# Switch: Executando SPARQL sobre o Neo4j

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

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

thiago.sant.helena@gmail.com, r.mello@ufsc.br

***Abstract.*** *This article details Switch, an approach that facilitates the construction of triplestores that query using SPARQL and store data in Neo4j. In order to do that, concepts of language processing and code generation through semantic actions are considered. The solution was developed and it is able to translate and execute several kinds of SPARQL queries. Experiments had demonstrated that the solution is viable in terms of processing time.*

***Resumo.*** *Este artigo detalha Switch, uma abordagem que facilita a construção de triplestores que suportam consultas utilizando SPARQL e armazenam dados no Neo4j. Para isso, são utilizados conceitos de processamento de linguagens e geração de código através de ações semânticas. A solução foi desenvolvida e é capaz de traduzir diversos tipos de consultas SPARQL. Experimentos demonstraram que a solução é viável em termos de tempo de processamento.*

## 1. Introdução

A partir da proposição da *World Wide Web (WWW)* no início dos anos 90, dados começaram a ser transmitidos através de continentes, criando uma rede quase unificada de informação [Berners-Lee et al. 1992]. Nesse contexto, o padrão para descrição de dados *RDF Resource Description Framework*[1] se popularizou como forma de descrever e publicar dados *online*. O RDF serviu de base para a criação de outras especificações utilizadas para enriquecer a descrição de informações, como a *Ontology Web Language* (OWL)[2]. Essas especificações são utilizadas para criar vocabulários (ou ontologias)[3] que aplicam semântica a conjuntos de dados, dando origem à *Web Semântica* [Berners-lee et al. 2001].

Os dados da Web Semântica, embora respeitem vocabulários pré-definidos, não geram dados adequados a tabelas. Dessa forma, o uso de bancos de dados (BDs) relacionais se torna menos interessante, e surge a ideia de sistemas de armazenamento de dados no formato RDF, chamados de *triplestores*. Um *triplestore* utiliza uma linguagem de consulta denominada *SPARQL (the Simple Protocol and RDF Query Language)*[4].

*Triplestores* é um tópico de pesquisa na literatura e diversos autores sugerem estruturas a serem utilizadas para a implementação deste conceito. Um exemplo é o trabalho de [Santana and Mello 2017], que propõe a combinação de diversas tecnologias na camada

[1]https://www.w3.org/RDF/
[2]https://www.w3.org/2001/sw/wiki/OWL
[3]https://www.w3.org/standards/semanticweb/ontology
[4]https://www.w3.org/TR/sparql11-query/

de armazenamento de dados. Um desafio nessas implementações é a execução de consultas SPARQL, independentemente da estrutura utilizada na camada de armazenamento.

Esta é a motivação principal para a abordagem *Switch*, um mecanismo de tradução de consultas SPARQL para a linguagem Cypher[5], que é a linguagem de consulta utilizada pelo sistema de gerência de BD (SGBD) Neo4j[6], um dos principais SGBDs orientados a grafos da atualidade, e que se mostra eficiente para o acesso a dados fortemente relacionados. Esse mecanismo de tradução pode viabilizar sistemas de busca baseados na Web Semântica que utilizam a estrutura de grafo para o armazenamento de *triplestores*.

A literatura pouco explora a conexão entre SPARQL e Cypher [Santana and Mello 2017, Lombardot et al. 2019, Fathy et al. 2020]. A abordagem aqui proposta se diferencia pelo suporte a consultas SPARQL complexas (suporte à múltiplos padrões de triplas no corpo da consulta) através de um mecanismo de tradução com baixo *overhead* de processamento.

Este artigo possui mais 5 seções. A Seção 2 apresenta a fundamentação teórica e a Seção 3 discute trabalhos relacionados. A Seção 4 detalha a abordagem proposta, a Seção 5 apresenta uma avaliação da abordagem e a Seção 6 é dedicada à conclusão.

