UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

MATHEUS TEIXEIRA PEREIRA

UMA FERRAMENTA PARA CONTROLE DE MAPEAMENTO DE DADOS RDF PARA BANCOS DE DADOS NOSQL GRAFO E CHAVE-VALOR

View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Institucional da UFSC

FLORIANÓPOLIS, 2016

Matheus Teixeira Pereira

Uma Ferramenta de Controle de Mapeamento de Dados RDF Para Bancos de Dados NoSQL Grafo e Chave-Valor

Trabalho Conclusão do Curso de Graduação em Ciências da Computação do Centro de Informática e Estatística da Universidade Federal de Santa Catarina como requisito para a obtenção do Título

de Bacharel em Ciências da Computação

Orientador: Prof. Dr. Ronaldo dos Santos Mello

Coorientador: Luiz Henrique Zambom Santana

Florianópolis

2017

Matheus Teixeira Pereira

UMA FERRAMENTA PARA CONTROLE DE MAPEAMENTO DE DADOS RDF PARA BANCOS DE DADOS NOSQL GRAFO E CHAVE-VALOR

Banca Examinadora:

____

Prof., Dr. Ronaldo dos Santos Mello

Orientador

Universidade Federal de Santa Catarina

____

Luiz Henrique Zambom Santana

Corientador

Universidade Federal de Santa Catarina

____

Prof., Dr. Roberto Willrich

Universidade Federal de Santa Catarina

____

Prof. Dr. Frank Augusto Siqueira

Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço primeiramente a minha companheira, Camilla Seifert, que com seu apoio incondicional e presença constante, me ajudou em todos os passos dessa jornada. Também a meus pais pelo apoio durante todo meu percurso e a todos que direta ou indiretamente me auxiliaram.

RESUMO

Este trabalho propõe um algoritmo e ferramenta para mapear triplas RDF para bancos de dados NoSQL baseados em Chave-Valor e Grafo, assim como a implementação de um módulo que realiza a tradução de consultas específicas de RDF para as linguagens de consulta utilizadas por estes bancos de dados. Atualmente existe um grande crescimento na quantidade de dados RDF sendo gerados na Web e poucos dos trabalhos relacionados propõem maneiras de realizar o mapeamento desses dados para bancos de dados NoSQL. Assim sendo, este trabalho visa desenvolver uma ferramenta que possa demonstrar as maneiras que se é possível trabalhar com NoSQL e RDF, especificamente em bancos de dados baseados em Chave-Valor e Grafo.

Palavras-chave: Banco de Dados. NoSQL. RDF.

ABSTRACT

The following academic work proposes both a algorithm and a tool to map RDF triples to Graph and Key-Value NoSQL's databases, and also the implementation of a module that translates RDF specific queries to this databases query language. Currently, the amount of RDF data has been growing across the Web and few existing projects proposes a way to map this data to NoSQL databases. So, this work aims at developing a tool that can show the way that it is possible to work with NoSQL and RDF, specifically in Graph and Key-Value based databases.

Keywords: Databases. RDF. NoSQL.

LISTA DE FIGURAS

Figura 1 - Formato chave-valor ………………………………………………………….. 15

Figura 2 - Exemplo de Inserção de Documento no MongoDB ……………………… 16

Figura 3 - Exemplo de Banco de Dados Colunar …………………………………….. 17

Figura 4 - Exemplo de Grafo ……………………………………………………………. 18

Figura 5 - Representação de uma Tripla RDF ……………………………………... 19

Figura 6 - Grafo RDF G_1 ………………………………………………………………. 20

Figura 7 - Exemplo de Consulta SPARQL ………………………………………..... 22

Figura 8 - Visão Geral da Arquitetura de Sistema do Rainbow ……………………... 25

Figura 9 - Visão Geral da Arquitetura de Sistema do ScalaRDF …………………… 27

Figura 10 - Comparação do Tempo de Performance de Consultas do ScalaRDF e Rainbow para os datasets LUBM-10 e LUBM-40 ……………………………………... 28

Figura 11 - Arquitetura da ferramenta RDF2Multimodel ……………………………… 30

Figura 12 - Visão de Alto Nível dos Módulos do RDF2Multimodel ………………….. 31

Figura 13 - Grafo utilizado de Exemplo ………………………………………………… 33

Figura 14 - Exemplo de Dados Inseridos com as duas Estratégias no Redis ……... 33

Figura 15 - Exemplo da utilização do UNWIND …………………………………….. 34

Figura 16 - Exemplo de Dado Inseridos no Neo4j …………………………………….. 34

Figura 17 - Estrutura de uma Consulta SPARQL ……………………………………… 36

Figura 18 - Visualização dos Dados Inseridos no Teste de Validação da Ferramenta no Neo4j e Redis………………………………………………………………………………. 45

Figura 19 - Gráfico de Tempo de Inserção em milissegundos ………………………   46

Figura 20 - Gráfico de Tempo de Inserção em milissegundos separado por BD ….   47

LISTA DE TABELAS

Tabela 1 - Conjunto de Dados de Validação ………………………………………….. 44

Tabela 2 - Tabela de Resultados do Experimento de Validação …………………… 45

Tabela 3 - Tempo Médio de Respostas de Consultas SPARQL …..……………….. 48

LISTA DE ABREVIATURAS E SIGLAS

RDF - Resource Description Framework

JSON - JavaScript Object Notation

BD - Banco de Dados

SUMÁRIO

SUMÁRIO 12

1. Introdução 12

1.1. Justificativa 14

1.2. Objetivos 15

1.2.1. Objetivo Geral 15

1.2.2 Objetivos Específicos 15

1.3 Organização do Documento 15

2. Fundamentação Teórica 16

2.1. NoSQL 17

2.2. RDF e SPARQL 24

3. Trabalhos Relacionados 28

3.1. Rainbow 28

3.2. ScalaRDF 30

4. Ferramenta Proposta: RDF2Multimodel 33

4.1. Rails API 34

4.2. Arquitetura do RDF2Multimodel 34

4.3. Inserindo Triplas RDF no RDF2Multimodel 36

4.4. Execução de consultas utilizando SPARQL no RDF2Multimodel 41

5. Avaliação da Ferramenta RDF2Multimodel 44

5.1. Testes de Validação da Ferramenta 44

5.2. Experimento de Inserção com dataset gerado aleatoriamente 46

5.3. Experimento de consultas com dataset gerado aleatoriamente 48

6. Conclusão 49

Referências Bibliográficas 51

Apêndice A - Artigo 56

Apêndice B - Código Fonte 66

1. Introdução

O foco deste trabalho está voltado aos conceitos de *Resource Description Framework*, (RDF 1.1 CONCEPTS, 2014) - o principal modelo de troca de dados para Web Semântica - e bancos de dados NoSQL (FOWLER e SADALAGE, 2012), especificamente os modelos de dados de grafos e chave-valor. RDF é o principal *framework* para representação de dados da Web, sendo recomendado e mantido pela *W3C*[1] (*World Wide Web Consortium*). Seu modelo de dados é baseado em grafos e sua estrutura segue um modelo de triplas, representadas por um sujeito, um predicado e um objeto. O conjunto destas triplas é chamado de grafo RDF (RDF GRAPHS, 2014).

Já NoSQL é um termo utilizado para descrever bancos de dados não relacionais, geralmente de alto desempenho, sendo reconhecidos também por sua escalabilidade, flexibilidade de esquema e alta disponibilidade. Os bancos de dados NoSQL podem utilizar os mais diversos modelos de persistência de dados como: documento, grafo, coluna e chave-valor. Recentemente, novos bancos de dados NoSQL possuem múltiplos modelos, sendo conhecidos como *multimodelos*. Com a popularização dos modelos NoSQL, o conceito de "Persistência Poliglota" cresceu no mercado, levando organizações a adotarem diversos bancos de dados diferentes para persistir os dados que geram e coletam. Os bancos de dados multimodelo surgem como uma alternativa para simplificar esses ambientes (ROUSE, 2017).

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

Este trabalho apresenta um algoritmo e uma ferramenta para realizar a persistência e a consulta de/a dados RDF em bancos de dados baseados nos modelos de dados de grafo e chave-valor. Esta ferramenta facilita trabalhos acadêmicos futuros que necessitem de flexibilidade e facilidade para converter dados RDF para esses modelos de bancos de dados NoSQL.

1.1. Justificativa

Tanto a qualidade quanto a quantidade dos dados RDF disponíveis na Web tem tido um grande crescimento nos últimos anos, com o RDF sendo adotado em diversos campos como: Redes Sociais (FOAF, OpenGraph), bases de Conhecimento (DBPedia, YAGO, Freebase) e Bioinformática (Bio2RDF, Uniprot), etc (GAI, CHEN e WANG, 2015).

Com o aumento dos dados disponíveis se torna necessário a utilização de diversos métodos de persistência tanto para escalabilidade e portabilidade desses dados, como para que possa ser possível aumentar a performance de consultas feitas sobre os mesmos.

As soluções existentes para persistência de RDF exploram pouco a utilização de bancos de dados NoSQL, como explorado na Seção 3. Desta forma, esse trabalho propõe uma ferramenta para o mapeamento de dados RDF para, especificamente, os modelos de BDs baseados em Grafo e Chave-valor. O mapeamento proposto, assim como a tradução de consultas SPARQL para as linguagens de consultas dos bancos propostos pode ser utilizado tanto para expandir projetos existentes, quanto para trabalhos futuros que queiram trabalhar com RDF e os formatos de NoSQL utilizados.

Persistir RDF como dados NoSQL permite utilizar-se das diversas características de NoSQL citadas na seção 2.1.

1.2. Objetivos

1.2.1. Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma ferramenta de mapeamento de dados RDF para os bancos de dados NoSQL, dos modelos de persistência em grafo e chave-valor.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

a. Desenvolver uma estratégia de mapeamento de dados RDF para bancos de dados baseados em grafos e chave-valor;

b. Desenvolver um analisador de consultas SPARQL para bancos de dados de baseados em grafos e chave-valor;

c. Validar o funcionamento da ferramenta.

1.3 Organização do Documento

A sequência deste documento está organizada da seguinte forma: a Seção 1.1 apresenta os objetivos gerais e específicos do trabalho; a Seção 2 discute a fundamentação teórica necessária para compreensão da proposta; a Seção 3, discute trabalhos relacionados; finalmente, a Seção 4 detalha o desenvolvimento deste trabalho de conclusão de curso.

2. Fundamentação Teórica

Esse trabalho está apoiado sobre dois pilares principais: os bancos de dados NoSQL e a Web Semântica.

Os bancos de dados NoSQL podem ser considerados parte de um movimento que inclui também conceitos como Computação em Nuvem e *Big Data*. A Computação em Nuvem refere-se a noção de utilizar aplicações que estão rodando em outros computadores, acessando-as de quaisquer dispositivos e em qualquer lugar, com a mesma facilidade de instalação. Sua principal vantagem é justamente poder ter acesso a aplicações a partir da *Internet*, sem a necessidade de que ela esteja instalada no dispositivo. Outras vantagens são: independência de sistema operacional, facilidade de trabalhos colaborativos, alta disponibilidade e a maior parte do processamento sendo feita remotamente e com boa escalabilidade (ALECRIM, 2008). A Big Data representa o crescimento exponencial na quantidade de dados armazenados e processados pelos sistemas de computação atuais (ARTHUR, 2013). No centro desse movimento estão os bancos de dados NoSQL, já que os tradicionais bancos de dados SQL - bancos de dados relacionais que geralmente realizam processamento de muitos *joins* entre tabelas grandes - dificulta o pleno uso da capacidade computacional oferecida pela Computação em Nuvem e impossibilita a manipulação eficiente de grandes quantidades de dados (Document Databases, 2016).

A Web Semântica trata das conexões que são necessárias (*links*) para que uma pessoa ou máquina possa explorar os dados disponíveis na Web a partir de um tema relacionado (BERNERS-LEE, 2006). Assim como em hipertexto, a Web de dados é construída com dados interligados. Tim Berners-Lee (BERNERS-LEE, 2006), propõe quatro regras para a formatação de documentos no que diz respeito ao potencial de crescimento da web:

- Usar URIs para dar nome às coisas;
- Usar HTTP URIs para que as pessoas possam procurar estes nomes;

- Quando alguém observar uma URI, prover informações úteis usando os padrões (RDF, SPARQL); e
- Incluir links para outras URIs, para que elas possam descobrir outras coisas.

A maneira mais simples de criar um link (BERNERS-LEE, 2006) é criar em um arquivo uma URI que aponta para outro. Por exemplo, em um arquivo RDF (mais detalhes sobre o padrão RDF podem ser encontrados na Seção 2.2) <http://exemplo.org/smith>, você pode utilizar identificadores dentro do arquivo que digam #albert, #brian e #carol, conforme descrito a seguir:

```
<rdf:Description about="#albert"
<fam:child rdf:Resource="#brian">
<fam:child rdf:Resource="#carol">
</rdf:Description>
```

Desta forma, quando a URI http://example.org/smith#carol for encontrada, é possível facilmente perceber o documento a qual ela pertence, apenas truncando o resultado antes da cerquilha e quem acessar esse documento vai receber informações sobre #carol.

Na sequência, a Seção 2.1. detalha os bancos de dados NoSQL e seus diversos modelos e a Seção 2.2. apresenta o RDF e SPARQL que representam implementações para a visão da Web Semântica.

2.1. NoSQL

O primeiro uso da palavra “NoSQL” aconteceu no final dos anos 90 como o nome de um banco de dados relacional, chamado de Strozzi NoSQL. (FOWLER e SADALAGE, 2012) O nome foi proposto pelo fato do banco de dados não utilizar SQL como uma

linguagem de consulta, mas a sua influência no que consideramos NoSQL hoje vai pouco além de seu nome.

