            WHERE type(_relation) = "ns0_hasPart"
        | [country, _relation, state]] AS triples
```

Código 32 – Filtro de triplas para `(?country dct:hasPart ?state)`

```
UNWIND [(brazil)-[_relation]-(b_BR_SC)
            WHERE type(_relation) = "ns0_hasPart"
                AND b_BR_SC.uri =
                    "http://ontologi.es/place/BR-SC"
              | [brazil, _relation, b_BR_SC]] AS triples
```

Código 33 – Filtro de triplas para `(?brazil dct:hasPart b:BR-SC)`

`predicate` é uma URI, como na tripla `(?country dct:hasPart ?state)`, precisamos adicionar filtros para levarmos em consideração apenas propriedades com o valor ou relações com o tipo `dct:hasPart`. O filtro para a tripla apresentada é demonstrado no Código 32. As quebras de linha foram inseridas para facilitar a leitura, porém o código segue o mesmo padrão apresentado anteriormente, com a adição de uma cláusula `WHERE` que filtra o valor da chave e o tipo da relação baseado na URI em `predicate`. O valor `ns0_hasPart` não pode ser obtido pelo código diretamente, porém ele pode ser construído a partir da chamada para funções do Neosemantics. Essa construção é detalhada na seção 4.5.1.3.

Quando o *object* é uma URI, o bloco que gera a lista de triplas que seriam valores de propriedade torna-se desnecessário, pois apenas *literals* são carregados como propriedades, então o código resultante precisa verificar apenas as relações entre nodos. Por exemplo, para a tripla `(?brazil dct:hasPart b:BR-SC)`, temos no Código 33 o filtro em Cypher. Além do filtro pelo tipo da relação, que também é uma URI, adicionamos um filtro pelo valor da propriedade `uri` do nodo.

Quando o *object* é um *literal*, é o bloco que gera a lista de triplas que seriam relações entre objetos que se torna desnecessário, uma vez que todas as *literals* foram carregadas como propriedades. Por exemplo, para a tripla `(?brazil b:name "Brazil")`, o Código 34 apresenta os filtros equivalentes. Além do filtro pelo nome da propriedade, filtramos também pelo valor da propriedade.

```
1  UNWIND [key in keys(brazil)
2              WHERE key = "ns1_name"
3                  AND brazil[key] = "Brazil"
4          | [brazil, key, brazil[key]]] AS triples
```

Código 34 – Filtro de triplas para `(?brazil b:name "Brazil")`

```
1  MATCH (b_BR) WHERE b_BR.uri = "http://ontologi.es/place/BR"
2  UNWIND [key in keys(b_BR)
3              WHERE key = "ns0_hasPart"
4          | [b_BR, key, b_BR[key]]] +
5          [(b_BR)-[_relation]-(brazilState)
6              WHERE type(_relation) = "ns0_hasPart"
7          | [b_BR, _relation, brazilState]] AS triples
8  WITH triples[2] AS brazilState
9  UNWIND [key in keys(brazilState)
10             WHERE key = "ns1_label"
11         | [brazilState, key, brazilState[key]]] +
12         [(brazilState)-[_relation]-(brazilStateName)
13             WHERE type(_relation) = "ns1_label"
14         | [brazilState, _relation, brazilStateName]] AS triples
15 WITH brazilState AS brazilState, triples[2] AS brazilStateName
```

Código 35 – Filtro para sequência de duas triplas

Essas regras podem ser combinadas e depender da tripla de entrada, e cada tripla forma uma cláusula `MATCH`, caso o *subject* ainda não tiver sido declarado, `UNWIND` caso não for uma cláusula com `rdf:type` como *predicate*, e termina com uma cláusula `WITH` que renomeia as colunas da variáveis criada *triples* de modo fazer com que os filtros da próxima tripla utilizem os dados já filtrados pela primeira. A cláusula é construída a partir das variáveis já utilizadas pela consulta até o momento, então a primeira tripla adicionará na cláusula `WITH` as variáveis usadas por ela, a segunda adicionará as variáveis usadas por ela e mais as variáveis utilizadas pelas anteriores e assim por diante. Um exemplo para a sequência de triplas `(b:BR dct:hasPart ?brazilState)` `(?brazilState rdfs:label ?brazilStateName)` é apresentado no Código 35.

A cláusula `RETURN` igual é adicionada a todos os blocos de código produzidos a partir de instâncias de `GraphPattern`, gerada a partir do campo `returning` da estrutura gerada para a consulta. A cláusula de retorno de Cypher é muito semelhante a de SPARQL, apenas listando o nome das variáveis que são retornadas, podendo haver a aplicação de uma renomeação ou criação de nova variável baseada no resultado de uma expressão utilizando a cláusula `ALIAS`.

Os códigos Cypher apresentados são gerados e colocados em uma lista (Código 27, linha 11). Para efetuar a união dos resultados, utilizamos a cláusula `UNION` de Cypher, que executa uma disjunção entre dois grupos de resultados. Colocamos então

```
1  MATCH (b_US) WHERE b_US.uri = "http://ontologi.es/place/US"
2  UNWIND [key in keys(b_US) WHERE key = "ns0_hasPart"
3          | [b_US, key, b_US[key]]] +
4          [(b_US)-[_relation]-(state) WHERE type(_relation) =
             "ns0_hasPart"
5          | [b_US, _relation, state]] AS triples
6  WITH triples[2] AS state
7  UNWIND [key in keys(state) WHERE key = "ns1_label"
8          | [state, key, state[key]]] +
9          [(state)-[_relation]-(stateName) WHERE type(_relation) =
             "ns1_label"
10         | [state, _relation, stateName]] AS triples
11 WITH triples[0] AS state, triples[2] AS stateName
12 RETURN stateName
13 UNION
14 MATCH (b_BR) WHERE b_BR.uri = "http://ontologi.es/place/BR"
15 UNWIND [key in keys(b_BR) WHERE key = "ns0_hasPart"
16         | [b_BR, key, b_BR[key]]] +
17         [(b_BR)-[_relation]-(state) WHERE type(_relation) =
             "ns0_hasPart"
18         | [b_BR, _relation, state]] AS triples
19 WITH triples[2] AS state
20 UNWIND [key in keys(state) WHERE key = "ns1_label"
21         | [state, key, state[key]]] +
22         [(state)-[_relation]-(stateName) WHERE type(_relation) =
             "ns1_label"
23         | [state, _relation, stateName]] AS triples
24 WITH triples[0] AS state, triples[2] AS stateName
25 RETURN stateName
```

Código 36 – Cypher para a consulta do Código 28

os blocos de código conectados em série por uma cláusula `UNION` para obter a consulta final.

A consulta em Cypher gerada para a consulta SPARQL do Código 28 está apresentada por completo no Código 36, com exceção dos valores utilizados nas cláusulas `WHERE` com a nomenclatura de *namespaces* definida internamente pela Neosemantics (`ns0_hasPart` e `ns1_label`). O nome dos *namespaces* não é acessado pelo Switch, uma vez que este não se conecta a instância de Neo4j. A forma como contornamos essa problema é descrito na seção 4.5.1.3 .

4.5.1.2 Transformação das expressões e funções nativas

As expressões em SPARQL e Cypher são muito semelhantes. Ambas contam com o mesmo conjunto de operadores aritméticos e lógicos. A classe `ExpressionHan-`

`dler`[9] implementa as funções que transformam cada classe associada com as expressões descritas na seção 4.3.5, de maneira recursiva. A classe `OrExpression` contém instâncias de `AndExpression`, então esta concatena o resultado das transformações das instâncias de `AndExpression` separadas pelo operador `OR` de Cypher. As classes que contém uma instância base e uma lista de tuplas de operação e instância, como `AdditiveExpression`, concatenam o resultado da transformação da primeira instância com a segunda através do operador na tupla da segunda, torna o resultado a base e repete o processo enquanto houverem tuplas. O Código 37 mostra partes da lógica descrita, e o código completo pode ser encontrado no repositório do projeto.

Para fazer o mapeamento das chamadas de função, a classe implementa a tradução de cada função como um método que recebe os parâmetros da chamada e cria a versão em Cypher. No construtor da classe criamos um dicionário mapeando o nome da chamada nativa SPARQL (valor do *token* identificado pelo *lexer*) para o método que gera a chamada equivalente em Cypher. No Código 37, o método `builtinfunction_to_cypher` recebe a instância de `BuiltInCall` da estrutura e usa o nome da chamada para obter do mapeamento o método gerador e o executa passando a lista de parâmetros.

Neste trabalho, por limitações de tempo e complexidade associada a tipagem em Cypher, apenas demonstramos o conceito para a transformação das funções através da funções de agregação `SUM`, `MAX`, `MIN`, `AVG` e `COUNT`.

4.5.1.3 Construção das URIs

Como descrito no capítulo 2.2.1.1, os *namespaces* dos dados carregados com o Neosemantics ficam armazenados em um nodo dedicado, e uma abreviação é associada aos elementos do *namespace* dentro dos nodos para criar as chaves e *labels* associadas a relações e nodos. No entanto, durante a transformação da consulta, o Switch não tem conexão com a base de dados, uma vez que tornaria a transformação mais custosa em termos de uso de recursos. A Neosemantics dispõe da função `n10s.rdf.shortFormFromFullUri` que pode ser utilizada durante as consultas em Cypher que recebe o endereço do *namespace* e retorna como *string* o prefixo associado ao endereço na nomenclatura das propriedades e *labels* dos dados.

Por exemplo, supondo que uma base de dados tenha sido carregada e esta use somente o *namespace http://purl.org/dc/terms/*, este teria sido associado ao prefixo `ns0_` e o valor da expressão

```
n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "label
```

quando utilizada durante uma consulta Cypher seria `"ns0_label"`.

[9] https://github.com/shthiago/switch/blob/v0.1.1-alpha/transpiler/expression_handler.p#L20

```
    class ExpressionHandler:
        def __init__(self):
            self.builtin_gen_map = {
                "COUNT": self._gen_call_for_count,
                "SUM": self._gen_call_for_sum,
                "MIN": self._gen_call_for_min,
                "MAX": self._gen_call_for_max,
                "AVG": self._gen_call_for_avg,
                [...]
            }
        [...]
        def value_orexpression(self, node: OrExpression) -> str:
            base_value = self.value_andexpression(node.base)
            if node.others:
                rest = " OR " + " OR ".join(
                    [f"({self.value_andexpression(n)})" for n in
                                                        node.others]
                )
                return f"{base_value}" + rest

            return base_value
        [...]
        def value_additiveexpression(self, node: AdditiveExpression) ->
                                                    str:
            value = self.value_multiplicativeexpression(node.base)
            if not node.others:
                return value

            for op, exp in node.others:
                op_str = self._add_operator_str(op)
                exp_str = self.value_multiplicativeexpression(exp)
                value += f" {op_str} {exp_str}"

            return f"({value})"
        [...]
        def builtinfunction_to_cypher(self, call: BuiltInFunction) -> str:
            return self.builtin_gen_map[call.name](call.params)

        def _gen_call_for_count(self, params: List[OrExpression]):
            arg = self.value_orexpression(params[0])

            return f"count({arg})"

        [...]
```

Código 37 – Trechos da `ExpressionHandler`

Para fazer a criação dessas expressões, o método `self.setup_namespaces` no Código 27, demonstrado no Código 38, cria um dicionário interno que faz o mapeamento da abreviação usada para o *namespace* na consulta de entrada, que é utilizada no método `ns_uri` que ao receber uma URI na forma `abbrev:name` usa o endereço do *namespace* para gerar um código que chama a função da Neosemantics e concatena o prefixo obtido com o nome de dentro do *namespace*.

Dessa forma, os códigos Cypher gerados pelo Switch não referenciam os prefixos gerados internamente, mas sim os próprios endereços dos *namespaces*, que estão acessíveis pela transformação. Um segundo uso para os *namespaces* está também demonstrado no Código 38, o método `full_uri` utiliza a abreviação para obter os endereços dos objetos referenciados e filtrar os nodos baseados na propriedade `uri`.

```
1     def setup_namespaces(self, namespaces: List[Namespace]):
2         self.namespaces = {nm.abbrev: nm.full for nm in namespaces}
3
4     def ns_uri(self, abbev_uri: str) -> str:
5         nms, name = abbev_uri.split(":")
6         full_nms = self.namespaces[nms]
7
8         return f'n10s.rdf.shortFormFromFullUri("{full_nms}") + "{name}"'
9
10    def full_uri(self, uri: str) -> str:
11        abbrev, name = uri.split(":")
12        base = self.namespaces[abbrev]
13
14        return f'"{base + name}"'
```

Código 38 – Processamento de namespaces

4.5.2 Campo `modifiers`

Para a implementação dos modificadores de resultado, utilizamos a ordem de avaliação de SPARQL para fazer uma operação equivalente em Cypher sem precisar modificar códigos já gerados. Os modificadores são as últimas cláusulas avaliadas e todas elas tem operações, implícitas ou explícitas, equivalentes em Cypher, então quando existem modificadores de resultado na consulta, usamos a cláusula `CALL` de Cypher para executar a consulta já gerada e aplicamos as modificações de resultado sobre elas.

Para as cláusulas `ORDER BY`, `LIMIT` e `OFFSET` extistem cláusulas semelhantes em Cypher. A consulta do Código 39, por exemplo, resulta na consulta em Cypher no Código 40, abreviada para melhor visualização dos modificadores de resultado.

Os modificadores de resultado `GROUP BY` e `HAVING`, por outro lado, não têm equivalente explícito em Cypher, porém podemos replicar seu comportamento através de operações nas cláusulas `RETURN`, `WITH` e `WHERE`. Como demonstrado no Código 42, re-

```
1  PREFIX b:<http://ontologi.es/place/>
2  PREFIX dct:<http://purl.org/dc/terms/>
3  PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
4  SELECT ?brazilStateName WHERE {
5      b:BR dct:hasPart ?brazilState .
6      ?brazilState rdfs:label ?brazilStateName
7  }
8  ORDER BY ?brazilStateName
9  LIMIT 10
10 OFFSET 1
```

Código 39 – Consulta SPARQL com modificador de resultado

```
1  CALL {
2      [...]
3      WITH triples[0] AS brazilState, triples[2] AS brazilStateName
4      RETURN brazilStateName
5  }
6  RETURN *
7  ORDER BY brazilStateName ASC
8  SKIP 1
9  LIMIT 10
```

Código 40 – Cypher para a consulta 39

```
1  PREFIX b:<http://www.geonames.org/ontology#>
2  PREFIX dct:<http://purl.org/dc/terms/>
3  SELECT ?name (COUNT(?state) AS ?stateCount) WHERE {
4      ?country a b:Country .
5      ?country dct:hasPart ?state .
6      ?country b:name ?name
7  } GROUP BY ?name
8  HAVING (COUNT(?state) > 20)
```

Código 41 – Consulta SPARQL com modificador de resultado

sultado abreviado da transformação para o Código 41, o uso de funções de agregação na cláusula de retorno (linha 4) implica uma agregação pelas colunas que não estão sendo agregadas. Os filtros sobre o resultado da agregação da cláusula `HAVING` são feitos com uma cláusula `WHERE`.

```
CALL {
    [...]
    WITH state AS state, triples[0] AS country, triples[2] AS name
    RETURN name, count(state) AS stateCount
}
WITH *
WHERE (stateCount > 20)
RETURN *
```

Código 42 – Cypher para a consulta 41

5 TESTS E RESULTADOS

5.1 VALIDAÇÃO

Para validar o Switch, diversas consultas em SPARQL foram criadas para serem executadas no Neo4j. Os testes consideraram sempre um mesmo conjunto de dados em RDF usando a biblioteca de Python `rdflib`[1], que carrega os dados em memória e executa consultas SPARQL sobre os mesmos, assim como um banco Neo4j carregado através da Neosemantics. Os dados RDF são persistidos no Neo4j conforme as regras de mapeamento descritas na Seção 2.2.1.1.

O conjunto de dados RDF utilizado para os testes é uma relação de países soberanos e territórios, disponível em *https://ontologi.es/place/*. Criamos um *script*, disponível no repositório do projeto, que faz o *download* das partes do dataset através dos links e consolida em um arquivo único, utilizado para este teste.

Todas as consultas utilizadas podem ser encontradas no repositório do projeto[2]. Duas delas são apresentadas aqui com a comparação de resultados.

5.1.1 Primeiro teste

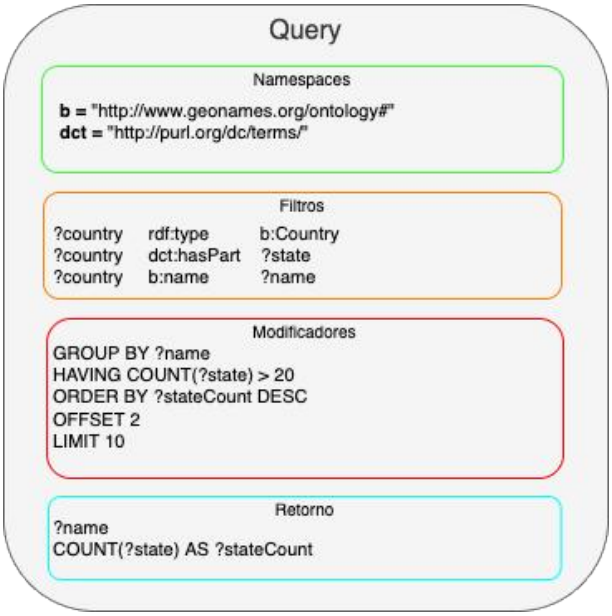
A primeira consulta demonstrada aqui é a do Código 43, que utiliza diversos recursos tratados por este trabalho sobre a transformação de expressões e aplicação de modificadores de resultado para buscar todos os nomes associados a nodos com a *label* `Country` com o número de territórios maior do que vinte, ordenando de maneira decrescente pelo número de territórios, descartando os dois primeiros e limitando a dez tuplas. O resultado da transformação dessa consulta é apresentado no Código 44. O resultado das consultas deste primeiro teste podem ser vistos nas Tabelas 4 e 5. Na Figura 10 está apresentada a visualização da estrutura intermediária para a consulta do teste.

---

[1] https://pypi.org/project/rdflib/

[2] https://github.com/shthiago/switch/tree/v0.1.1-alpha/test_queries

Figura 10 – Primeiro teste - Visualização da estrutura



Fonte: produção própria

```
1  PREFIX b:<http://www.geonames.org/ontology#>
2  PREFIX dct:<http://purl.org/dc/terms/>
3  SELECT ?name (COUNT(?state) AS ?stateCount) WHERE {
4      ?country a b:Country .
5      ?country dct:hasPart ?state .
6      ?country b:name ?name
7  } GROUP BY ?name
8  HAVING (COUNT(?state) > 20)
9  ORDER BY DESC (?stateCount)
10 OFFSET 2
11 LIMIT 10
```

Código 43 – Primeiro teste - consulta SPARQL

```
CALL {
MATCH (country) WHERE n10s.rdf.shortFormFromFullUri("http://www.geonames.org/ontology#") +
    "Country" IN labels(country)
WITH country AS country

UNWIND [key in keys(country) WHERE key =
    n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"
| [country, key, country[key]]] +
[(country)-[_relation]-(state) WHERE type(_relation) =
    n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"
| [country, _relation, state]] AS triples
WITH triples[0] AS country, triples[2] AS state

UNWIND [key in keys(country) WHERE key =
    n10s.rdf.shortFormFromFullUri("http://www.geonames.org/ontology#") + "name"
| [country, key, country[key]]] +
[(country)-[_relation]-(name)
WHERE type(_relation) = n10s.rdf.shortFormFromFullUri("http://www.geonames.org/ontology#") +
    "name"
| [country, _relation, name]] AS triples
WITH state AS state, triples[0] AS country, triples[2] AS name

RETURN name, count(state) AS stateCount
}
WITH *
WHERE (stateCount > 20)
RETURN *
ORDER BY stateCount DESC
SKIP 2
LIMIT 10
```

Código 44 – Primeiro teste - consulta Cypher gerada pelo Switch

?name	?stateCount
Georgia	78
Azerbaijan	77
Thailand	77
Viet Nam	60
United States	57
Moldova	50
Poland	49
Algeria	47
Japan	46
Burkina Faso	45

Tabela 4 – Primeiro teste - resultado da consulta SPARQL

name	stateCount
"Georgia"	78
"Thailand"	77
"Azerbaijan"	77
"Viet Nam"	60
"United States"	57
"Moldova"	50
"Poland"	49
"Algeria"	47
"Japan"	46
"Burkina Faso"	45

Tabela 5 – Primeiro teste - resultado da consulta Cypher

Percebe-se que o resultado das duas consultas é o mesmo, ou seja, o mapeamento realizado pelo Switch preserva a semântica da consulta quando executada no Neo4j. Ainda, os tempos de execução das consultas SPARQL e Cypher foram, respectivamente, 57ms e 2,76s. O tempo de conversão da consulta foi de 1,8ms. Podemos observar como a execução de agregações é mais custosa no Neo4j, no entanto, não aparenta ser um tempo proibitivo.

5.1.2 Segundo teste

O segundo teste considera a consulta do Código 45, que utiliza uma cláusula `UNION` em paralelo com uma tripla, demonstrando o funcionamento da união de resultados e planificação dos padrões de grafo apresentados na Seção 4.3.3. A consulta busca os nomes dos primeiros 20 territórios pertencentes ao Brasil ou aos Estados Unidos em ordem alfabética. A consulta Cypher é apresentada no Código 46, e o resultado das mesmas nos cenários descritos é apresentado nas Tabelas 6 e 7, respectivamente.