## 2. Fundamentação Teórica

### 2.1. RDF e SPARQL

RDF é uma das principais tecnologias da Web Semântica para representação de dados. Um dado em RDF é expresso por uma *tripla (S,P,O)* que define um relacionamento entre 2 recursos. Triplas RDF podem ser modeladas como grafos onde os recursos, denominados *sujeito (S)* e *objeto (O)*, são vértices, e o relacionamento, chamado *predicado (P)*, é uma aresta direcionada de S para O. Pode-se definir, por exemplo, um predicado *:knows* (conhece) entre 2 recursos do tipo *:person* (pessoa - S e O).

SPARQL é uma linguagem de consulta para dados RDF. Um exemplo de consulta SPARQL, que busca o número de territórios de 10 países, é mostrado no código a seguir:

```
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
7 LIMIT 10
```

A cláusula `PREFIX` permite a definição de um prefixo (geralmente uma URI) que pode ser utilizado ao longo da consulta. A cláusula `SELECT` define o resultado da consulta, sendo possível definir variáveis com o prefixo '`?`' e chamar funções especificadas pela linguagem para modificação do resultado, como é o caso da função `COUNT`. A cláusula `WHERE` define a condição da consulta através da especificação de padrões de busca a serem encontrados em um grafo RDF. Por fim, a SPARQL dispõe de recursos para modificação de resultados, utilizando cláusulas similares à linguagem SQL: `LIMIT`, `OFFSET`, `GROUP BY` e `HAVING`.

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

[6]https://neo4j.com

## 2.2. Neo4j e Cypher

O Neo4j é o SGBD NoSQL orientado a grafos mais popular atualmente. Um item de dado é representado por um vértice (ou nodo) que tem relações com outros itens através de arestas. Tanto nodos quanto arestas podem ter propriedades, como mostra a Figura 1.

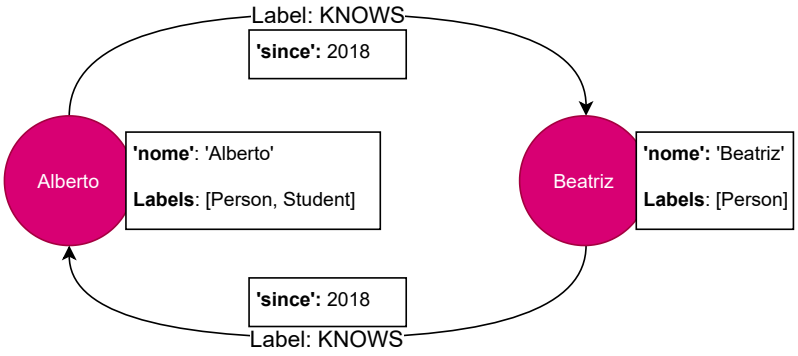


**Figure 1. Exemplo de um BD no Neo4j**

Nodos e arestas podem ter também rótulos (ou *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*, permitindo que um mesmo nodo represente mais de um tipo.

Cypher é uma linguagem declarativa, criada inicialmente para execução de consultas no SGBD Neo4j. O código a seguir exemplifica uma consulta em Cypher que retorna nomes de pessoas e suas quantidades de pessoas conhecidas:

```
MATCH (person)-[:KNOWS]->(friend)
WHERE person.nome IS NOT NULL
RETURN person.nome, COUNT(friend) AS count
```

A cláusula `SELECT` da SPARQL é representada em Cypher com a cláusula `RETURN`, incluindo a função de agregação `COUNT`, que em Cypher conta com uma operação `GROUP BY` implícita. A cláusula `WHERE` em SPARQL precisa de uma combinação de duas cláusulas (`MATCH` e `WHERE`) em Cypher.

## 2.3. Neosemantics

A extensão *Neosemantics* do Neo4j adiciona funções ao Neo4j para manipulação de dados em RDF. Entretanto, ela não conta com suporte para pesquisa sobre os dados armazenados utilizando SPARQL, sendo esta a lacuna que este trabalho busca preencher. Essas funções são usadas como base para o desenvolvimento da solução proposta, uma vez que elas já implementam um mecanismo de carga de dados em RDF para um esquema de grafos através de um mapeamento específico sumarizado na Tabela 1 [Barrasa 2023]. A Figura 2 mostra um mapeamento considerando o *object* como um conteúdo não-literal.