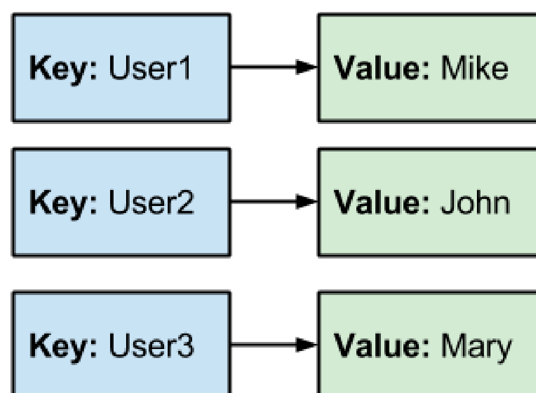
A origem do termo é traçada a 11 de Junho de 2009, quando um encontro entre desenvolvedores foi feito em São Francisco nos Estados Unidos, organizado por Johan Oskarsson. O convite para esse primeiro encontro pedia que fossem bancos de dados “open-source, distribuídos e não relacionais”. O nome NoSQL foi sugerido pela comunidade de desenvolvedores Cassandra - um importante banco de dados NoSQL - da época. Apesar do termo ter grande aderência tanto pelo mercado quanto pela da comunidade acadêmica, sua definição ainda não é consensual.

Segundo Fowler e Sadalage (FOWLER e SADALAGE, 2012), já que não existe uma definição consolidada para o termo “NoSQL”, é mais fácil discutir suas características gerais: bancos de dados NoSQL não utilizam o modelo relacional, muitos deles não utilizam a linguagem SQL como interface de acesso, a grande maioria são projetos de código aberto, muitos são dirigidos pela necessidade de execução em *clusters* de computadores, permitem operar bancos de dados sem a definição de esquemas permitindo a adição livre de campos nos registros, sem precisar fazer modificações na estrutura. Isso se tornou bastante útil ao lidar com dados não uniformes e campos personalizados, que eram um empecilho em bancos de dados relacionais, que exigiam a criação de campos extras que aumentavam a dificuldade de processamento de dados. Todas as características citadas acima ajudam a descrever o modelo de dados NoSQL. Assim, apesar de não existir uma definição fortemente aceita, esse conjunto de características é considerado para definir os bancos de dados NoSQL ao longo deste trabalho.

Segundo (MCMURTRY et al., 2013; VIEIRA et al., 2012), os bancos de dados NoSQL atualmente podem ser categorizados, de acordo com a maneira que fazem a persistência de seus dados, em quatro principais modelos de dados: chave-valor, documento, colunar e baseado em grafos.

Um banco de dados **chave-valor** é um banco de dados NoSQL otimizado para aplicações que necessitam muito de leituras ou de computações intensivas. Esses bancos de dados são geralmente usados para a criação de estruturas de *cache* em memória ou disco, para melhorar o desempenho de um ponto específico de uma aplicação. Essas chaves são então guardadas em memória com baixa latência de acesso e estão geralmente associadas a resultados de consultas frequentes (IN-MEMORY KEY-VALUE, 2016). Segundo (TIWARI, 2011), um banco de dados chave-valor pode ser representado por um *Hash Map*, uma das estruturas de dados mais simples, como é possível ver na Figura 1. Essas estruturas de dados são populares, pois permitem algoritmos de acesso muito eficientes com complexidades relativas a *big O(1).* Alguns exemplos de bancos de dados chave-valor são Redis, Voldemort , Riak e Amazon DynamoDB.

Figura 1 - Formato chave-valor



Fonte: http://www.kdnuggets.com/2016/06/top-nosql-database-engines.html

O Redis é um banco de dados chave-valor *open source*. Sua estrutura de armazenamento é feita em memória e é geralmente usado como uma ferramenta de *cache*. Ele suporta dados como: strings, hashes, listas, conjuntos, conjuntos ordenados, bitmaps, *hyperloglogs* e índices geoespaciais.

Redis foi escolhido neste trabalho para representar os BDs baseados em chave-valor por ser amplamente utilizado pela comunidade de desenvolvimento e a indústria como um todo[2]. Alguns exemplos de aplicações que utilizam o Redis são: Twitter, GitHub, StackOverflow, Snapchat e Pinterest. (WHO USES, 2017)

Bancos de dados baseados em **documentos** possuem dados pouco estruturados, consistindo majoritariamente de pares de conjuntos chave/valor com formatos semelhantes ao *JSON*[3] *(JavaScript Object Notation)*. Um exemplo de um dado nesse formato é mostrado na Figura 2 para um documento no banco de dados MongoDB. Bancos de dados documento tratam os documentos como um todo e evitam separar este em vários pares de nome/valor (TIWARI, 2011). Na Figura 2 é possível ver que temos dois documentos sendo inseridos: o primeiro deles possui titulo, descrição e alguns outros atributos. Uma das grandes vantagens da utilização do JSON é seu formato de fácil entendimento, estrutura simples e leve que é independente da linguagem utilizada.

Essa categoria de banco de dados geralmente possui mecanismos de consulta poderosos e ferramentas de indexação. Os dois exemplos de bancos de dados documento mais utilizados são o MongoDB e o CouchDB (DB-Engines, 2017).

Bancos de dados colunares priorizam o armazenamento de dados no sentido de evitar o desperdício de espaço de armazenamento simplesmente por não armazenarem uma coluna quando o valor dela não existe (TIWARI, 2011). Esses bancos podem diminuir drasticamente o tempo de busca em consultas analíticas, pois reduzem o tempo de escrita e leitura no disco e a quantidade total e dados que são

[2]https://db-engines.com/en/system/Redis
[3] https://www.w3schools.com/js/js_json_intro.asp

carregados do disco (DB-Engines, 2017). Na Figura 3 é possível observar um exemplo de como um dado é armazenado em um banco de dados colunar. No exemplo, a coluna preço é armazenada como uma coluna sequencial. Logo, essa coluna pode inteira ser percorrida e consultas como SUM, AVG e COUNT podem ser realizadas de forma muito mais eficiente. Exemplos populares de bancos de dados colunares são o Apache Cassandra, Apache HBase e Amazon RedShift (DB-Engines, 2017).

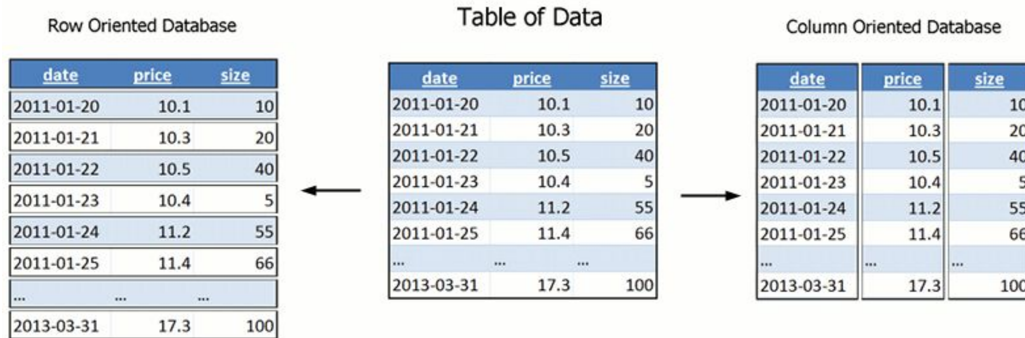
Figura 2 - Exemplo de Inserção de Documento no MongoDB

```
>db.post.insert([
   {
      title: 'MongoDB Overview',
      description: 'MongoDB is no sql database',
      by: 'tutorials point',
      url: 'http://www.tutorialspoint.com',
      tags: ['mongodb', 'database', 'NoSQL'],
      likes: 100
   },

   {
      title: 'NoSQL Database',
      description: 'NoSQL database doesn't have tables',
      by: 'tutorials point',
      url: 'http://www.tutorialspoint.com',
      tags: ['mongodb', 'database', 'NoSQL'],
      likes: 20,
      comments: [
         {
            user:'user1',
            message: 'My first comment',
            dateCreated: new Date(2013,11,10,2,35),
            like: 0
         }
      ]
   }
])
```

Fonte: https://www.tutorialspoint.com/mongodb/mongodb_insert_document.htm

Figura 3 - Exemplo de Banco de Dados Colunar



Row Oriented Database

date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

Table of Data

date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

Column Oriented Database

date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

Fonte: http://www.timestored.com/time-series-data/what-is-a-column-oriented-database

Os bancos de dados baseados em **grafos** baseiam-se na definição formal de um grafo, ou seja, uma coleção de vértices, ou nó, que é a unidade fundamental de um grafo e as arestas que os ligam (ROBINSON, WEBBER, EIFREM, 2015). As entidades são representadas como vértices, como mostrado na Figura 4, e entre tais entidades são criadas arestas para representar relacionamentos. Ainda na Figura 4 é possível notar que os vértices e arestas possuem cores distintas. Isso ocorre devido ao tipo de informação que está sendo representado, já que em um BD baseado em grafo é possível que cada vértice ou aresta possua um tipo diferente, composto por diferentes atributos.

Figura 4 - Exemplo de Dado de um BD baseado em Grafo

Fonte: https://neo4j.com/blog/node-js-react-js-developers-neo4j-movies-template/

Bancos de dados baseados em grafos são geralmente construídos para o uso em aplicações que utilizem dados que possuem muitos relacionamentos entre si, como um gerenciador de catálogo de Filmes ou uma rede social, por exemplo. Dentre os bancos de dados baseados em grafos mais populares se encontram o TitanDB e o Neo4j.

O Neo4J é o mais popular banco de dados NoSQL baseado em grafos[4]. Sua estrutura foi criada para permitir o gerenciamento rápido de armazenamento e travessia de vértices e relacionamentos entre grafos. A maneira que o Neo4J realiza suas consultas e armazenamento permite que a travessia de entidades tenha um desempenho de até 4 milhões de saltos em uma única thread.

Independentemente do modelo de armazenamento, o uso de dados NoSQL traz consigo diversas vantagens em relação aos bancos de dados relacionais, dentre elas: os dados são tratados como unidades independentes, o que torna o desempenho

[4] https://db-engines.com/en/system/Neo4j

melhor e facilita a distribuição de dados por vários servidores; a lógica da aplicação se torna mais fácil de ser escrita, já que não é necessário traduzir entre objetos da aplicação e consultas SQL, sendo possível mapear os modelos diretamente para um documento, chave-valor, coluna ou grafo; dados não estruturados podem ser armazenados facilmente, os esquemas podem evoluir automaticamente; migrações com altos custos de degradação e riscos ao sistema são evitadas, já que não é preciso conhecer o esquema.

2.2. RDF e SPARQL

RDF é a sigla para *Resource Description Framework,* que é o principal padrão para a representação de dados na Web. Atualmente, o RDF é usado em uma grande variedade de áreas do conhecimento, como: ciência, bioinformática, redes sociais, grafos de conhecimento e automatizadores de perguntas e respostas (HU, WANG, YANG e WO).

Atualmente, a comunidade de pesquisa e da indústria usa o modelo RDF para manipular a crescente quantidade de dados pouco estruturados para serem mantidos em bancos de dados, mas que contém estruturas regulares o suficiente para serem exploradas na formulação e execução de consultas (FAYE, CURE e BLIN, 2012).

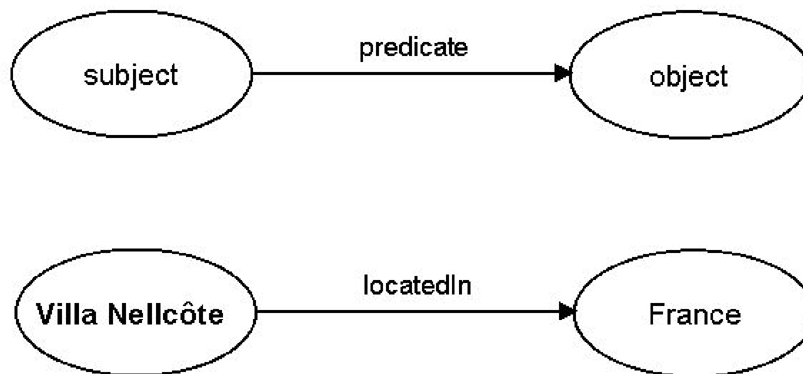
Em sua concepção, o RDF apresenta os seguintes objetivos (RDF 1.1 CONCEPTS, 2014):

- Ter um modelo de dados simples;
- Ter uma semântica formal;
- Usar um vocabulário extensível baseado em URIs;
- Usar uma sintaxe baseada em XML; e
- Suportar o uso de esquemas de XML.

A estrutura básica de uma expressão RDF consiste em uma coleção de triplas. Cada uma dessas triplas contém um *sujeito*, um *predicado* e um *objeto*, e pode ser

representada como um grafo orientado, sendo que a direção do arco dá significado à relação, apontando sempre do sujeito para o objeto. A aresta entre um nó sujeito e um nó objeto é o predicado (RDF 1.1 CONCEPTS, 2014). A Figura 5 apresenta um exemplo de uma tripla e também um exemplo prático, em que é feito o mapeamento da cidade de *Villa Nellcôte* com o predicado mostrando que ela está localizada na França.

Figura 5 - Representação de uma Tripla RDF



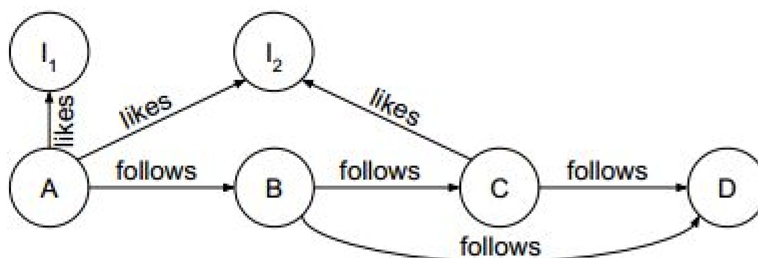
Fonte: https://objectcomputing.com/resources/publications/sett/february-2011-the-semantic-web/

Dessa forma, qualquer fato do mundo real pode ser representado por uma tripla RDF de forma que o predicado nomeia o relacionamento entre duas coisas, que são o sujeito e o objeto. As triplas podem ser representadas em um banco de dados relacional através de uma tabela possuindo duas colunas, uma correspondendo ao sujeito e a outra ao objeto da tripla RDF, sendo o nome da tabela o predicado (RDF 1.1 CONCEPTS, 2014).

Formalmente, uma tripla RDF é definida como t = (s, p, o), onde *s* é o sujeito, *p* o predicado e o objeto. Com essa tripla pode-se inferir que "*s* tem a propriedade *p* com o valor *o*", que pode ser interpretada como uma aresta de s para o chamada p, tal que "$s \rightarrow^{p} o$". Assim, um grafo G pode ser representado pelo conjunto de triplas, como por exemplo, G = {(A, *segue*, B), (B, *segue*, C), (B, *segue*, D), (C, *segue*, D), (A, *curte*, I_1),

(A, *curte*, I_2), (C, *curte*, I_2)}. A Figura 6 apresenta uma representação visual do grafo G (SCHÄTZLE et al).