```
1  PREFIX b:<http://ontologi.es/place/>
2  PREFIX dct:<http://purl.org/dc/terms/>
3  PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
4  SELECT ?stateName WHERE {
5      ?state rdfs:label ?stateName .
6      {
7          b:BR dct:hasPart ?state
8      } UNION {
9          b:US dct:hasPart ?state
10     }
11 }
12 ORDER BY ?stateName
13 LIMIT 20
```

Código 45 – Segundo teste - consulta SPARQL

```
CALL {
MATCH (b_US) WHERE b_US.uri = "http://ontologi.es/place/US"
UNWIND [key in keys(b_US) WHERE key =
    n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart" | [b_US, key,
    b_US[key]]] + [(b_US)-[_relation]-(state) WHERE type(_relation) =
    n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart" | [b_US,
    _relation, state]] AS triples
WITH triples[2] AS state

UNWIND [key in keys(state) WHERE key =
    n10s.rdf.shortFormFromFullUri("http://www.w3.org/2000/01/rdf-schema#") + "label"
| [state, key, state[key]]] +
[(state)-[_relation]-(stateName) WHERE type(_relation) =
    n10s.rdf.shortFormFromFullUri("http://www.w3.org/2000/01/rdf-schema#") + "label"
| [state, _relation, stateName]] AS triples
WITH triples[0] AS state, triples[2] AS stateName
RETURN stateName

UNION

MATCH (b_BR) WHERE b_BR.uri = "http://ontologi.es/place/BR"
UNWIND [key in keys(b_BR) WHERE key =
    n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"
| [b_BR, key, b_BR[key]]] +
[(b_BR)-[_relation]-(state) WHERE type(_relation) =
    n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"
| [b_BR, _relation, state]] AS triples
WITH triples[2] AS state

UNWIND [key in keys(state) WHERE key =
    n10s.rdf.shortFormFromFullUri("http://www.w3.org/2000/01/rdf-schema#") + "label"
| [state, key, state[key]]] +
[(state)-[_relation]-(stateName) WHERE type(_relation) =
    n10s.rdf.shortFormFromFullUri("http://www.w3.org/2000/01/rdf-schema#") + "label"
| [state, _relation, stateName]] AS triples
WITH triples[0] AS state, triples[2] AS stateName
RETURN stateName
}
RETURN *
ORDER BY stateName ASC
LIMIT 20
```

Código 46 – Segundo teste - consulta Cypher gerada

?stateName
Acre
Alabama
Alagoas
Alaska
Amapá
Amazonas
American Samoa
Arizona
Arkansas
Bahia
California
Ceará
Colorado
Connecticut
Delaware
District of Columbia
Distrito Federal
Espírito Santo
Florida
Georgia

Tabela 6 – Segundo teste - resultado da consulta SPARQL

stateName
"Acre"
"Alabama"
"Alagoas"
"Alaska"
"Amapá"
"Amazonas"
"American Samoa"
"Arizona"
"Arkansas"
"Bahia"
"California"
"Ceará"
"Colorado"
"Connecticut"
"Delaware"
"District of Columbia"
"Distrito Federal"
"Espírito Santo"
"Florida"
"Georgia"

Tabela 7 – Segundo teste - resultado da consulta Cypher

Percebe-se, novamente, que o resultado das duas consultas é o mesmo. Com relação aos os tempos de execução das consultas SPARQL e Cypher, temos 77,4ms e 19ms. O tempo de conversão da consulta foi de 1,6ms. Nesse caso, podemos observar como uma consulta que não envolve agregação executa mais rápido no Neo4j.

5.2 TESTES QUANTITATIVOS E RESTRIÇÕES

Além dos testes detalhados anteriormente, foram executados mais dez testes, para avaliar o funcionamento da lógica implementada. Os tempos de execução em SPARQL, tradução e execução em Cypher são apresentados na Table a8. De todos os testes, apenas um não apresentou exatamente o mesmo resultado. O teste do arquivo `test_query_10.sparql` apresentou dados diferentes. No entanto, a diferença é causada pelos tempos de execução, pois essa consulta testa a chamada para a função `NOW()`, que retorna o instante de tempo em que a função foi executada. Portanto, apesar dos resultados não serem iguais, a tradução continua estando correta.

Teste	**SPARQL (s)**	**Switch (ms)**	**Cypher (s)**	**Resulados iguais**
test_query_1.sparql	0.002	2.2	0.009	Sim
test_query_2.sparql	0.004	1.8	0.02	Sim
test_query_3.sparql	0.003	2.0	0.011	Sim
test_query_4.sparql	0.003	2.3	0.012	Sim
test_query_5.sparql	0.079	1.7	0.027	Sim
test_query_6.sparql	0.042	2.5	0.246	Sim
test_query_7.sparql	0.003	1.7	0.167	Sim
test_query_8.sparql	0.058	1.7	2.808	Sim
test_query_9.sparql	0.057	1.9	2.683	Sim
test_query_10.sparql	0.001	1.6	3.195	Não
test_query_11.sparql	0.095	1.7	2.992	Sim
test_query_12.sparql	0.087	1.9	3.048	Sim

Tabela 8 – Todos os testes

Todos os arquivos fonte dos testes estão disponíveis no código do projeto. Muitas funcionalidades da linguagem precisaram ser deixadas de lado por conta das modificações aplicadas e lógicas não implementadas por conta do tempo disponível, destacando as cláusulas `FILTER` e `MINUS`.

6 CONCLUSÃO

A construção de *triplestores*, por mais que o conceito não seja novo, ainda é um tema em desenvolvimento e estudo. Nesse trabalho, pudemos demonstrar um dos desafios associados a construção de sistemas de armazenamento de tripla e propomos uma ferramenta para ser utilizada na construção dessa tecnologia. A utilização da linguagem Python com a biblioteca PLY se distancia da implementação do Neo4j, que utiliza a linguagem Java[1], porém existem diversas maneiras de se fazer a integração entre aplicações em diferentes linguagens, além da possibilidade da recriação da lógica do Switch em Java.

Durante a execução deste trabalho, pudemos observar a complexidade envolvida na tradução entre as duas linguagens de consulta. As diferentes formas de expressar operações e descrever o resultado desejado foram dificuldades maiores que o esperado durante a definição de escopo do trabalho, além do desafio que seria fazer mapeamentos entre as funções nativas de cada linguagem, que também se revelou maior que o previsto. No entanto, em termos de resultados, como demonstrado nos testes, podemos traduzir consultas relativamente complexas, realizando agregações e filtros baseados na estrutura dos dados, o que representa uma parte considerável das consultas SPARQL.

Trabalhos futuros incluem estudos para aprimoramento da tradução, adicionando suporte para as cláusulas `FILTER` e `MINUS`, e para as funções nativas que precisaram ser deixadas de lado neste trabalho. Para expandir o número de consultas traduzíveis, uma extensão para Neo4j poderia ser criada, implementando as funções removidas pela alteração do item 4 do Capítulo 4.2, que poderia então ser revertida. Outro avanço para o Switch seria a implementação de uma camada de abstração entre o mecanismo de tradução e o banco de modo que o usuário não precise se preocupar com o armazenamento dos dados. Além disso, uma API para facilitar a integração do Switch a outros softwares também poderia ser desenvolvida.

[1] https://www.java.com/pt-BR/

REFERÊNCIAS

BARRASA, Jesús. **Importing RDF data into Neo4j**. [*S.l.*: *s.n.*], fev. 2021. Disponível em: <https://jbarrasa.com/2016/06/07/importing-rdf-data-into-neo4j/>. Citado na p. 23.

BERNERS-LEE, Tim. **Principles of Design**. [*S.l.*: *s.n.*], 1998. Disponível em: <https://www.w3.org/DesignIssues/Principles.html>. Citado na p. 13.

BERNERS-LEE, Tim *et al.* **Relational Databases on the Semantic Web**. [*S.l.*: *s.n.*], set. 1998. Disponível em: <https://www.w3.org/DesignIssues/RDB-RDF>. Citado na p. 14.

BERNERS-LEE, Tim; CAILLIAU, Robert; GROFF, Jean-François. The World-Wide Web. **Comput. Networks ISDN Syst.**, v. 25, n. 4-5, p. 454–459, 1992. DOI: 10.1016/0169-7552(92)90039-S. Disponível em: <https://doi.org/10.1016/0169-7552(92)90039-S>. Citado na p. 13.

BERNERS-LEE, Tim; LASSILA, Ora; HENDLER, James. The Semantic Web. **The Scientific America**, p. 28–37, mai. 2001. Citado nas pp. 13, 15, 17.

BIZER, Christian; HEATH, Tom; BERNERS-LEE, Tim. Linked Data - The Story So Far. **Int. J. Semantic Web Inf. Syst.**, v. 5, n. 3, p. 1–22, 2009. DOI: 10.4018/jswis.2009081901. Disponível em: <https://doi.org/10.4018/jswis.2009081901>. Citado na p. 13.

FATHY, Naglaa *et al.* Querying Heterogeneous Property Graph Data Sources Based on a Unified Conceptual View. *In:* PROCEEDINGS of the 2020 9th International Conference on Software and Information Engineering (ICSIE). Cairo, Egypt: Association for Computing Machinery, 2020. (ICSIE 2020), p. 113–118. DOI: 10.1145/3436829.3436855. Disponível em: <https://doi.org/10.1145/3436829.3436855>. Citado nas pp. 28, 32, 34.

KITCHENHAM, Barbara. **Procedures for Performing Systematic Reviews**. Department of Computer Science, Keele University, UK, 2004. Citado na p. 27.

LADWIG, Gunter; HARTH, Andreas. CumulusRDF: Linked Data Management on Nested Key-Value Stores. **SSWS 2011**, 2011. Citado na p. 17.

LOMBARDOT, Thierry *et al.* Updates in Rhea: SPARQLing biochemical reaction data. **Nucleic Acids Res.**, v. 47, Database-Issue, p. d596–d600, 2019. DOI: 10.1093/nar/gky876. Disponível em: <https://doi.org/10.1093/nar/gky876>. Citado nas pp. 26, 28–31, 54.

MICHEL, Franck *et al.* Translation of Heterogeneous Databases into RDF, and Application to the Construction of a SKOS Taxonomical Reference. *In:* MONFORT, Valérie *et al.* (Ed.). **Web Information Systems and Technologies**. Cham: Springer International Publishing, 2016. P. 275–296. Citado na p. 32.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. An analysis of mapping strategies for storing RDF data into NoSQL databases. *In:* HUNG, Chih-Cheng *et al.* (Ed.). **SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020**. [*S.l.*]: ACM, 2020. P. 386–392. DOI: 10.1145/3341105.3375753. Disponível em: <https://doi.org/10.1145/3341105.3375753>. Citado na p. 15.

SANTANA, Luiz Henrique Zambom; SANTOS MELLO, Ronaldo dos. Workload-Aware RDF Partitioning and SPARQL Query Caching for Massive RDF Graphs stored in NoSQL Databases. *In:* HARA, Carmem S. (Ed.). **XXXII Simpósio Brasileiro de Banco de Dados, Uberlandia, MG, Brazil, October 4-7, 2017**. [*S.l.*]: SBC, 2017. P. 184–195. Disponível em: <http://sbbd.org.br/2017/wp-content/uploads/sites/3/2018/02/p184-195.pdf>. Citado nas pp. 14, 21, 28, 32.

SHADBOLT, Nigel; BERNERS-LEE, Tim; HALL, Wendy. The Semantic Web Revisited. **IEEE Intell. Syst.**, v. 21, n. 3, p. 96–101, 2006. DOI: 10.1109/MIS.2006.62. Disponível em: <https://doi.org/10.1109/MIS.2006.62>. Citado na p. 15.

THAKKAR, Harsh *et al.* Two for One: Querying Property Graph Databases Using SPARQL via Gremlinator. *In:* PROCEEDINGS of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences amp; Systems (GRADES) and Network Data Analytics (NDA). Houston, Texas: Association for Computing Machinery, 2018. (GRADES-NDA '18). DOI: 10.1145/3210259.3210271. Disponível em: <https://doi.org/10.1145/3210259.3210271>. Citado nas pp. 28, 29, 34.

THORSTEN. **Neo4j: A Reasonable RDF Graph Database Reasoning Engine [Community Post]**. [*S.l.*: *s.n.*], out. 2018. Disponível em:

<https://neo4j.com/blog/neo4j-rdf-graph-database-reasoning-engine/>. Citado na p. 15.

Apêndices

APÊNDICE A – GRAMÁTICA SELECT SPARQL

```
Query                    ::= Prologue SelectQuery
Prologue                 ::= BaseDecl Prologue
Prologue                 ::= PrefixDecl Prologue
Prologue                 ::= &
BaseDecl                 ::= 'BASE' IRIREF
PrefixDecl               ::= 'PREFIX' PNAME_NS IRIREF
SelectQuery              ::= SelectClause WhereClause SolutionModifier
SelectClause             ::= 'SELECT' SelectClauseAux1
SelectClauseAux1         ::= SelectClauseAux2
SelectClauseAux1         ::= 'DISTINCT' SelectClauseAux2
SelectClauseAux1         ::= 'REDUCED' SelectClauseAux2
SelectClauseAux2         ::= '*'
SelectClauseAux2         ::= Var SelectClauseAux3
SelectClauseAux2         ::= '(' Expression 'AS' Var ')' SelectClauseAux3
SelectClauseAux3         ::= Var SelectClauseAux3
SelectClauseAux3         ::= '(' Expression 'AS' Var ')' SelectClauseAux3
SelectClauseAux3         ::= &
WhereClause              ::= GroupGraphPattern
WhereClause              ::= 'WHERE' GroupGraphPattern
SolutionModifier       ::= SolutionModifierAux1 SolutionModifierAux2 SolutionModifierAux3 SolutionModifierAux4
SolutionModifierAux1     ::= &
SolutionModifierAux1     ::= GroupClause
SolutionModifierAux2     ::= &
SolutionModifierAux2     ::= HavingClause
SolutionModifierAux3     ::= &
SolutionModifierAux3     ::= OrderClause
SolutionModifierAux4     ::= &
SolutionModifierAux4     ::= LimitOffsetClauses
GroupClause              ::= 'GROUP' 'BY' GroupCondition GroupClauseAux
GroupClauseAux           ::= &
GroupClauseAux           ::= GroupCondition GroupClauseAux
GroupCondition           ::= BuiltInCall
GroupCondition           ::= (' Expression GroupConditionAux ')'
GroupCondition           ::= Var
GroupConditionAux        ::= 'AS' Var
GroupConditionAux        ::= &
HavingClause             ::= 'HAVING' HavingCondition HavingClauseAux
HavingClauseAux          ::= &
HavingClauseAux          ::= HavingCondition HavingClauseAux
HavingCondition          ::= Constraint
OrderClause              ::= 'ORDER' 'BY' OrderCondition OrderClauseAux
OrderClauseAux           ::= &
OrderClauseAux           ::= OrderCondition OrderClauseAux
OrderCondition           ::= Constraint
OrderCondition           ::= Var
OrderCondition           ::= OrderConditionAux Bracketted Expression
OrderConditionAux        ::= 'ASC'
OrderConditionAux        ::= 'DESC'
LimitOffsetClauses       ::= LimitClause LimitOffsetClausesAux1
LimitOffsetClauses       ::= OffsetClause LimitOffsetClausesAux2
LimitOffsetClausesAux1   ::= &
LimitOffsetClausesAux1   ::= OffsetClause
LimitOffsetClausesAux2   ::= &
LimitOffsetClausesAux2   ::= LimitClause
LimitClause              ::= 'LIMIT' INTEGER
OffsetClause             ::= 'OFFSET' INTEGER
```

```
GroupGraphPattern         ::= '{' GroupGraphPatternSub '}'
GroupGraphPatternSub     ::= GroupGraphPatternSubAux1 GroupGraphPatternSubAux2
GroupGraphPatternSubAux1  ::= TriplesBlock
GroupGraphPatternSubAux1  ::= &
GroupGraphPatternSubAux2  ::= GraphPatternNotTriples GroupGraphPatternSubAux3 GroupGraphPatternSubAux1 GroupGraphPatternSubAux
GroupGraphPatternSubAux2  ::= &
GroupGraphPatternSubAux3  ::= '.'
GroupGraphPatternSubAux3  ::= &
TriplesBlock              ::= TriplesSameSubjectPath TriplesBlockAux1
TriplesBlockAux1          ::=  '.' TriplesBlockAux2
TriplesBlockAux1          ::= &
TriplesBlockAux2          ::= TriplesBlock
TriplesBlockAux2          ::= &
GraphPatternNotTriples    ::= GroupOrUnionGraphPattern
GraphPatternNotTriples    ::= OptionalGraphPattern
GraphPatternNotTriples    ::= MinusGraphPattern
GraphPatternNotTriples    ::= Filter
OptionalGraphPattern      ::= 'OPTIONAL' GroupGraphPattern
MinusGraphPattern         ::= 'MINUS' GroupGraphPattern
GroupOrUnionGraphPattern  ::= GroupGraphPattern GroupOrUnionGraphPatternAux
GroupOrUnionGraphPatternAux ::= 'UNION' GroupGraphPattern GroupOrUnionGraphPatternAux
GroupOrUnionGraphPatternAux ::= &
Filter                    ::= 'FILTER' Constraint
Constraint                ::= BrackettedExpression
Constraint                ::= BuiltInCall
ExpressionList            ::= NIL
ExpressionList            ::= '(' Expression ExpressionListAux ')'
ExpressionListAux         ::= ',' Expression ExpressionListAux
ExpressionListAux         ::= &
PropertyListNotEmpty      ::= Verb ObjectList PropertyListNotEmptyAux2
PropertyListNotEmptyAux1  ::= Verb ObjectList
PropertyListNotEmptyAux1  ::= &
PropertyListNotEmptyAux2  ::= ';' PropertyListNotEmptyAux1 PropertyListNotEmptyAux2
PropertyListNotEmptyAux2  ::= &
Verb                      ::= VarOrIri
Verb                      ::= 'a'
ObjectList                ::= Object ObjectListAux
ObjectListAux             ::= ',' Object ObjectListAux
ObjectListAux             ::= &
Object                    ::= GraphNode
TriplesSameSubjectPath    ::= VarOrTerm PropertyListPathNotEmpty
TriplesSameSubjectPath    ::= TriplesNodePath PropertyListPath
PropertyListPath          ::= PropertyListPathNotEmpty
PropertyListPath          ::= &
PropertyListPathNotEmpty    ::= PropertyListPathNotEmptyAux1 ObjectListPath PropertyListPathNotEmptyAux2
PropertyListPathNotEmptyAux1 ::= VerbPath
PropertyListPathNotEmptyAux1 ::= VerbSimple
PropertyListPathNotEmptyAux2 ::= ';' PropertyListPathNotEmptyAux3
PropertyListPathNotEmptyAux2 ::= &
PropertyListPathNotEmptyAux3 ::= PropertyListPathNotEmptyAux1 ObjectList
PropertyListPathNotEmptyAux3 ::= &
VerbPath                  ::= Path
VerbSimple                ::= Var
ObjectListPath            ::= ObjectPath ObjectListPathAux
ObjectListPathAux         ::= ',' ObjectPath
ObjectListPathAux         ::= &
ObjectPath                ::= GraphNodePath
Path                      ::= PathAlternative
PathAlternative           ::= PathSequence PathAlternativeAux
PathAlternativeAux        ::= '|' PathSequence PathAlternativeAux
PathAlternativeAux        ::= &
```

```
PathSequence              ::= PathEltOrInverse PathSequenceAux
PathSequenceAux           ::= '/' PathEltOrInverse PathSequenceAux
PathSequenceAux           ::= &
PathElt                   ::= PathPrimary PathEltAux
PathEltAux                ::= PathMod
PathEltAux                ::= &
PathEltOrInverse          ::= PathElt
PathEltOrInverse          ::= '^' PathElt
PathMod                   ::= '?'
PathMod                   ::= '*'
PathMod                   ::= '+'
PathPrimary               ::= iri
PathPrimary               ::= 'a'
PathPrimary               ::= '!' PathNegatedPropertySet
PathPrimary               ::= '(' Path ')'
PathNegatedPropertySet    ::= PathOneInPropertySet
PathNegatedPropertySet    ::= '(' PathNegatedPropertySetAux1 ')'
PathNegatedPropertySetAux1   ::= PathOneInPropertySet PathNegatedPropertySetAux2
PathNegatedPropertySetAux1    ::= &
PathNegatedPropertySetAux2   ::= '|' PathOneInPropertySet PathNegatedPropertySetAux2
PathNegatedPropertySetAux2    ::= &
PathOneInPropertySet      ::= iri
PathOneInPropertySet      ::= 'a'
PathOneInPropertySet      ::= '^' PathOneInPropertySetAux
PathOneInPropertySetAux   ::= iri
PathOneInPropertySetAux   ::= 'a'
TriplesNode               ::= Collection
TriplesNode               ::= BlankNodePropertyList
BlankNodePropertyList     ::= '[' PropertyListNotEmpty ']'
TriplesNodePath           ::= CollectionPath
TriplesNodePath           ::= BlankNodePropertyListPath
BlankNodePropertyListPath ::= '[' PropertyListPathNotEmpty ']'
Collection                ::= '(' GraphNode CollectionAux ')'
CollectionAux             ::= GraphNode CollectionAux
CollectionAux             ::= &
CollectionPath            ::= '(' GraphNodePath CollectionPathAux ')'
CollectionPathAux         ::= GraphNodePath CollectionPathAux
CollectionPathAux         ::= &
GraphNode                 ::= TriplesNode
GraphNode                 ::= VarOrTerm
GraphNodePath             ::= VarOrTerm
GraphNodePath             ::= TriplesNodePath
VarOrTerm                 ::= Var
VarOrTerm                 ::= GraphTerm
VarOrIri                  ::= Var
VarOrIri                  ::= iri
Var                       ::= VAR1
Var                       ::= VAR2
GraphTerm                 ::= iri
GraphTerm                 ::= RDFLiteral
GraphTerm                 ::= NumericLiteral
GraphTerm                 ::= BooleanLiteral
GraphTerm                 ::= BlankNode
GraphTerm                 ::= NIL
Expression                ::= ConditionalOrExpression
ConditionalOrExpression  ::= ConditionalAndExpression ConditionalOrExpressionAux
ConditionalOrExpressionAux ::= '||' ConditionalAndExpression ConditionalOrExpressionAux
ConditionalOrExpressionAux ::= &
ConditionalAndExpression  ::= ValueLogical ConditionalAndExpressionAux
ConditionalAndExpressionAux ::= '&&' ValueLogical ConditionalAndExpressionAux
ConditionalAndExpressionAux ::= &
ValueLogical              ::= RelationalExpression
```