# 3. Trabalhos relacionados

Para a busca de referências relacionadas ao tema deste trabalho, uma metodologia de Revisão Sistemática (RS) foi utilizada [Kitchenham 2004]. A *string* de busca `SPARQL AND (Cypher OR Neo4j)` foi definida e executada sobre os principais repositórios

**Table 1. Mapeamento de RDF para um grafo Neo4j**

| esquema RDF | esquema Neo4j |
|---|---|
| *namespace* | nodo com *label* `_NsPrefDef` e com propriedade cujo nome é a abreviação do *namespace* e o valor é a URL base |
| *subject* | nodo com *label* `Resource` e com propriedade `uri` que mantém o valor do sujeito |
| *object* | mapeamento idêntico ao *subject*, caso não seja um valor literal |
| *predicate* | propriedade do nodo *subject* com valor igual ao valor do *object*, caso o objeto seja um valor literal. Caso contrário, gera-se uma aresta entre os nodos criados para *subject* e *object* |

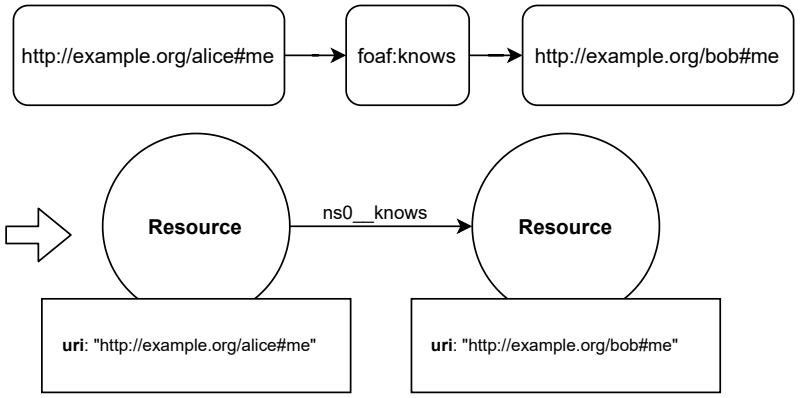


**Figure 2. Exemplo de mapeamento RDF para grafo Neo4j**

de publicações científicas. Ao final da RS, os resultados obtidos são mostrados na Tabela 2. A exclusão da grande maioria dos trabalhos ocorreu após a leitura do resumo ou do texto complexo por estarem fora de escopo, como trabalhos que realizam comparações de desempenho entre triplestores e SGBDs orientados a grafos. Além dos 3 artigos selecionados, foi adicionada manualmente uma tese de doutorado.

**Table 2. Resultado da aplicação da revisão sistemática**

| 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 |

O primeiro trabalho discorre sobre o uso de SPARQL em *Gremlin*, uma linguagem de consulta sobre grafos, trazendo subsídios para a utilização de SPARQL sobre Neo4j [Thakkar et al. 2018]. Mesmo assim, não há uma relação entre SPARQL e Cypher.

Um segundo trabalho propõe uma ferramenta similar à abordagem pretendida por este trabalho [Lombardot et al. 2019]. Ela constroi um *parser* para a linguagem SPARQL e associa ações semânticas que geram a consulta equivalente em Cypher, também supondo uma base de dados RDF carregada em Neo4j utilizando o Neosemantics. No entanto, os autores não descrevem a tradução de consultas que definem mais de uma tripla na cláusula `WHERE` e como a tradução lida com a importação de *namespaces* pelo Neosemantics.

Um terceiro trabalho apresenta uma forma de tradução SPARQL-Cypher através de uma álgebra intermediária chamada *xR2RML* [Fathy et al. 2020]. A tradução se dá em 3 passos: uma etapa de tradução de SPARQL para expressões xR2RML, manipulações sobre as expressões geradas para evitar duplicações dos dados e o uso de expressões algébricas reescritas para geração da consulta em Cypher. A principal desvantagem desta proposta é justamente o *overhead* introduzido com a utilização da *xR2RML*.