Figura 6 - Grafo RDF G_1



Fonte: http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf

O poder do RDF advém da flexibilidade em se representar estruturas arbitrárias sem a necessidade de definir esquemas *a priori*. Por isso, o RDF oferece a habilidade de especificar conceitos e assim ligá-los em um grafo de dados. Uma outra vantagem do RDF como linguagem de armazenamento é a possibilidade de ligar fontes de dados diferentes adicionando apenas poucas triplas adicionais especificando as relações entre os conceitos. Além disso, não existe restrição quanto ao tamanho do grafo, ao contrário de campos de SGBDs onde o esquema deve ser conciso (FAYE, CURE e BLIN, 2012).

Para consulta a grafos RDF foi proposta a linguagem de consultas SPARQL. SPARQL é um acrônimo recursivo para SPARQL Protocol And RDF Query Language (SPARQL, 2008). É a linguagem de consulta da Web Semântica, permitindo que sejam consultados valores de dados semiestruturados, exploradas relações desconhecidas, realizadas junções complexas entre grafos em uma única consulta e transformar dados RDF de um vocabulário para outro (FEIGENBAUM, 2017).

Uma expressão de consulta SPARQL define padrões de triplas a serem descobertas e recuperadas de um grafo RDF. A Figura 7 mostra um exemplo de uma

consulta SPARQL que define 3 padrões de tuplas e seu objetivo é retornar os estudantes que gostam de nadar e cujo professor é o Professor1.

Figura 7 - Exemplo de Consulta SPARQL

```
select ?x where {
?x type Student .
?x like Swimming .
Professor1 teach ?x .}
```

Fonte: (RAINBOW, 2016)

Mais especificamente, uma consulta SPARQL consiste de: declaração de prefixos para abreviar URIs; definição do *dataset*, começando com o grafo RDF a ser consultado; cláusula de resultado, que identifica que informação retornar para a consulta; um padrão de consulta, que especifica o que consultar no *dataset*; e modificadores de consulta, que podem ser cláusulas de ordenação, *slicing* e outras maneiras de rearranjar os resultados da consulta (FEIGENBAUM, 2017). Desta forma, uma consulta SPARQL possui a seguinte estrutura:

```
# declaração de prefixos
PREFIX foo: <http://example.com/resources/>
...
# definição do dataset
FROM ...
# cláusula de resultado
SELECT ...
# padrão de consulta
WHERE {  ...}
# modificadores de consulta
ORDER BY …
```

3. Trabalhos Relacionados

Apesar das vantagens no uso de RDF para armazenamento de dados, diferentes autores discutem as limitações de sistemas centralizados de RDF para armazenamento de grafos com um número grande de triplas (HU, WANG, YANG e WO). Dessa forma, recentemente algumas abordagens distribuídas de armazenamento RDF baseado em bancos de dados NoSQL foram propostas para melhorar a escalabilidade do sistema e acelerar o desempenho de consultas. Este capítulo apresenta dois principais trabalhos relacionados ao armazenamento de dados RDF em bancos de dados NoSQL.

3.1. Rainbow

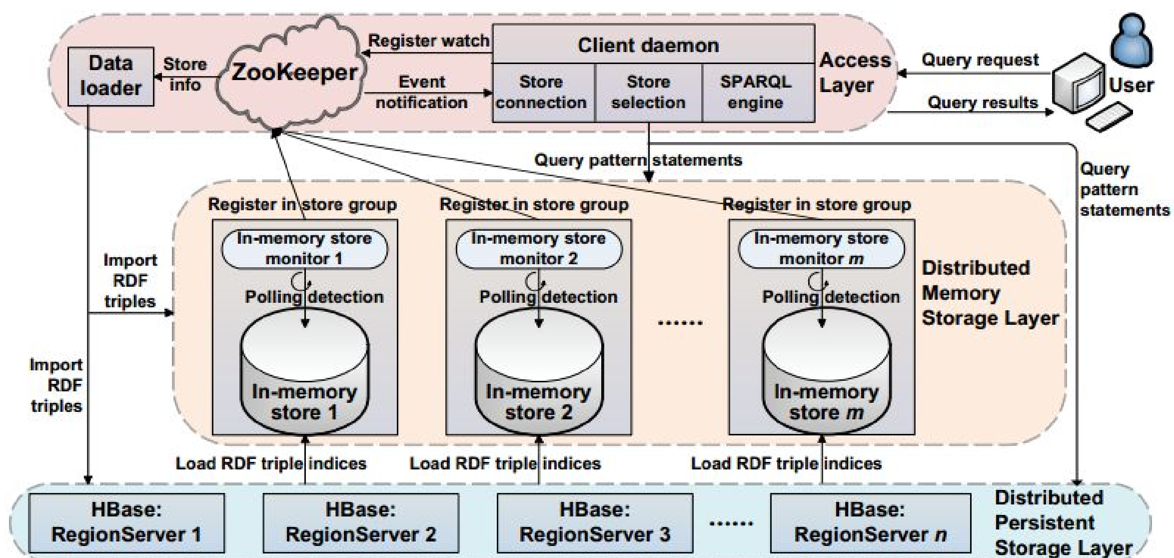
Rainbow, proposto por Gu, Hu e Huang (GU, HU e HUANG, 2014), é uma arquitetura distribuída e hierárquica para o armazenamento de triplas RDF que utiliza o banco de dados colunar HBase para fazer a persistência e técnicas de armazenamento em memória baseado no banco de dados chave-valor Redis para melhorar o desempenho de consultas.

Rainbow utiliza um esquema de índices, salvando os padrões mais recorrentes de triplas em memória. Os autores deste trabalho afirmam que os padrões SP*, *PO e *P* são mais recorrentes (WEIS et al., 2008) na maioria de seus benchmarks, onde S se refere ao Sujeito, P ao Predicado e O ao Objeto. Por isso, construir índices eficientes em cima desses padrões é uma ótima estratégia para melhorar o tempo de consulta. O carregador de dados do Rainbow importa simultaneamente as triplas RDF tanto na memória quando na sua persistência de dados (HBase). Utilizando o padrão de qual foi o último dado usado (LRU), os dados são substituídos durante

carregamento e migrações. Para esse armazenamento em memória é utilizado o banco de dados chave-valor Redis.

Como apresentado na Figura 8, o Rainbow é dividido em três camadas: a camada inferior é o armazenamento persistente no HBase, na camada intermediária fica o sistema distribuído de persistência em memória e na camada superior se encontra o cliente que recebe consultas dos usuários. O cliente possui três módulos importantes. O primeiro é um *engine* SPARQL responsável pela análise das consultas SPARQL e geração de planos de otimização para essas consultas. O segundo, chamado de *Store Selection*, é quem decide executar as consultas em algum dos armazenamentos em memória ou trocar para o HBase subjacente. O terceiro é chamado *Store Connection*, que gerencia as conexões de armazenamento e monitora o estado dos armazenamentos em memória.

Figura 8 - Visão Geral da Arquitetura de Sistema do Rainbow



Fonte: (RAINBOW, 2014)

O Rainbow possui tolerância a falhas e alta disponibilidade. Sua arquitetura de persistência herda do HBase a tolerância a falhas, utilizando o método de duplicação de dados. Porém, duplicar dados através de armazenamento em memória diminui a capacidade de armazenamento em memória. Por isso, o Rainbow utiliza uma estratégia de migração de dados da memória para sua persistência de maneira “lazy”, ou seja, sendo apenas carregado conforme a demanda. Baseado em análises estatísticas apresentadas no trabalho é possível notar que o Rainbow se comporta muito bem, aproveitando-se bastante de suas características principais de tolerância a falhas e escalabilidade dinâmica. A utilização do Redis como um esquema de cache em memória permite acelerar grandes consultas. A escalabilidade mostra-se notável em ambientes distribuídos, onde o aumento no número de máquinas não afeta o desempenho.

Em relação ao Rainbow, as principais diferenças do RDF2Multimodel são: a utlização do Redis no RDF2Multimodel para resolução de consultas, não só como um sistema de cache auxiliar para a aceleração de consultas grandes; e o mapeamento de dados RDF para um banco de dados baseado em grafo a fim de aproveitar a semelhança de formato para acelerar consultas que percorrem grafos.

3.2. ScalaRDF

O ScalaRDF (HU, WANG, YANG e WO, 2016) se propõe a ser um mecanismo de armazenamento e consultas de triplas RDF que possui as características de ser distribuído, em memória que seja tolerante a falhas e escalável. Os autores definem o projeto como um protocolo de *hashing* consistente que otimiza a atribuição de dados RDF, operações de dados e alcança uma re-distribuição elástica de dados em casos de um nó de cluster sair ou se unir, evitando a oscilação holística, quando é necessário utilizar heurísticas para determinar, de um armazenamento de dados.

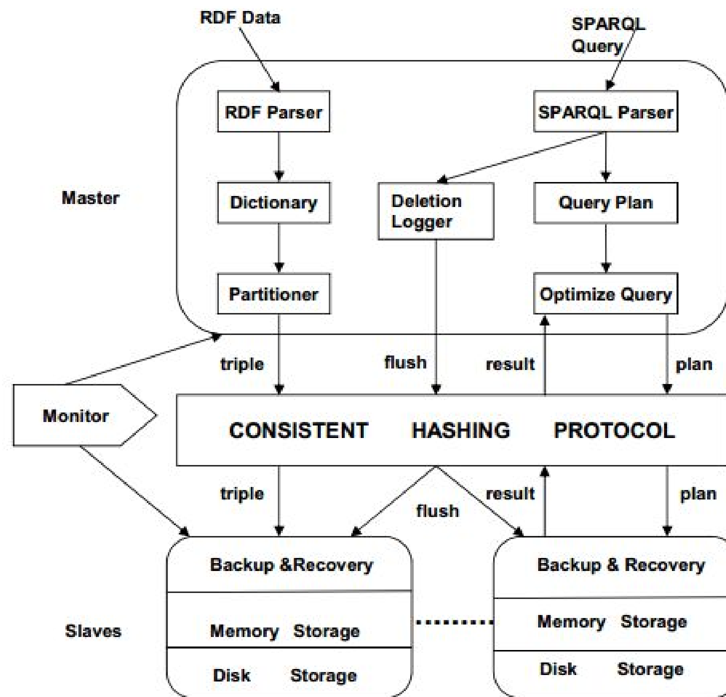
A principal motivação para o ScalaRDF é o gargalo de escalabilidade e confiabilidade dos sistemas centralizados de RDF. O sistema baseado em grafos

precisa executar um reparticionamento holístico de dados entre nós existentes para fazer o balanceamento caso novos dados ou novas máquinas sejam adicionados. Este processo é bastante custoso computacionalmente e consome um grande tempo quando considerado em larga escala.

Assim como o Rainbow, o ScalaRDF também é implementado com base no Redis e utiliza o Sesame como mecanismo de consultas. A arquitetura segue um modelo *master-slave* onde o *master* é responsável por analisar o arquivo RDF e distribuir as triplas entre diferentes *slaves*.

Conforme apresentado na Figura 9, o nó mestre também analisa as consultas SPARQL e gera o seu plano de execução. Seu primeiro componente é o *RDF Parser*, que possui a responsabilidade de analisar arquivos RDF (representados no formato TTL/N3), dos quais lê os arquivos e extrai as triplas. Logo após, no *Dicionário*, as triplas geradas pelo *RDF Parser* originalmente em um formato de string são sumarizadas e cada parte da tripla é substituída por um ID para reduzir o espaço ocupado. O mapeamento <ID, String> é guardado no Dicionário. As triplas geradas pelo Dicionário passam pelo *Partitioner*, que converte elas em 6 índices chave-valor "(i.e., SP_O, PO_S, SO_P, P_SO, O_SP, S_PO)", onde S determina o Sujeito, P o predicado e O o objeto. Esses índices são distribuídos para os nós *slaves* para permitir que o protocolo de *hashing* consistente execute caso haja inserção de dados.

Figura 9 - Visão Geral da Arquitetura de Sistema do ScalaRDF



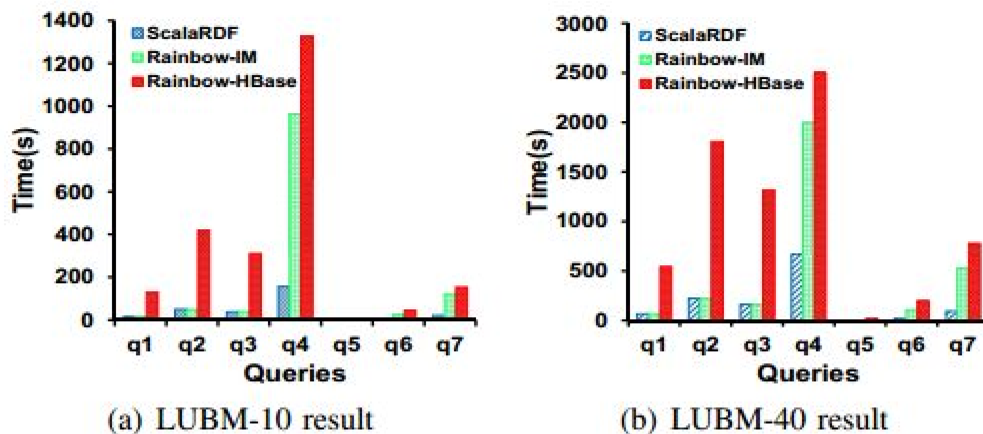
Fonte: (SCALARDF, 2016)

Outro componente do ScalaRDF é o *SPARQL Parser & Query Plan & Plan Opimitized Query*. Esse analisador é responsável por processar consultas de entrada e converter em um plano de consulta. Esse plano então é separado em tarefas de computação paralela e executado dentro do *cluster*. Considerando os custos da operação de deleção foi adicionado também um componente chamado *Deletion Logger*, que utiliza *logs* para salvar as operações em disco. Os logs são salvos no *master* em caso de perda de dados. Os componentes *Backup & Recovery* e *Memory & Disk Storage* são utilizados pelos *slaves* para salvar índices em disco ou memória como *backup*. Por fim, há um componente *Monitor* responsável por notificar os nós correspondentes, uma vez que um nó se une ou sai, para que lidem com os eventos e ativem recuperações e ajustes.