```
RelationalExpression      ::= NumericExpression RelationalExpressionAux
RelationalExpressionAux   ::= '=' NumericExpression
RelationalExpressionAux   ::= '!=' NumericExpression
RelationalExpressionAux   ::= '<' NumericExpression
RelationalExpressionAux   ::= '>' NumericExpression
RelationalExpressionAux   ::= '<=' NumericExpression
RelationalExpressionAux   ::= '>=' NumericExpression
RelationalExpressionAux   ::= 'IN' ExpressionList
RelationalExpressionAux   ::= 'NOT' 'IN' ExpressionList
RelationalExpressionAux   ::= &
NumericExpression         ::= AdditiveExpression
AdditiveExpression       ::= MultiplicativeExpression AdditiveExpressionAux1
AdditiveExpressionAux1   ::= '+' MultiplicativeExpression AdditiveExpressionAux1
AdditiveExpressionAux1   ::= '-' MultiplicativeExpression AdditiveExpressionAux1
AdditiveExpressionAux1    ::= &
MultiplicativeExpression  ::= UnaryExpression MultiplicativeExpressionAux
MultiplicativeExpressionAux ::= '*' UnaryExpression MultiplicativeExpressionAux
MultiplicativeExpressionAux ::= '/' UnaryExpression MultiplicativeExpressionAux
MultiplicativeExpressionAux ::= &
UnaryExpression           ::= '!' PrimaryExpression
UnaryExpression           ::= '+' PrimaryExpression
UnaryExpression           ::= '-' PrimaryExpression
UnaryExpression           ::= PrimaryExpression
PrimaryExpression         ::= BrackettedExpression
PrimaryExpression         ::= BuiltInCall
PrimaryExpression         ::= iri
PrimaryExpression         ::= RDFLiteral
PrimaryExpression         ::= NumericLiteral
PrimaryExpression         ::= BooleanLiteral
PrimaryExpression         ::= Var
BrackettedExpression      ::= '(' Expression ')'
BuiltInCall               ::= Aggregate
BuiltInCall               ::= 'RAND' NIL
BuiltInCall               ::= 'ABS' '(' Expression ')'
BuiltInCall               ::= 'CEIL' '(' Expression ')'
BuiltInCall               ::= 'FLOOR' '(' Expression ')'
BuiltInCall               ::= 'ROUND' '(' Expression ')'
BuiltInCall               ::= 'CONCAT' ExpressionList
BuiltInCall               ::= SubstringExpression
BuiltInCall               ::= 'STRLEN' '(' Expression ')'
BuiltInCall               ::= StrReplaceExpression
BuiltInCall               ::= 'UCASE' '(' Expression ')'
BuiltInCall               ::= 'LCASE' '(' Expression ')'
BuiltInCall               ::= 'CONTAINS' '(' Expression ',' Expression ')'
BuiltInCall               ::= 'STRSTARTS' '(' Expression ',' Expression ')'
BuiltInCall               ::= 'STRENDS' '(' Expression ',' Expression ')'
BuiltInCall               ::= 'YEAR' '(' Expression ')'
BuiltInCall               ::= 'MONTH' '(' Expression ')'
BuiltInCall               ::= 'DAY' '(' Expression ')'
BuiltInCall               ::= 'HOURS' '(' Expression ')'
BuiltInCall               ::= 'MINUTES' '(' Expression ')'
BuiltInCall               ::= 'SECONDS' '(' Expression ')'
BuiltInCall               ::= 'TIMEZONE' '(' Expression ')'
BuiltInCall               ::= 'TZ' '(' Expression ')'
BuiltInCall               ::= 'NOW' NIL
BuiltInCall               ::= 'COALESCE' ExpressionList
BuiltInCall               ::= RegexExpression
RegexExpression        ::= 'REGEX' '(' Expression ',' Expression RegexExpressionAux ')'
RegexExpressionAux        ::= ',' Expression
RegexExpressionAux        ::= &
```

```
SubstringExpression     ::= 'SUBSTR' '(' Expression ',' Expression SubstringExpressionAux ')'
SubstringExpressionAux    ::= ',' Expression
SubstringExpressionAux    ::= &
StrReplaceExpression    ::= 'REPLACE' '(' Expression ',' Expression ',' Expression StrReplaceExpressionAux ')'
StrReplaceExpressionAux   ::= ',' Expression
StrReplaceExpressionAux   ::= &
Aggregate                 ::= 'COUNT' '(' AggregateAux1 AggregateAux2 ')'
Aggregate                 ::= 'SUM' '(' AggregateAux1 Expression ')'
Aggregate                 ::= 'MIN' '(' AggregateAux1 Expression ')'
Aggregate                 ::= 'MAX' '(' AggregateAux1 Expression ')'
Aggregate                 ::= 'AVG' '(' AggregateAux1 Expression ')'
AggregateAux1             ::= 'DISTINCT'
AggregateAux1             ::= &
AggregateAux2             ::= '*'
AggregateAux2             ::= Expression
RDFLiteral                ::= String RDFLiteralAux1
RDFLiteralAux1            ::= LANGTAG
RDFLiteralAux1            ::= '^^' iri
RDFLiteralAux1            ::= &
NumericLiteral            ::= NumericLiteralUnsigned
NumericLiteral            ::= NumericLiteralPositive
NumericLiteral            ::= NumericLiteralNegative
NumericLiteralUnsigned    ::= INTEGER
NumericLiteralUnsigned    ::= DECIMAL
NumericLiteralUnsigned    ::= DOUBLE
NumericLiteralPositive    ::= INTEGER_POSITIVE
NumericLiteralPositive    ::= DECIMAL_POSITIVE
NumericLiteralPositive    ::= DOUBLE_POSITIVE
NumericLiteralNegative    ::= INTEGER_NEGATIVE
NumericLiteralNegative    ::= DECIMAL_NEGATIVE
NumericLiteralNegative    ::= DOUBLE_NEGATIVE]
BooleanLiteral            ::= 'true'
BooleanLiteral            ::= 'false'
String                    ::= STRING_LITERAL1
String                    ::= STRING_LITERAL2
String                    ::= STRING_LITERAL_LONG1
String                    ::= STRING_LITERAL_LONG2
iri                       ::= IRIREF
iri                       ::= PrefixedName
PrefixedName              ::= PNAME_LN
PrefixedName              ::= PNAME_NS
BlankNode                 ::= BLANK_NODE_LABEL
BlankNode                 ::= ANON
```

APÊNDICE B – DEFINIÇÃO DOS TOKENS DA SELECT SPARQL

```
IRI_REF           ::= '<' ([^<>"{}|^`\]-[#x00-#x20])* '>'
PNAME_NS          ::= PN_PREFIX? ':'
PNAME_LN          ::= PNAME_NS PN_LOCAL
BLANK_NODE_LABEL  ::= '_:' PN_LOCAL
VAR1              ::= '?' VARNAME
VAR2              ::= '$' VARNAME
LANGTAG           ::= '@' [a-zA-Z]+ ('-' [a-zA-Z0-9]+)*
INTEGER           ::= [0-9]+
DECIMAL           ::= [0-9]+ '.' [0-9]* | '.' [0-9]+
DOUBLE         ::= [0-9]+ '.' [0-9]* EXPONENT | '.' ([0-9])+ EXPONENT | ([0-9])+ EXPONENT
INTEGER_POSITIVE  ::= '+' INTEGER
DECIMAL_POSITIVE  ::= '+' DECIMAL
DOUBLE_POSITIVE   ::= '+' DOUBLE
INTEGER_NEGATIVE  ::= '-' INTEGER
DECIMAL_NEGATIVE  ::= '-' DECIMAL
DOUBLE_NEGATIVE   ::= '-' DOUBLE
EXPONENT          ::= [eE] [+-]? [0-9]+
STRING_LITERAL1   ::= "'" ( ([^#x27#x5C#xA#xD]) | ECHAR )* "'"
STRING_LITERAL2   ::= '"' ( ([^#x22#x5C#xA#xD]) | ECHAR )* '"'
STRING_LITERAL_LONG1 ::= "'''" ( ( "'" | "''" )? ( [^'\] | ECHAR ) )* "'''"
STRING_LITERAL_LONG2 ::= '"""' ( ( '"' | '""' )? ( [^"\] | ECHAR ) )* '"""'
ECHAR             ::= '\' [tbnrf\"']
NIL               ::= '(' WS* ')'
WS                ::= #x20 | #x9 | #xD | #xA
ANON              ::= '[' WS* ']'
PN_CHARS_BASE     ::= [A-Z] | [a-z]
                    | [#x00C-1-#x00D6]
                    | [#x00D8-#x00F6]
                    | [#x00F8-#x02FF]
                    | [#x0370-#x037D]
                    | [#x037F-#x1FFF]
                    | [#x200C-#x200D]
                    | [#x2070-#x218F]
                    | [#x2C00-#x2FEF]
                    | [#x3001-#xD7FF]
                    | [#xF900-#xFDCF]
                    | [#xFDF0-#xFFFD]
                    | [#x10000-#xEFFFF]
PN_CHARS_U        ::= PN_CHARS_BASE | '_'
VARNAME         ::= ( PN_CHARS_U | [0-9] ) ( PN_CHARS_U | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040] )*
PN_CHARS        ::= PN_CHARS_U | '-' | [0-9] | #x00B7 | [#x0300-#x036F] | [#x203F-#x2040]
PN_PREFIX         ::= PN_CHARS_BASE ((PN_CHARS|'.')* PN_CHARS)?
PN_LOCAL          ::= ( PN_CHARS_U | [0-9] ) ((PN_CHARS|'.')* PN_CHARS)?
```

APÊNDICE C – DERIVAÇÕES A PARTIR DE EXPRESSION

```
Expression                ::= ConditionalOrExpression
ConditionalOrExpression   ::= ConditionalAndExpression
                              ( '||' ConditionalAndExpression )*
ConditionalAndExpression  ::= ValueLogical ( '&&' ValueLogical )*
ValueLogical              ::= RelationalExpression
RelationalExpression      ::= NumericExpression
                              ( '=' NumericExpression
                               | '!=' NumericExpression
                               | '<' NumericExpression
                               | '>' NumericExpression
                               | '<=' NumericExpression
                               | '>=' NumericExpression
                               | 'IN' ExpressionList
                               | 'NOT' 'IN' ExpressionList )?
NumericExpression         ::= AdditiveExpression
AdditiveExpression        ::= MultiplicativeExpression
                              ( '+' MultiplicativeExpression
                               | '-' MultiplicativeExpression )*
MultiplicativeExpression  ::= UnaryExpression
                              ( '*' UnaryExpression
                               | '/' UnaryExpression )*
UnaryExpression           ::= '!' PrimaryExpression
                            | '+' PrimaryExpression
                            | '-' PrimaryExpression
                            | PrimaryExpression
PrimaryExpression         ::= BrackettedExpression
                            | BuiltInCall
                            | iri
                            | RDFLiteral
                            | NumericLiteral
                            | BooleanLiteral
                            | Var

BrackettedExpression      ::= '(' Expression ')'
RDFLiteral                ::= String ( LANGTAG | ( '^^' iri ) )?
NumericLiteral            ::= NumericLiteralUnsigned
                            | NumericLiteralPositive
                            | NumericLiteralNegative
NumericLiteralUnsigned    ::= INTEGER | DECIMAL | DOUBLE
NumericLiteralPositive    ::= INTEGER_POSITIVE
                            | DECIMAL_POSITIVE
                            | DOUBLE_POSITIVE
NumericLiteralNegative    ::= INTEGER_NEGATIVE
                            | DECIMAL_NEGATIVE
                            | DOUBLE_NEGATIVE
BooleanLiteral            ::= 'true' | 'false'
String                    ::= STRING_LITERAL1
                            | STRING_LITERAL2
                            | STRING_LITERAL_LONG1
                            | STRING_LITERAL_LONG2
iri                       ::= IRIREF | PrefixedName
PrefixedName              ::= PNAME_LN | PNAME_NS
BlankNode                 ::= BLANK_NODE_LABEL | ANON
```

APÊNDICE D – CÓDIGO FONTE

Neste apêndice é apresentado o código fonte em Python. O código completo com definição de dependências e testes pode ser encontrado no seguinte link: https://github.com/shthiago/switch/releases/tag/v0.1.1-alpha.

transpiler/expression_handler.py

```python
from typing import List

from transpiler.structures.nodes.expression import (
    AdditiveExpression,
    AdditiveOperator,
    AndExpression,
    BuiltInFunction,
    LogOperator,
    MultiplicativeExpression,
    MultiplicativeOperator,
    OrExpression,
    PrimaryExpression,
    PrimaryType,
    RelationalExpression,
    UnaryExpression,
    UnaryOperator,
)


class ExpressionHandler:
    def __init__(self):
        self.builtin_gen_map = {
            "COUNT": self._gen_call_for_count,
            "SUM": self._gen_call_for_sum,
            "MIN": self._gen_call_for_min,
            "MAX": self._gen_call_for_max,
            "AVG": self._gen_call_for_avg,
            "STRLEN": self._gen_call_for_strlen,
            "SUBSTR": self._gen_call_for_substr,
            "UCASE": self._gen_call_for_ucase,
            "LCASE": self._gen_call_for_lcase,
            "STRSTARTS": self._gen_call_for_strstarts,
            "STRENDS": self._gen_call_for_strends,
            "CONTAINS": self._gen_call_for_contains,
            "CONCAT": self._gen_call_for_concat,
            "REGEX": self._gen_call_for_regex,
            "REPLACE": self._gen_call_for_replace,
            "COALESCE": self._gen_call_for_coalesce,
            "ABS": self._gen_call_for_abs,
            "ROUND": self._gen_call_for_round,
```

```
41              "CEIL": self._gen_call_for_ceil,
42              "FLOOR": self._gen_call_for_floor,
43              "RAND": self._gen_call_for_rand,
44              "NOW": self._gen_call_for_now,
45              "YEAR": self._gen_call_for_year,
46              "MONTH": self._gen_call_for_month,
47              "DAY": self._gen_call_for_day,
48              "HOURS": self._gen_call_for_hours,
49              "MINUTES": self._gen_call_for_minutes,
50              "SECONDS": self._gen_call_for_seconds,
51              "TIMEZONE": self._gen_call_for_timezone,
52              "TZ": self._gen_call_for_tz,
53          }
54
55      def value_orexpression(self, node: OrExpression) -> str:
56          base_value = self.value_andexpression(node.base)
57          if node.others:
58              rest = " OR " + " OR ".join(
59                  [f"({self.value_andexpression(n)})" for n in node.others]
60              )
61              return f"{base_value}" + rest
62
63          return base_value
64
65      def value_andexpression(self, node: AndExpression) -> str:
66          base_value = self.value_relationalexpression(node.base)
67          if node.others:
68              rest = " AND " + " AND ".join(
69                  [f"({self.value_relationalexpression(n)})" for n in
                                                        node.others]
70              )
71              return f"({base_value})" + rest
72
73          return base_value
74
75      def value_relationalexpression(self, node: RelationalExpression) ->
                                              str:
76          first = self.value_additiveexpression(node.first)
77
78          if node.second:
79              op, value = node.second
80              second = self.value_additiveexpression(value)
81              fmt_string = self._log_operator_fmt_str(op)
82
83              return f"({fmt_string.format(first, second)})"
84
85          return first
86
```

```
87      def value_additiveexpression(self, node: AdditiveExpression) -> str:
88          value = self.value_multiplicativeexpression(node.base)
89          if not node.others:
90              return value
91
92          for op, exp in node.others:
93              op_str = self._add_operator_str(op)
94              exp_str = self.value_multiplicativeexpression(exp)
95              value += f" {op_str} {exp_str}"
96
97          return f"({value})"
98
99      def value_multiplicativeexpression(self, node:
                                           MultiplicativeExpression) -> str:
100         value = self.value_unaryexpression(node.base)
101         if not node.others:
102             return value
103
104         for op, exp in node.others:
105             op_str = self._mult_operator_str(op)
106             exp_str = self.value_unaryexpression(exp)
107             value += f" {op_str} {exp_str}"
108
109         return f"({value})"
110
111     def value_unaryexpression(self, node: UnaryExpression) -> str:
112         value = self.value_primaryexpression(node.value)
113         if not node.op:
114             return value
115
116         op = self._unary_operator_str(node.op)
117
118         return f"({op}{value})"
119
120     def value_primaryexpression(self, node: PrimaryExpression) -> str:
121         if self.is_literal(node) or self.is_iri(node):
122             return node.value
123
124         if self.is_var(node):
125             return node.value.replace("?", "")
126
127         if self.is_builtinfunction(node):
128             return self.builtinfunction_to_cypher(node.value)
129
130         return self.value_orexpression(node.value)
131
132     def builtinfunction_to_cypher(self, call: BuiltInFunction) -> str:
133         return self.builtin_gen_map[call.name](call.params)
```

```
134
135     def _gen_call_for_count(self, params: List[OrExpression]):
136         arg = self.value_orexpression(params[0])
137
138         return f"count({arg})"
139
140     def _gen_call_for_sum(self, params: List[OrExpression]):
141         arg = self.value_orexpression(params[0])
142
143         return f"sum({arg})"
144
145     def _gen_call_for_min(self, params: List[OrExpression]):
146         arg = self.value_orexpression(params[0])
147
148         return f"min({arg})"
149
150     def _gen_call_for_max(self, params: List[OrExpression]):
151         arg = self.value_orexpression(params[0])
152
153         return f"max({arg})"
154
155     def _gen_call_for_avg(self, params: List[OrExpression]):
156         arg = self.value_orexpression(params[0])
157
158         return f"avg({arg})"
159
160     def _gen_call_for_strlen(self, params: List[OrExpression]):
161         raise NotImplementedError
162
163     def _gen_call_for_substr(self, params: List[OrExpression]):
164         raise NotImplementedError
165
166     def _gen_call_for_ucase(self, params: List[OrExpression]):
167         raise NotImplementedError
168
169     def _gen_call_for_lcase(self, params: List[OrExpression]):
170         raise NotImplementedError
171
172     def _gen_call_for_strstarts(self, params: List[OrExpression]):
173         raise NotImplementedError
174
175     def _gen_call_for_strends(self, params: List[OrExpression]):
176         raise NotImplementedError
177
178     def _gen_call_for_contains(self, params: List[OrExpression]):
179         raise NotImplementedError
180
181     def _gen_call_for_concat(self, params: List[OrExpression]):
```

```
182         raise NotImplementedError
183
184     def _gen_call_for_regex(self, params: List[OrExpression]):
185         raise NotImplementedError
186
187     def _gen_call_for_replace(self, params: List[OrExpression]):
188         raise NotImplementedError
189
190     def _gen_call_for_coalesce(self, params: List[OrExpression]):
191         raise NotImplementedError
192
193     def _gen_call_for_abs(self, params: List[OrExpression]):
194         raise NotImplementedError
195
196     def _gen_call_for_round(self, params: List[OrExpression]):
197         raise NotImplementedError
198
199     def _gen_call_for_ceil(self, params: List[OrExpression]):
200         raise NotImplementedError
201
202     def _gen_call_for_floor(self, params: List[OrExpression]):
203         raise NotImplementedError
204
205     def _gen_call_for_rand(self, params: List[OrExpression]):
206         raise NotImplementedError
207
208     def _gen_call_for_now(self, params: List[OrExpression]):
209         return "datetime()"
210
211     def _gen_call_for_year(self, params: List[OrExpression]):
212         raise NotImplementedError
213
214     def _gen_call_for_month(self, params: List[OrExpression]):
215         raise NotImplementedError
216
217     def _gen_call_for_day(self, params: List[OrExpression]):
218         raise NotImplementedError
219
220     def _gen_call_for_hours(self, params: List[OrExpression]):
221         raise NotImplementedError
222
223     def _gen_call_for_minutes(self, params: List[OrExpression]):
224         raise NotImplementedError
225
226     def _gen_call_for_seconds(self, params: List[OrExpression]):
227         raise NotImplementedError
228
229     def _gen_call_for_timezone(self, params: List[OrExpression]):
```