Por fim, a tese de doutorado apresenta o *triplestore WA-RDF*, capaz de armazenar dados em múltiplos BDs NoSQL [Santana and Mello 2017]. Os SGBDs escolhidos foram o MongoDB e o Neo4j. No caso do Neo4j, ele adota uma premissa muito forte: os dados são persistidos no Neo4j com base nas topologias de consulta mais frequentes. Dessa forma, as traduções são diretas e relativamente simples. A abordagem aqui proposta (*Switch*) define um método para tradução de consultas SPARQL para Cypher sem a necessidade dos dados passarem por um pré-processamento para serem armazenados de acordo com as estruturas definidas no corpo da consulta SPARQL. Ela é detalhada a seguir.

## 4. A Abordagem *Switch*

A abordagem *Switch* é um processo de tradução de uma consulta SPARQL para uma consulta equivalente Cypher a ser executada no SGBD Neo4j. Uma visão geral da *Switch* é apresentada na Figura 3. Ela considera como entrada uma versão enxuta do comando de consulta SPARQL original que considera consultas iniciadas com a palavra `SELECT` e cláusulas que obtêm dados de múltiplas fontes RDF da *Web*. Desconsidera-se funções definidas pelo usuário, subconsultas e diversas funções nativas da SPARQL, como `STR...`, `EXISTS`, `is...` e `IF`, devido às suas complexidades. Maiores detalhes podem ser encontrados no trabalho de [Silva 2022]. Uma gramática desta versão foi definida para ser utilizada no primeiro passo da abordagem (*Parser*).

O *Parser* é responsável pela verificação de uma consulta SPARQL válida como entrada. Utiliza-se esta 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. Um exemplo de consulta SPARQL válida para a *Switch* é mostrado na Seção 2.1.

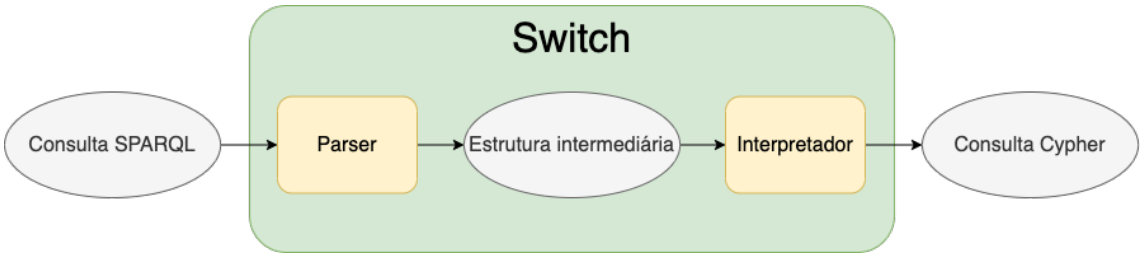


**Figure 3. Visão geral da abordagem *Switch***

Criar ações semânticas no *Parser* para serem associadas às produções da consulta SPARQL e gerar diretamente a consulta equivalente em Cypher seria complexo, dado que as estruturas das linguagens são significativamente diferentes. Dessa forma, optou-se 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 intermediária (listagem 1) contém as referências para as partes principais da consulta SPARQL: padrão do grafo pesquisado (`graph_pattern`), variáveis existentes (`variables`) (prefixadas com '?'), modificadores de resultado (`modifiers`), *namespaces* e bloco de variáveis retornadas pela consultas (`returning`). Em suma, a estrutura `GraphPattern` descreve o conteúdo da cláusula `WHERE` da consulta SPARQL. Esta cláusula pode conter padrões de triplas, como exempliicado nas linhas 4 e 5 da consulta SPARQL da Seção 2.1, e algumas cláusulas especiais, como `UNION`, `MINUS` e `FILTER`. A listagem 2 mostra um exemplo de instância da subestrutura `GraphPattern` para a cláusula WHERE da consulta SPARQL da Seção 2.1. Outras subestruturas foram definidas para a estrutura intermediária [Silva 2022].

```
@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

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

Listing 2: Exemplo de instância de `GraphPattern`

A linguagem *Python* foi utilizada para o desenvolvimento da *Switch*, com o apoio da biblioteca *PLY*[7] para a geração dos analisadores léxico e sintático para a consulta SPARQL utilizados pelo *Parser* para a geração da estrutura intermediária. O código completo da *Switch*, incluindo a gramática da consulta SPARQL, está disponível para acesso[8].

A partir da estrutura intermediária, o segundo passo da abordagem (*Interpretador*) produz a consulta correspondente em Cypher. Ele aplica transformações em um objeto da classe *Query*. A função `generate` recebe o código SPARQL e faz as chamadas para construção dos blocos de código em Cypher (listagem 3). O método `self.parse_query` constroi a estrutura intermediária e seu retorno é armazenado em `query`. Os métodos `self.setup_aliases` e `setup_namespaces` associam valores ao próprio objeto de modo facilitar a construção de outras funções.

[7]https://www.dabeaz.com/ply/

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