O artigo enfatiza que o ScalaRDF oferece um grande ganho de desempenho em experimentos com tempos de consulta e tempo de atualizações. Em geral, seu ganho

pode ser de 87% a 90% em relação a outros trabalhos similares, como pode ser observado na Figura 10. Porém o trabalho apresenta ainda alguns pontos passíveis de melhoria: o mecanismo centralizado de buscas causa um gargalo no nó *master*, que é responsável por todas as requisições de consulta. Apesar do aumento de velocidade que é possível com as técnicas aplicadas, o gargalo no nó master pode ocasionar sérios problemas de estabilidade e confiabilidade, visto seu potencial para ser um ponto único de falhas.

Figura 10 - Comparação do Tempo de Performance de Consultas do ScalaRDF e Rainbow para os datasets LUBM-10 e LUBM-40



Fonte: (SCALARDF, 2016)

4. Ferramenta Proposta: RDF2Multimodel

Este capítulo apresenta o desenvolvimento da ferramenta proposta neste trabalho de conclusão de curso (*RDF2Multimodel*), bem como o algoritmo para o mapeamento de dados RDF para bancos de dados NoSQL baseados em grafo e chave-valor. A RDF2Multimodel é uma ferramenta que, a partir de entradas de dados RDF, persiste esses dados nos bancos de dados NoSQL Redis (chave-valor) e Neo4j

(grafo). Ela também realiza a tradução de consultas SPARQL sobre estes dados de forma transparente, ou seja, sem que o usuário tenha conhecimento de que estes dados estão persistidos em bancos de dados NoSQL e que uma tradução para os mecanismos de acesso destes bancos de dados é realizada.

A seguir são explicadas as tecnologias utilizadas no desenvolvimento da RDF2Multimodel e a descrição de seus principais módulos.

4.1. Rails API

Rails é um *framework* de desenvolvimento para aplicações web desenvolvido para o Ruby. A proposta deste *framework* é facilitar o desenvolvimento de aplicações web utilizando-se da simplicidade de assumir convenções ao invés de configurações (RAILS, 2017).

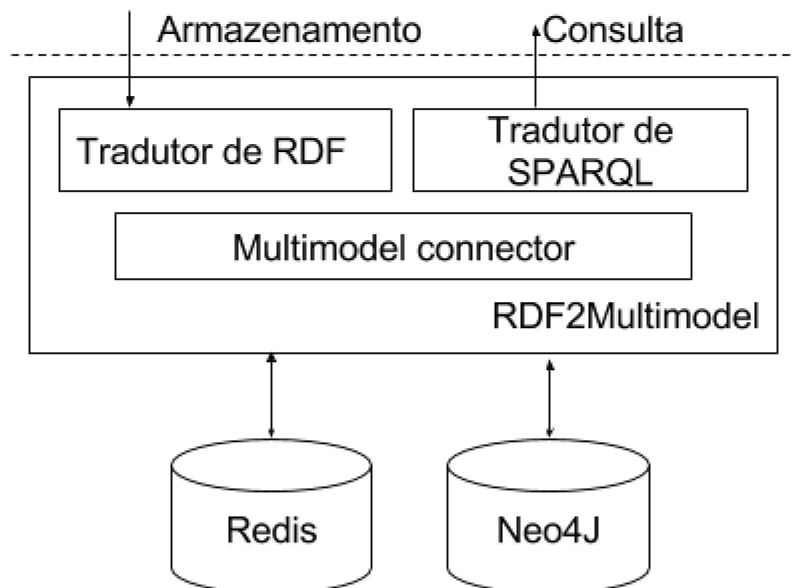
Para esse trabalho foi utilizado uma biblioteca chamada rails_api, que consiste em um subconjunto do Rails otimizado para aplicações do tipo API (RAILS API, 2017). Desta forma, o projeto da RDF2Multimodel pode utilizar todas as facilidades oferecidas pelas convenções do Rails e ser otimizado para seu propósito de API.

4.2. Arquitetura do RDF2Multimodel

O objetivo principal deste trabalho é implementar uma ferramenta de mapeamento multimodelo de dados RDF para bancos de dados NoSQL, para que possa se demonstrar que faz sentido apostar em persistência poliglota para RDF, já que diferentes modelos de dados podem responder melhor a tipos diferentes de consultas. Conforme apresentado na Figura 11, esse mapeamento permite a persistência de RDF nos bancos de dados grafo Neo4j e chave-valor Redis.

A ferramenta RDF2Multimodel consiste de um analisador SPARQL e de uma estratégia de mapeamento de dados RDF que permitem, respectivamente, a consulta e a inserção em bancos de dados de baseados em grafos e chave-valor.

Figura 11 - Arquitetura de alto nível da ferramenta RDF2Multimodel



A RDF2Multimodel funciona como uma API a partir da qual é possível inserir triplas RDF e fazer as consultas às mesmas. Foi utilizado o padrão de API *RESTful*, que utiliza protocolo HTTP para executar transações dos tipos (RODRIGUEZ):

- POST, quando se precisa criar um objeto na aplicação;
- GET, para recuperar objetos;
- PUT, para modificar o estado de um objeto;
- DELETE, para remover um registro.

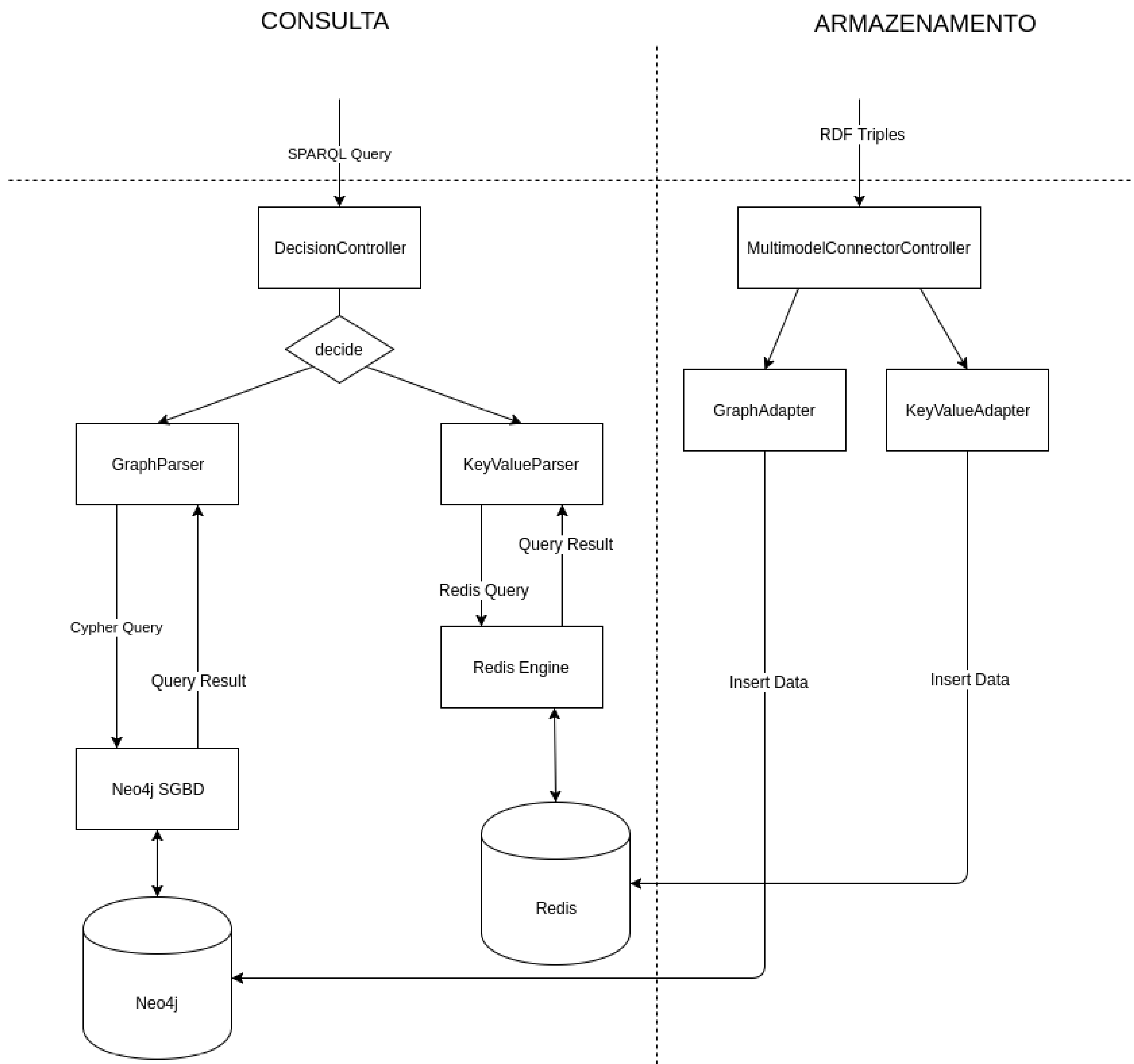
Com base nos padrões REST foram criados *endpoints*, pelos quais a aplicação Web utiliza esses recursos. As seções a seguir detalham as operações realizadas pela ferramenta. O fluxo de execução das operações é mostrado na Figura 12.

4.3. Inserindo Triplas RDF no RDF2Multimodel

As rotas desenvolvidas para a inserção de triplas RDF são:

- POST “/save-triple-to-redis”;
- POST “/save-triple-to-neo4j”;
- POST “/insert-data”.

Figura 12 - Visão de alto nível da execução de operações através da RDF2Multimodel

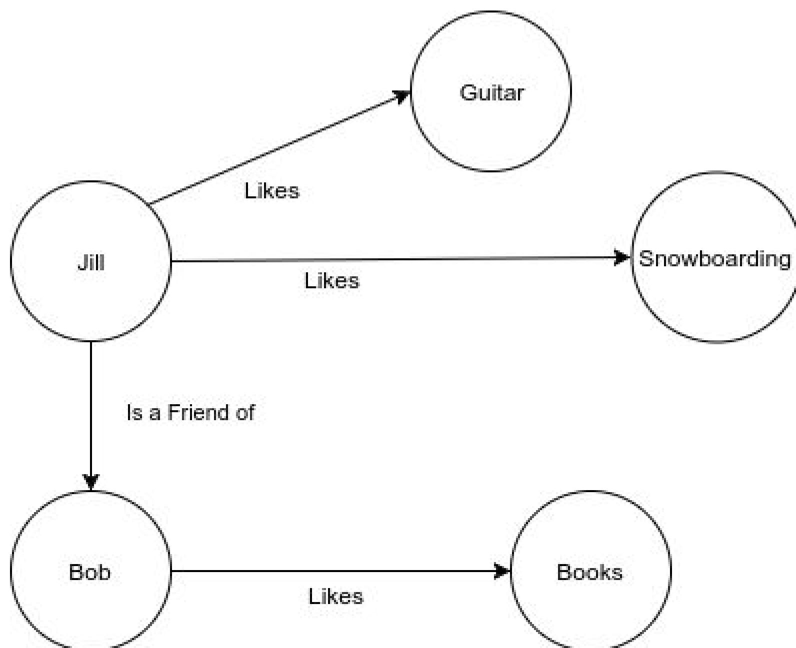


As duas primeiras funcionam individualmente caso se deseje inserir o dado em apenas um dos dois bancos de dados. Já a terceira recebe as triplas RDFs formatadas como um vetor de elementos no formato JSON[1], como por exemplo: “[‘João’, ‘Gosta’, ‘Futebol’], [‘João’, ‘é amigo de', ‘Pedro’]” e insere esta tripla RDF ao mesmo tempo no Neo4j e no Redis.

Ao receber a requisição para os caminhos citados anteriormente, a camada de controle, através do módulo *MultimodelConnectorController* chama os módulos *KeyValueAdapter* e *GraphAdapter*, que realizam a persistência dos dados nos dois bancos de dados.

O módulo *KeyValueAdapter* é responsável por realizar o mapeamento do dado RDF recebido para o banco de dado baseado em chave-valor Redis. A estratégia adotada para salvar o dado no Redis foi considerar a chave do elemento salvo no Redis como sendo uma concatenação do *sujeito* e *predicado* da tripla RDF, com o valor sendo, por consequência, o *objeto*. Foi adotado o padrão do Redis para nomenclaturas de chaves, que consiste em usar dois pontos para separação de espaços. Por isso, a chave é montada da forma “Sujeito:Predicado”. Essa estratégia de utilização do sujeito e predicado como chave foi definida para que o Redis possa ser utilizado para consultas simples, onde, a partir de um sujeito e predicado se quer saber um objeto, ou com apenas uma variável de busca. A Figura 14 mostra como os dados inseridos utilizando essa estratégia ficam organizados no cliente do Redis.

Figura 13 - Grafo utilizado de Exemplo



Para que as buscas por padrões de uma variável em consultas SPARQL sejam possíveis, foi adotada também uma inserção do tipo “Predicado => Sujeito:Predicado:Objeto”. Essas duas formas de inserção no Redis ocorrem simultaneamente e permitem buscas também por um predicado a fim de encontrar os sujeitos e objetos que se relacionam a partir dele.

Figura 14 - Exemplo de dados Inseridos com as duas estratégias no Redis

```
$redis-cli
127.0.0.1:6379> KEYS *
1) "JILL:IS A FRIEND OF"
2) "LIKES"
3) "JILL:LIKES"
4) "IS A FRIEND OF"
5) "BOB:LIKES"
127.0.0.1:6379> SMEMBERS "LIKES"
1) "JILL:LIKES:SNOWBOARDING"
2) "JILL:LIKES:GUITAR"
3) "BOB:LIKES:BOOKS"
127.0.0.1:6379> SMEMBERS "JILL:LIKES"
1) "SNOWBOARDING"
2) "GUITAR"
127.0.0.1:6379> SMEMBERS "JILL:IS A FRIEND OF"
1) "BOB"
127.0.0.1:6379> _
```

O mapeamento de dados RDF para o banco de dados baseado Neo4j foi definido considerando as particularidades da linguagem de consulta e inserção Cypher (What is CYPHER, 2017), nativa do Neo4j, para permitir um mapeamento direto. O algoritmo do módulo *GraphAdapter* utiliza a cláusula *UNWIND* do Cypher que converte uma dada lista em uma coluna de valores, como mostrado na Figura 15 (UNWIND).