```
230         raise NotImplementedError
231
232     def _gen_call_for_tz(self, params: List[OrExpression]):
233         raise NotImplementedError
234
235     def _unary_operator_str(self, op: UnaryOperator) -> str:
236         match op:
237             case UnaryOperator.PLUS:
238                 return "+"
239             case UnaryOperator.MINUS:
240                 return "-"
241             case UnaryOperator.NOT:
242                 return "NOT "
243
244     def _mult_operator_str(self, op: MultiplicativeOperator) -> str:
245         match op:
246             case MultiplicativeOperator.MULT:
247                 return "*"
248             case MultiplicativeOperator.DIV:
249                 return "/"
250
251     def _add_operator_str(self, op: AdditiveOperator) -> str:
252         match op:
253             case AdditiveOperator.SUM:
254                 return "+"
255             case AdditiveOperator.SUB:
256                 return "-"
257
258     def _log_operator_fmt_str(self, op: LogOperator) -> str:
259         match op:
260             case LogOperator.EQ:
261                 return "{} = {}"
262             case LogOperator.NEQ:
263                 return "{} <> {}"
264             case LogOperator.LT:
265                 return "{} < {}"
266             case LogOperator.GT:
267                 return "{} > {}"
268             case LogOperator.LTE:
269                 return "{} <= {}"
270             case LogOperator.GTE:
271                 return "{} >= {}"
272             case LogOperator.IN:
273                 return "{} IN {}"
274             case LogOperator.NOT_IN:
275                 return "NOT {} IN {}"
276
277     def is_literal(self, node: PrimaryExpression) -> bool:
```

```
278         literal_types = [
279             PrimaryType.NUM_LITERAL,
280             PrimaryType.STR_LITERAL,
281             PrimaryType.BOOL_LITERAL,
282         ]
283 
284         return node.type in literal_types
285 
286     def is_builtinfunction(self, node: PrimaryExpression) -> bool:
287         return node.type == PrimaryType.FUNC
288 
289     def is_iri(self, node: PrimaryExpression) -> bool:
290         return node.type == PrimaryType.IRI
291 
292     def is_var(self, node: PrimaryExpression) -> bool:
293         return node.type == PrimaryType.VAR
```

transpiler/cypher_generator.py

```
1  from enum import Enum, auto
2  from typing import Any, List, Optional
3 
4  from loguru import logger
5 
6  from transpiler.structures.nodes.expression import BuiltInFunction,
                                                      OrExpression
7  from transpiler.structures.nodes.modifiers import ModifiersNode, OrderNode
8  from transpiler.structures.nodes.namespace import Namespace
9  from transpiler.structures.nodes.variables import SelectedVar
10 
11 from .expression_handler import ExpressionHandler
12 from .parser import SelectSparqlParser
13 from .structures.nodes import Triple
14 from .structures.query import GraphPattern, Query
15 
16 
17 class CypherGenerationException(Exception):
18     """Failure on a generation process"""
19 
20 
21 class TriplePartType(Enum):
22     URI = auto()
23     VAR = auto()
24     LIT = auto()
25 
26 
27 class CypherGenerator:
28     def __init__(self):
```

```
29          self.parser = SelectSparqlParser()
30
31          self.used_variables: List[str] = []
32
33          self.expression_handler = ExpressionHandler()
34
35      def reset_variables(self):
36          self.used_variables = []
37
38      def get_triple_part_type(self, part: Any) -> TriplePartType:
39          if not isinstance(part, str):
40              return TriplePartType.LIT
41
42          if part[0] == "?":
43              return TriplePartType.VAR
44
45          elif ":" in part:
46              return TriplePartType.URI
47
48          return TriplePartType.LIT
49
50      def ns_uri(self, abbev_uri: str) -> str:
51          nms, name = abbev_uri.split(":")
52          full_nms = self.namespaces[nms]
53
54          return f'n10s.rdf.shortFormFromFullUri("{full_nms}") + "{name}"'
55
56      def cypher_var_for(self, triple_part: str) -> str:
57          part_type = self.get_triple_part_type(triple_part)
58          if part_type == TriplePartType.VAR:
59              var = triple_part[1:]
60
61          elif part_type == TriplePartType.URI:
62              var = triple_part.replace(":", "_")
63
64          else:
65              raise CypherGenerationException(
66                  f"Cannot generate var for a literal: {triple_part}"
67              )
68
69          return var
70
71      def case_property_where_clause(self, triple: Triple) -> str:
72          obj_type = self.get_triple_part_type(triple.object)
73          if obj_type == TriplePartType.URI:
74              return ""
75
76          pred_type = self.get_triple_part_type(triple.predicate)
```

```
77
78          filters = []
79
80          if pred_type == TriplePartType.URI:
81              filters.append(f"key = {self.ns_uri(triple.predicate)}")
82
83          if obj_type == TriplePartType.LIT:
84              sub_var = self.cypher_var_for(triple.subject)
85              filters.append(f'{sub_var}[key] = "{triple.object}"')
86
87          if not filters:
88              return ""
89
90          base = "WHERE "
91
92          return base + " AND ".join(filters) + " "
93
94      def case_object_where_clause(self, triple: Triple) -> str:
95          obj_type = self.get_triple_part_type(triple.object)
96          if obj_type == TriplePartType.LIT:
97              return ""
98
99          pred_type = self.get_triple_part_type(triple.predicate)
100
101         filters = []
102         if pred_type == TriplePartType.URI:
103             filters.append(f"type(_relation) =
                                                {self.ns_uri(triple.predicate)}")
104
105         if obj_type == TriplePartType.URI:
106             obj_var = self.cypher_var_for(triple.object)
107             filters.append(f"{obj_var}.uri =
                                                {self.full_uri(triple.object)}")
108
109         if not filters:
110             return ""
111
112         base = "WHERE "
113
114         return base + " AND ".join(filters) + " "
115
116     def filter_case_property(self, triple: Triple) -> Optional[str]:
117         obj_type = self.get_triple_part_type(triple.object)
118
119         if obj_type == TriplePartType.URI:
120             return None
121
122         where_clause = self.case_property_where_clause(triple)
```

```
123         subject = self.cypher_var_for(triple.subject)
124
125         return (
126             f"[key in keys({subject}) {where_clause}| [{subject}, key,
                                                    {subject}[key]]]"
127         )
128
129     def filter_case_object(self, triple: Triple) -> Optional[str]:
130         object_type = self.get_triple_part_type(triple.object)
131
132         if object_type == TriplePartType.LIT:
133             return None
134
135         where_clause = self.case_object_where_clause(triple)
136         subject = self.cypher_var_for(triple.subject)
137         obj = self.cypher_var_for(triple.object)
138
139         return f"[({subject})-[_relation]-({obj}) {where_clause}|
                                            [{subject}, _relation,
                                            {obj}]]"
140
141     def with_clause_rdf_type(self, triple: Triple) -> str:
142         self.used_variables.append(self.cypher_var_for(triple.subject))
143
144         with_parts: List[str] = []
145
146         for var in reversed(self.used_variables):
147             with_parts.insert(0, f"{var} AS {var}")
148
149         return "WITH " + ", ".join(with_parts)
150
151     def with_clause(self, triple: Triple) -> Optional[str]:
152         parts = [triple.subject, triple.predicate, triple.object]
153         types = list(map(self.get_triple_part_type, parts))
154
155         used: List[str] = []
156         with_parts: List[str] = []
157         for i, (part, type_) in enumerate(zip(parts, types)):
158             if type_ == TriplePartType.VAR:
159                 varname = self.cypher_var_for(part)
160                 used.append(varname)
161                 with_parts.append(f"triples[{i}] AS {varname}")
162
163         for var in reversed(self.used_variables):
164             if var not in used:
165                 with_parts.insert(0, f"{var} AS {var}")
166
167         for used_var in used:
```

```
168             if used_var not in self.used_variables:
169                 self.used_variables.append(used_var)
170 
171         if not with_parts:
172             return None
173 
174         return "WITH " + ", ".join(with_parts)
175 
176     def match_clause(self, triple: Triple) -> str:
177         subject_type = self.get_triple_part_type(triple.subject)
178 
179         if subject_type == TriplePartType.LIT:
180             raise CypherGenerationException("Subject cannot be a literal")
181 
182         sub_var = self.cypher_var_for(triple.subject)
183 
184         if sub_var in self.used_variables:
185             return ""
186 
187         clause = f"MATCH ({sub_var})"
188 
189         if self.is_rdf_type(triple):
190             clause += f" WHERE {self.ns_uri(triple.object)} IN
                                                labels({sub_var})"
191 
192         elif subject_type == TriplePartType.URI:
193             clause += f" WHERE {sub_var}.uri =
                                                {self.full_uri(triple.subject)}"
194 
195         return clause
196 
197     def is_rdf_type(self, triple: Triple) -> bool:
198         return triple.predicate == "rdf:type"
199 
200     def full_uri(self, uri: str) -> str:
201         abbrev, name = uri.split(":")
202         base = self.namespaces[abbrev]
203 
204         return f'"{base + name}"'
205 
206     def return_clause(self, query: Query) -> str:
207         variables = query.returning
208         return_parts = []
209         for var in variables:
210             if var.alias is None:
211                 suffix = ""
212 
213             else:
```

```
214                 suffix = f" AS {self.cypher_var_for(var.alias)}"
215
216             if isinstance(var.value, str):
217                 if var.value == "*":
218                     value = "*"
219                 else:
220                     value = self.cypher_var_for(var.value)
221
222             else:
223                 value =
                                                    self.expression_handler.value_or
224
225             return_parts.append(f"{value}{suffix}")
226
227         return "RETURN " + ", ".join(return_parts)
228
229     def unwind_clause(self, triple: Triple) -> str:
230         cases = [
231             self.filter_case_property(triple),
232             self.filter_case_object(triple),
233         ]
234         cases = list(filter(lambda k: k is not None, cases))
235
236         return "UNWIND " + " + ".join(cases) + " AS triples"
237
238     def result_modifier(self, modifiers: ModifiersNode) -> str:
239         """Given a result modifiers node, generate the code block"""
240         mods = []
241
242         if modifiers.order:
243             mods.append(self.order_by_clause(modifiers.order))
244
245         if modifiers.offset:
246             mods.append(self.skip_clause(modifiers.offset))
247
248         if modifiers.limit:
249             mods.append(self.limit_clause(modifiers.limit))
250
251         if not mods:
252             return ""
253
254         return "\n".join(mods)
255
256     def order_by_clause(self, order_node: OrderNode) -> str:
257         base = "ORDER BY "
258
259         cases: List[str] = []
260
```

```
261         for cond in order_node.conditions:
262             if isinstance(cond.value, str):
263                 value = self.cypher_var_for(cond.value)
264 
265             elif isinstance(cond.value, OrExpression):
266                 value =
                                                    self.expression_handler.value_or
267 
268             else:
269                 value =
                                                    self.expression_handler.builtinf
270 
271             cases.append(f"{value} {cond.order}")
272 
273         return base + ", ".join(cases)
274 
275     def skip_clause(self, skip: int) -> str:
276         return f"SKIP {skip}"
277 
278     def limit_clause(self, limit: int) -> str:
279         return f"LIMIT {limit}"
280 
281     def parse_query(self, query: str) -> Query:
282         return self.parser.parse(query)
283 
284     def setup_namespaces(self, namespaces: List[Namespace]):
285         self.namespaces = {nm.abbrev: nm.full for nm in namespaces}
286 
287     def code_block_for_triple(self, triple: Triple) -> str:
288         match_clause = self.match_clause(triple)
289         if self.is_rdf_type(triple):
290             unwind_clause = None
291             with_clause = self.with_clause_rdf_type(triple)
292 
293         else:
294             unwind_clause = self.unwind_clause(triple)
295             with_clause = self.with_clause(triple)
296 
297         return "\n".join(
298             filter(lambda k: bool(k), [match_clause, unwind_clause,
                                                with_clause])
299         )
300 
301     def code_block_for_pattern(self, pattern: GraphPattern, query: Query) -
                                        > str:
302         self.reset_variables()
303         codes = []
304 
```

```
305         if pattern.filters or pattern.minus or pattern.optionals:
306             logger.warning(
307                 "Some features used in the sparql query were not
                                                    implemented yet."
308             )
309
310         for and_triple in pattern.and_triples:
311             codes.append(self.code_block_for_triple(and_triple))
312
313         return "\n".join(codes) + f"\n{self.return_clause(query)}"
314
315     def split_pattern(self, graph: GraphPattern) -> List[GraphPattern]:
316         """Split graph to generate queries
317
318         Given a graph pattern, split it into a list of graph
319         pattern without or_blocks, to concatenate or blocks after
320         """
321         if graph.or_blocks is None:
322             return [graph]
323
324         to_check_graphs = [graph]
325         patterns = []
326         while to_check_graphs:
327             curr_graph = to_check_graphs.pop(0)
328             if curr_graph.or_blocks:
329                 for block in curr_graph.or_blocks:
330                     for pattern in block:
331                         pattern.and_triples.extend(curr_graph.and_triples)
332                         to_check_graphs.append(pattern)
333
334             elif curr_graph.and_triples:
335                 patterns.append(curr_graph)
336
337         return patterns
338
339     def having_clause(self, node: ModifiersNode) -> str:
340         if node.having is None:
341             return ""
342
343         conditions: List[str] = []
344         for cond in node.having.constraints:
345             if isinstance(cond, OrExpression):
346                 converted =
                                                    self.expression_handler.value_or
347
348             elif isinstance(cond, BuiltInFunction):
349                 converted =
                                                    self.expression_handler.builtinf
```

```
350
351             for exp, alias in self.exp_aliases:
352                 converted = converted.replace(exp, alias)
353
354             conditions.append(converted)
355
356         return "WITH *\nWHERE " + " AND ".join(conditions)
357
358     def setup_aliases(self, ret_vars: List[SelectedVar]):
359         """Setup aliases to use in having clause for replacing values"""
360         self.exp_aliases = [
361             (
362                 self.cypher_var_for(var.value)
363                 if isinstance(var.value, str)
364                 else self.expression_handler.value_orexpression(var.value),
365                 self.cypher_var_for(var.alias),
366             )
367             for var in ret_vars
368             if var.alias
369         ]
370
371     def generate(self, sparql_query: str) -> str:
372         """Generate cypher from sparql"""
373         query = self.parse_query(sparql_query)
374
375         self.setup_aliases(query.returning)
376
377         self.setup_namespaces(query.namespaces)
378
379         patterns = self.split_pattern(query.graph_pattern)
380
381         code_blocks = [
382             self.code_block_for_pattern(p, query)
383             for p in patterns
384             if len(p.and_triples) > 0
385         ]
386
387         united_code = "\nUNION\n".join(code_blocks)
388
389         modifiers = self.result_modifier(query.modifiers)
390         having_part = self.having_clause(query.modifiers)
391
392         if modifiers or having_part:
393             ret_clause = "RETURN *"
394             modified_code = (
395                 "CALL {\n"
396                 + united_code
397                 + "\n}"
```

```
                + ("\n" + having_part if having_part else "")
                + "\n"
                + ret_clause
                + ("\n" + modifiers if modifiers else "")
            )

        else:
            modified_code = united_code

        return modified_code
```

transpiler/__init__.py

transpiler/parser.py

```
from typing import List, Union

from ply import yacc

from .lexer import SelectSparqlLexer
from .structures import nodes
from .structures.query import Query


class SelectSparqlParser:
    def __init__(self, **kwargs):
        self.lexer = SelectSparqlLexer()
        self.tokens = self.lexer.tokens

        self.yacc = yacc.yacc(module=self, check_recursion=False, **kwargs)

        self.query = Query()

    def parse(self, source_code: str) -> Query:
        return self.yacc.parse(source_code, lexer=self.lexer)

    def p_production_0(self, p):
        """QueryUnit : Query"""
        p[0] = self.query

    def p_production_2(self, p):
        """Query : Prologue SelectQuery"""
        pass

    def p_production_4(self, p):
        """Prologue : BaseDecl Prologue"""
        pass
```

```
33
34      def p_production_5(self, p):
35          """Prologue : PrefixDecl Prologue"""
36          pass
37
38      def p_production_6(self, p):
39          """Prologue : empty"""
40          pass
41
42      def p_production_8(self, p):
43          """BaseDecl : KW_BASE IRIREF"""
44          # TODO
45          raise NotImplementedError
46
47      def p_production_11(self, p):
48          """PrefixDecl : KW_PREFIX PNAME_NS IRIREF"""
49          self.query.namespaces.append(
50              nodes.Namespace(
51                  p[2].replace(":", ""), p[3].replace("<", "").replace(">",
                                                                         "")
52              )
53          )
54
55      def p_production_13(self, p):
56          """SelectQuery : SelectClause WhereClause SolutionModifier"""
57          pass
58
59      def p_production_15(self, p):
60          """SelectClause : KW_SELECT SelectClauseAux1"""
61          pass
62
63      def p_production_16(self, p):
64          """SelectClauseAux1 : SelectClauseAux2"""
65          self.query.returning = p[1]
66
67      def p_production_17(self, p):
68          """SelectClauseAux1 : KW_DISTINCT SelectClauseAux2"""
69          # TODO
70          raise NotImplementedError
71
72      def p_production_18(self, p):
73          """SelectClauseAux1 : KW_REDUCED SelectClauseAux2"""
74          # TODO
75          raise NotImplementedError
76
77      def p_production_19(self, p):
78          """SelectClauseAux2 : SYMB_ASTERISK"""
79          p[0] = [nodes.SelectedVar(value="*")]
```

```
80
81      def p_production_20(self, p):
82          """SelectClauseAux2 : Var SelectClauseAux3"""
83          p[0] = [nodes.SelectedVar(value=p[1]), *p[2]]
84
85      def p_production_21(self, p):
86          """SelectClauseAux2 : SYMB_LP Expression KW_AS Var SYMB_RP
                                                    SelectClauseAux3"""
87          p[0] = [nodes.SelectedVar(value=p[2], alias=p[4]), *p[6]]
88
89      def p_production_22(self, p):
90          """SelectClauseAux3 : Var SelectClauseAux3"""
91          p[0] = [nodes.SelectedVar(value=p[1]), *p[2]]
92
93      def p_production_23(self, p):
94          """SelectClauseAux3 : SYMB_LP Expression KW_AS Var SYMB_RP
                                                    SelectClauseAux3"""
95          p[0] = [nodes.SelectedVar(value=p[2], alias=p[4]), *p[6]]
96
97      def p_production_24(self, p):
98          """SelectClauseAux3 : empty"""
99          p[0] = []
100
101     def p_production_26(self, p):
102         """WhereClause : GroupGraphPattern"""
103         self.query.graph_pattern = p[1]
104
105     def p_production_27(self, p):
106         """WhereClause : KW_WHERE GroupGraphPattern"""
107         self.query.graph_pattern = p[2]
108
109     def p_production_29(self, p):
110         """SolutionModifier : SolutionModifierAux1 SolutionModifierAux2
                                                    SolutionModifierAux3
                                                    SolutionModifierAux4"""
111         pass
112
113     def p_production_30(self, p):
114         """SolutionModifierAux1 : empty"""
115         pass
116
117     def p_production_31(self, p):
118         """SolutionModifierAux1 : GroupClause"""
119         pass
120
121     def p_production_32(self, p):
122         """SolutionModifierAux2 : empty"""
123         pass
```

```
124
125     def p_production_33(self, p):
126         """SolutionModifierAux2 : HavingClause"""
127         pass
128
129     def p_production_34(self, p):
130         """SolutionModifierAux3 : empty"""
131         pass
132
133     def p_production_35(self, p):
134         """SolutionModifierAux3 : OrderClause"""
135         pass
136
137     def p_production_36(self, p):
138         """SolutionModifierAux4 : empty"""
139         pass
140
141     def p_production_37(self, p):
142         """SolutionModifierAux4 : LimitOffsetClauses"""
143         pass
144
145     def p_production_39(self, p):
146         """GroupClause : KW_GROUP KW_BY GroupCondition GroupClauseAux"""
147         conds = [p[3], *p[4]]
148         self.query.modifiers.group = nodes.GroupClauseNode(conds)
149
150     def p_production_40(self, p):
151         """GroupClauseAux : empty"""
152         p[0] = []
153
154     def p_production_41(self, p):
155         """GroupClauseAux : GroupCondition GroupClauseAux"""
156         conds = [p[1], *p[2]]
157         p[0] = conds
158
159     def p_production_43(self, p):
160         """GroupCondition : BuiltInCall"""
161         p[0] = nodes.GroupCondition(value=p[1])
162
163     def p_production_44(self, p):
164         """GroupCondition : SYMB_LP Expression GroupConditionAux SYMB_RP"""
165         p[0] = nodes.GroupCondition(value=p[2], alias=p[3])
166
167     def p_production_45(self, p):
168         """GroupCondition : Var"""
169         p[0] = nodes.GroupCondition(value=p[1])
170
171     def p_production_46(self, p):
```

```
172         """GroupConditionAux : KW_AS Var"""
173         p[0] = p[2]
174 
175     def p_production_47(self, p):
176         """GroupConditionAux : empty"""
177         p[0] = None
178 
179     def p_production_49(self, p):
180         """HavingClause : KW_HAVING HavingCondition HavingClauseAux"""
181         conds = p[3]
182         conds.append(p[2])
183         self.query.modifiers.having = nodes.HavingClauseNode(conds)
184 
185     def p_production_50(self, p):
186         """HavingClauseAux : empty"""
187         p[0] = []
188 
189     def p_production_51(self, p):
190         """HavingClauseAux : HavingCondition HavingClauseAux"""
191         conds = p[2]
192         conds.append(p[1])
193         p[0] = conds
194 
195     def p_production_53(self, p):
196         """HavingCondition : Constraint"""
197         p[0] = p[1]
198 
199     def p_production_55(self, p):
200         """OrderClause : KW_ORDER KW_BY OrderCondition OrderClauseAux"""
201         conds = [p[3], *p[4]]
202         self.query.modifiers.order = nodes.OrderNode(conds)
203 
204     def p_production_56(self, p):
205         """OrderClauseAux : empty"""
206         p[0] = []
207 
208     def p_production_57(self, p):
209         """OrderClauseAux : OrderCondition OrderClauseAux"""
210         conds = [p[1], *p[2]]
211         p[0] = conds
212 
213     def p_production_59(self, p):
214         """OrderCondition : Constraint"""
215         p[0] = nodes.OrderCondition(value=p[1])
216 
217     def p_production_60(self, p):
218         """OrderCondition : Var"""
219         p[0] = nodes.OrderCondition(value=p[1])
```

```
220
221     def p_production_61(self, p):
222         """OrderCondition : OrderConditionAux Bracketted Expression"""
223         p[0] = nodes.OrderCondition(order=p[1], value=p[2])
224
225     def p_production_62(self, p):
226         """OrderConditionAux : KW_ASC"""
227         p[0] = p[1]
228
229     def p_production_63(self, p):
230         """OrderConditionAux : KW_DESC"""
231         p[0] = p[1]
232
233     def p_production_65(self, p):
234         """LimitOffsetClauses : LimitClause LimitOffsetClausesAux1"""
235         pass
236
237     def p_production_66(self, p):
238         """LimitOffsetClauses : OffsetClause LimitOffsetClausesAux2"""
239         pass
240
241     def p_production_67(self, p):
242         """LimitOffsetClausesAux1 : empty"""
243         pass
244
245     def p_production_68(self, p):
246         """LimitOffsetClausesAux1 : OffsetClause"""
247         pass
248
249     def p_production_69(self, p):
250         """LimitOffsetClausesAux2 : empty"""
251         pass
252
253     def p_production_70(self, p):
254         """LimitOffsetClausesAux2 : LimitClause"""
255         pass
256
257     def p_production_72(self, p):
258         """LimitClause : KW_LIMIT INTEGER"""
259         self.query.modifiers.limit = int(p[2])
260
261     def p_production_74(self, p):
262         """OffsetClause : KW_OFFSET INTEGER"""
263         self.query.modifiers.offset = int(p[2])
264
265     def p_production_79(self, p):
266         """GroupGraphPattern : SYMB_LCB GroupGraphPatternSub SYMB_RCB"""
267         p[0] = p[2]
```

```
268
269     def p_production_81(self, p):
270         """GroupGraphPatternSub : GroupGraphPatternSubAux1
                                            GroupGraphPatternSubAux2"""
271         and_triples = p[1] or []
272         and_triples.extend(p[2].pop("triples"))
273
274         p[0] = nodes.GraphPattern(and_triples=and_triples, **p[2])
275
276     def p_production_82(self, p):
277         """GroupGraphPatternSubAux1 : TriplesBlock"""
278         p[0] = p[1]
279
280     def p_production_83(self, p):
281         """GroupGraphPatternSubAux1 : empty"""
282         pass
283
284     def p_production_84(self, p):
285         """GroupGraphPatternSubAux2 : GraphPatternNotTriples
                                            GroupGraphPatternSubAux3
                                            GroupGraphPatternSubAux1
                                            GroupGraphPatternSubAux2"""
286         data = p[4]
287         data[p[1]["type"]].append(p[1]["value"])
288
289         if p[3] is not None:
290             data["triples"].extend(p[3])
291
292         p[0] = data
293
294     def p_production_85(self, p):
295         """GroupGraphPatternSubAux2 : empty"""
296         p[0] = {
297             "or_blocks": [],
298             "filters": [],
299             "minus": [],
300             "optionals": [],
301             "triples": [],
302         }
303
304     def p_production_86(self, p):
305         """GroupGraphPatternSubAux3 : SYMB_DOT"""
306         pass
307
308     def p_production_87(self, p):
309         """GroupGraphPatternSubAux3 : empty"""
310         pass
311
```

```
312     def p_production_89(self, p):
313         """TriplesBlock : TriplesSameSubjectPath TriplesBlockAux1"""
314         triples = p[1]
315         if p[2] is not None:
316             triples.extend(p[2])
317 
318         p[0] = triples
319 
320     def p_production_90(self, p):
321         """TriplesBlockAux1 : SYMB_DOT TriplesBlockAux2"""
322         p[0] = p[2]
323 
324     def p_production_91(self, p):
325         """TriplesBlockAux1 : empty"""
326         p[0] = []
327 
328     def p_production_92(self, p):
329         """TriplesBlockAux2 : TriplesBlock"""
330         p[0] = p[1]
331 
332     def p_production_93(self, p):
333         """TriplesBlockAux2 : empty"""
334         p[0] = []
335 
336     def p_production_95(self, p):
337         """GraphPatternNotTriples : GroupOrUnionGraphPattern"""
338         p[0] = {"type": "or_blocks", "value": p[1]}
339 
340     def p_production_96(self, p):
341         """GraphPatternNotTriples : OptionalGraphPattern"""
342         p[0] = {"type": "optionals", "value": p[1]}
343 
344     def p_production_97(self, p):
345         """GraphPatternNotTriples : MinusGraphPattern"""
346         p[0] = {"type": "minus", "value": p[1]}
347 
348     def p_production_98(self, p):
349         """GraphPatternNotTriples : Filter"""
350         p[0] = {"type": "filters", "value": p[1]}
351 
352     def p_production_102(self, p):
353         """OptionalGraphPattern : KW_OPTIONAL GroupGraphPattern"""
354         p[0] = p[2]
355 
356     def p_production_133(self, p):
357         """MinusGraphPattern : KW_MINUS GroupGraphPattern"""
358         p[0] = p[2]
359 
```

```
360     def p_production_135(self, p):
361         """GroupOrUnionGraphPattern : GroupGraphPattern
                                            GroupOrUnionGraphPatternAux"""
362         patterns = p[2]
363         patterns.append(p[1])
364 
365         p[0] = patterns
366 
367     def p_production_136(self, p):
368         """GroupOrUnionGraphPatternAux : KW_UNION GroupGraphPattern
                                            GroupOrUnionGraphPatternAux"""
369         patterns = p[3]
370         patterns.append(p[2])
371         p[0] = patterns
372 
373     def p_production_137(self, p):
374         """GroupOrUnionGraphPatternAux : empty"""
375         p[0] = []
376 
377     def p_production_139(self, p):
378         """Filter : KW_FILTER Constraint"""
379         p[0] = nodes.FilterNode(p[2])
380 
381     def p_production_141(self, p):
382         """Constraint : BrackettedExpression"""
383         p[0] = p[1]
384 
385     def p_production_142(self, p):
386         """Constraint : BuiltInCall"""
387         p[0] = p[1]
388 
389     def p_production_144(self, p):
390         """ExpressionList : NIL"""
391         p[0] = []
392 
393     def p_production_145(self, p):
394         """ExpressionList : SYMB_LP Expression ExpressionListAux SYMB_RP"""
395         p[0] = [p[2], *p[3]]
396 
397     def p_production_146(self, p):
398         """ExpressionListAux : SYMB_COMMA Expression ExpressionListAux"""
399         p[0] = [p[2], *p[3]]
400 
401     def p_production_147(self, p):
402         """ExpressionListAux : empty"""
403         p[0] = []
404 
405     def p_production_149(self, p):
```

```
406         """PropertyListNotEmpty : Verb ObjectList
                                              PropertyListNotEmptyAux2"""
407         props = [(p[1], p[2])]
408         props.extend(p[3])
409 
410         p[0] = props
411 
412     def p_production_150(self, p):
413         """PropertyListNotEmptyAux1 : Verb ObjectList"""
414         p[0] = (p[1], p[2])
415 
416     def p_production_151(self, p):
417         """PropertyListNotEmptyAux1 : empty"""
418         p[0] = None
419 
420     def p_production_152(self, p):
421         """PropertyListNotEmptyAux2 : SYMB_SEMICOLON
                                              PropertyListNotEmptyAux1
                                              PropertyListNotEmptyAux2"""
422         props = p[3]
423 
424         if p[3] is not None:
425             props.append(p[2])
426 
427         p[0] = props
428 
429     def p_production_153(self, p):
430         """PropertyListNotEmptyAux2 : empty"""
431         p[0] = []
432 
433     def p_production_155(self, p):
434         """Verb : VarOrIri"""
435         p[0] = p[1]
436 
437     def p_production_156(self, p):
438         """Verb : SYMB_a"""
439         p[0] = "rdf:type"
440 
441     def p_production_158(self, p):
442         """ObjectList : Object ObjectListAux"""
443         obj_list = p[2]
444         obj_list.append(p[1])
445 
446         p[0] = obj_list
447 
448     def p_production_159(self, p):
449         """ObjectListAux : SYMB_COMMA Object ObjectListAux"""
450         obj_list = p[3]
```

```
451         obj_list.append(p[2])
452 
453         p[0] = obj_list
454 
455     def p_production_160(self, p):
456         """ObjectListAux : empty"""
457         p[0] = []
458 
459     def p_production_162(self, p):
460         """Object : GraphNode"""
461         if isinstance(p[1], str):
462             p[0] = p[1]
463 
464         else:
465             raise NotImplementedError
466 
467     def p_production_164(self, p):
468         """TriplesSameSubjectPath : VarOrTerm PropertyListPathNotEmpty"""
469         triples = []
470         for prop, obj in p[2]:
471             triples.append(nodes.Triple(p[1], prop, obj))
472 
473         p[0] = triples
474 
475     def p_production_165(self, p):
476         """TriplesSameSubjectPath : TriplesNodePath PropertyListPath"""
477         # TODO
478         raise NotImplementedError
479 
480     def p_production_167(self, p):
481         """PropertyListPath : PropertyListPathNotEmpty"""
482         # TODO
483         raise NotImplementedError
484 
485     def p_production_168(self, p):
486         """PropertyListPath : empty"""
487         # TODO
488         raise NotImplementedError
489 
490     def p_production_170(self, p):
491         """PropertyListPathNotEmpty : PropertyListPathNotEmptyAux1
                                        ObjectListPath
                                        PropertyListPathNotEmptyAux2"""
492         p[0] = p[3]
493         for obj in p[2]:
494             p[0].append((p[1], obj))
495 
496     def p_production_171(self, p):
```

```
497         """PropertyListPathNotEmptyAux1 : VerbPath"""
498         p[0] = p[1]
499
500     def p_production_172(self, p):
501         """PropertyListPathNotEmptyAux1 : VerbSimple"""
502         p[0] = p[1]
503
504     def p_production_173(self, p):
505         """PropertyListPathNotEmptyAux2 : SYMB_SEMICOLON
                                              PropertyListPathNotEmptyAux3"""
506         p[0] = p[2]
507
508     def p_production_174(self, p):
509         """PropertyListPathNotEmptyAux2 : empty"""
510         p[0] = []
511
512     def p_production_175(self, p):
513         """PropertyListPathNotEmptyAux3 : PropertyListPathNotEmptyAux1
                                              ObjectList"""
514         subs = []
515         for obj in p[2]:
516             subs.append((p[1], obj))
517         p[0] = subs
518
519     def p_production_176(self, p):
520         """PropertyListPathNotEmptyAux3 : empty"""
521         p[0] = []
522
523     def p_production_178(self, p):
524         """VerbPath : Path"""
525         p[0] = p[1]
526
527     def p_production_180(self, p):
528         """VerbSimple : Var"""
529         p[0] = p[1]
530
531     def p_production_182(self, p):
532         """ObjectListPath : ObjectPath ObjectListPathAux"""
533         objs = [p[1]]
534         if p[2] is not None:
535             objs.append(p[2])
536
537         p[0] = objs
538
539     def p_production_183(self, p):
540         """ObjectListPathAux : SYMB_COMMA ObjectPath"""
541         p[0] = p[2]
542
```

```
543     def p_production_184(self, p):
544         """ObjectListPathAux : empty"""
545         pass
546 
547     def p_production_186(self, p):
548         """ObjectPath : GraphNodePath"""
549         p[0] = p[1]
550 
551     def p_production_188(self, p):
552         """Path : PathAlternative"""
553         p[0] = p[1]
554 
555     def p_production_190(self, p):
556         """PathAlternative : PathSequence PathAlternativeAux"""
557         p[0] = p[1]
558 
559     def p_production_191(self, p):
560         """PathAlternativeAux : SYMB_PIPE PathSequence
                                                PathAlternativeAux"""
561         # TODO
562         raise NotImplementedError
563 
564     def p_production_192(self, p):
565         """PathAlternativeAux : empty"""
566         # TODO
567         pass
568 
569     def p_production_194(self, p):
570         """PathSequence : PathEltOrInverse PathSequenceAux"""
571         # TODO
572         p[0] = p[1]
573 
574     def p_production_195(self, p):
575         """PathSequenceAux : SYMB_SLASH PathEltOrInverse PathSequenceAux"""
576         # TODO
577         raise NotImplementedError
578 
579     def p_production_196(self, p):
580         """PathSequenceAux : empty"""
581         # TODO
582         pass
583 
584     def p_production_198(self, p):
585         """PathElt : PathPrimary PathEltAux"""
586         # TODO
587         p[0] = p[1]
588 
589     def p_production_199(self, p):
```

```
590         """PathEltAux : PathMod"""
591         # TODO
592         raise NotImplementedError
593 
594     def p_production_200(self, p):
595         """PathEltAux : empty"""
596         # TODO
597         pass
598 
599     def p_production_202(self, p):
600         """PathEltOrInverse : PathElt"""
601         # TODO
602         p[0] = p[1]
603 
604     def p_production_203(self, p):
605         """PathEltOrInverse : SYMB_CIRCUMFLEX PathElt"""
606         # TODO
607         raise NotImplementedError
608 
609     def p_production_205(self, p):
610         """PathMod : SYMB_QUESTION"""
611         # TODO
612         raise NotImplementedError
613 
614     def p_production_206(self, p):
615         """PathMod : SYMB_ASTERISK"""
616         # TODO
617         raise NotImplementedError
618 
619     def p_production_207(self, p):
620         """PathMod : SYMB_PLUS"""
621         # TODO
622         raise NotImplementedError
623 
624     def p_production_209(self, p):
625         """PathPrimary : iri"""
626         p[0] = p[1]
627 
628     def p_production_210(self, p):
629         """PathPrimary : SYMB_a"""
630         p[0] = "rdf:type"
631 
632     def p_production_211(self, p):
633         """PathPrimary : SYMB_EXCLAMATION PathNegatedPropertySet"""
634         # TODO
635         raise NotImplementedError
636 
637     def p_production_212(self, p):
```

```
638         """PathPrimary : SYMB_LP Path SYMB_RP"""
639         # TODO
640         raise NotImplementedError
641 
642     def p_production_215(self, p):
643         """PathNegatedPropertySet : PathOneInPropertySet"""
644         # TODO
645         raise NotImplementedError
646 
647     def p_production_216(self, p):
648         """PathNegatedPropertySet : SYMB_LP PathNegatedPropertySetAux1
                                        SYMB_RP"""
649         # TODO
650         raise NotImplementedError
651 
652     def p_production_217(self, p):
653         """PathNegatedPropertySetAux1 : PathOneInPropertySet
                                        PathNegatedPropertySetAux2"""
654         # TODO
655         raise NotImplementedError
656 
657     def p_production_218(self, p):
658         """PathNegatedPropertySetAux1 : empty"""
659         # TODO
660         raise NotImplementedError
661 
662     def p_production_219(self, p):
663         """PathNegatedPropertySetAux2 : SYMB_PIPE PathOneInPropertySet
                                        PathNegatedPropertySetAux2"""
664         # TODO
665         raise NotImplementedError
666 
667     def p_production_220(self, p):
668         """PathNegatedPropertySetAux2 : empty"""
669         # TODO
670         raise NotImplementedError
671 
672     def p_production_222(self, p):
673         """PathOneInPropertySet : iri"""
674         # TODO
675         raise NotImplementedError
676 
677     def p_production_223(self, p):
678         """PathOneInPropertySet : SYMB_a"""
679         p[0] = "rdf:type"
680 
681     def p_production_224(self, p):
682         """PathOneInPropertySet : SYMB_CIRCUMFLEX
```

```
                                            PathOneInPropertySetAux"""
683         # TODO
684         raise NotImplementedError
685
686     def p_production_225(self, p):
687         """PathOneInPropertySetAux : iri"""
688         # TODO
689         raise NotImplementedError
690
691     def p_production_226(self, p):
692         """PathOneInPropertySetAux : SYMB_a"""
693         p[0] = "rdf:type"
694
695     def p_production_228(self, p):
696         """TriplesNode : Collection"""
697         # TODO
698         raise NotImplementedError
699
700     def p_production_229(self, p):
701         """TriplesNode : BlankNodePropertyList"""
702         # TODO
703         raise NotImplementedError
704
705     def p_production_231(self, p):
706         """BlankNodePropertyList : SYMB_LSB PropertyListNotEmpty
                                            SYMB_RSB"""
707         # TODO
708         raise NotImplementedError
709
710     def p_production_233(self, p):
711         """TriplesNodePath : CollectionPath"""
712         # TODO
713         raise NotImplementedError
714
715     def p_production_234(self, p):
716         """TriplesNodePath : BlankNodePropertyListPath"""
717         # TODO
718         raise NotImplementedError
719
720     def p_production_236(self, p):
721         """BlankNodePropertyListPath : SYMB_LSB PropertyListPathNotEmpty
                                            SYMB_RSB"""
722         # TODO
723         raise NotImplementedError
724
725     def p_production_238(self, p):
726         """Collection : SYMB_LP GraphNode CollectionAux SYMB_RP"""
727         # TODO
```

```
728         raise NotImplementedError
729
730     def p_production_239(self, p):
731         """CollectionAux : GraphNode CollectionAux"""
732         # TODO
733         raise NotImplementedError
734
735     def p_production_240(self, p):
736         """CollectionAux : empty"""
737         # TODO
738         raise NotImplementedError
739
740     def p_production_242(self, p):
741         """CollectionPath : SYMB_LP GraphNodePath CollectionPathAux
                                                SYMB_RP"""
742         # TODO
743         raise NotImplementedError
744
745     def p_production_243(self, p):
746         """CollectionPathAux : GraphNodePath CollectionPathAux"""
747         # TODO
748         raise NotImplementedError
749
750     def p_production_244(self, p):
751         """CollectionPathAux : empty"""
752         # TODO
753         raise NotImplementedError
754
755     def p_production_246(self, p):
756         """GraphNode : TriplesNode"""
757         p[0] = p[1]
758
759     def p_production_247(self, p):
760         """GraphNode : VarOrTerm"""
761         p[0] = p[1]
762
763     def p_production_249(self, p):
764         """GraphNodePath : VarOrTerm"""
765         p[0] = p[1]
766
767     def p_production_250(self, p):
768         """GraphNodePath : TriplesNodePath"""
769         p[0] = p[1]
770
771     def p_production_252(self, p):
772         """VarOrTerm : Var"""
773         p[0] = p[1]
774
```

```
775     def p_production_253(self, p):
776         """VarOrTerm : GraphTerm"""
777         p[0] = p[1]
778 
779     def p_production_255(self, p):
780         """VarOrIri : Var"""
781         p[0] = p[1]
782 
783     def p_production_256(self, p):
784         """VarOrIri : iri"""
785         p[0] = p[1]
786 
787     def p_production_258(self, p):
788         """Var : VAR1
789         | VAR2"""
790         p[0] = p[1]
791 
792         if p[0] not in self.query.variables:
793             self.query.variables.append(p[0])
794 
795     def p_production_261(self, p):
796         """GraphTerm : iri"""
797         p[0] = p[1]
798 
799     def p_production_262(self, p):
800         """GraphTerm : RDFLiteral"""
801         p[0] = p[1]
802 
803     def p_production_263(self, p):
804         """GraphTerm : NumericLiteral"""
805         p[0] = p[1]
806 
807     def p_production_264(self, p):
808         """GraphTerm : BooleanLiteral"""
809         p[0] = p[1]
810 
811     def p_production_265(self, p):
812         """GraphTerm : BlankNode"""
813         p[0] = None
814 
815     def p_production_266(self, p):
816         """GraphTerm : NIL"""
817         p[0] = None
818 
819     def p_production_268(self, p):
820         """Expression : ConditionalOrExpression"""
821         p[0] = p[1]
822 
```

```
823     def p_production_270(self, p):
824         """ConditionalOrExpression : ConditionalAndExpression
                                           ConditionalOrExpressionAux"""
825         p[0] = nodes.OrExpression(p[1], p[2])
826
827     def p_production_271(self, p):
828         """ConditionalOrExpressionAux : SYMB_OR ConditionalAndExpression
                                           ConditionalOrExpressionAux"""
829         p[0] = [p[2], *p[3]]
830
831     def p_production_272(self, p):
832         """ConditionalOrExpressionAux : empty"""
833         p[0] = []
834
835     def p_production_274(self, p):
836         """ConditionalAndExpression : ValueLogical
                                           ConditionalAndExpressionAux"""
837         p[0] = nodes.AndExpression(p[1], p[2])
838
839     def p_production_275(self, p):
840         """ConditionalAndExpressionAux : SYMB_AND ValueLogical
                                           ConditionalAndExpressionAux"""
841         p[0] = [p[2], *p[3]]
842
843     def p_production_276(self, p):
844         """ConditionalAndExpressionAux : empty"""
845         p[0] = []
846
847     def p_production_278(self, p):
848         """ValueLogical : RelationalExpression"""
849         p[0] = p[1]
850
851     def p_production_280(self, p):
852         """RelationalExpression : NumericExpression
                                           RelationalExpressionAux"""
853         p[0] = nodes.RelationalExpression(p[1], p[2])
854
855     def p_production_281(self, p):
856         """RelationalExpressionAux : SYMB_EQ NumericExpression"""
857         p[0] = (nodes.LogOperator.EQ, p[2])
858
859     def p_production_282(self, p):
860         """RelationalExpressionAux : SYMB_NEQ NumericExpression"""
861         p[0] = (nodes.LogOperator.NEQ, p[2])
862
863     def p_production_283(self, p):
864         """RelationalExpressionAux : SYMB_LT NumericExpression"""
865         p[0] = (nodes.LogOperator.LT, p[2])
```

```
866
867     def p_production_284(self, p):
868         """RelationalExpressionAux : SYMB_GT NumericExpression"""
869         p[0] = (nodes.LogOperator.GT, p[2])
870
871     def p_production_285(self, p):
872         """RelationalExpressionAux : SYMB_LTE NumericExpression"""
873         p[0] = (nodes.LogOperator.LTE, p[2])
874
875     def p_production_286(self, p):
876         """RelationalExpressionAux : SYMB_GTE NumericExpression"""
877         p[0] = (nodes.LogOperator.GTE, p[2])
878
879     def p_production_287(self, p):
880         """RelationalExpressionAux : KW_IN ExpressionList"""
881         p[0] = (nodes.LogOperator.IN, p[2])
882
883     def p_production_288(self, p):
884         """RelationalExpressionAux : KW_NOT KW_IN ExpressionList"""
885         p[0] = (nodes.LogOperator.NOT_IN, p[2])
886
887     def p_production_289(self, p):
888         """RelationalExpressionAux : empty"""
889         p[0] = None
890
891     def p_production_291(self, p):
892         """NumericExpression : AdditiveExpression"""
893         p[0] = p[1]
894
895     def p_production_292(self, p):
896         """AdditiveExpression : MultiplicativeExpression
                                  AdditiveExpressionAux1"""
897         p[0] = nodes.AdditiveExpression(p[1], p[2])
898
899     def p_production_293(self, p):
900         """AdditiveExpressionAux1 : SYMB_PLUS MultiplicativeExpression
                                      AdditiveExpressionAux1"""
901         p[0] = [(nodes.AdditiveOperator.SUB, p[2]), *p[3]]
902
903     def p_production_294(self, p):
904         """AdditiveExpressionAux1 : SYMB_MINUS MultiplicativeExpression
                                      AdditiveExpressionAux1"""
905         p[0] = [(nodes.AdditiveOperator.SUB, p[2]), *p[3]]
906
907     def p_production_295(self, p):
908         """AdditiveExpressionAux1 : empty"""
909         p[0] = []
910
```

```
911     def p_production_304(self, p):
912         """MultiplicativeExpression : UnaryExpression
                                            MultiplicativeExpressionAux"""
913         p[0] = nodes.MultiplicativeExpression(p[1], p[2])
914 
915     def p_production_305(self, p):
916         """MultiplicativeExpressionAux : SYMB_ASTERISK UnaryExpression
                                            MultiplicativeExpressionAux"""
917         p[0] = [(nodes.MultiplicativeOperator.MULT, p[2]), *p[3]]
918 
919     def p_production_306(self, p):
920         """MultiplicativeExpressionAux : SYMB_SLASH UnaryExpression
                                            MultiplicativeExpressionAux"""
921         p[0] = [(nodes.MultiplicativeOperator.DIV, p[2]), *p[3]]
922 
923     def p_production_307(self, p):
924         """MultiplicativeExpressionAux : empty"""
925         p[0] = []
926 
927     def p_production_309(self, p):
928         """UnaryExpression : SYMB_EXCLAMATION PrimaryExpression"""
929         p[0] = nodes.UnaryExpression(value=p[1],
                                            op=nodes.UnaryOperator.NOT)
930 
931     def p_production_310(self, p):
932         """UnaryExpression : SYMB_PLUS PrimaryExpression"""
933         p[0] = nodes.UnaryExpression(value=p[1],
                                            op=nodes.UnaryOperator.PLUS)
934 
935     def p_production_311(self, p):
936         """UnaryExpression : SYMB_MINUS PrimaryExpression"""
937         p[0] = nodes.UnaryExpression(value=p[1],
                                            op=nodes.UnaryOperator.MINUS)
938 
939     def p_production_312(self, p):
940         """UnaryExpression : PrimaryExpression"""
941         p[0] = nodes.UnaryExpression(p[1])
942 
943     def p_production_314(self, p):
944         """PrimaryExpression : BracketedExpression"""
945         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.EXP, p[1])
946 
947     def p_production_315(self, p):
948         """PrimaryExpression : BuiltInCall"""
949         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.FUNC, p[1])
950 
951     def p_production_316(self, p):
952         """PrimaryExpression : iri"""
```

```
953         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.IRI, p[1])
954
955     def p_production_317(self, p):
956         """PrimaryExpression : RDFLiteral"""
957         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.STR_LITERAL, p[1])
958
959     def p_production_318(self, p):
960         """PrimaryExpression : NumericLiteral"""
961         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.NUM_LITERAL, p[1])
962
963     def p_production_319(self, p):
964         """PrimaryExpression : BooleanLiteral"""
965         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.BOOL_LITERAL,
                                           p[1])
966
967     def p_production_320(self, p):
968         """PrimaryExpression : Var"""
969         p[0] = nodes.PrimaryExpression(nodes.PrimaryType.VAR, p[1])
970
971     def p_production_322(self, p):
972         """BrackettedExpression : SYMB_LP Expression SYMB_RP"""
973         p[0] = p[2]
974
975     def p_production_324(self, p):
976         """BuiltInCall : Aggregate"""
977         p[0] = p[1]
978
979     def p_production_325(self, p):
980         """BuiltInCall : FUNC_RAND NIL"""
981         p[0] = nodes.BuiltInFunction("RAND", [])
982
983     def p_production_326(self, p):
984         """BuiltInCall : FUNC_ABS SYMB_LP Expression SYMB_RP"""
985         p[0] = nodes.BuiltInFunction("ABS", [p[3]])
986
987     def p_production_327(self, p):
988         """BuiltInCall : FUNC_CEIL SYMB_LP Expression SYMB_RP"""
989         p[0] = nodes.BuiltInFunction("CEIL", [p[3]])
990
991     def p_production_328(self, p):
992         """BuiltInCall : FUNC_FLOOR SYMB_LP Expression SYMB_RP"""
993         p[0] = nodes.BuiltInFunction("FLOOR", [p[3]])
994
995     def p_production_329(self, p):
996         """BuiltInCall : FUNC_ROUND SYMB_LP Expression SYMB_RP"""
997         p[0] = nodes.BuiltInFunction("ROUND", [p[3]])
998
999     def p_production_330(self, p):
```

```
1000         """BuiltInCall : FUNC_CONCAT ExpressionList"""
1001         p[0] = nodes.BuiltInFunction("CONCAT", p[2])
1002
1003     def p_production_331(self, p):
1004         """BuiltInCall : SubstringExpression"""
1005         p[0] = p[1]
1006
1007     def p_production_332(self, p):
1008         """BuiltInCall : FUNC_STRLEN SYMB_LP Expression SYMB_RP"""
1009         p[0] = nodes.BuiltInFunction("STRLEN", [p[3]])
1010
1011     def p_production_333(self, p):
1012         """BuiltInCall : StrReplaceExpression"""
1013         p[0] = p[1]
1014
1015     def p_production_334(self, p):
1016         """BuiltInCall : FUNC_UCASE SYMB_LP Expression SYMB_RP"""
1017         p[0] = nodes.BuiltInFunction("UCASE", [p[3]])
1018
1019     def p_production_335(self, p):
1020         """BuiltInCall : FUNC_LCASE SYMB_LP Expression SYMB_RP"""
1021         p[0] = nodes.BuiltInFunction("LCASE", [p[3]])
1022
1023     def p_production_336(self, p):
1024         """BuiltInCall : FUNC_CONTAINS SYMB_LP Expression SYMB_COMMA
                                                Expression SYMB_RP"""
1025         p[0] = nodes.BuiltInFunction("CONTAINS", [p[3], p[5]])
1026
1027     def p_production_337(self, p):
1028         """BuiltInCall : FUNC_STRSTARTS SYMB_LP Expression SYMB_COMMA
                                                Expression SYMB_RP"""
1029         p[0] = nodes.BuiltInFunction("STRSTARTS", [p[3], p[5]])
1030
1031     def p_production_338(self, p):
1032         """BuiltInCall : FUNC_STRENDS SYMB_LP Expression SYMB_COMMA
                                                Expression SYMB_RP"""
1033         p[0] = nodes.BuiltInFunction("STRENDS", [p[3], p[5]])
1034
1035     def p_production_339(self, p):
1036         """BuiltInCall : FUNC_YEAR SYMB_LP Expression SYMB_RP"""
1037         p[0] = nodes.BuiltInFunction("YEAR", [p[3]])
1038
1039     def p_production_340(self, p):
1040         """BuiltInCall : FUNC_MONTH SYMB_LP Expression SYMB_RP"""
1041         p[0] = nodes.BuiltInFunction("MONTH", [p[3]])
1042
1043     def p_production_341(self, p):
1044         """BuiltInCall : FUNC_DAY SYMB_LP Expression SYMB_RP"""
```

```
1045        p[0] = nodes.BuiltInFunction("DAY", [p[3]])
1046
1047    def p_production_342(self, p):
1048        """BuiltInCall : FUNC_HOURS SYMB_LP Expression SYMB_RP"""
1049        p[0] = nodes.BuiltInFunction("HOURS", [p[3]])
1050
1051    def p_production_343(self, p):
1052        """BuiltInCall : FUNC_MINUTES SYMB_LP Expression SYMB_RP"""
1053        p[0] = nodes.BuiltInFunction("MINUTES", [p[3]])
1054
1055    def p_production_344(self, p):
1056        """BuiltInCall : FUNC_SECONDS SYMB_LP Expression SYMB_RP"""
1057        p[0] = nodes.BuiltInFunction("SECONDS", [p[3]])
1058
1059    def p_production_345(self, p):
1060        """BuiltInCall : FUNC_TIMEZONE SYMB_LP Expression SYMB_RP"""
1061        p[0] = nodes.BuiltInFunction("TIMEZONE", [p[3]])
1062
1063    def p_production_346(self, p):
1064        """BuiltInCall : FUNC_TZ SYMB_LP Expression SYMB_RP"""
1065        p[0] = nodes.BuiltInFunction("TZ", [p[3]])
1066
1067    def p_production_347(self, p):
1068        """BuiltInCall : FUNC_NOW NIL"""
1069        p[0] = nodes.BuiltInFunction("NOW", [])
1070
1071    def p_production_348(self, p):
1072        """BuiltInCall : FUNC_COALESCE ExpressionList"""
1073        p[0] = nodes.BuiltInFunction("COALESCE", p[2])
1074
1075    def p_production_349(self, p):
1076        """BuiltInCall : RegexExpression"""
1077        p[0] = p[1]
1078
1079    def p_production_351(self, p):
1080        """RegexExpression : FUNC_REGEX SYMB_LP Expression SYMB_COMMA
                                          Expression
                                          RegexExpressionAux SYMB_RP"""
1081        params = [p[4], p[5]]
1082        params.extend(p[6])
1083        p[0] = nodes.BuiltInFunction("REGEX", p)
1084
1085    def p_production_352(self, p):
1086        """RegexExpressionAux : SYMB_COMMA Expression"""
1087        p[0] = p[2]
1088
1089    def p_production_353(self, p):
1090        """RegexExpressionAux : empty"""
```

```
1091        p[0] = []
1092
1093    def p_production_355(self, p):
1094        """SubstringExpression : FUNC_SUBSTR SYMB_LP Expression SYMB_COMMA
                                   Expression
                                   SubstringExpressionAux
                                   SYMB_RP"""
1095        params = [p[4], p[5]]
1096        params.extend(p[6])
1097        p[0] = nodes.BuiltInFunction("SUBSTR", params)
1098
1099    def p_production_356(self, p):
1100        """SubstringExpressionAux : SYMB_COMMA Expression"""
1101        p[0] = p[2]
1102
1103    def p_production_357(self, p):
1104        """SubstringExpressionAux : empty"""
1105        p[0] = []
1106
1107    def p_production_359(self, p):
1108        """StrReplaceExpression : FUNC_REPLACE SYMB_LP Expression
                                   SYMB_COMMA Expression
                                   SYMB_COMMA Expression
                                   StrReplaceExpressionAux
                                   SYMB_RP"""
1109        params = [p[4], p[5], p[7]]
1110        params.extend(p[8])
1111        p[0] = nodes.BuiltInFunction("REPLACE", params)
1112
1113    def p_production_360(self, p):
1114        """StrReplaceExpressionAux : SYMB_COMMA Expression"""
1115        p[0] = p[1]
1116
1117    def p_production_361(self, p):
1118        """StrReplaceExpressionAux : empty"""
1119        p[0] = []
1120
1121    def p_production_363(self, p):
1122        """Aggregate : FUNC_COUNT SYMB_LP AggregateAux1 AggregateAux2
                                   SYMB_RP"""
1123        p[0] = nodes.BuiltInFunction("COUNT", [p[4]])
1124
1125    def p_production_364(self, p):
1126        """Aggregate : FUNC_SUM SYMB_LP AggregateAux1 Expression SYMB_RP"""
1127        p[0] = nodes.BuiltInFunction("SUM", [p[4]])
1128
1129    def p_production_365(self, p):
1130        """Aggregate : FUNC_MIN SYMB_LP AggregateAux1 Expression SYMB_RP"""
```

```
1131        p[0] = nodes.BuiltInFunction("MIN", [p[4]])
1132
1133    def p_production_366(self, p):
1134        """Aggregate : FUNC_MAX SYMB_LP AggregateAux1 Expression SYMB_RP"""
1135        p[0] = nodes.BuiltInFunction("MAX", [p[4]])
1136
1137    def p_production_367(self, p):
1138        """Aggregate : FUNC_AVG SYMB_LP AggregateAux1 Expression SYMB_RP"""
1139        p[0] = nodes.BuiltInFunction("AVG", [p[4]])
1140
1141    def p_production_368(self, p):
1142        """AggregateAux1 : KW_DISTINCT"""
1143        raise NotImplementedError
1144
1145    def p_production_369(self, p):
1146        """AggregateAux1 : empty"""
1147        p[0] = None
1148
1149    def p_production_370(self, p):
1150        """AggregateAux2 : SYMB_ASTERISK"""
1151        p[0] = nodes.OrExpression(
1152            nodes.AndExpression(
1153                nodes.RelationalExpression(
1154                    first=nodes.AdditiveExpression(
1155                        nodes.MultiplicativeExpression(
1156                            nodes.UnaryExpression(
1157                                value=nodes.PrimaryExpression(
1158                                    type=nodes.PrimaryType.STR_LITERAL,
                                                                    value="*"
1159                                )
1160                            )
1161                        )
1162                    )
1163                )
1164            )
1165        )
1166
1167    def p_production_371(self, p):
1168        """AggregateAux2 : Expression"""
1169        p[0] = p[1]
1170
1171    def p_production_373(self, p):
1172        """RDFLiteral : String RDFLiteralAux1"""
1173        p[0] = p[1] + p[2]
1174
1175    def p_production_374(self, p):
1176        """RDFLiteralAux1 : LANGTAG"""
1177        p[0] = "@" + p[1]
```

```
1178
1179    def p_production_375(self, p):
1180        """RDFLiteralAux1 : SYMB_C2 iri"""
1181        p[0] = p[1] + p[2]
1182
1183    def p_production_376(self, p):
1184        """RDFLiteralAux1 : empty"""
1185        p[0] = ""
1186
1187    def p_production_378(self, p):
1188        """NumericLiteral : NumericLiteralUnsigned"""
1189        p[0] = p[1]
1190
1191    def p_production_379(self, p):
1192        """NumericLiteral : NumericLiteralPositive"""
1193        p[0] = p[1]
1194
1195    def p_production_380(self, p):
1196        """NumericLiteral : NumericLiteralNegative"""
1197        p[0] = p[1]
1198
1199    def p_production_382(self, p):
1200        """NumericLiteralUnsigned : INTEGER"""
1201        p[0] = int(p[1])
1202
1203    def p_production_383(self, p):
1204        """NumericLiteralUnsigned : DECIMAL"""
1205        p[0] = float(p[1])
1206
1207    def p_production_384(self, p):
1208        """NumericLiteralUnsigned : DOUBLE"""
1209        p[0] = float(p[1])
1210
1211    def p_production_386(self, p):
1212        """NumericLiteralPositive : INTEGER_POSITIVE"""
1213        p[0] = int(p[1])
1214
1215    def p_production_387(self, p):
1216        """NumericLiteralPositive : DECIMAL_POSITIVE"""
1217        p[0] = float(p[1])
1218
1219    def p_production_388(self, p):
1220        """NumericLiteralPositive : DOUBLE_POSITIVE"""
1221        p[0] = float(p[1])
1222
1223    def p_production_390(self, p):
1224        """NumericLiteralNegative : INTEGER_NEGATIVE"""
1225        p[0] = int(p[1])
```

```
1226
1227    def p_production_391(self, p):
1228        """NumericLiteralNegative : DECIMAL_NEGATIVE"""
1229        p[0] = float(p[1])
1230
1231    def p_production_392(self, p):
1232        """NumericLiteralNegative : DOUBLE_NEGATIVE"""
1233        p[0] = float(p[1])
1234
1235    def p_production_394(self, p):
1236        """BooleanLiteral : SYMB_TRUE"""
1237        p[0] = True
1238
1239    def p_production_395(self, p):
1240        """BooleanLiteral : SYMB_FALSE"""
1241        p[0] = False
1242
1243    def p_production_397(self, p):
1244        """String : STRING_LITERAL1"""
1245        p[0] = p[1][1:-1]
1246
1247    def p_production_398(self, p):
1248        """String : STRING_LITERAL2"""
1249        p[0] = p[1][1:-1]
1250
1251    def p_production_399(self, p):
1252        """String : STRING_LITERAL_LONG1"""
1253        p[0] = p[1][3:-3]
1254
1255    def p_production_400(self, p):
1256        """String : STRING_LITERAL_LONG2"""
1257        p[0] = p[1][3:-3]
1258
1259    def p_production_402(self, p):
1260        """iri : IRIREF"""
1261        p[0] = p[1]
1262
1263    def p_production_403(self, p):
1264        """iri : PrefixedName"""
1265        p[0] = p[1]
1266
1267    def p_production_405(self, p):
1268        """PrefixedName : PNAME_LN"""
1269        p[0] = p[1]
1270
1271    def p_production_406(self, p):
1272        """PrefixedName : PNAME_NS"""
1273        p[0] = p[1]
```

```
1274
1275        def p_production_408(self, p):
1276            """BlankNode : BLANK_NODE_LABEL"""
1277            p[0] = ""
1278
1279        def p_production_409(self, p):
1280            """BlankNode : ANON"""
1281            p[0] = ""
1282
1283        def p_empty(self, p):
1284            """empty :"""
1285            pass
1286
1287        def p_error(self, p):
1288            print("ERROR!")
1289            print(p)
```

transpiler/lexer.py

```
1  from loguru import logger
2  from ply import lex
3
4  from .exceptions import InvalidTokenError
5
6
7  class SelectSparqlLexer:
8      def __init__(self, **kwargs):
9          self.lexer = lex.lex(module=self, **kwargs)
10         self._input = ""
11
12     def input(self, source_code: str, **kwargs):
13         self._input = source_code
14         self.lexer.input(source_code, **kwargs)
15
16     keywords = {
17         "BASE": "KW_BASE",
18         "PREFIX": "KW_PREFIX",
19         "SELECT": "KW_SELECT",
20         "DISTINCT": "KW_DISTINCT",
21         "REDUCED": "KW_REDUCED",
22         "WHERE": "KW_WHERE",
23         "HAVING": "KW_HAVING",
24         "ORDER": "KW_ORDER",
25         "BY": "KW_BY",
26         "ASC": "KW_ASC",
27         "DESC": "KW_DESC",
28         "LIMIT": "KW_LIMIT",
29         "OFFSET": "KW_OFFSET",
```

```
30          "OPTIONAL": "KW_OPTIONAL",
31          "MINUS": "KW_MINUS",
32          "UNION": "KW_UNION",
33          "FILTER": "KW_FILTER",
34          "AS": "KW_AS",
35          "NOT": "KW_NOT",
36          "IN": "KW_IN",
37          "GROUP": "KW_GROUP",
38      }
39
40      builtin_calls = {
41          "RAND": "FUNC_RAND",
42          "ABS": "FUNC_ABS",
43          "CEIL": "FUNC_CEIL",
44          "FLOOR": "FUNC_FLOOR",
45          "ROUND": "FUNC_ROUND",
46          "CONCAT": "FUNC_CONCAT",
47          "STRLEN": "FUNC_STRLEN",
48          "UCASE": "FUNC_UCASE",
49          "LCASE": "FUNC_LCASE",
50          "CONTAINS": "FUNC_CONTAINS",
51          "STRSTARTS": "FUNC_STRSTARTS",
52          "STRENDS": "FUNC_STRENDS",
53          "YEAR": "FUNC_YEAR",
54          "MONTH": "FUNC_MONTH",
55          "DAY": "FUNC_DAY",
56          "HOURS": "FUNC_HOURS",
57          "MINUTES": "FUNC_MINUTES",
58          "SECONDS": "FUNC_SECONDS",
59          "TIMEZONE": "FUNC_TIMEZONE",
60          "TZ": "FUNC_TZ",
61          "NOW": "FUNC_NOW",
62          "COALESCE": "FUNC_COALESCE",
63          "REGEX": "FUNC_REGEX",
64          "COUNT": "FUNC_COUNT",
65          "SUM": "FUNC_SUM",
66          "MIN": "FUNC_MIN",
67          "MAX": "FUNC_MAX",
68          "AVG": "FUNC_AVG",
69          "SUBSTR": "FUNC_SUBSTR",
70          "REPLACE": "FUNC_REPLACE",
71      }
72
73      tokens = [
74          # Keywords
75          *keywords.values(),
76          # Builtin calls
77          *builtin_calls.values(),
```

```
78          # Terminals
79          "ANON",
80          "BLANK_NODE_LABEL",
81          "DECIMAL",
82          "DECIMAL_NEGATIVE",
83          "DECIMAL_POSITIVE",
84          "DOUBLE",
85          "DOUBLE_NEGATIVE",
86          "DOUBLE_POSITIVE",
87          "INTEGER",
88          "INTEGER_NEGATIVE",
89          "INTEGER_POSITIVE",
90          "IRIREF",
91          "LANGTAG",
92          "NIL",
93          "PNAME_LN",
94          "PNAME_NS",
95          "STRING_LITERAL1",
96          "STRING_LITERAL2",
97          "STRING_LITERAL_LONG1",
98          "STRING_LITERAL_LONG2",
99          "VAR1",
100         "VAR2",
101         # Symbols
102         "SYMB_ASTERISK",
103         "SYMB_LP",
104         "SYMB_RP",
105         "SYMB_LCB",
106         "SYMB_RCB",
107         "SYMB_LSB",
108         "SYMB_RSB",
109         "SYMB_DOT",
110         "SYMB_COMMA",
111         "SYMB_SEMICOLON",
112         "SYMB_a",
113         "SYMB_PIPE",
114         "SYMB_SLASH",
115         "SYMB_CIRCUMFLEX",
116         "SYMB_QUESTION",
117         "SYMB_EXCLAMATION",
118         "SYMB_PLUS",
119         "SYMB_MINUS",
120         "SYMB_OR",
121         "SYMB_AND",
122         "SYMB_C2",
123         "SYMB_EQ",
124         "SYMB_NEQ",
125         "SYMB_GT",
```

```
126         "SYMB_LT",
127         "SYMB_GTE",
128         "SYMB_LTE",
129         "SYMB_TRUE",
130         "SYMB_FALSE",
131     ]
132
133     # Shared constructions
134     HEX = r"[0-9A-Fa-f]"
135     WS = r" "
136     ECHAR = r"\\[tbnrf\"\']"
137     PN_LOCAL_ESC = r"\\[_~\.\-!$&\'\(\)\*\+,;=/\?#@%]"
138     PERCENT = "%" + HEX + HEX
139     PLX = PERCENT + r"|" + PN_LOCAL_ESC
140     PN_CHARS_BASE = r"[A-Za-z]"
141     PN_CHARS_U = PN_CHARS_BASE + r"|_"
142     PN_CHARS = PN_CHARS_U + r"|\-|[0-9]"
143     PN_PREFIX = PN_CHARS_BASE + r"((" + PN_CHARS + r")|\.)*" + PN_CHARS +
                                                r")?"
144     EXPONENT = r"[eE][+-]?[0-9]+"
145     VARNAME = r"(" + PN_CHARS_U + r"|[0-9])+"
146
147     PN_LOCAL = (
148         r"("
149         + PN_CHARS_U
150         + r"|:|[0-9]|"
151         + PLX
152         + r")(("
153         + PN_CHARS
154         + r"|\.|:|"
155         + PLX
156         + ")*("
157         + PN_CHARS
158         + "|:|"
159         + PLX
160         + r"))?"
161     )
162
163     # Regexes
164
165     t_IRIREF = r'<([^<>"{}|^`\\ ])*>'
166     t_INTEGER = r"[0-9]+"
167     t_LANGTAG = r"@[a-zA-Z]+(-[a-zA-Z0-9]+)*"
168     t_DECIMAL = r"[0-9]*\.[0-9]+"
169     t_INTEGER_POSITIVE = r"\+" + t_INTEGER
170     t_INTEGER_NEGATIVE = r"-" + t_INTEGER
171     t_DECIMAL_POSITIVE = r"\+" + t_DECIMAL
172     t_DECIMAL_NEGATIVE = r"\-" + t_DECIMAL
```

```
173     t_DOUBLE = (
174         r"([0-9]+\.[0-9]*"
175         + EXPONENT
176         + ")"
177         + r"|(\.[0-9]+"
178         + EXPONENT
179         + ")"
180         + r"|([0-9]+"
181         + EXPONENT
182         + ")"
183     )
184     t_DOUBLE_POSITIVE = r"\+(" + t_DOUBLE + ")"
185     t_DOUBLE_NEGATIVE = r"\-(" + t_DOUBLE + ")"
186     t_VAR1 = r"\?" + VARNAME
187     t_VAR2 = r"\$" + VARNAME
188     t_NIL = r"\(\)"
189     t_ANON = r"\[\]"
190     t_STRING_LITERAL1 = r"\'.*\'"
191     t_STRING_LITERAL2 = r'".*"'
192     t_PNAME_NS = PN_PREFIX + r":"
193     t_PNAME_LN = t_PNAME_NS + PN_LOCAL
194     t_BLANK_NODE_LABEL = (
195         r"_:((" + PN_CHARS_U + r")|[0-9])((" + PN_CHARS + r")|.)(" +
                                                PN_CHARS + r"))?"
196     )
197     t_STRING_LITERAL_LONG1 = r"'''(.|\n)*'''"
198     t_STRING_LITERAL_LONG2 = r'"""(.|\n)*"""'
199
200     t_KW_BASE = r"BASE"
201     t_KW_PREFIX = r"PREFIX"
202     t_KW_SELECT = r"SELECT"
203     t_KW_DISTINCT = r"DISTINCT"
204     t_KW_REDUCED = r"REDUCED"
205     t_KW_WHERE = r"WHERE"
206     t_KW_HAVING = r"HAVING"
207     t_KW_ORDER = r"ORDER"
208     t_KW_BY = r"BY"
209     t_KW_ASC = r"ASC"
210     t_KW_DESC = r"DESC"
211     t_KW_LIMIT = r"LIMIT"
212     t_KW_OFFSET = r"OFFSET"
213     t_KW_OPTIONAL = r"OPTIONAL"
214     t_KW_MINUS = r"MINUS"
215     t_KW_UNION = r"UNION"
216     t_KW_FILTER = r"FILTER"
217     t_KW_AS = r"AS"
218     t_KW_NOT = r"NOT"
219     t_KW_IN = r"IN"
```

```
220     t_KW_GROUP = r"GROUP"
221     t_FUNC_RAND = r"RAND"
222     t_FUNC_ABS = r"ABS"
223     t_FUNC_CEIL = r"CEIL"
224     t_FUNC_FLOOR = r"FLOOR"
225     t_FUNC_ROUND = r"ROUND"
226     t_FUNC_CONCAT = r"CONCAT"
227     t_FUNC_STRLEN = r"STRLEN"
228     t_FUNC_UCASE = r"UCASE"
229     t_FUNC_LCASE = r"LCASE"
230     t_FUNC_CONTAINS = r"CONTAINS"
231     t_FUNC_STRSTARTS = r"STRSTARTS"
232     t_FUNC_STRENDS = r"STRENDS"
233     t_FUNC_YEAR = r"YEAR"
234     t_FUNC_MONTH = r"MONTH"
235     t_FUNC_DAY = r"DAY"
236     t_FUNC_HOURS = r"HOURS"
237     t_FUNC_MINUTES = r"MINUTES"
238     t_FUNC_SECONDS = r"SECONDS"
239     t_FUNC_TIMEZONE = r"TIMEZONE"
240     t_FUNC_TZ = r"TZ"
241     t_FUNC_NOW = r"NOW"
242     t_FUNC_COALESCE = r"COALESCE"
243     t_FUNC_REGEX = r"REGEX"
244     t_FUNC_COUNT = r"COUNT"
245     t_FUNC_SUM = r"SUM"
246     t_FUNC_MIN = r"MIN"
247     t_FUNC_MAX = r"MAX"
248     t_FUNC_AVG = r"AVG"
249     t_FUNC_SUBSTR = r"SUBSTR"
250     t_FUNC_REPLACE = r"REPLACE"
251     t_SYMB_ASTERISK = r"\*"
252     t_SYMB_LP = r"\("
253     t_SYMB_RP = r"\)"
254     t_SYMB_LCB = r"{"
255     t_SYMB_RCB = r"}"
256     t_SYMB_LSB = r"\["
257     t_SYMB_RSB = r"\]"
258     t_SYMB_DOT = r"\."
259     t_SYMB_COMMA = r","
260     t_SYMB_SEMICOLON = r";"
261     t_SYMB_a = r"a"
262     t_SYMB_PIPE = r"\|"
263     t_SYMB_SLASH = r"/"
264     t_SYMB_CIRCUMFLEX = r"\^"
265     t_SYMB_QUESTION = r"\?"
266     t_SYMB_EXCLAMATION = r"\!"
267     t_SYMB_PLUS = r"\+"
```

```
    t_SYMB_MINUS = r"\-"
    t_SYMB_OR = r"\|\|"
    t_SYMB_AND = r"&&"
    t_SYMB_C2 = r"\^\^"
    t_SYMB_EQ = r"="
    t_SYMB_NEQ = r"!="
    t_SYMB_GT = r">"
    t_SYMB_LT = r"<"
    t_SYMB_GTE = r">="
    t_SYMB_LTE = r"<="
    t_SYMB_TRUE = r"true"
    t_SYMB_FALSE = r"false"

    t_ignore = " "

    def t_newline(self, token: lex.Token):
        r"\n+"
        token.lexer.lineno += token.value.count("\n")

    def token(self) -> lex.Token:
        return self.lexer.token()

    def t_error(self, token: lex.Token):
        raise InvalidTokenError(
            "Illegal character '%s' at line %s, column %s"
            % (token.value[0], token.lexer.lineno, self.find_column(token))
        )

    def find_column(self, token: lex.Token):
        line_start = self._input.rfind("\n", 0, token.lexpos) + 1
        return (token.lexpos - line_start) + 1
```

transpiler/exceptions.py

```
class InvalidTokenError(Exception):
    """Invalid tokenization on input source  code"""
```

transpiler/structures/query.py

```
"""Main node for the structure"""
from dataclasses import dataclass, field
from typing import List, Optional

from .nodes import GraphPattern, ModifiersNode, Namespace, SelectedVar


@dataclass
class Query:
```

```
    graph_pattern: Optional[GraphPattern] = None
    variables: List[str] = field(default_factory=list)
    modifiers: ModifiersNode = field(default_factory=ModifiersNode)
    namespaces: List[Namespace] = field(default_factory=list)
    returning: List[SelectedVar] = field(default_factory=list)

    def __eq__(self, other):
        if not isinstance(other, Query):
            return False

        return (
            self.graph_pattern == other.graph_pattern
            and set(self.variables) == set(other.variables)
            and self.modifiers == other.modifiers
            and set(self.namespaces) == set(other.namespaces)
            and self.returning == other.returning
        )
```

transpiler/structures/__init__.py

transpiler/structures/nodes/graph_pattern.py

```
from dataclasses import dataclass, field
from typing import List

from .filter import FilterNode


@dataclass
class Triple:
    subject: str
    predicate: str
    object: str

    def __hash__(self):
        return hash(self.subject + self.predicate + self.object)


@dataclass
class GraphPattern:
    and_triples: List[Triple] = field(default_factory=list)
    or_blocks: List[List["GraphPattern"]] = field(default_factory=list)
    filters: List[FilterNode] = field(default_factory=list)
    minus: List["GraphPattern"] = field(default_factory=list)
    optionals: List["GraphPattern"] = field(default_factory=list)

    def __eq__(self, other):
```

```
        if not isinstance(other, GraphPattern):
            return False

        return (
            str(self.and_triples) == str(other.and_triples)
            and str(self.or_blocks) == str(other.or_blocks)
            and str(self.optionals) == str(other.optionals)
            and str(self.minus) == str(other.minus)
            and str(self.filters) == str(other.filters)
        )

    def __hash__(self):
        return hash(str(self))
```

transpiler/structures/nodes/variables.py

```
from dataclasses import dataclass
from typing import Optional, Union

from .expression import OrExpression


@dataclass
class SelectedVar:
    value: Union[str, OrExpression]
    alias: Optional[str] = None
```

transpiler/structures/nodes/__init__.py

```
from .expression import *
from .filter import *
from .graph_pattern import *
from .modifiers import *
from .namespace import *
from .variables import *
```

transpiler/structures/nodes/filter.py

```
from dataclasses import dataclass
from typing import Union

from .expression import BuiltInFunction, OrExpression


@dataclass
class FilterNode:
    constraint: Union[OrExpression, BuiltInFunction]

    def __hash__(self):
```

```
12          return hash(str(self))
```

transpiler/structures/nodes/modifiers.py

```
1  """Modifiers node"""
2  from dataclasses import dataclass
3  from typing import List, Optional, Union
4
5  from .expression import BuiltInFunction, OrExpression
6
7
8  @dataclass
9  class GroupCondition:
10     value: Union[str, OrExpression, BuiltInFunction]
11     alias: Optional[str] = None
12
13
14 @dataclass
15 class GroupClauseNode:
16     conditions: List[GroupCondition]
17
18
19 @dataclass
20 class OrderCondition:
21     value: Optional[Union[str, OrExpression, BuiltInFunction]] = None
22     order: str = "ASC"
23
24
25 @dataclass
26 class OrderNode:
27     conditions: List[OrderCondition]
28
29
30 @dataclass
31 class HavingClauseNode:
32     constraints: List[Union[OrExpression, BuiltInFunction]]
33
34
35 @dataclass
36 class ModifiersNode:
37     group: Optional[GroupClauseNode] = None
38     having: Optional[HavingClauseNode] = None
39     order: Optional[OrderNode] = None
40     limit: Optional[int] = None
41     offset: Optional[int] = None
```

transpiler/structures/nodes/namespace.py

```python
from dataclasses import dataclass


@dataclass
class Namespace:
    abbrev: str
    full: str

    def __hash__(self):
        return hash(self.abbrev + self.full)
```

transpiler/structures/nodes/expression.py

```python
from dataclasses import dataclass, field
from enum import Enum, auto
from typing import List, Optional, Tuple, Union


class PrimaryType(Enum):
    EXP = auto()
    IRI = auto()
    NUM_LITERAL = auto()
    BOOL_LITERAL = auto()
    STR_LITERAL = auto()
    VAR = auto()
    FUNC = auto()


@dataclass
class BuiltInFunction:
    name: str
    params: List["OrExpression"] = field(default_factory=list)

    def __eq__(self, __o: object) -> bool:
        if not isinstance(__o, BuiltInFunction):
            return False

        return self.name == __o.name and self.params == __o.params


@dataclass
class PrimaryExpression:
    type: PrimaryType
    value: Union[str, int, float, "OrExpression", BuiltInFunction]

    def __eq__(self, __o: object) -> bool:
        if not isinstance(__o, PrimaryExpression):
            return False
```

```
36
37          return self.type == __o.type and self.value == __o.value
38
39
40  class UnaryOperator(Enum):
41      PLUS = auto()
42      MINUS = auto()
43      NOT = auto()
44
45
46  @dataclass
47  class UnaryExpression:
48      value: PrimaryExpression
49      op: Optional[UnaryOperator] = None
50
51      def __eq__(self, __o: object) -> bool:
52          if not isinstance(__o, UnaryExpression):
53              return False
54
55          return self.value == __o.value and self.op == __o.op
56
57
58  class MultiplicativeOperator(Enum):
59      MULT = auto()
60      DIV = auto()
61
62
63  @dataclass
64  class MultiplicativeExpression:
65      base: UnaryExpression
66      others: List[Tuple[MultiplicativeOperator, UnaryExpression]] = field(
67          default_factory=list
68      )
69
70      def __eq__(self, __o: object) -> bool:
71          if not isinstance(__o, MultiplicativeExpression):
72              return False
73
74          return self.base == __o.base and set(self.others) ==
                                                 set(__o.others)
75
76
77  class AdditiveOperator(Enum):
78      SUM = auto()
79      SUB = auto()
80
81
82  @dataclass
```

```
83  class AdditiveExpression:
84      base: MultiplicativeExpression
85      others: List[Tuple[AdditiveOperator, MultiplicativeExpression]] =
                                                field(
86          default_factory=list
87      )
88
89      def __eq__(self, __o: object) -> bool:
90          if not isinstance(__o, AdditiveExpression):
91              return False
92
93          return self.base == __o.base and set(self.others) ==
                                                    set(__o.others)
94
95
96  class LogOperator(Enum):
97      EQ = auto()
98      NEQ = auto()
99      LT = auto()
100     GT = auto()
101     LTE = auto()
102     GTE = auto()
103     IN = auto()
104     NOT_IN = auto()
105
106
107 @dataclass
108 class RelationalExpression:
109     first: AdditiveExpression
110     second: Optional[Tuple[LogOperator, AdditiveExpression]] = None
111
112     def __eq__(self, __o: object) -> bool:
113         if not isinstance(__o, RelationalExpression):
114             return False
115
116         return self.first == __o.first and self.second == __o.second
117
118
119 @dataclass
120 class AndExpression:
121     base: RelationalExpression
122     others: List[RelationalExpression] = field(default_factory=list)
123
124     def __eq__(self, __o: object) -> bool:
125         if not isinstance(__o, AndExpression):
126             return False
127
128         return self.base == __o.base and set(self.others) ==
```

```
                                                set(__o.others)
129
130
131 @dataclass
132 class OrExpression:
133     base: AndExpression
134     others: List[AndExpression] = field(default_factory=list)
135
136     def __eq__(self, __o: object) -> bool:
137         if not isinstance(__o, OrExpression):
138             return False
139
140         return self.base == __o.base and set(self.others) ==
                                                set(__o.others)
```

APÊNDICE E – ARTIGO

Neste apêndice será apresentado o artigo no formato SBC, referente ao presente projeto.

Instructions for Authors of SBC Conferences Papers and Abstracts

Thiago Sant'Helena da Silva[1], **Ronaldo dos Santos Mello**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

sh.thiago@pm.me, r.mello@ufsc.br

Abstract. *This article brings the development of Switch, a tool that aims to facilitate the construction of semantic databases (triplestores) that query using SPARQL and store data in Neo4j. To this do so, concepts of language processing and code generation are used through semantic actions. In addition to the development of the tool, this work also brings a literature review on the State of the Art in the use and development of triplestores. The tool was successfully developed and was able to translate various forms of consultation. However, there are improvements to be made cited as future work.*

Resumo. *Este artigo trás o desenvolvimento da Switch, ferramenta que visa facilitar a construção de bancos de dados semânticos (triplestores) que implementem consultas utilizando SPARQL e armazenem dados em Neo4j. Para isso, são utilizados conceitos de processamento de linguagens e geração de código através de ações semânticas. Além do desenvolvimento da ferramenta, esse trabalho também trás uma revisão bibliográfica a respeito do Estado da Arte do uso e desenvolvimento de triplestores. A ferramenta foi desenvolvida com sucesso e foi capaz de traduzir diversas formas de consulta. No entanto, existem melhorias a serem feitas citadas como trabalhos futuros.*

1. Introdução

A partir da proposição da World Wide Web (WWW) no início dos anos 90, por Tim Berners-Lee, dados começaram a ser transmitidos através de continentes, criando uma rede quase unificada de informação. A estrutura e protocolos propostos trouxeram ao mundo um novo paradigma de geração e compartilhamento de conhecimento [Berners-Lee et al. 1992].

Nesse contexto, o padrão para descrição de dados em *Resource Description Framework* (RDF[1]), proposto por um grupo de pesquisa da W3C[2] em 1997, se populariza como forma de descrever e publicar dados online. O RDF serviu de base para a criação de diversas outras especificações utilizadas para enriquecer a descrição de informações, como a *Ontology Web Language* (OWL[3]) e a *Simple Knowledge Organization System* (SKOS[4]). Essas especificações são utilizadas para criar vocabulários (ou ontologias[5])

[1]https://www.w3.org/RDF/
[2]https://www.w3c.br/
[3]https://www.w3.org/2001/sw/wiki/OWL
[4]https://www.w3.org/2001/sw/wiki/SKOS
[5]https://www.w3.org/standards/semanticweb/ontology

abertos que aplicam semântica em conjuntos de dados, dando origem a ideia de *Web Semântica* [Berners-lee et al. 2001].

O uso de SPARQL se dá sobre dados em RDF, e a linguagem traz uma sintaxe pouco verbosa, onde diversas estruturas da álgebra relacional podem ser simplificadas para relações entre triplas. É possível armazenar dados em RDF de diversas formas, a medida que se defina um padrão de mapeamento entre o dado com o sistema de armazenamento utilizado. Existem mapeamentos propostos para esquemas relacionais, por [Berners-lee et al. 1998] e esquemas de grafos no banco de dados Neo4j[6] com a extensão *Neosemantics*[7], por exemplo.

2. Conceitos básicos

2.1. Web Semântica

A Web Semântica, proposta por Tim Berners-Lee, é uma extensão da WWW criada pelo mesmo autor anos antes, pensada para que os dados compartilhados pelas páginas tenham formato mais amigável para máquinas. Dessa forma, agentes automatizados conseguiriam fazer interpretações sobre os dados encontrados por estes em páginas da web de modo a inferir significados ou encontrar informações solicitadas por um usuário.

Esse conceito é importante para este trabalho a medida que, para se operar com dados carregados de significado, tecnologias que possibilitem o armazenamento dessa semântica sejam aprimoradas e popularizadas.

2.2. RDF

Junto com a XML[8], o RDF foi apontado como uma das principais tecnologias para a Web Semântica por [Berners-lee et al. 2001]. Uma formalização dos conceitos estruturais do RDF é dada por [Ladwig and Harth 2011] (tradução livre):

> (Tripla RDF, Termo RDF, Grafo RDF) Dado um conjunto de URIs $\mathcal{I}$, um conjunto de nodos vazios $\mathcal{B}$ e um conjunto de valores literais $\mathcal{L}$:
>
> $$(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$$
>
> é chamada de Tripla RDF. Nós chamamos elementos de $\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ termos RDF. Conjuntos de Triplas RDF são chamados Grafos RDF.

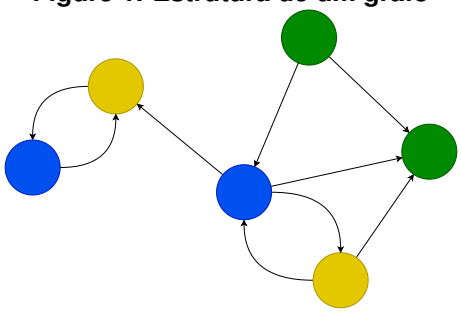
O formato RDF herda o sistema de tipagem do *Extensible Markup Language* (XML), onde estão definidas os tipos primitivos entendidos pelo padrão como valores literais possíveis. Existem algumas formas de serialização de dados RDF, como o próprio XML e o formato *Turtle*[9].

2.3. SPARQL

SPARQL é uma sigla recursiva para *SPARQL Protocol and RDF Query Language*[10]. É uma linguagem declarativa criada especificamente para executar consultas em dados no

[6]https://neo4j.com/
[7]https://neo4j.com/labs/neosemantics/
[8]https://www.w3.org/TR/REC-xml/
[9]https://www.w3.org/TR/turtle/
[10]https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

Figure 1. Estrutura de um grafo



formato RDF. De maneira semelhante ao SQL para dados relacionais, a linguagem oferece recursos para consulta, inserção, remoção e atualização para dados no formato RDF em páginas web ou *triplestores*, padronizada pela W3C.[11].

2.4. Triplestore

Triplestores ou *RDF Stores* são sistemas de armazenamento de triplas RDF. Implementações de tais sistemas são feitas com base em outros sistemas de armazenamento de dados, como bancos de dados relacionais ou de documentos, ou criados especificamente para o padrão.

Exemplo notável de *triplestore*, o *Openlink Virtuoso*[12] é capaz de armazenar bilhões de triplas em suas versões mais recentes e serve de base para projetos abertos de catalogação de informação, como o *WikiData*[13].

2.5. Banco de dados orientados a grafos

Bancos de dados orientados a grafos são, de modo geral, a implementação de estruturas de armazenamento fortemente baseadas na teoria de grafos. Dessa forma, os conceitos centrais desse tipo de banco são objetos e as relações entre eles, como demonstrado na Figura 1. São especializados para o armazenamento de entidades na forma de vértices (ou nodos), e as relações entre essas entidades são feitas através de arestas. Toda aresta tem, necessariamente, um nodo de partida e um de chegada, um tipo e uma direção

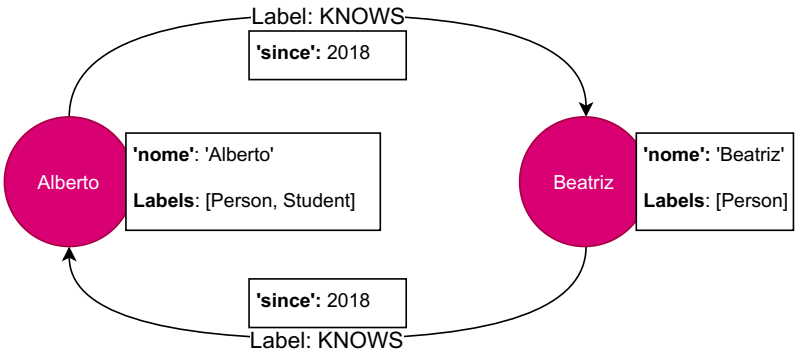
2.6. Neo4j

O Neo4j é um dos bancos de dados orientados a grafos mais populares atualmente. Cada elemento no banco é representado por um vértice (ou nodo) que tem relações com outros elementos através de arestas. Tanto vértices quanto arestas podem ter um conjunto de propriedades, como mostra a Figura 2.

Tanto nodos quanto arestas podem ter *labels*, que são usados para associar tipos, facilitando o processo de consulta e aumentando a consistência das estruturas armazenadas. Arestas devem ter um *label*, enquanto nodos podem ter um número qualquer de *labels*, criando a possibilidade de um mesmo nodo representar mais de um tipo, como mostrado na Figura 2 onde os nodos tem os *labels Person* e/ou *Student*.

[11]https://www.w3.org/TR/sparql11-overview/
[12]https://virtuoso.openlinksw.com/
[13]https://www.wikidata.org/wiki

Figure 2. Exemplo de dados em grafos



2.7. Neosemantics

A extensão *Neosemantics* adiciona um conjunto de funções ao Neo4j para manipulação de dados em RDF, porém esta não conta com suporte para pesquisa sobre os dados armazenados utilizando SPARQL, sendo esse o espaço que este trabalho busca preencher.

Essas funções serão usadas como base do desenvolvimento da ferramenta proposta, uma vez que elas já implementam um mecanismo de carregamento de dados em RDF para o esquema de grafos utilizado através de um mapeamento específico.

2.8. Cypher

Cypher é uma linguagem declarativa, criada inicialmente para execução de consultas no banco de dados Neo4j. Dessa forma, ela é especializada para a descrição das relações dentro de um conjunto de dados estruturado baseado em grafos. Em 2015, a linguagem foi tornada um projeto de código aberto independente pela iniciativa *openCypher*[14].

3. Trabalhos relacionados

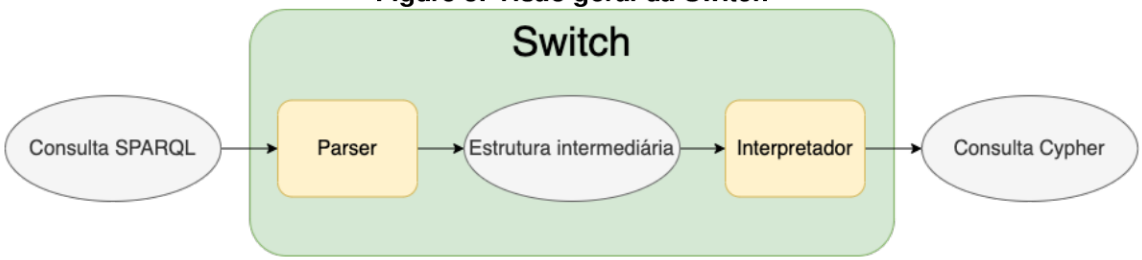
3.1. A Middleware For Workload-Aware Manipulation of RDF Data Stored Into NoSQL Databases

Nesta tese os autores demonstram os resultados de uma série de estudos e publicações passadas a respeito do desenvolvimento de um *triplestore* denominado WA-RDF, capaz de armazenar dados em múltiplos bancos de dados NoSQL [Santana and dos Santos Mello 2017]. A principal motivação por trás da utilização de diferentes bancos de dados NoSQL para a implementação dos autores é tentar utilizar as operações mais eficientes que cada sistema oferece.

A importância do WA-RDF para este trabalho é a demonstração feita pelos autores de que é possível utilizar o Neo4j como camada de armazenamento de dados para construção de *triplestores*. Nesse trabalho, tentaremos abrir ainda mais essa possibilidade, apresentando um método para tradução de consultas SPARQL para Cypher sem a necessidade de que os dados armazenados precisem ter passado por um pré-processamento específico para armazenar os dados de acordo com as estruturas que ele forme.

[14]http://opencypher.org/

Figure 3. Visão geral da Switch



3.2. Querying Heterogeneous Property Graph Data Sources based on a Unified Conceptual View

[Fathy et al. 2020] traz uma forma de traduzir as duas linguagens através de uma álgebra intermediária chamada xR2RML, proposta por [Michel et al. 2016]. A tradução envolve um processo de três passos, envolvendo uma etapa de tradução de SPARQL para expressões em xR2RML, manipulações sobre as expressões geradas para evitar duplicações dos dados e o uso das expressões algébricas reescritas para geração da consulta em Cypher.

Os autores efetuam a tradução de um conjunto de consultas sobre duas populares bases de dados e apresentam os resultados na forma do tempo necessário para as traduções. A tradução para uma álgebra intermediária que é posteriormente otimizada antes da geração da consulta correspondente em Cypher parece ser o gargalo desta abordagem.

3.3. SARQLing Neo4j

Os autores propõem uma ferramenta nos mesmos moldes da pretendida no presente trabalho [Lombardot et al. 2019]. A abordagem assumida por eles foi de construir um *parser* para a linguagem SPARQL utilizando o *framework* PEG.js[15] e associar as ações semânticas que gerariam a consulta equivalente em Cypher, também supondo uma base de dados RDF carregada em Neo4j utilizando o Neosemantics.

De modo geral, os autores descrevem um processo de tradução entre SPARQL e Cypher baseado em ações semânticas. No entanto, não são feitas considerações ou demonstrações que levem em conta as funções nativas ou os modificadores de resultado (`LIMIT`, `GROUP BY` e outros) da SPARQL. Essa lacuna é a que o presente trabalho busca cobrir.

4. Switch: executando SPARQL sobre Neo4j

Todo código criado para este trabalho está disponível no repositório deste trabalho e a versão aqui apresentada pode ser acessada pela *tag* `v0.1.1-alpha`[16]. Uma visão geral da ferramenta é apresentada na Figura 3. No decorrer desta seção, cada parte da figura será explicada em detalhes.

É importante destacar que, para a delimitação de escopo, a linguagem SPARQL foi modificada para este trabalho, removendo algumas funções nativas, estruturas de

[15]https://pegjs.org
[16]https://github.com/shthiago/switch/tree/v0.1.1-alpha

```python
@dataclass
class Query:
    graph_pattern: Optional[GraphPattern] = None
    variables: List[str] = field(default_factory=list)
    modifiers: ModifiersNode = field(default_factory=ModifiersNode)
    namespaces: List[Namespace] = field(default_factory=list)
    returning: List[SelectedVar] = field(default_factory=list)
```

Listing 1: Raiz da estrutura intermediária

interconexão entre bases de dados disponibilizadas em *endpoints* de dados semânticos públicos e toda consulta que não inicie por `SELECT`. Os passos feitos para a modificação da linguagem e os estados intermediários desta estão disponíveis no repositório do projeto.

4.1. Parser

Para criar a estrutura intermediária, as ações semânticas operam sobre uma instância da estrutura intermediária, de modo preencher ela com os valores encontrados na consulta de entrada. O parser foi criado utilizando a biblioteca PLY[17] e o código pode ser encontrado no repositório na pasta `transpiler/parser.py`. Este faz uso do *lexer*, também disponível no repositório para criar um analisador sintático e criar as instâncias de cada bloco da estrutura a medida que a consulta de entrada é processada.

4.2. Estrutura intermediária

A estrutura proposta se assemelha a uma árvore de análise sintática, porém reorganizada para agrupar as características relevantes da consulta de entrada. O código da raiz da estrutura (apresentado no Código 1) contém as referências para o que definimos como as partes principais de uma consulta SPARQL: padrão do grafo pesquisado(`graph_pattern`, instância de `GraphPattern`), variáveis existentes (`variables`, lista de `strings`), modificadores de resultado (`modifiers`, instância de `ModifiersNode`), *namespaces* (`namespaces`, lista de instâncias de `Namespace` e bloco de variáveis retornadas pela consultas (`returning`, lista de instâncias de `SelectedVar`).

4.3. Interpretador

O interpretador da estrutura executa um processo iterativo sobre a estrutura intermediária, gerando um bloco de consulta para cada tripla da consulta de entrada. Os blocos são concatenados de maneira que as variáveis definidas por um bloco sejam consideradas nos subsequentes, e que as cláusulas `UNION` sejam respeitadas de modo que o resultado da consulta Cypher sobre Neo4j seja equivalente a SPARQL sobre uma *triplestore*.

Tomando como exemplo a consulta no Código 2, a consulta Cypher correspondente gerada pela Switch é a apresentada no Código 3. Cada tripla da consulta é convertida em um bloco de código que filtra os nodos e arestas de maneira equivalente. Para a primeira tripla, da linha 4 do Código 2, o bloco gerado vai da linha 1 até a linha 5 do Código 3. O predicado `rdf:type` é transformado em uma *label* do nodo, então o nodo nomeado por `country` na consulta em Cypher é selecionado baseado em se o nodo tem a *label*.

[17] http://www.dabeaz.com/ply/

```
1  PREFIX b:<http://www.geonames.org/ontology#>
2  PREFIX dct:<http://purl.org/dc/terms/>
3  SELECT ?country (COUNT(?state) AS ?stateCount) WHERE {
4      ?country rdf:type b:Country .
5      ?country dct:hasPart ?state
6  } GROUP BY ?country
```

Listing 2: Consulta SPARQL exemplo

```
1  MATCH (country)
2  WHERE
3      n10s.rdf.shortFormFromFullUri("http://www.geonames.org/ontology#") + "Country"
4      IN labels(country)
5  WITH country AS country
6  UNWIND [key in keys(country)
7      WHERE key =
8          n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"
9     | [country, key, country[key]]] + [(country)-[relation]-(state)
10     WHERE type(relation) =
11         n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"
12    | [country, relation, state]] AS triples
13 WITH triples[0] AS country, triples[2] AS state
14 RETURN country, count(state) AS stateCount
```

Listing 3: Consulta Cypher exemplo

Na linha 3 do Código 3, a função `n10s.rdf.shortFormFromFullUri` é utilizada para obter o nome da *label* gerada pela Neosemantics no carregamento dos dados, dado que os *namespaces* são abreviados e sem consultar o banco previamente não necessariamente sabemos como cada *namespace* é nomeado. A URI completa do *namespace* é obtida da estrutura gerada, e a *string* `"Country"` é obtida da própria tripla.

A cláusula `WITH` é utilizada ao final de cada bloco, de modo que todas as transformações e filtros aplicados no bloco atual seja levado em conta no próximo de maneira contextualizada e genérica do ponto de vista da implementação.

A tripla da linha 5 do Código 2 gera o bloco das linhas 6 até 13 do Código 3. A cláusula `UNWIND` é aplicada sobre a concatenação de duas listas geradas a partir de duas sub-consultas. A primeira lista é gerada a partir das propriedades do nodo `country`, supondo que a tripla buscada tenha sido carregada como uma tripla pela Neosemantics. A comparação `key = n10s.rdf.shortFormFromFullUri("http://purl.org/dc/terms/") + "hasPart"` garante que apenas as propriedades com a chave correspondente ao predicado usado na tripla será levada em conta. O resultado é uma lista de tuplas de três elementos, onde o primeiro é o nodo do *subject* da tripla, o segundo é o valor da chave da propriedade representando o *predicate* da tripla e o último o valor da chave para o nodo, representando o *object* da tripla.

A segunda lista é gerada por uma sub-consulta que supõe que a tripla tenha sido carregada como uma relação entre dois nodos. Então a sub-consulta busca todos os nodos que se relacionem com o *subject* através de uma relação qualquer e filtra baseado no valor do tipo da relação, que é ditado pela URI do predicado. Caso o *object* da tripla fosse uma URI, esse também seria usado para filtrar os resultados pela propriedade `uri` do nodo na posição de *object*. Da mesma forma que a lista anterior, essa é composta por tuplas de

três elementos.

A cláusula `UNWIND` transforma a lista de listas em uma tabela de três colunas, e suas colunas são renomeadas na linha 13 pela cláusula `WITH`. Para finalizar a consulta, a cláusula `RETURN` é adicionada ao final de acordo com o bloco de retorno da consulta de entrada. Nesse caso, o `GROUP BY` é reproduzido implicitamente na consulta em Cypher pela chamada para a função `count(state)`.

Outros exemplos de consultas utilizadas para teste estão disponíveis no repositório do projeto, junto a um script para a execução desta.

5. Conclusão

De modo geral, a ferramenta conseguiu traduzir consultas com sucesso. A consulta resultante teve resultados equivalentes para todas as consultas testadas. Apesar de o escopo da ferramenta ter sido reduzido de diversas formas ao longo do desenvolvimento, dentro de suas limitações a tradução funciona como esperado.

O tempo de tradução das consultas varia de acordo com a consulta de entrada. A tabela 1 apresenta os tempos de tradução e execução das consultas para fins comparativos. Podemos observar que os tempos de execução da Switch é relativamente constante para as entradas utilizadas. O tempo de execução das consultas varia muito de acordo com as operações necessárias. As consultas em SPARQL foram executadas com os dados em memória, enquanto as em Cypher foram executadas sobre o banco que armazena os dados em disco, portanto a comparação entre as duas não é interessante.

Como trabalhos futuros, incluímos o ajuste da ferramenta para reduzir suas limitações e aproximar a ferramenta de uma tradução completa da linguagem SPARQL. Diversas funções de SPARQL se mantém não traduzíveis pela Switch, implementar estas também faz parte do desenvolvimento para trablalhos futuros. Uma extensão para Neo4j cobrindo as funções que não puderam ser reproduzidas facilitaria esse trabalho. Por fim, a ferramenta poderia também ser adaptada para a linguagem Java de modo que pudesse ser incluída na extensão Neosemantics ou mesmo criar uma nova extensão.

Teste	**SPARQL (s)**	**Switch (ms)**	**Cypher (s)**	**Resulados iguais**
test_query_1.sparql	0.002	2.2	0.009	Sim
test_query_2.sparql	0.004	1.8	0.02	Sim
test_query_3.sparql	0.003	2.0	0.011	Sim
test_query_4.sparql	0.003	2.3	0.012	Sim
test_query_5.sparql	0.079	1.7	0.027	Sim
test_query_6.sparql	0.042	2.5	0.246	Sim
test_query_7.sparql	0.003	1.7	0.167	Sim
test_query_8.sparql	0.058	1.7	2.808	Sim
test_query_9.sparql	0.057	1.9	2.683	Sim
test_query_10.sparql	0.001	1.6	3.195	Sim
test_query_11.sparql	0.095	1.7	2.992	Sim
test_query_12.sparql	0.087	1.9	3.048	Sim

Table 1. Todos os testes

References

Berners-Lee, T., Cailliau, R., and Groff, J. (1992). The world-wide web. *Comput. Networks ISDN Syst.*, 25(4-5):454–459.

Berners-lee, T. et al. (1998). Relational databases on the semantic web.

Berners-lee, T., Lassila, O., and Hendler, J. (2001). The semantic web. *The Scientific America*, page 28–37.

Fathy, N., Gad, W., Badr, N., and Hashem, M. (2020). Querying heterogeneous property graph data sources based on a unified conceptual view. In *Proceedings of the 2020 9th International Conference on Software and Information Engineering (ICSIE)*, ICSIE 2020, page 113–118, New York, NY, USA. Association for Computing Machinery.

Ladwig, G. and Harth, A. (2011). Cumulusrdf: Linked data management on nested key-value stores. *SSWS 2011*.

Lombardot, T., Morgat, A., Axelsen, K. B., Aimo, L., Hyka-Nouspikel, N., Niknejad, A., Ignatchenko, A., Xenarios, I., Coudert, E., Redaschi, N., and Bridge, A. (2019). Updates in rhea: Sparqling biochemical reaction data. *Nucleic Acids Res.*, 47(Database-Issue):D596–D600.

Michel, F., Djimenou, L., Faron-Zucker, C., and Montagnat, J. (2016). Translation of heterogeneous databases into rdf, and application to the construction of a skos taxonomical reference. In Monfort, V., Krempels, K.-H., Majchrzak, T. A., and Turk, Ž., editors, *Web Information Systems and Technologies*, pages 275–296, Cham. Springer International Publishing.

Santana, L. H. Z. and dos Santos Mello, R. (2017). Workload-aware RDF partitioning and SPARQL query caching for massive RDF graphs stored in nosql databases. In Hara, C. S., editor, *XXXII Simpósio Brasileiro de Banco de Dados, Uberlandia, MG, Brazil, October 4-7, 2017*, pages 184–195. SBC.