```
1     def generate(self, sparql_query: str) -> str:
2         query = self.parse_query(sparql_query)
3         self.setup_aliases(query.returning)
4         self.setup_namespaces(query.namespaces)
5         patterns = self.split_pattern(query.graph_pattern)
6         code_blocks = [
7             self.code_block_for_pattern(p, query)
8             for p in patterns
9             if len(p.and_triples) > 0
10         ]
11         united_code = "\nUNION\n".join(code_blocks)
12         modifiers = self.result_modifier(query.modifiers)
13         having_part = self.having_clause(query.modifiers)
14        if modifiers or having_part:
15             ret_clause = "RETURN *"
16             modified_code = (
17               "CALL {\n" + united_code + "\n}"
18               + ("\n" + having_part if having_part else "") + "\n"
19               + ret_clause + ("\n" + modifiers if modifiers else "")
20             )
21         else:
22             modified_code = united_code
23         return modified_code
```

Listing 3: Função para geração da consulta Cypher

O método `self.split_pattern` é responsável pela transformação do campo `graph_pattern` da estrutura intermediária devido a sua característica recursiva. Isso ocorre por que o campo `or_blocks` (ver listagem 2) pode conter uma lista de outras instâncias de `GraphPattern`, cada uma com sua própria lista de triplas no campo `and_triples`. Esta transformação basicamente aplica a propriedade distributiva $P1 \wedge (P2 \vee P3) = (P1 \wedge P2) \vee (P1 \wedge P3)$ para condições complexas ou blocos de filtros conectados por `UNION` em consultas SPARQL. A estrutura formada por uma disjunção entre conjunções é mais simples de ser representada em Cypher.

A consulta Cypher a seguir é o resultado da conversão realizada pela *Switch* para a consulta SPARQL apresentada na Seção 2.1:

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