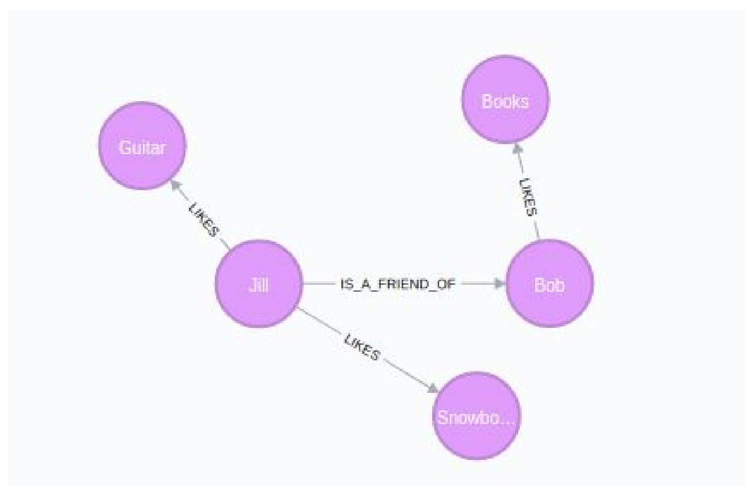
Figura 15 - Exemplo da utilização do UNWIND

```
UNWIND [1, 2, 3] AS x
RETURN x

# Resultado:
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 linhas
```

Depois disso, cada um dos elementos presentes no parâmetro da cláusula UNWIND passado para a API é inserido, de acordo com a sua posição no vetor, utilizando a cláusula MERGE do Cypher, que encontra ou cria uma relação com o nome especificado. Esse método de inserção do Neo4j requer que os vértices e arestas do grafo possuam um tipo. Como não essa informação não é relevante neste trabalho, ela é ignorada e apenas especificado como um tipo padrão “default”. A Figura 16 mostra como os dados ficam organizados na interface gráfica do Neo4j, com o predicado agindo como o valor da aresta e os objetos e sujeitos como vértices.

Figura 16 - Exemplo de dados inseridos no Neo4j



4.4. Execução de consultas utilizando SPARQL no RDF2Multimodel

A ferramenta RDF2Multimodel também é capaz de processar alguns tipos de consultas SPARQL. Para tanto, é utilizado a rota GET "/parse" passando o parâmetro "query" na URL, que contém a consulta a ser feita. Ao submeter essa consulta, ela primeiro passa por um avaliador chamado *DecisionController* (ver Figura 12), que através de uma heurística simples decide se a consulta será executada no Redis ou no Neo4j.

Caso a consulta tenha menos de dois parâmetros, ela é executada no Redis, aproveitando-se das estratégias de inserção citadas na seção anterior. Caso seja uma consulta composta, utilizando-se de padrões de triplas que tenham relacionamentos entre si, ela é roteada para o módulo *GraphParser* que faz uma interpretação e tradução da consulta SPARQL para uma consulta em Cypher.

Para isso, o parâmetro recebido na consulta é transformado em um vetor e suas partes são separadas para que possam ser traduzidas. Essa partes são: *Padrão de Resultado*, *Padrão da Consulta* e *Modificadores da Consulta*. A definição dessas partes é baseada na estrutura de uma consulta SPARQL definida pela W3C (FEIGENBAUM, 2009), que especifica as partes de uma consulta SPARQL como:

- **Declaração de prefixo**, para abreviação de URIs;
- **Definição do conjunto de dados**, especificando quais grafos RDF estão sendo requeridos;
- **Cláusula de resultado**, identificando que informação retornar da consulta;
- **Padrão da consulta**, especificando o que buscar no conjunto de dados;
- **Modificadores da Consulta**: fatiamento, ordenação, limites e outras maneiras de organizar os resultados das consultas.

Para os exemplos utilizados neste trabalho foram consideradas apenas as partes centrais da consulta, ou seja, *Cláusula de Resultado, Padrão de Consulta e Modificadores da Consulta*.

A extração da *Cláusula de Resultado* foi realizada com a utilização do método *split* das strings do Ruby, que divide uma string em substrings com base em um dado delimitador e retorna um vetor com essas substrings. O delimitador utilizado para a extração da *Cláusula de Resultado* foi “WHERE” que, como é possível observar na Figura 16, marca o final da *Cláusula de Resultado* e o início do *Padrão de Consulta*.

Uma vez realizada esta extração, o primeiro elemento do vetor resultante é considerado, ou seja, todo texto entre “SELECT” e “WHERE” e é realizada a tradução. Esta parte da consulta possui uma tradução direta simples para o Cypher, apenas trocando a palavra-chave “SELECT” por “RETURN”.

Figura 17 - Estrutura de uma Consulta SPARQL

```
# prefix declarations
PREFIX foo: <http://example.com/resources/>
...
# dataset definition
FROM ...
# result clause
SELECT ...
# query pattern
WHERE {
    ...
}
# query modifiers
ORDER BY ...
```

Fonte: (FEIGENBAUM, 2009)

Para o *Padrão de Consulta* foi utilizada a mesma separação da string original a partir do “WHERE”. Porém, neste caso foi utilizada a segunda parte do vetor resultante. Depois disso é feito um mapeamento de quais elementos serão utilizados como sujeito, predicado e objeto na consulta.. Para os dois primeiros componentes, considerando as convenções de consultas SPARQL, foi realizada uma busca por elementos com o caractere “?”, que mostra que é uma variável. Após essa identificação é realizado um mapeamento para uma chave *hash* de sujeito e objeto que é aplicado ao modelo de consulta Cypher.

A identificação do objeto é realizada utilizando dois métodos nativos da linguagem Ruby chamados *index* e *rindex*. O primeiro encontra o indíce de um determinado caractere dentro de uma string e o segundo encontra o índice reverso de um caractere, começando a contagem a partir do último elemento. Com esses dois dados são buscadas a primeira e a última ocorrência de um caractere do tipo aspa na string, que denota o objeto da consulta.

Depois dos três elementos terem sido mapeados, eles são inseridos em uma consulta Cypher. Para determinar o tipo de consulta a ser feita, como por exemplo: “Amigo de Um Amigo”, ”Gosta de”, é feita uma varredura pela string para encontrar o elemento central que identifica o predicado. Isso é feito procurando por um elemento na string que não contenha um dos elementos que mostram que ela é uma variável (por exemplo: ?x) ou um objeto (por exemplo: ‘X’) ou uma das palavras reservadas de modificação da consulta.

Se a decisão for de que a consulta deve ser executada no Redis, isso significa que ela possui uma complexidade mais baixa. Dessa forma, o algoritmo para o mapeamento é feito de maneira trivial. Uma busca em todas as chaves disponíveis no Redis é realizada e, após separar o sujeito do predicado, a partir do separador “:”, é verificado se o segundo elemento da chave separada é igual ao valor de predicado. Caso seja, ele monta novamente a chave usando o separador (:) e envia a requisição de consulta para o Redis solicitando todos os membros daquele conjunto determinado através do uso da função *SMEMBERS*.

No caso da consulta do Redis possuir um predicado e um objeto e se deseja descobrir quem é o sujeito, é feita a separação novamente da consulta e separado quem é o predicado. Após ter essa informação, aproveita-se da estratégia *P* inserida no Redis para recuperar todas as triplas que apresentam este sujeito. Uma vez tendo todas as triplas, é realizada uma varredura em cada uma delas, procurando qual delas possui o objeto especificado na consulta.

O próximo capítulo apresenta uma avaliação da ferramenta proposta.

5. Avaliação da Ferramenta RDF2Multimodel

Este capítulo apresenta os testes realizados para a avaliação do funcionamento e do desempenho da RDF2Multimodel. Todos os testes foram realizados em um computador com a seguinte configuração:

- Processador: Intel Core i7-5500U @ 2.40 GHz x 4;
- 8GB de memória RAM;
- Disco Rígido de 1TB e;
- Sistema Operacional Ubuntu 17.04.

A ferramenta roda em um servidor Puma 3.10.0 utilizando Ruby 2.4.0, Rails 5.05, Redis versão 3.0.5 e Neo4j versão 3.2.3. A seguir são detalhados os testes realizados.

5.1. Testes de Validação da Ferramenta

O primeiro teste proposto para a validação da ferramenta foi utilizar uma massa de dados pequena para verificar se as estratégias de inserção no Redis e Neo4j estavam se comportando de acordo com o esperado. Para esse teste foi usado o conjunto de dados apresentado na Tabela 1.

Tabela 1 - Conjunto de Dados de Validação

Sujeito	**Predicado**	**Objeto**
John	Is a Friend of	James
John	Is a Friend of	Jill
James	Is a Friend of	Jesse
James	Is a Friend of	Jill
Jill	Likes	Snowboarding
Jill	Is a Friend of	Doug
Doug	Plays	RPG
Snowboarding	Is a	Sport

Já para os testes de desempenho (ver Tabela 2) foi utilizado o módulo *Benchmark* disponível no Ruby visando testar o tempo de resposta de todos *endpoints* comentados no capítulo anterior. O módulo Benchmark tem como métricas:

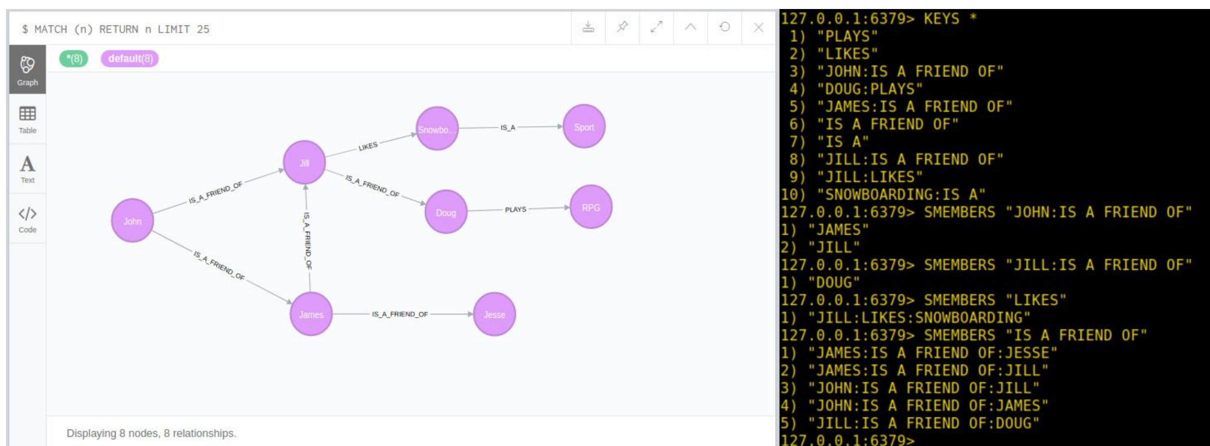
- *user* CPU time, que representa o tempo total que o processador trabalhou para o código em específico;
- *system* CPU time, que mostra o tempo que o processador trabalhou em funções do sistema operacional relacionadas ao programa;
- *total,* que é uma soma desses dois valores;
- *real*, que mostra o tempo real que se passou para a execução do trecho de código.

Todos os tempos estão expressos em segundos.

Tabela 2 - Tabela de Resultados do Experimento de Validação

endpoint	user	system	total	real
/insert-data	0.020000	0.010000	0.030000	0.601163
/save-triple-to-redis	0.010000	0.000000	0.010000	0.016510
/save-triple-to-neo4j	0.020000	0.000000	0.020000	0.482063
/parse	0.010000	0.000000	0.010000	0.004832

Figura 18 - Visualização dos dados inseridos no teste de validação da ferramenta no Neo4j e Redis



A Figura 17 mostra a visualização dos dados inseridos no Redis e no Neo4j após o primeiro teste de validação utilizando os dados citados anteriormente e as estratégias de mapeamento explicadas no capítulo 4.

5.2. Experimento de Inserção com *dataset* gerado aleatoriamente

Esta seção apresenta um experimento de inserção para quantidades diferentes de triplas. Para gerar esses dados foi utilizada uma extensão da biblioteca de código

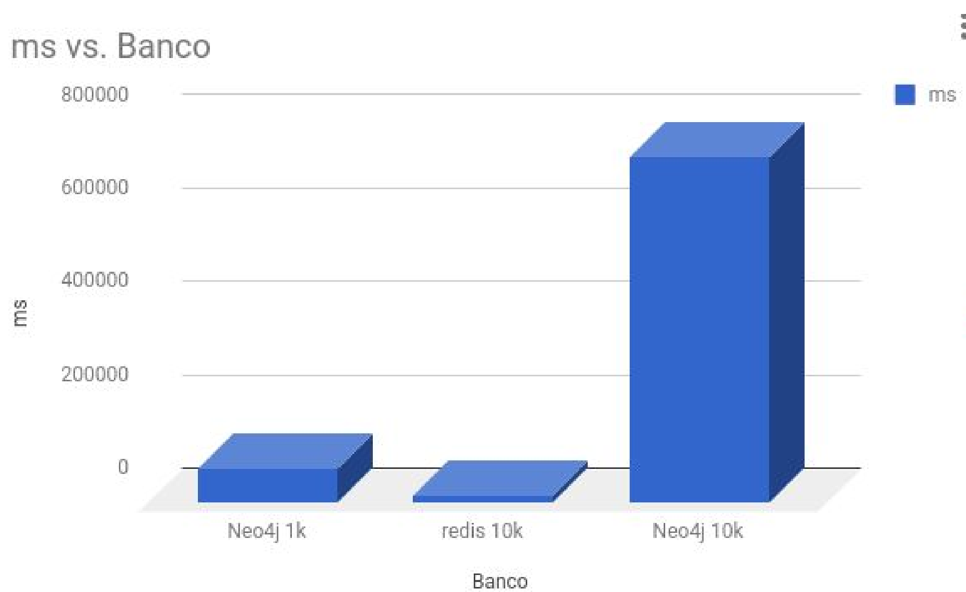
aberto *Faker*[5] para gerar triplas aleatórias. Devido à aleatoriedade dos dados, esse *dataset* foi utilizado apenas para testes de inserção.

A Figura 18 mostra os tempos de inserção, em milissegundos, para a inserção de mil, 10 mil e 100 mil Triplas, no formato [Sujeito, Predicado, Objeto]. Para mil triplas tem-se um tempo de 1 minuto e 48 segundos, para 10 mil triplas um tempo de 13 minutos e para 100 mil triplas com tempo de aproximadamente 120 minutos. Desta forma os tempos de inserção variam entre 9.25 triplas por segundo e 13.9 triplas por segundo. Os tempos de inserção obtidos para o Neo4j ficaram abaixo do que era esperado, e podem ser melhorados modificando o algoritmo de inserção no Neo4j para utilizar maneiras diferentes de fazer o *merge* de nós, sem necessidade de fazer o UNWIND citado na seção anterior.

Figura 19 - Gráfico de tempo de inserção em milissegundos

[5] https://github.com/stympy/faker

Figura 20 - Gráfico de tempo de Inserção em milissegundos separado por BD



A Figura 19 mostra a distribuição de tempo dividida entre os dois bancos de dados para as quantidades de mil e 10 mil triplas. A partir dessa Figura é notável que o fadaptador para Neo4j é o que está afetando de forma crítica o tempo de inserção e precisa ser otimizado. O tempo de inserção no Redis se mostrou bastante performático, inserindo mil Triplas em menos de 1 segundo (408 ms) e 10 mil triplas em 14 segundos.

5.3. Experimento de consultas com dataset gerado aleatoriamente

Neste experimento foi utilizado o mesmo *dataset* de 100 mil triplas gerado na seção anterior e foi testado o tempo de consultas diferentes para medir o tempo de resposta quando se utiliza o Neo4j, o Redis no padrão SP*, bem como o Redis no padrão *P*. Para o cálculo do tempo médio médio de resposta foi utilizada uma média simples entre o tempo de 10 execuções de cada consulta. A primeira consulta da Tabela 3 foi escolhida por possuir uma junção, permitindo assim utilizar-se do mapeamento do Neo4j para percorrer os grafos. As outras duas consultas são mais simples, com a segunda não possuindo a especificação de um Objeto, para que seja

explorado o padrão SP* e a última com um Objeto especificado para demonstrar o padrão *P*.

Tabela 3 - Tempo Médio de Respostas de Consultas SPARQL

Consulta	Acesso	Tempo Médio de Resposta
SELECT ?p WHERE { ?p CHATS ?x . ?x EATS 'Lasagne' }	Neo4j	18.8ms
SELECT ?r WHERE { ?p EATS ?x }	Redis - Estratégia SP*	31.6ms
SELECT ?r WHERE { ?p EATS 'SOUVLAKI' }	Redis - Estratégia *P*	1.3ms

Com base nos resultados apresentados na Tabela 3, é possível ver que a estratégia de mapeamento *P* do Redis se mostrou extremamente eficiente em relação a estratégia SP*, tendo uma velocidade de resposta 24 vezes mais rápida

6. Conclusão

Este trabalho apresenta a RDF2Multimodel, uma ferramenta que é capaz de fazer a persistência de triplas RDF em bancos de dados NoSQL de dois modelos distintos (Redis (Chave-valor) e Neo4j (Grafo)), bem como traduzir consultas SPARQL para que possam ser executadas nos mesmos.

O mapeamento proposto para o banco de dados baseado em Chave-Valor se mostrou bastante eficiente tanto em tempo de inserção quanto ao tempo de consulta. Já para o Neo4j obteve-se um desempenho não muito alto, porém esse tempo pode

ser otimizado com técnicas de paralelização, distribuição e com escala de processamento.

Os algoritmos, técnicas de mapeamento e considerações apresentados neste trabalho podem ser utilizados em trabalhos futuros para otimizar o mapeamento e velocidade de acesso de ferramentas que lidem com RDF, tanto utilizando-se da API proposta quanto baseando-se em suas características para que possam ser incorporadas em outras ferramentas.

O RDF2Multimodel cumpre os requisitos propostos nesse trabalho, fazendo o mapeamento de Triplas RDF para os bancos Neo4j e Redis assim como a tradução de alguns padrões de consulta SPARQL para que seja possível acessar esses dados.

Como trabalhos futuros é possível apontar:

- Extensão dos padrões de consulta Cypher para que seja possível mapear consultas ainda mais complexas para serem feitas no Neo4j;
- Processamento paralelo distribuído da inserção no Neo4j a fim de aumentar o tempo de inserção, que está em volta de 0.06s por Tripla;
- Implementação de um módulo que faça a tradução do formato RDF na entrada do RDF2Multimodel. Atualmente, só é possível inserir um vetor simulando uma Tripla RDF. Com a implementação desse módulo pode-se estender as capacidades do RDF2Multimodel para receber um arquivo no formato RDF/XML e proceder a sua análise e tradução;
- Implementação de um *Cliente* que facilite a manipulação dos endpoints criados na API;
- Realização de experimentos com outros modelos de dados NoSQL;
- Realização de experimentos utilizando o LUBM, para que seja possível comparar o RDF2Multimodel ao estado da arte de ferramentas que trabalham com RDF.

Referências Bibliográficas

RDF 1.1 CONCEPTS and Abstract Syntax, Disponível em: <https://www.w3.org/TR/rdf11-concepts>. Acessado em 24 Nov. 2016.

O QUE É NoSQL?, Disponível em: <https://aws.amazon.com/pt/nosql/>. Acessado em 24 Nov. de 2016.

Data modeling with MULTI-MODEL databases, Disponível em: <https://www.oreilly.com/ideas/data-modeling-with-multi-model-databases>. Acessado em 5 de Dez. de 2016.

Resource Description Framework (RDF), Disponível em: <https://www.w3.org/RDF/>. Acessado em 24 Nov. 2016.

RDF GRAPHS, Disponível em: <https://www.w3.org/TR/rdf11-concepts/#section-rdf-graph>. Acessado em 24 Nov. 2016.

GAI, Lei, CHEN, Wei e WANG, Tengjiao., **An Efficient Summary Graph Driven Method for RDF Query Processing**; 2015. Disponível em: <https://arxiv.org/pdf/1510.07749.pdf>

RDF 1.1 JSON Alternate Serialization (RDF/JSON), Disponível em: <https://dvcs.w3.org/hg/rdf/raw-file/default/rdf-json/index.html>. Acessado em 24 Nov. 2016.

RUBY, Disponível em: <https://www.ruby-lang.org/pt/>. Acessado em 24 Nov. 2016.

Ruby's FLEXIBILITY, Disponível em: <https://www.ruby-lang.org/en/about>. Acessado em 5 Dez. 2016.

DB-Engines Ranking Disponível em <https://db-engines.com/en/ranking>. Acessado em 28 Jun. 2017.

FOWLER, Martin e SADALAGE, Pramod J., **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**, 1. ed. Addison-Wesley Professional; 2012, p. 29

FOWLER, Martin e SADALAGE, Pramod J., **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**, 1. ed. Addison-Wesley Professional; 2012, p. 51

ALECRIM, Emerson, **O que é cloud computing?**, Disponível em: <https://www.infowester.com/cloudcomputing.php>, 2008. Acessado em: 28 de Jun. de 2017.

What is an IN-MEMORY KEY-VALUE Store? Disponível em: <https://aws.amazon.com/pt/nosql/key-value/>, Acessado em: 29 de Mar. 2017.

MCMURTRY, D. et al. **Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence**. MICROSOFT, 2013. Disponível em: <https://msdn.microsoft.com/en-us/library/dn271399.aspx>, Acessado em: 29 de Mar. 2017.

TIWARI, S. **Professional NoSQL**. John Wiley & Sons, 2011.

Document Databases, Disponível em: <https://www.mongodb.com/document-databases>, Acessado em: 29 de Mar. 2017.

ROBINSON, Ian e WEBBER e EIFRAM, **Graph Databases: New Opportunites for Connected Data**, 2. ed. O'Reilly. 2015.

HU, WANG, YANG e WO, **ScalaRDF: a Distributed, Elastic and Scalable In-Memory RDF Triple Store**, 2016

GU, HU e HUANG, **Rainbow: A distributed and hierarchical RDF triple store with dynamic scalability**, 2014. Disponível em: <http://ieeexplore.ieee.org/abstract/document/7004274/>

FAYE, CURE, BLIN, **A Survey of RDF Storage Approaches.** Arima Journal, 15:11–35, 2012. Disponível em <http://arima.inria.fr/015/pdf/Vol.15.pp.11-35.pdf>

BERNERS-LEE, **Design Issues**, Disponível em: <https://www.w3.org/DesignIssues/LinkedData.htm>, 2006. Acessado em: 19 de Jun. 2017

WEIS, KARRAS e BERNSTEIN, **HEXASTORE: Sextuple Indexing for Semantic Web Data Management**. PVLDB, 2008, pp. 1008-1019. Disponível em: <http://www.vldb.org/pvldb/1/1453965.pdf>

Introduction to REDIS, Disponível em: <https://redis.io/topics/introduction>. Acessado em: 19 de Jun. 2017

Who Uses Redis?, Disponível em: <http://techstacks.io/tech/redis>, Acessado em: 19 de Jul. 2017

SPARQL Query Language for RDF, Disponível em : <https://www.w3.org/TR/rdf-sparql-query/>, 2008. Acessado em: 28 de Jun. de 2017

FEIGENBAUM Lee, **SPARQL by Example**, Disponível em: .<http://www.cambridgesemantics.com/semantic-university/sparql-by-example>. Acessado em: 28 de Jun. de 2017.

ROUSE, Margaret**.** **What is Multimodel Database?**, Disponível em: <http://searchdatamanagement.techtarget.com/definition/multimodel-database>. Acessado em 29 de Jun. de 2017.

ARTHUR, Lisa. **What is Big Data**?, Disponível em: <https://www.forbes.com/sites/lisaarthur/2013/08/15/what-is-big-data/#5cfaa97a5c85>. Acessado em 30 de Jul. de 2017.

UNWIND, Neo4j, Disponível em: <https://neo4j.com/docs/developer-manual/current/cypher/clauses/unwind/#unwind-create-nodes-from-a-list-parameter>, Acessado em 22 de Setembro de 2017

What is CYPHER?, Neo4j. Disponível em: <https://neo4j.com/docs/developer-manual/current/cypher/#cypher-intro>, Acessado em: 17 de Outubro de 2017.

RODRIGUEZ, Alex. **RESTful Web services: The basics**, Disponível Em: <https://www.ibm.com/developerworks/library/ws-restful/index.html>, Acessado em 01 de Outubro de 2017.

SPLIT, Ruby Documentation. Disponível em: <https://apidock.com/ruby/String/split>, Acessado em 10 de Outubro de 2017.

RAILS, Disponível em: <http://guides.rubyonrails.org/getting_started.html>. Acessado em 11 de Outubro de 2017.

GUO Yuanbo, PAN hengxiang, e HEFLIN Jeff, **LUBM: A Benchmark for OWL Knowledge Base Systems**, 2005. Disponível em <http://www.websemanticsjournal.org/index.php/ps/article/view/70/68>

RAILS API, Disponível em <https://github.com/rails-api/rails-api>. Acessado em 11 de Outubro de 2017.

APÊNDICE A - ARTIGO

Uma Ferramenta Para Mapeamento de Dados RDF Para Bancos de Dados NoSQL Grafo e Chave-valor

Matheus Teixeira Pereira, Luiz Henrique Zambom Santana, Ronaldo dos Santos Mello[1]

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

{matheustxrp,lhzsantana}@gmail.com, r.mello@ufsc.br

Abstract. *The work proposes both a algorithm and a tool to map RDF triples to Graph and Key-Value NoSQL's databases, and also the implementation of a module that translates RDF specific queries to this databases query language. Currently, the amount of RDF data has been growing across the Web and few existing projects proposes a way to map this data to NoSQL databases. So, this work aims at developing a tool that can show the way that it is possible to work with NoSQL and RDF, specifically in Graph and Key-Value based databases.*

Resumo. *Este trabalho propõe um algoritmo e ferramenta para mapear triplas RDF para bancos de dados NoSQL baseados em Chave-Valor e Grafo, assim como a implementação de um módulo que realiza a tradução de consultas específicas de RDF para as linguagens de consulta utilizadas por estes bancos de dados. Atualmente existe um grande crescimento na quantidade de dados RDF sendo gerados na Web e poucos dos trabalhos relacionados propõem maneiras de realizar o mapeamento desses dados para bancos de dados NoSQL. Assim sendo, este trabalho visa desenvolver uma ferramenta que possa demonstrar as maneiras que se é possível trabalhar com NoSQL e RDF, especificamente em bancos de dados baseados em Chave-Valor e Grafo.*

1. Introdução

Tanto a qualidade quanto a quantidade dos dados RDF disponíveis na Web tem tido um grande crescimento nos últimos anos, com o RDF sendo adotado em diversos campos como: Redes Sociais (FOAF, OpenGraph), bases de Conhecimento (DBPedia, YAGO, Freebase) e Bioinformática (Bio2RDF, Uniprot), etc (GAI, CHEN e WANG, 2015).

Com o aumento dos dados disponíveis se torna necessário a utilização de diversos métodos de persistência tanto para escalabilidade e portabilidade desses dados, como para que possa ser possível aumentar a performance de consultas feitas sobre os mesmos.

As soluções existentes para persistência de RDF exploram pouco a utilização de bancos de dados NoSQL, como explorado na Seção 3. Desta forma, esse trabalho propõe uma ferramenta para o mapeamento de dados RDF para, especificamente, os modelos de BDs baseados em Grafo e Chave-valor. O mapeamento proposto, assim como a tradução de consultas SPARQL para as linguagens de consultas dos bancos propostos pode ser utilizado tanto para expandir projetos existentes, quanto para trabalhos futuros que queiram trabalhar com RDF e os formatos de NoSQL utilizados.

O foco deste trabalho está voltado aos conceitos de *Resource Description Framework*, (RDF 1.1 CONCEPTS, 2014) - o principal modelo de troca de dados para Web Semântica - e bancos de dados NoSQL (FOWLER e SADALAGE, 2012), especificamente os modelos de dados de grafos e chave-valor. RDF é o principal *framework* para representação de dados da Web, sendo recomendado e mantido pela *W3C*[6] (*World Wide Web Consortium*). Seu modelo de dados é baseado em grafos e sua estrutura segue um modelo de triplas, representadas por um sujeito, um predicado e um objeto. O conjunto destas triplas é chamado de grafo RDF (RDF GRAPHS, 2014).

Já NoSQL é um termo utilizado para descrever bancos de dados não relacionais, geralmente de alto desempenho, sendo reconhecidos também por sua escalabilidade, flexibilidade de esquema e alta disponibilidade.

Este trabalho apresenta um algoritmo e uma ferramenta para realizar a persistência e a consulta de/a dados RDF em bancos de dados baseados nos modelos de dados de grafo e chave-valor. Esta ferramenta facilita trabalhos acadêmicos futuros que necessitem de flexibilidade e facilidade para converter dados RDF para esses modelos de bancos de dados NoSQL.

---

[6] https://www.w3.org/

2. Trabalhos Correlatos

Rainbow, proposto por Gu, Hu e Huang (GU, HU e HUANG, 2014), é uma arquitetura distribuída e hierárquica para o armazenamento de triplas RDF que utiliza o banco de dados colunar HBase para fazer a persistência e técnicas de armazenamento em memória baseado no banco de dados chave-valor Redis para melhorar o desempenho de consultas.

Rainbow utiliza um esquema de índices, salvando os padrões mais recorrentes de triplas em memória. Os autores deste trabalho afirmam que os padrões SP*, *PO e *P* são mais recorrentes (WEIS et al., 2008) na maioria de seus benchmarks, onde S se refere ao Sujeito, P ao Predicado e O ao Objeto. Por isso, construir índices eficientes em cima desses padrões é uma ótima estratégia para melhorar o tempo de consulta. O carregador de dados do Rainbow importa simultaneamente as triplas RDF tanto na memória quando na sua persistência de dados (HBase). Utilizando o padrão de qual foi o último dado usado (LRU), os dados são substituídos durante carregamento e migrações. Para esse armazenamento em memória é utilizado o banco de dados chave-valor Redis.

O ScalaRDF (HU, WANG, YANG e WO, 2016) se propõe a ser um mecanismo de armazenamento e consultas de triplas RDF que possui as características de ser distribuído, em memória que seja tolerante a falhas e escalável. Os autores definem o projeto como um protocolo de *hashing* consistente que otimiza a atribuição de dados RDF, operações de dados e alcança uma re-distribuição elástica de dados em casos de um nó de cluster sair ou se unir, evitando a oscilação holística, quando é necessário utilizar heurísticas para determinar, de um armazenamento de dados.

A principal motivação para o ScalaRDF é o gargalo de escalabilidade e confiabilidade dos sistemas centralizados de RDF. O sistema baseado em grafos precisa executar um reparticionamento holístico de dados entre nós existentes para fazer o balanceamento caso novos dados ou novas máquinas sejam adicionados. Este processo é bastante custoso computacionalmente e consome um grande tempo quando considerado em larga escala.

Assim como o Rainbow, o ScalaRDF também é implementado com base no Redis e utiliza o Sesame como mecanismo de consultas. A arquitetura segue um modelo *master-slave*

onde o *master* é responsável por analisar o arquivo RDF e distribuir as triplas entre diferentes *slaves*.

3. Aplicação Proposta

A ferramenta RDF2Multimodel consiste de um analisador SPARQL e de uma estratégia de mapeamento de dados RDF que permitem, respectivamente, a consulta e a inserção em bancos de dados de baseados em grafos e chave-valor.

Com base nos padrões REST foram criados *endpoints*, pelos quais a aplicação Web utiliza esses recursos. As seções a seguir detalham as operações realizadas pela ferramenta. O fluxo de execução das operações é mostrado na Figura 1.

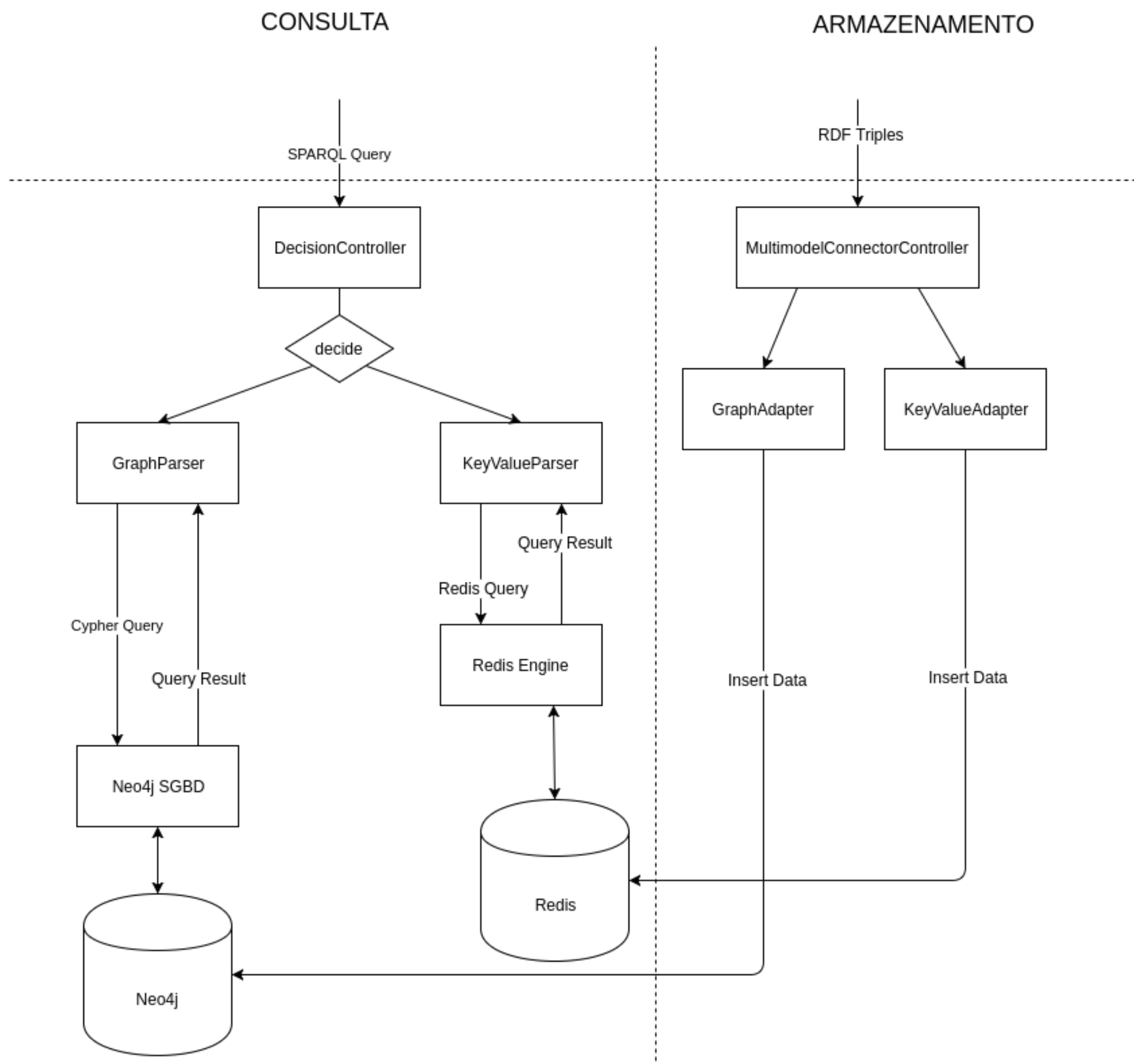


Figura 1. Visão Geral do RDF2Multimodel

As rotas desenvolvidas para a inserção de triplas RDF são:

- POST "/save-triple-to-redis";
- POST "/save-triple-to-neo4j";
- POST "/insert-data".

As duas primeiras funcionam individualmente caso se deseje inserir o dado em apenas um dos dois bancos de dados. Já a terceira recebe as triplas RDFs formatadas como um vetor de elementos no formato JSON[1], como por exemplo: "[‘João’, ‘Gosta’, ‘Futebol’], [‘João’, ‘é amigo de', ‘Pedro’]" e insere esta tripla RDF ao mesmo tempo no Neo4j e no Redis.

O módulo *KeyValueAdapter* é responsável por realizar o mapeamento do dado RDF recebido para o banco de dado baseado em chave-valor Redis. A estratégia adotada para salvar o dado no Redis foi considerar a chave do elemento salvo no Redis como sendo uma concatenação do *sujeito* e *predicado* da tripla RDF, com o valor sendo, por consequência, o *objeto*. Foi adotado o padrão do Redis para nomenclaturas de chaves, que consiste em usar dois pontos para separação de espaços. Por isso, a chave é montada da forma "Sujeito:Predicado". Essa estratégia de utilização do sujeito e predicado como chave foi definida para que o Redis possa ser utilizado para consultas simples, onde, a partir de um sujeito e predicado se quer saber um objeto, ou com apenas uma variável de busca. A Figura 2 mostra como os dados inseridos utilizando essa estratégia ficam organizados no cliente do Redis.

Para que as buscas por padrões de uma variável em consultas SPARQL sejam possíveis, foi adotada também uma inserção do tipo "Predicado => Sujeito:Predicado:Objeto". Essas duas formas de inserção no Redis ocorrem simultaneamente e permitem buscas também por um predicado a fim de encontrar os sujeitos e objetos que se relacionam a partir dele.

```
$redis-cli
127.0.0.1:6379> KEYS *
1) "JILL:IS A FRIEND OF"
2) "LIKES"
3) "JILL:LIKES"
4) "IS A FRIEND OF"
5) "BOB:LIKES"
127.0.0.1:6379> SMEMBERS "LIKES"
1) "JILL:LIKES:SNOWBOARDING"
2) "JILL:LIKES:GUITAR"
3) "BOB:LIKES:BOOKS"
127.0.0.1:6379> SMEMBERS "JILL:LIKES"
1) "SNOWBOARDING"
2) "GUITAR"
127.0.0.1:6379> SMEMBERS "JILL:IS A FRIEND OF"
1) "BOB"
127.0.0.1:6379> _
```

Figura 2. Dados Inseridos no Redis

O mapeamento de dados RDF para o banco de dados baseado Neo4j foi definido considerando as particularidades da linguagem de consulta e inserção Cypher (What is CYPHER, 2017), nativa do Neo4j, para permitir um mapeamento direto.

4. Experimentos e Resultados

Esta seção apresenta um experimento de inserção para quantidades diferentes de triplas. Para gerar esses dados foi utilizada uma extensão da biblioteca de código aberto *Faker*[7] para gerar triplas aleatórias. Devido à aleatoriedade dos dados, esse *dataset* foi utilizado apenas para testes de inserção.

A Figura 3 mostra os tempos de inserção, em milissegundos, para a inserção de mil, 10 mil e 100 mil Triplas, no formato [Sujeito, Predicado, Objeto]. Para mil triplas tem-se um tempo de 1 minuto e 48 segundos, para 10 mil triplas um tempo de 13 minutos e para 100 mil triplas com tempo de aproximadamente 120 minutos. Desta forma os tempos de inserção variam entre 9.25 triplas por segundo e 13.9 triplas por segundo.

---

[7] https://github.com/stympy/faker

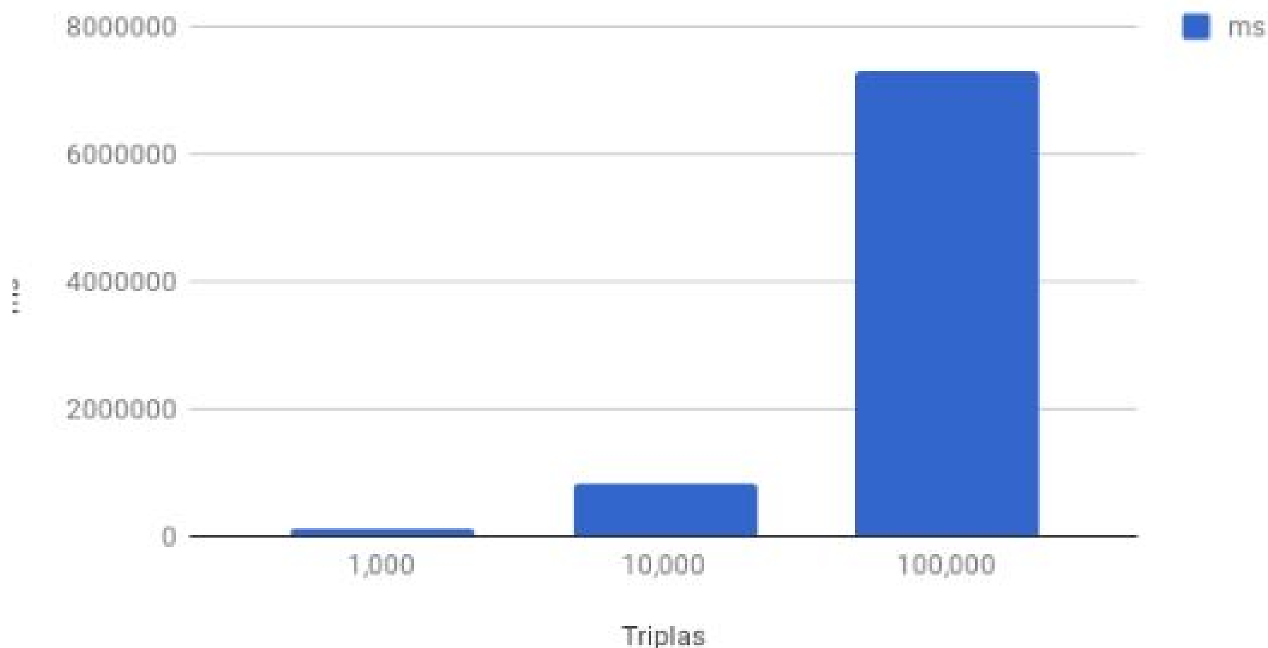


Figura 3 - Gráfico de tempo de inserção em milissegundos

Neste experimento foi utilizado o mesmo *dataset* de 100 mil triplas gerado na seção anterior e foi testado o tempo de consultas diferentes para medir o tempo de resposta quando se utiliza o Neo4j, o Redis no padrão SP*, bem como o Redis no padrão *P*. Para o cálculo do tempo médio médio de resposta foi utilizada uma média simples entre o tempo de 10 execuções de cada consulta. A primeira consulta da Tabela 3 foi escolhida por possuir uma junção, permitindo assim utilizar-se do mapeamento do Neo4j para percorrer os grafos. As outras duas consultas são mais simples, com a segunda não possuindo a especificação de um Objeto, para que seja explorado o padrão SP* e a última com um Objeto especificado para demonstrar o padrão *P*.