```
15 RETURN country, count(state) AS stateCount}
16 RETURN * LIMIT 10
```

Para a transformação das triplas presentes em uma consulta SPARQL, é necessário encontrar nodos e arestas correspondentes em um grafo no Neo4j. Para tanto, deve-se primeiro avaliar o *sujeito* da tripla. Se ele for uma variável, verifica-se se esta variável já foi usada na consulta resultante. Se não tiver sido usada, adiciona-se uma cláusula `MATCH` para o nome desta variável (linha 2). Do contrário, esse passo não é necessário. Se o *sujeito* for uma URI, adiciona-se 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. Um caso específico ocorre quando o *predicado* for um `rdf:type`. Nesse caso, a cláusula `WHERE` deve filtrar se a URI especificada no *objeto* está inclusa na lista de *labels* do nodo (linhas 3 a 5).

Após a avaliação do *sujeito*, é necessário avaliar *predicado* e *objeto*. Para essa etapa, leva-se em conta a estratégia adotada pelo trabalho de [Lombardot et al. 2019]. 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 2 nodos nos demais casos. O caso que envolve a propriedade `rdf:type` foi explicado no parágrafo anterior. Para os demais casos, é necessário criar blocos de código (linhas 6 a 10 da listagem 3), conforme explicado a seguir. Caso vários blocos sejam gerados, eles são posteriormente unificados através de uma cláusula `UNION` do Cypher (linha 11 da listagem 3).

Supondo a tripla `(?s ?p ?o)` em uma consulta SPARQL, a sua representação em Cypher (desconsiderando a cláusula `MATCH`) gera uma cláusula `UNWIND` seguida de 2 construções de lista renomeadas como `triples` (linhas 6 a 13). A primeira lista trata o caso em que o *objeto* é valor de uma propriedade `predicado`. Sendo assim, para cada propriedade `key` na lista de propriedades (retornada pela chamada para a função `keys(`*`nodo`*`)`), cria-se uma sublista de 3 elementos composta pelo nodo `s`, a propriedade `key` e o valor da propriedade para aquele nodo `s[key]` (linhas 6 a 9). A segunda lista (linhas 10 a 13) cobre o caso do *objeto* ser um outro nodo ao qual o *sujeito* se relaciona através de uma relação qualquer. Nesse caso, cria-se uma consulta interna com o formato `(s)-[_relation]-(o)` onde a variável `s` já foi definida anteriormente e as demais são definidas na consulta SPARQL. Para cada resultado dessa consulta interna, ou seja, todas as relações em qualquer direção ligando a variável `s` com outro nodo, forma-se uma lista de 3 elementos contendo o nodo `s`, a relação `_relation` e outro nodo `o`. As 2 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. Já a renomeação como `triples` permite aplicar novas operações sobre essa lista de resultados. Ainda, uma cláusula `RETURN` é adicionada a cada bloco de código gerado (linha 16), que são unidas por uma cláusula `UNION` quando mais de um bloco é definido.

As linhas 3-4, 7-8 e 11-12 exemplificam o mapeamento de namespaces em uma consulta SPARQL através do uso da função `n10s.rdf.shortFormFromFullUri` da Neosemantics. Ela recebe o endereço do *namespace* e retorna como *string* o prefixo associado ao endereço na nomenclatura das propriedades e *labels* dos dados.

As linhas 12 a 21 da listagem 3 tratam modificadores de resultado que podem estar presentes em uma consulta SPARQL. Para tanto, utiliza-se a cláusula `CALL` da Cypher

para executar blocos de consulta já gerados e aplicar modificações de resultado sobre ela. Os modificadores `ORDER BY`, `LIMIT` e `OFFSET` possuem cláusulas semelhantes em Cypher, sendo a tradução direta. Já `GROUP BY` e `HAVING` não têm equivalência em Cypher, porém pode-se replicar seu comportamento utilizando as cláusulas `RETURN`, `WITH` e `WHERE`. A linha 15 da consulta Cypher implementa o equivalente ao uso de um `GROUP BY` com funções de agregação. Já os filtros da cláusula `HAVING` são tratados com uma cláusula `WHERE` associada ao último `RETURN` da consulta Cypher.

## 5. Avaliação

Para avaliar a *Switch*, 12 consultas em SPARQL foram criadas para serem executadas no Neo4j. O *dataset* RDF utilizado para os testes é uma relação de países soberanos e territórios[9]. Um *script* que faz o *download* do *dataset* e o persiste no Neo4j utilizando as regras de mapeamento do Neosemantics foi implementado. Todas as consultas podem ser encontradas no repositório do projeto[10]. Consultas foram definidas com diferentes níveis de complexidade e com o uso de diferentes recursos da SPARQL, como diferentes padrões de triplas, expressões lógicas e modificadores de resultado.