Tabela 1 - Tempo Médio de Respostas de Consultas SPARQL

Consulta	Acesso	Tempo Médio de Resposta
SELECT ?p WHERE { ?p CHATS ?x . ?x EATS 'Lasagne' }	Neo4j	18.8ms
SELECT ?r WHERE { ?p EATS ?x }	Redis - Estratégia SP*	31.6ms

SELECT ?r WHERE { ?p EATS 'SOUVLAKI' }	Redis - Estratégia *P*	1.3ms

Com base nos resultados apresentados na Tabela 1, é possível ver que a estratégia de mapeamento *P* do Redis se mostrou extremamente eficiente em relação a estratégia SP*, tendo uma velocidade de resposta 24 vezes mais rápida

4. Considerações Finais e Trabalhos Futuros

Este trabalho apresenta a RDF2Multimodel, uma ferramenta que é capaz de fazer a persistência de triplas RDF em bancos de dados NoSQL de dois modelos distintos (Redis (Chave-valor) e Neo4j (Grafo)), bem como traduzir consultas SPARQL para que possam ser executadas nos mesmos.

O mapeamento proposto para o banco de dados baseado em Chave-Valor se mostrou bastante eficiente tanto em tempo de inserção quanto ao tempo de consulta. Já para o Neo4j obteve-se um desempenho não muito alto, porém esse tempo pode ser otimizado com técnicas de paralelização, distribuição e com escala de processamento.

Os algoritmos, técnicas de mapeamento e considerações apresentados neste trabalho podem ser utilizados em trabalhos futuros para otimizar o mapeamento e velocidade de acesso de ferramentas que lidem com RDF, tanto utilizando-se da API proposta quanto baseando-se em suas características para que possam ser incorporadas em outras ferramentas.

O RDF2Multimodel cumpre os requisitos propostos nesse trabalho, fazendo o mapeamento de Triplas RDF para os bancos Neo4j e Redis assim como a tradução de alguns padrões de consulta SPARQL para que seja possível acessar esses dados.

Como trabalhos futuros é possível apontar:

- Extensão dos padrões de consulta Cypher para que seja possível mapear consultas ainda mais complexas para serem feitas no Neo4j;
- Processamento paralelo distribuído da inserção no Neo4j a fim de aumentar o tempo de inserção, que está em volta de 0.06s por Tripla;

- Implementação de um módulo que faça a tradução do formato RDF na entrada do RDF2Multimodel. Atualmente, só é possível inserir um vetor simulando uma Tripla RDF. Com a implementação desse módulo pode-se estender as capacidades do RDF2Multimodel para receber um arquivo no formato RDF/XML e proceder a sua análise e tradução;
- Implementação de um *Cliente* que facilite a manipulação dos endpoints criados na API;
- Realização de experimentos com outros modelos de dados NoSQL;
- Realização de experimentos utilizando o LUBM, para que seja possível comparar o RDF2Multimodel ao estado da arte de ferramentas que trabalham com RDF.

Referências

GAI, Lei, CHEN, Wei e WANG, Tengjiao., **An Efficient Summary Graph Driven Method for RDF Query Processing**; 2015. Disponível em: <https://arxiv.org/pdf/1510.07749.pdf>

RDF 1.1 CONCEPTS and Abstract Syntax, Disponível em: <https://www.w3.org/TR/rdf11-concepts>. Acessado em 24 Nov. 2016.

FOWLER, Martin e SADALAGE, Pramod J., **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**, 1. ed. Addison-Wesley Professional; 2012, p. 29

RDF GRAPHS, Disponível em: <https://www.w3.org/TR/rdf11-concepts/#section-rdf-graph>. Acessado em 24 Nov. 2016.

GU, HU e HUANG, **Rainbow: A distributed and hierarchical RDF triple store with dynamic scalability**, 2014. Disponível em: <http://ieeexplore.ieee.org/abstract/document/7004274/>

WEIS, KARRAS e BERNSTEIN, **HEXASTORE: Sextuple Indexing for Semantic Web Data Management**. PVLDB, 2008, pp. 1008-1019. Disponível em: <http://www.vldb.org/pvldb/1/1453965.pdf>

HU, WANG, YANG e WO, **ScalaRDF: a Distributed, Elastic and Scalable In-Memory RDF Triple Store**, 2016

APÊNDICE B - CÓDIGO FONTE

config/application.rb

```
require_relative 'boot'

require "rails"
# Pick the frameworks you want:
require "active_model/railtie"
require "active_job/railtie"
require "active_record/railtie"
require "action_controller/railtie"
# require "action_mailer/railtie"
require "action_view/railtie"

# require "action_cable/engine"
# require "sprockets/railtie"
# require "rails/test_unit/railtie"

# Require the gems listed in Gemfile, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(*Rails.groups)

module Rdf2multimodel
  class Application < Rails::Application
    # Settings in config/environments/* take precedence over
those specified here.
    # Application configuration should go into files in
config/initializers
    # -- all .rb files in that directory are automatically
loaded.

    # Only loads a smaller set of middleware suitable for API
only apps.
    # Middleware like session, flash, cookies can be added back
manually.
```

```
    # Skip views, helpers and assets when generating a new
resource.
    config.api_only = true
    config.autoload_paths += %W(#{config.root}/lib)
  end
End
```

Gemfile

```
source 'https://rubygems.org'

git_source(:github) do |repo_name|
  repo_name = "#{repo_name}/#{repo_name}" unless
repo_name.include?("/")
  "https://github.com/#{repo_name}.git"
end


# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.0.1'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use Puma as the app server
gem 'puma', '~> 3.0'

gem 'redis'
gem 'neo4j-core'

group :development, :test do
  gem 'byebug', platform: :mri
  gem 'rspec-rails', '~> 3.6'
  gem 'pry'
end

group :development do
  gem 'listen', '~> 3.0.5'
  gem 'spring'
  gem 'pry'
  gem 'spring-watcher-listen', '~> 2.0.0'
```

```
  gem 'rspec-rails', '~> 3.6'
end
```

config/routes.rb

```
Rails.application.routes.draw do
  # Insert data into Redis (Key/Value)
  post '/save-triple-to-redis', to: 'key_value_adapter#insert'

  # Redis get by key
  get '/recover-object-redis', to: 'key_value_adapter#load_data'

  # Insert data into Neo4j (Graph)
  post '/save-triple-to-neo4j', to: 'graph_adapter#insert'

  # Insert data both to Neo4j (Graph) and Redis (Key/Value)
  post '/insert-data', to: 'multimodel_connector#insert'

  # Parse SPARQL query -- To Cypher
  get '/parse', to: 'decision#decide'
end
```

lib/tasks/neo4j.rake

```
namespace :neo4j do
  task :install => :environment do
    load 'neo4j/rake_tasks/neo4j_server.rake'
  end
End
```

lib/database/session.rb

```
require 'neo4j-core'

module Database
  class Session
    SESSION = Neo4j::Session.open(:server_db,
                                  'http://localhost:7474',
                                  basic_auth: { username:
'neo4j', password: '123qwe' })
```

```
    def self.instance
      SESSION
    end
  end
end
```

app/controllers/decision_controller.rb

```
class DecisionController < ApplicationController
  def decide
    result = if params[:query].split('{').second.count('?') > 2
               GraphParser.new(params[:query]).parse
             else
               KeyValueParser.new(params[:query]).parse
             end

    render json: result, status: :ok
  end
end
```

app/controllers/graph_adapter_controller.rb

```
class GraphaAdapterController < ApplicationController
  def insert
    ::GraphAdapter.new(data).insert_data
  end

  private

  def data
    JSON.parse(params[:data])
  end
end
```

app/controllers/key_value_adapter_controller.rb

```
class KeyValueAdapterController < ApplicationController
```

```
  def insert
    ::KeyValueAdapter.new(data).insert_data

    head :ok
  end

  private

  def data
    JSON.parse(params[:data])
  end
end
```

app/controllers/multimodel_connector_controller.rb

```
class MultimodelConnectorController < ApplicationController
  def insert
    ::GraphAdapter.new(data).insert_data
    ::KeyValueAdapter.new(data).insert_data

    render json: 'Data has been insert with success!'.to_json,
status: :ok
  end

  private

  def data
    JSON.parse(params[:data])
  end
end
```

app/controllers/parser_controller.rb

```
class ParserController < ApplicationController
  def parse
    result = execute(translate_pattern + translate_result +
query_modifier)

    render json: result
  end
```

```
  def translate_result
    translated = ''

    result_clause.first.split.each do |r|
      next if r == 'SELECT'
      translated << "#{r.remove('?')} "
    end

    translated.prepend('RETURN ')
  end

  def translate_pattern
    translated = 'MATCH p=()-'

    translated += query_pattern.first.sub('{',
'[').remove("?p").sub(' . ', ']->()').remove(" ")
  end

  def result_clause
    params[:query].split('WHERE')
  end

  def query_pattern
    result_clause.second.split('}')
  end

  def query_modifier
    query_pattern.last
  end

  def execute(query)
    session.query(query)
  end

  def session
    Database::Session.instance
  end
```

```
end
```

lib/graph_adapter.rb

```
class GraphAdapter
  attr_reader :data

  def initialize(data)
    @data = data
  end

  def insert_data
    data.each do |d|
      session.query(insert_query(d), triples: [d])
    end
  end

  private

  def insert_query(data)
    types = type(data)

    insert_query = <<-QUERY
      UNWIND {triples} as triple
      MERGE (p1:#{types[:subject]} {name:triple[0]})
      MERGE (p2:#{types[:object]} {name:triple[2]})
      MERGE (p1)-[:#{types[:predicate]}]-(p2)
    QUERY
  end

  def type(d)
    { predicate: normalize_predicate(d[1]), subject: 'default',
object: 'default' }
  end

  def normalize_predicate(string)
    string.parameterize.underscore.upcase
  end
```

```
  def session
    Database::Session.instance
  end
end
```

lib/graph_parser.rb

```
class GraphParser < ApplicationController
  def initialize(query)
    @query = query
  end

  def parse
    query_result = execute(translate_pattern + translate_result
+ query_modifier)
    result = []

    query_result.to_a.each_with_object(result) do |res|
      result << res.p[:name]
    end
  end

  def translate_result
    translated = ''

    result_clause.first.split.each do |r|
      next if r == 'SELECT'
      translated << "#{r.remove('?')} "
    end

    translated.prepend(' RETURN ')
  end

  def translate_pattern
    str = result_clause[1].split(' ')
    arr = []

    str.each do |element|
```

```
      arr << element if element.include?('?')
    end

    pattern_hash = { n0: arr[0].remove('?'), n1:
arr[1].remove('?'), n2: find_object }

    build_pattern_translation(pattern_hash[:n0],
pattern_hash[:n1], pattern_hash[:n2])
  end

  def build_pattern_translation(predicate, subject, object)
    subjects = extract_subjects
    "MATCH
(#{predicate})-[:#{subjects[0]}]->(#{subject})-[:#{subjects[1]}]
->(n2 { name: #{object} })"
  end

  def extract_subjects
    subjects = []

    result_clause[1].split('{')[1].split(' ').each do |keyword|
      subjects << keyword unless KEY_TERMS.any? { |term|
keyword.include?(term) }
    end

    subjects
  end

  def result_clause
    @query.split('WHERE')
  end

  def query_modifier
    @query.split('}')[1].to_s
  end

  def find_object
    start_of = result_clause[1].index("'")
```

```
    end_of = result_clause[1].rindex("'")

    result_clause[1][start_of..end_of]
  end

  def execute(query)
    session.query(query)
  end

  def session
    Database::Session.instance
  end

  KEY_TERMS = ['.', '}', '?', 'LIMIT', 'ORDER', 'BY',
"'"].freeze
end
```

lib/key_value_adapter.rb

```
class KeyValueAdapter
  attr_reader :data

  def initialize(data)
    @data = data
  end

  def insert_data
    data.each do |triple|
      save_triple_sp(triple)
      save_triple_p(triple)
    end
  end

  # Save Triple as SP* => O
  def save_triple_sp(triple)
    redis.sadd(key_sp(triple), value_sp(triple))
  end

  def load_data
```

```
    key = "#{data.first.upcase}:#{data.second.upcase}"

    redis.smembers(key)
  end

  def key_sp(triple)
    "#{triple.first.upcase}:#{triple.second.upcase}"
  end

  def value_sp(triple)
    triple[2].upcase
  end

  # Save Triple as *P* => SPO
  def save_triple_p(triple)
    redis.sadd(key_p(triple), value_p(triple))
  end

  def key_p(triple)
    triple.second.upcase
  end

  def value_p(triple)

"#{triple.first.upcase}:#{triple.second.upcase}:#{triple.last.up
case}"
  end

  private

  def redis
    Redis.new
  end
end

lib/key_value_parser.rb
class KeyValueParser < ApplicationController
  def initialize(query)
```

```
    @query = query
  end

  def parse
    result = []

    if query_object == 'NO-OBJ'
      parse_key.each do |key|
        next if key == ':'
        result << sanitize_redis_key(key)[0]
      end
    else
      redis.smembers(query_predicate).each do |member|
        if sanitize_redis_key(member).last == query_object
          result << sanitize_redis_key(member).first
        end
      end
    end

    result
  end

  private

  def parse_key
    redis.keys.each_with_object([]) do |key, keys|
      splitted_key = sanitize_redis_key(key)

      if splitted_key[1] == query_predicate
        keys << "#{splitted_key[0]}:#{splitted_key[1]}"
      end
    end
  end

  def query_predicate
    @query.split('{')[1].split(' ').each do |element|
      return element.humanize.upcase unless
element.include?('?')
```

```
    end
  end

  def query_object
    result_clause = @query.split('WHERE')
    start_of = result_clause[1].index("'")
    end_of = result_clause[1].rindex("'")

    result_clause[1][start_of..end_of].tr("'", '')
  rescue
    return 'NO-OBJ'
  end

  def sanitize_redis_key(key)
    key.split(':')
  end

  def redis
    Redis.new
  end
end
```