A Tabela 3 apresenta os resultados dos testes. Cada teste mostra os tempos gastos pela execução direta das consultas *SPARQL* sobre o *dataset* RDF já carregado em memória principal, pela tradução realizada pela *Switch* e pela execução no *Neo4j*. Os tempos são uma média de 10 execuções de cada consulta. Para a execução direta foi utilizada a biblioteca `rdflib` do Python[11], que carrega dados RDF em memória e executa consultas SPARQL sobre eles. Foi verificado também se o resultado da consulta executada pela *Switch* é igual ao resultado da execução direta da consulta SPARQL.

**Table 3. Testes de avaliação da Switch (tempos em segundos)**

| **Teste** | ***SPARQL*** | ***Switch*** | ***Neo4j*** | **Resultados iguais** |
|---|---|---|---|---|
| test_query_1.sparql | 0.002 | 0.0022 | 0.009 | Sim |
| test_query_2.sparql | 0.004 | 0.0018 | 0.020 | Sim |
| test_query_3.sparql | 0.003 | 0.0020 | 0.011 | Sim |
| test_query_4.sparql | 0.003 | 0.0023 | 0.012 | Sim |
| test_query_5.sparql | 0.079 | 0.0017 | 0.027 | Sim |
| test_query_6.sparql | 0.042 | 0.0025 | 0.246 | Sim |
| test_query_7.sparql | 0.003 | 0.0017 | 0.167 | Sim |
| test_query_8.sparql | 0.058 | 0.0017 | 2.808 | Sim |
| test_query_9.sparql | 0.057 | 0.0019 | 2.683 | Sim |
| test_query_10.sparql | 0.001 | 0.0016 | 3.195 | Não |
| test_query_11.sparql | 0.095 | 0.0017 | 2.992 | Sim |
| test_query_12.sparql | 0.087 | 0.0019 | 3.048 | Sim |

O resultado da grande maioria das consultas foi o mesmo. Apenas uma consulta apresentou resultado diferente, pois ela invoca o método `NOW()`, que retorna o instante de tempo da sua execução. Mesmo assim, a tradução realizada pela *Switch* foi correta.

[9]https://ontologi.es/place/

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

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

Em termos de tempo de processamento, o principal *overhead* na comparação entre a execução direta e a execução via *Switch* é o processamento da consulta pelo *Neo4j*, como esperado, pois os dados estão persistidos em memória secundária. Entretanto, o *overhead* com a tradução realizada pela *Switch* não foi considerado proibitivo, pois na grande maioria dos casos o tempo gasto pela *Switch* foi inferior ao tempo *SPARQL*, sendo, em média, 19 vezes mais rápida. Alguns *outliers* nesses resultados devem ainda ser analisados.

## 6. Conclusão

A construção de *triplestores* é um tópico de pesquisa na comunidade de BD. Esse trabalho contribui com esta problemática através da *Switch*: uma abordagem de tradução de consultas SPARQL para consultas semanticamente equivalentes sobre o SGBD Neo4j. Poucos trabalhos lidam com esta tradução, porém eles ou não tratam consultas SPARQL complexas ou seus mecanismos de tradução são custosos. *Switch*, por outro lado, traduz consultas SPARQL com diferentes graus de complexidade e baixo *overhead*.

Trabalhos futuros incluem o suporte às cláusulas `FILTER`, `MINUS` e `EXISTS`, bem como funções nativas da SPARQL [Silva 2022]. Uma prova formal também pode ser desenvolvida para avaliar a corretude semântica das consultas transformadas, assim como uma API para integrar a *Switch* a outros SGBDs.

## References


Barrasa, J. (2023). Importing RDF Data into Neo4j. `https://jbarrasa.com/2016/06/07/importing-rdf-data-into-neo4j/`. Access: 2023-02-15.

Berners-Lee, T. et al. (1992). The World-Wide Web. *Comput. Networks ISDN Syst.*, 25(4-5):454–459.

Berners-lee, T. et al. (2001). The Semantic Web. *The Scientific America*, page 28–37.

Fathy, N. et al. (2020). Querying Heterogeneous Property Graph Data Sources Based on a Unified Conceptual View. In *IX International Conference on Software and Information Engineering (ICSIE)*, page 113–118. ACM.

Kitchenham, B. (2004). Procedures for Performing Systematic Reviews. Keele university. technical report tr/se-0401, Department of Computer Science, Keele University, UK.

Lombardot, T. et al. (2019). Updates in Rhea: SPARQLing Biochemical Reaction Data. *Nucleic Acids Res.*, 47(Database-Issue):D596–D600.

Santana, L. H. Z. and Mello, R. (2017). Workload-Aware RDF Partitioning and SPARQL Query Caching for Massive RDF Graphs Stored in NoSQL Databases. In *XXXII Simpósio Brasileiro de Banco de Dados (SBBD)*, pages 184–195.

Silva, T. S. d. (2022). Switch: Executando SPARQL sobre Neo4j. Bachelor's thesis, Depto. de Informática e Estatística, Universidade Federal de Santa Catarina, Brasil.

Thakkar, H. et al. (2018). Two for One: Querying Property Graph Databases Using SPARQL via Gremlinator. In *SIGMOD Joint Int. Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM.