UNIVERSIDADE FEDERAL DE SANTA CATARINA

Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL do Tipo Grafos

Salomão Rodrigues Jacinto

Florianópolis
2019

Salomão Rodrigues Jacinto

Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL do Tipo Grafos

Trabalho de Conclusão de Curso submetido como parte dos requisitos para obtenção do Título de Bacharel do Curso de Ciências da Computação, da Universidade Federal de Santa Catarina.

Orientador: Ronaldo dos Santos Mello.

Florianópolis, Novembro de 2019

Salomão Rodrigues Jacinto

Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL do Tipo Grafos

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de Bacharel em Ciências da Computação, e aprovado em sua forma final pelo Curso de Ciências da Computação da Universidade Federal de Santa Catarina.

Prof. Ronaldo dos Santos Mello
Orientador
Universidade Federal de Santa Catarina

Profª. Patrícia Vilain
Membro da Banca
Universidade Federal de Santa Catarina

Prof. Renato Fileto
Membro da Banca
Universidade Federal de Santa Catarina

Florianópolis, Novembro de 2019

Dedico este trabalho a todos que estiveram ao meu lado e, de alguma forma, antes, durante e continuamente me ajudam nesta e em futuras etapas em minha vida.

Agradecimentos

Agradeço principalmente à minha família, ao meus pais e minha irmã, por sempre me proporcionarem a melhor educação e pela liberdade em deixar eu escolher meu próprio caminho.

A todos os meus amigos, presentes em toda a graduação nos momentos de seriedade e de diversão.

A todo o corpo docente do CTC-UFSC, pelo aprendizado dentro e fora da sala de aula. Em especial ao meu professor orientador Ronaldo, pelo incentivo durante o desenvolvimento deste trabalho, por ter compreendido a minha forma de trabalhar e por estar sempre à disposição quando necessário. Aos membros da banca, professora Patrícia e professor Renato, por aceitarem o convite.

“A learning experience is one of those things that says, ’You know that thing you just did? Don’t do that.”

(Douglas Adams)

Resumo

Atualmente, uma grande quantidade de dados heterogêneos são gerados e consumidos em uma escala sem precedentes, o que motivou a criação de sistemas gerenciadores de bancos de dados que levam o nome de NoSQL. Esses bancos de dados possuem capacidade para lidar com um grande volume de dados e não necessariamente possuem um esquema implícito como os bancos de dados relacionais. Mesmo assim, o conhecimento de como os dados estão sendo armazenados estruturalmente é de suma importância para diversas tarefas, como integração ou análise de dados. Existem trabalhos na literatura que extraem o esquema de dados semiestruturados de forma geral e trabalhos que propõem um modelo teórico de esquema para bancos de dados do tipo grafo. Como diferencial, o presente trabalho visa o desenvolvimento de uma ferramenta para extração de um esquema de um banco de dados NoSQL do tipo grafo para um formato do tipo JSON *Schema*, assim como a elaboração de um documento contendo os estudos e testes realizados sobre a ferramenta implementada. Avaliações experimentais demonstram que a ferramenta produz uma representação adequada de um esquema com uma complexidade linear.

Palavras-chave: NoSQL, Grafos, JSON, Esquema, Extração de esquema.

Abstract

Currently, a large volume of heterogeneous data is generated and consumed on the network in an unprecendented scale which led to the creation of database models named NoSQL. These databases are capable of handling a large volume of data and are *schemaless*, in other words, they do not have an implicit schema such as relational databases. But the knowledge of how data is structurally stored is of great importance for the development of an application or an data analysis. There are works in the literature that extract the schema from a semistructured data in general and works that propose a theoretical schema model for graph databases. Diffferent from them, this work aims to develop a tool to extract a schema from an existing graph NoSQL database to a JSON Schema format, as well as the elaboration of a document containing the studies and tests carried out on top of the implemented application. Experimental evaluations show that the proposed tool generates a suitable schema representation with a linear complexity.

Keywords: NoSQL, Graph, JSON, Schema, Schema extraction.

Lista de ilustrações

Figura 1 – Grafo exemplo com localidades de Florianópolis 35
Figura 2 – Possível resultado para busca em profundidade a partir do vértice Centro 35
Figura 3 – Possível resultado para busca em largura a partir do vértice Centro . . 36
Figura 4 – Nomenclatura Neo4j [Neo4j 2018] 36
Figura 5 – Especificação de um objeto JSON 38
Figura 6 – Especificação de um valor JSON 38
Figura 7 – Exemplo de especificação em JSON *Schema* 39
Figura 8 – Exemplo do uso das chaves *"definitions"* e *"$ref"* 40
Figura 9 – Exemplo do uso da chave *"allOf"* para estruturar uma herança 40
Figura 10 – Exemplo do tipo *{"type": "array"}* 41
Figura 11 – Organização das funções da ferramenta 50
Figura 12 – Visão geral da ferramenta 52
Figura 13 – Dicionário de vértices 53
Figura 14 – Dicionário de relacionamentos 53
Figura 15 – Grafo exemplo 54
Figura 16 – Exemplo para um vértice de rótulo ator 68
Figura 17 – Esquema Airbnb no BD Neo4j 70
Figura 18 – JSON *Schema* para o relacionamento *HOSTS* 70
Figura 19 – JSON *Schema* para o vértice *Host* 71
Figura 20 – Esquema IMDB no BD Neo4j 72
Figura 21 – JSON *Schema* para o vértice *Person* 73
Figura 22 – JSON *Schema* 1/2 para o vértice *Actor* 74
Figura 23 – JSON *Schema* 2/2 para o vértice *Actor* 75
Figura 24 – Tempo de processamento X Tamanho do grafo 78

Lista de tabelas

Tabela 1 – Mapeamento de tipos Neo4j para Python 66
Tabela 2 – Mapeamento de tipos Python para JSON *Schema* 66
Tabela 3 – Tamanhos das Bases de Dados 76
Tabela 4 – Tempos de Processamento 1/3 77
Tabela 5 – Tempos de Processamento 2/3 77
Tabela 6 – Média de Tempo de Processamento 77

Lista de Algoritmos

1 Agrupamento de Vértices 54

2 Contabilizar propriedades 56

3 Popular relacionamentos 57

4 Agrupamento de relacionamentos 58

5 Extração de vértices com rótulo único 60

6 Encontrar intersecções 64

7 Processar intersecções 64

8 Intersectar propriedades 65

Lista de abreviaturas e siglas

ACID	*Atomicity, Consistency, Isolation, Durability*
API	*Application Programming Interface*
BASE	*Basically Available, Soft State, Eventual Consistency*
BD	Banco de Dados
DDL	*Data Definition Language*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
NoSQL	*Not SQL ou Not Only SQL*
REST	*Representational State Transfer*
RF	Requisito Funcional
RNF	Requisito Não-Funcional
SGBD	Sistema de Gerenciamento de Banco e Dados
TCC	Trabalho de Conclusão de Curso
UFSC	Universidade Federal de Santa Catarina

Sumário

1	**INTRODUÇÃO**	**27**
1.1	**Justificativa**	**28**
1.2	**Objetivos**	**29**
1.2.1	Objetivo Geral	29
1.2.2	Objetivos Específicos	29
1.3	**Metodologia**	**29**
1.4	**Estrutura do Trabalho**	**30**
2	**FUNDAMENTAÇÃO TEÓRICA**	**31**
2.1	***Big Data***	**31**
2.2	**Bancos de Dados NoSQL**	**32**
2.2.1	Modelo Chave-Valor	32
2.2.2	Modelo Colunar	32
2.2.3	Modelo de Documento	33
2.2.4	Modelo de Grafo	33
2.3	**Grafos**	**34**
2.4	**Neo4j**	**36**
2.5	**JSON *Schema***	**37**
3	**TRABALHOS RELACIONADOS**	**43**
3.1	**Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL Orientados a Documentos**	**44**
3.2	***Logical Design of Graph Databases from an Entity-Relationship Conceptual Model***	**45**
3.3	***Modeling Graph Database Schema***	**46**
3.4	***Schema Validation and Evolution for Graph Databases***	**46**
3.5	**Esquemas Neo4j e OrientDB**	**47**
4	**FERRAMENTA PARA EXTRAÇÃO DE ESQUEMAS DE BANCOS DE DADOS NOSQL DE GRAFO**	**49**
4.1	**Projeto**	**49**
4.2	**Processo de Extração de Esquemas**	**51**
4.2.1	Agrupamento	53
4.2.2	Extração	58
4.2.3	Mapeamento	66

5 AVALIAÇÃO DA FERRAMENTA 69
5.1 Avaliação Qualitativa 69
5.1.1 Airbnb 69
5.1.2 IMDB 72
5.2 Análise de Tempo de Processamento 76

6 CONCLUSÃO 79

REFERÊNCIAS 81

APÊNDICES 83

APÊNDICE A – CÓDIGO 85

APÊNDICE B – ESQUEMA AIRBNB 97

APÊNDICE C – ESQUEMA IMDB 107

APÊNDICE D – ARTIGO FORMATO SBC 117

1 Introdução

Com o crescimento contínuo da Internet e de aplicações lidando com um grande volume de dados, se tornou necessário desenvolver Bancos de Dados (BDs) que fossem capazes de armazenar e processar esses dados de forma efetiva, ou seja, com um alto desempenho em termos de operações de leitura e escrita [Han et al. 2011].

BDs tradicionais (BDs relacionais) oferecem mecanismos para gerenciar dados com forte consistência, tendo alcançado um nível de estabilidade e confiabilidade ao longo de muitos anos de desenvolvimento. Entretanto, dados gerados por redes sociais ou redes de sensores, por exemplo, são naturalmente volumosos e heterogêneos, além de não apresentarem, em geral, um esquema associado, o que faz com que não consigam ser gerenciados de forma eficiente por BDs relacionais [Gessert et al. 2017].

Logo, escalar verticalmente (estendendo as capacidades computacionais de um servidor de dados, por exemplo) não é mais viável econômica e fisicamente. Assim, para solucionar esses problemas percebeu-se a oportunidade de escalar horizontalmente por intermédio de BDs distribuídos, isto é, incluindo mais nodos servidores de dados à infraestrutura computacional. Porém, como BDs tradicionais não foram projetados para funcionar de forma distribuída, não se adequando bem ao gerenciamento de particionamento e fragmentação de dados sob demanda, surgiu uma nova família de BDs denominada *NoSQL*, um acrônimo para *Not only SQL* [Han et al. 2011].

Os BDs NoSQL surgem, então, para suprir essa necessidade de desempenho de leitura e escrita sobre uma grande massa de dados de forma escalável, concorrente e eficiente. Para atingir esses objetivos, esses BDs dispensam as tradicionais propriedades ACID (*Atomicity, Consistency, Isolation, Durability*) dos BDs relacionais e adotam as propriedades BASE (*Basically Available, Soft State, Eventual Consistency*), relaxando a consistência para garantir alta disponibilidade [Costa 2017].

Dentre uma das famílias de BDs NoSQL encontra-se BDs do tipo grafo. Em contraste com BDs relacionais, BDs do tipo grafo são especializados em gerenciar, de modo eficiente, dados extremamente conectados, pois o custo de operações como *joins* recursivos podem ser substituídos por operações de travessias no grafo. Os principais BDs do tipo grafo são baseados em grafos direcionados com múltiplos relacionamentos e propriedades. Neste caso, vértices e arestas são objetos com propriedades do tipo chave-valor [Hecht e Jablonski 2011].

Em BDs relacionais é necessário ter uma estrutura de tabelas pré-definidas antes que se possam armazenar os dados, também denominada de *esquema*. Já em BDs NoSQL, não é preciso existir uma estrutura prévia para começar a armazenar os dados, sendo possível

que dados com representações diferentes possam coexistir no BD. Essa característica dos BDs NoSQL é chamada *schemaless* [Sadalage e Fowler 2012].

A ausência de controle de esquemas dos dados permite uma flexibilidade de representação dos dados, evitando, por exemplo, que existam diversas colunas sem valor, como no caso de BDs relacionais que possuem esquemas rígidos. Essa característica trouxe grande popularidade aos BDs NoSQL e, apesar de ser um ponto positivo, muitas vezes a estrutura dos dados pode estar implicitamente agregada ao código da aplicação, dificultando o crescimento e manutenção da mesma [Sadalage e Fowler 2012].

Os esquemas que BDs do tipo grafo providenciam são tipicamente *descritivos*, ou seja, apenas refletem os dados que estão ali inseridos, mas não criam nenhum tipo de restrição. Este esquema pode ser alterado simplesmente mudando os dados, isto é, ao inserir um vértice que seja completamente diferente de todos que já existem na base, o esquema dos dados do BD muda naturalmente. Essa flexibilidade inicialmente é percebida como uma ótima característica, principalmente nos estágios iniciais de desenvolvimento, porém na medida em que a aplicação vai maturando, essas mudanças precisam ser feitas com maior cuidado, criando uma demanda por restrições de esquemas, melhor dizendo, um esquema mais *prescritivo* [Bonifati et al. 2019].

1.1 Justificativa

O conhecimento de como os dados estão estruturados e armazenados é de suma importância para o desenvolvimento e manutenção de uma aplicação. Contudo, quando isto está implícito no código da aplicação pode se tornar um grande problema, porquanto dificulta a integração entre múltiplos aplicativos, em um mundo onde a necessidade desse conhecimento é essencial [Sadalage e Fowler 2012].

Trabalhos já desenvolvidos no Grupo de BD da UFSC (GBD UFSC) realizaram a extração de esquemas de BDs NoSQL do tipo *Documento* [Costa 2017], do tipo *Colunar* [Defreyn 2019] e *Orientado a Agregados* [Frozza e Mello 2018] para o formato canônico JSON *Schema*[1]. Esse formato foi escolhido devido à grande popularidade do padrão JSON para representação e intercâmbio de dados.

A intenção deste trabalho é contribuir com uma pesquisa em nível de pós-graduação, que visa extrair esquemas de todos os modelos de dados NoSQL para uma posterior integração de dados através da extração de esquemas de BDs do tipo grafo.

Trabalhos relacionados na literatura propõem um modelo teórico de esquema para bancos de dados do tipo grafo [Roy-Hubara et al. 2017] ou extraem algum tipo de esquema a partir de dados semiestruturados de forma geral [Nestorov, Abiteboul e Motwani 1998].

[1] http://json-schema.org/

No entanto, nenhum deles extrai o esquema especificamente de um BD do tipo grafo.

Assim sendo, a ferramenta proposta por este TCC tem como objetivo extrair o esquema de um BD NoSQL do tipo grafo, em particular o SGBD Neo4j[2], para um formato JSON *Schema*, com o intuito de auxiliar no desenvolvimento de aplicações, integração entre serviços e validação de dados. O Neo4j foi escolhido por ser o SGBD do tipo grafo mais utilizado no mundo, segundo o site *db-engines*[3].

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral do presente Trabalho de Conclusão de Curso consiste no desenvolvimento de uma ferramenta para extração de esquemas de um BD NoSQL do tipo grafo para um formato JSON *Schema*.

1.2.2 Objetivos Específicos

Os objetivos específicos são os seguintes:

- Mapear o conjunto de tipos e atributos dos vértices e arestas de um BD do tipo grafo para uma versão reduzida e refinada, utilizando o SGBD Neo4j e operações de conjuntos;
- Extrair o mapeamento realizado para um esquema no padrão JSON *Schema*;
- Avaliar a ferramenta proposta através de estudos de caso e testes de desempenho;
- Disponibilizar todo código produzido de forma livre.

1.3 Metodologia

A metodologia de pesquisa e desenvolvimento deste trabalho dividiu-se em três etapas:

- **Etapa 1:** consistiu em uma pesquisa bibliográfica dos assuntos necessários para desenvolver a aplicação, investigando o estado da arte em *"esquemas de BDs NoSQL do tipo grafo"*. A pesquisa foi realizada em artigos, documentação das ferramentas, *surveys* e outras publicações relacionadas ao assunto.

[2] https://neo4j.com/
[3] https://db-engines.com/

- **Etapa 2:** Análise de requisitos e desenvolvimento da aplicação para extração de esquemas de BDs do tipo grafos. Também foram consideradas pesquisas acerca das tecnologias a serem utilizadas e dos algoritmos a serem usados e/ou criados.

- **Etapa 3:** Documentação, teste e avaliação da ferramenta desenvolvida para análise dos resultados, verificando a qualidade dos esquemas extraídos de grandes *datasets*, bem como simular cenários de serviços como IMDB[4] e Airbnb[5].

1.4 Estrutura do Trabalho

O presente trabalho se divide em 6 capítulos. Este primeiro apresenta a introdução do trabalho. No capítulo 2 são apresentados os conceitos básicos necessários para o entendimento e desenvolvimento da ferramenta proposta. No capítulo 3 é realizada uma análise dos trabalhos relacionados. O capítulo 4 traz o processo do desenvolvimento da ferramenta e dos algoritmos para extração do esquema, assim como as decisões tecnológicas e a arquitetura escolhida. Os resultados produzidos são apresentados no capítulo 5, através de estudos de caso e medidas de desempenho que avaliam a qualidade do esquema. Por fim, no capítulo 6 é reforçada a contribuição deste trabalho e os possíveis trabalhos futuros.

[4] https://www.imdb.com/
[5] https://www.airbnb.com

2 Fundamentação Teórica

Este capítulo apresenta os conceitos utilizados no entendimento e desenvolvimento do trabalho, quais sejam, *Big Data*, BDs NoSQL e seus diferentes modelos de dados, além da teoria de grafos, o BD Neo4J e o formato JSON *Schema*.

2.1 *Big Data*

Big data é um termo usado para se referir a um grande volume de dados, tanto estruturados quanto não estruturados, de difícil processamento, gerenciamento e armazenamento [Han et al. 2011]. O fenômeno *Big Data* também é frequentemente descrito por meio dos chamados 5 Vs: Volume, Velocidade, Variedade, Veracidade e Valor [Marr 2014].

O *volume* se refere a vasta quantidade de dados gerada por pessoas ou máquinas, como e-mails, mensagens e fotos que são produzidos e compartilhados. Se for comparado todo o volume de dados gerado desde o início da Internet até o ano de 2008, a mesma quantidade é gerada a todo minuto hoje em dia. Isso torna os conjuntos atuais de dados grandes e de difícil armazenamento e análise. Com a tecnologia *Big Data* é possível armazenar e manipular esses conjuntos de dados com a ajuda de sistemas distribuídos e métodos de processamento paralelo massivo.

Por sua vez, a *velocidade* diz respeito a velocidade com que novos dados são gerados, transmitidos e utilizados por diversas fontes. A tecnologia *Big Data* permite analisar esses dados à medida que são gerados, ajudando na tomada de decisões.

Já a *variedade*, consiste nos diferentes tipos de dados utilizados atualmente. No passado só eram considerados dados estruturados que se encaixassem no formato relacional, mas, atualmente, com a tecnologia da *Big Data* é possível guardar dados de sensores, gravações de áudio e vídeo, além de manipular todos de forma conjunta.

Quanto à *veracidade*, trata-se da acurácia dos dados, que são menos controláveis e podem ser irrelevantes. É necessário que se saiba como trabalhar com esses dados através de filtros que garantam dados com maior qualidade.

Por último, mas não menos importante, o *valor* indica que não adianta nada ter um volume massivo de dados se não for possível gerar informação e conhecimento útil deles.

2.2 Bancos de Dados NoSQL

Os BDs NoSQL são uma família de gerenciadores de dados que não seguem o modelo de dados relacional [Sadalage e Fowler 2012]. Eles visam melhor desempenho de acesso, armazenamento de um grande volume de dados, escalabilidade e disponibilidade, tudo isso ao custo de abdicar as tradicionais propriedades ACID (*Atomicity, Consistency, Isolation, Durability*) [Han et al. 2011].

Em vez de as propriedades ACID, os BDs NoSQL se baseiam nas propriedades BASE. Um sistema BASE se preocupa em garantir alta disponibilidade (*Basically Available*) e não precisa estar continuamente consistente (*Soft State*), uma vez que eventualmente um sistema estará no referido estado em momento futuro (*Eventually Consistent*). Destaca-se que geralmente as atualizações são propagadas para todos os nodos de um BD distribuídos conforme o dado é requisitado por este nodo, evitando, assim, um baixo desempenho com operações de atualização e necessidade de atualização síncrona [Costa 2017].

Conforme comentando anteriormente, BDs NoSQL são uma família de gerenciadores de dados, não existindo apenas um único modelo de dados utilizado. Eles são organizados em quatro categorias, de acordo com o modelo de dados adotado, podendo ser chave-valor, colunar, documento e grafo [Hecht e Jablonski 2011], que serão detalhadas nas seções seguintes.

2.2.1 Modelo Chave-Valor

BDs do tipo chave-valor são similares a mapas e dicionários, onde o dado é endereçado por uma chave única e não é interpretável pelo sistema. Como não há controle sobre o conteúdo dos dados, questões como relacionamentos e restrições de integridade devem ser tratadas na lógica da aplicação.

Além de suportar o armazenamento em massa, os BDs chave-valor oferecem um alto desempenho em operações de leitura e escrita concorrentes, estando entre os mais populares o Voldemort, DynamoDB e Redis [Hecht e Jablonski 2011].

2.2.2 Modelo Colunar

Os BDs colunares, também chamados de BDs de famílias de colunas, oferecem uma arquitetura altamente escalável. Possuem uma similaridade com BDs chave-valor, porém com um modelo de representação mais complexo. Diferentemente dos BDs relacionais, que são orientados a tuplas, os BDs colunares são orientados a atributos (colunas).

Dados em um BD colunar são acessados por chaves que mapeiam para grupos de valores mantidas em famílias de colunas. Uma família de colunas, como o próprio nome indica, mantém um conjunto de atributos, não sendo obrigatório todas as chaves

conservarem o mesmo conjunto fixo de atributos. Logo, cada registro de dados pode ter seu próprio conjunto de colunas, oferecendo flexibilidade de representação para registros armazenados no BD. Entre os BDs colunares mais conhecidos estão o Cassandra e o HBase [Gessert et al. 2017].

2.2.3 Modelo de Documento

BDs do tipo documento encapsulam pares chave-valor no formato JSON[1] (documento JSON), ou formatos similares a JSON, como XML. Dentro de cada documento as chaves de cada atributo (seus nomes) devem ser únicas, e cada documento possui uma chave especial que também é única no conjunto de documentos.

Diferente do modelo chave-valor, aqui a estrutura dos documentos é conhecida pelo SGBD, ou seja, dados podem ser manipulados pelo SGBD, com a vantagem de suportar tipos de dados.

O número de atributos de um documento não é limitado e novos campos podem ser adicionados dinamicamente a um documento. Um documento pode conter subdocumentos ou mesmo listas de subdocumentos, gerando uma estrutura na forma de árvore. Os BDs populares nessa categoria são o CouchDB e o MongoDB [Costa 2017].

Um ponto em comum entre BDs chave-valor, documento e colunar é que todos podem manter dados não normalizados, isto é, dados aninhados dentro de outros dados mais complexos com intuito de facilitar o acesso a dados relacionados sem precisar executar operações de junção. Entretanto, isso pode gerar alta redundância de dados e, como eles não controlam integridade referencial, não há garantias de que os relacionamentos estejam corretos [Hecht e Jablonski 2011].

2.2.4 Modelo de Grafo

Os BDs do tipo grafo podem ser definidos como estruturas onde o esquema e suas instâncias são modeladas como grafos (ver Seção 2.3) ou generalizações de grafos, sendo a manipulação de dados expressa em operações que percorrem grafos. Estão entre os principais BDs deste tipo o Neo4j, GraphDB e FlockDB.

Convém mencionar que esse modelo surgiu juntamente com BDs do tipo orientado a objetos nos anos 80, mas foi deixado de lado com o surgimento de BDs semiestruturados [Angles e Gutierrez 2008]. Contudo, recentemente, a necessidade em gerenciar *Big Data* na forma de grafos fez com que BDs do tipo grafo se tornassem relevantes novamente.

BDs nesta categoria geralmente mantêm grafos direcionados e rotulados, melhor dizendo, dados são organizados em vértices e arestas (relacionamentos) rotuladas. Além

[1] http://json.org/

disso, tanto os vértices quanto as arestas podem ter propriedadas (atributos) [Angles e Gutierrez 2008]. Ainda, a manipulação de dados é realizada por meio de típicas operações sobre grafos, como navegação através de arestas e noções de vizinhança, subgrafos e conectividade, operações essas que serão descritas na seção seguinte (Seção 2.3).

Os BDs do tipo grafo se aplicam a áreas onde a informação sobre conexões entre dados e sua topologia é bastante importante ou tão importante quanto os dados em si [Angles e Gutierrez 2008].

2.3 Grafos

Algumas definições acerca de uma estrutura de dados em grafo são necessárias para o desenvolvimento deste trabalho. As definições a seguir foram retiradas do livro eletrônico sobre Teoria de Grafos do Professor Antônio Carlos Mariani, da Universidade Federal de Santa Catarina [Mariani 2018].

Um *Grafo* pode ser definido por dois conjuntos V e A, onde V são os vértices do grafo e A um conjunto de pares pertencentes a V que definem suas arestas. Este grafo também pode ser um dígrafo, que consiste em um grafo orientado onde suas arestas possuem direção.

Outros conceitos associados a uma estrutura de grafo são os seguintes:

- *Adjacência*: dois vértices são ditos adjacentes (vizinhos) se existe uma aresta entre eles.
- *Grau*: o grau de um vértice é dado pelo número de arestas que lhe são incidentes. Um grafo orientado pode ser dividido em grau de emissão e grau de recepção.
- *Cadeia*: é uma sequência qualquer de arestas adjacentes que ligam dois vértices. O conceito de cadeia vale também para grafos orientados, bastando que se ignore o sentido da orientação. Uma cadeia é dita *elementar* se não passa duas vezes pelo mesmo vértice. Por outro lado, é *simples* se não passa duas vezes pela mesma aresta.
- *Caminho*: é uma cadeia na qual todos os arcos possuem a mesma orientação, sendo aplicada, portanto, somente a grafos orientados.
- *Grafo conexo*: quando há pelo menos uma cadeia ligando cada par de vértices deste grafo.

Buscas (ou percorrimentos) podem ser realizadas sobre um grafo. Dentre as principais formas de busca estão a *busca em profundidade* e a *busca em largura*, sendo que ambas partem de um vértice e vão construindo uma árvore de busca. Em uma busca

em largura, cada iteração varre todos os nodos adjacentes, e depois os adjacentes dos adjacentes, e assim sucessivamente. Já em uma busca em profundidade, procura-se uma cadeia o máximo que for possível antes de realizar um *backtracking*.

As Figuras 2 e 3 representam possíveis resultados para uma busca em profundidade e uma busca em largura, respectivamente, sobre o grafo da Figura 1.

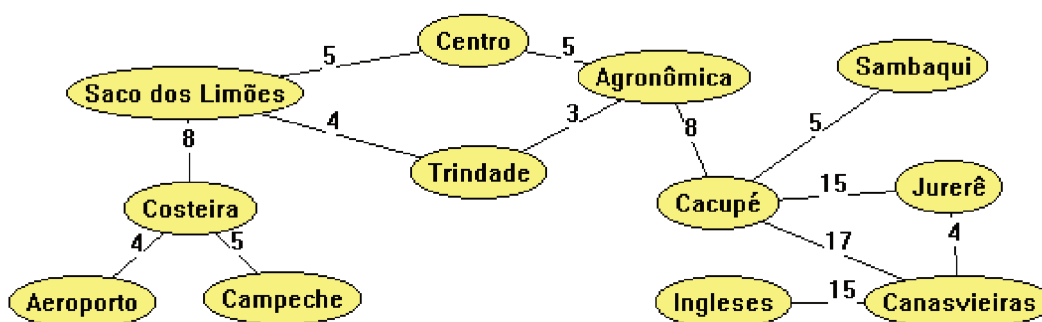

Figura 1 – Grafo exemplo com localidades de Florianópolis

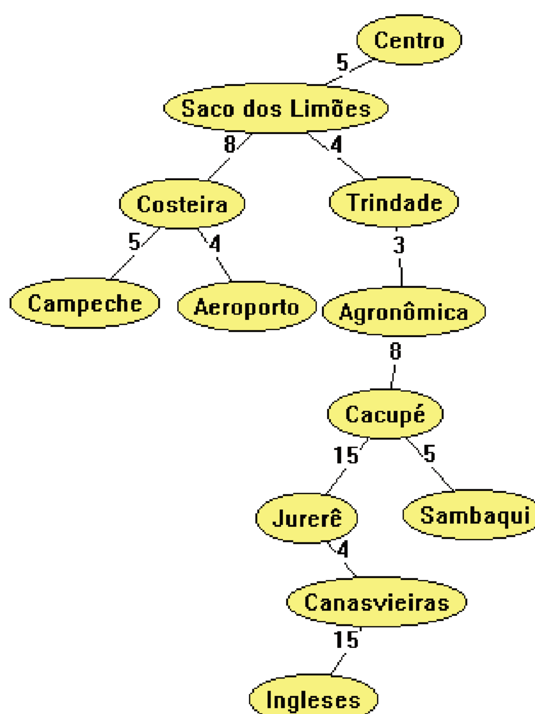

Figura 2 – Possível resultado para busca em profundidade a partir do vértice Centro

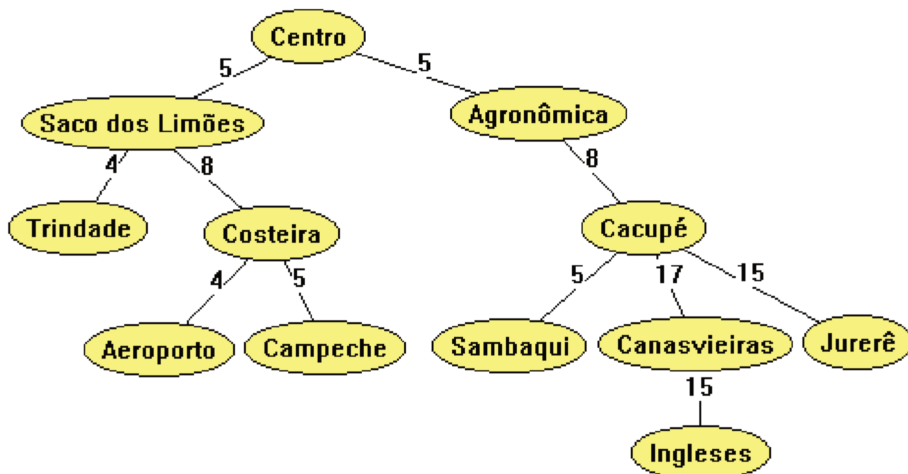

Figura 3 – Possível resultado para busca em largura a partir do vértice Centro

2.4 Neo4j

O BD NoSQL do tipo grafo Neo4J foi o escolhido para o desenvolvimento da ferramenta proposta neste trabalho. A escolha se justifica pelo fato deste BD ser altualmente um dos principais representantes desta categoria, bem como por possuir uma vasta gama de artigos e documentação. Além disso, o Neo4J apresenta um modelo de dados mais robusto e flexível que outros BDs similares. A nomenclatura do Neo4j é descrita na Figura 4.

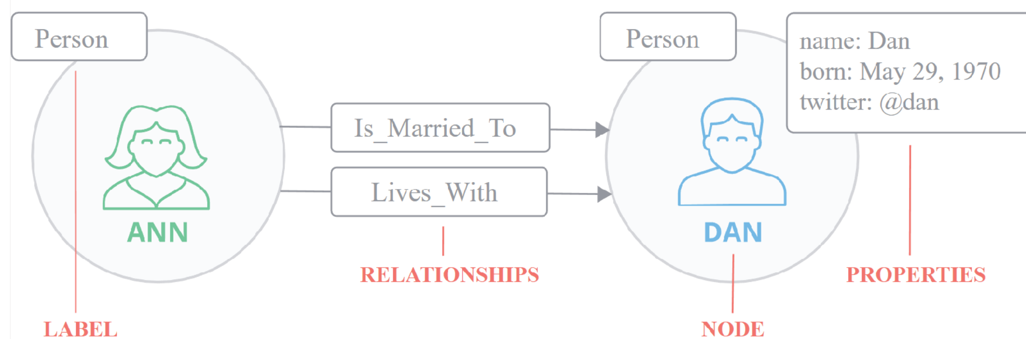

Figura 4 – Nomenclatura Neo4j [Neo4j 2018]

Os conceitos principais do modelo de dados do Neo4j são os seguintes:

- *Vértices*: consiste no elemento principal do modelo de dados do Neo4j. Eles podem ser conectados por meio de *relacionamentos*, ter uma ou mais *propriedades* (atributos guardados como um par chave-valor) e ter um ou múltiplos *rótulos* (identificador do tipo do vértice).
- *Relacionamentos*: conectam dois vértices e são orientados, ou seja, possuem direção.

Eles podem apresentar uma ou mais *propriedades*, e também possuem um *tipo*, que tem a mesma função dos rótulos dos vértices, porém no caso de relacionamentos são limitados a um único tipo.

- *Propriedades*: são valores nomeados, em outras palavras, pares chave-valor. A chave sempre é uma *String* e seu valor pode ser um número (*Integer* ou *Float*), uma *String*, um *Boolean*, um tipo espacial *Point*, uma gama de tipos temporais (*Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* e *Duration*). Listas e mapas de tipos simples também são permitidos.

Ainda, o Neo4j utiliza um armazenamento nativo em grafo. É também chamado de *index-free adjacency*, onde cada vértice aponta fisicamente para sua localização na memória, melhorando o desempenho de acesso [Neo4j 2018].

A linguagem de consulta do Neo4j é a *Cypher*, a qual é inspirada na linguagem SQL e adota o conceito de combinação de padrões da linguagem SPARQL[2]. Cypher descreve os vértices, os relacionamentos e as propriedades como se formassem um desenho utilizando caracteres ASCII, tornando, assim, as consultas mais fáceis de ler e entender [Cypher, The Graph Query Language 2018].

2.5 JSON *Schema*

Antes de apresentar JSON *Schema* é necessário se ter o conhecimento do que é JSON. JSON (*JavaScript Object Notation*) é um formato leve para troca de dados, sendo de fácil entendimento e leitura por humanos e também de simples manipulação por máquinas. Trata-se de um formato de texto completamente independente de linguagem, mas que usa convenções familiares com diversas linguagens de programação como C, C++, C#, *Java*, *JavaScript*, *Perl* e *Python*. Essas propriedades fazem com que JSON seja ideal para troca de informações[3].

JSON é projetado sobre duas estruturas universais, ou seja, estruturas suportadas por todas as linguagens de programação modernas. São elas:

- Uma *coleção de pares chave-valor*, também conhecida nas linguagens de programação como *record*, *struct*, *hash table* ou *dictionary*.
- Uma *lista ordenada de valores*, também denominada nas linguagens de programação como *array*, *vector* ou *list*.

Com base nessas estruturas, uma representação no formato JSON apresenta os seguintes conceitos básicos, conforme ilustram as Figuras 5 e 6:

[2] https://www.w3.org/TR/sparql11-query/
[3] https://www.json.org/

- Um *objeto (Object)*, que é uma sequência não ordenada de chave-valor. É definido entre chaves ('{', '}'), sendo os pares chave-valor internos separados por vírgula.

- Um *valor*, que pode ser do tipo *String*, *Number*, *Object*, *Array*, *Boolean* ou *Null*.

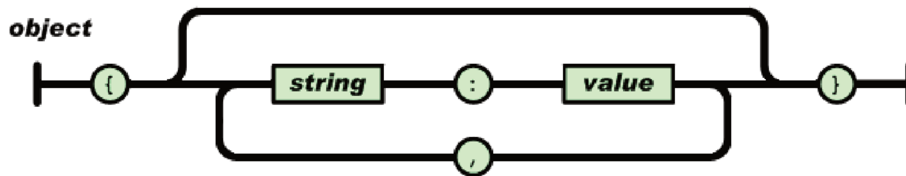

Figura 5 – Especificação de um objeto JSON

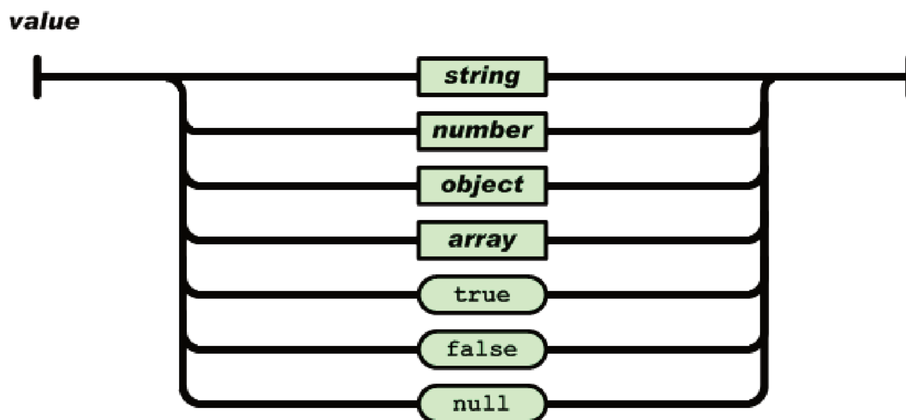

Figura 6 – Especificação de um valor JSON

Atualmente, JSON tem um papel chave no desenvolvimento de aplicações. Um programa que execute funções de forma remota precisa estabelecer um protocolo preciso de comunicação para receber dados e respondê-los, o que é chamado de API (*Application Programming Interface*). Como JSON é uma linguagem facilmente compreendida por humanos e máquinas, acabou se tornando o formato mais popular para se enviar pedidos a APIs por intermédio do protocolo HTTP [Pezoa et al. 2016].

Por exemplo, considere a requisição para um *web service* que deseja saber o clima de um determinado local. Uma chamada hipotética a essa API conteria os seguintes dados em formato JSON:

```
{"Country": "Chile", "City": "Santiago"}
```

A requisição está solicitando o clima para o país *Chile*, na cidade de *Santiago*. Assim sendo, a API poderia responder com o seguinte JSON:

```
{"timestamp": "14/10/2015 11:59:07",
 "temperature": 25, "Country": "Chile",
 "City": "Santiago", "description": "Sunny"}
```

A API então responde indicando que a temperatura para aquela data é de 25 graus Celsius e que o dia está ensolarado. O exemplo mostra a simplicidade e legibilidade do formato JSON.

Em diversos cenários deste tipo é possível se beneficiar de uma forma declarativa de especificar um esquema, isto é, uma forma de definir como a requisição deve ser formatada para evitar chamadas mal formadas para a API. Desta maneira, utiliza-se o JSON *Schema* para validar expressões no formato JSON [Pezoa et al. 2016].

JSON *Schema* é uma linguagem que permite ao usuário restringir a estrutura de documentos JSON e prover um *framework* com a finalidade de verificar a integridade das requisições de acordo com a API.

```
{
    "type": "object",
    "properties": {
        "Country": {"type": "string"},
        "City": {"type": "string"},
    },
    "required": ["Country", "City"]
    "additionalProperties": false
}
```

Figura 7 – Exemplo de especificação em JSON *Schema*

A Figura 7 mostra como seria um JSON *Schema* para averiguar a requisição de clima. Ela indica que a requisição deve conter um objeto com as chaves *"Country"* e *"City"*, do tipo *String*, pois estão dentro da chave *"properties"*. Além disso, ambas são obrigatórias, visto que estão compreendidas na lista da chave *"required"*. Também indica que não é possível adicionar propriedades adicionais.

Na recomendação JSON *Schema*, além das chaves *"properties"* e *"required"*, também são utilizadas neste trabalho as chaves *"definitions"* e *"allOf"*, assim como o tipo *{"type": "array"}*, que possuem algumas construções diferentes.

A chave *"definitions"* permite o reuso de uma estrutura complexa dentro do esquema. Ela pode ser utilizada aliada à chave *"$ref"*, que permite referenciar esquemas ou estruturas presentes em outros arquivos. Quando utilizado com o sufixo #, a chave *"$ref"* referencia dentro do próprio arquivo.

Na Figura 8 é utilizado o *"$ref"* dentro de *"definitions"* para criar uma definição local do tipo *"host"*, que aponta para um arquivo externo. Na seção *"properties"* é definido, então, uma propriedade do tipo *"host"* que indica a definição criada em *"definitions"*, com o uso do sufixo #.

```
{
    "definitions": {
        "host": {
            "type": "object",
            "\$ref": "hosts.json"
        },
    },
    "type": "object",
    "properties": {
        "host": {
            "type": "object",
            "\$ref": "#/definitions/host"
        },
    }
}
```

Figura 8 – Exemplo do uso das chaves *"definitions"* e *"$ref"*

A chave *"allOf"* é usada aliada também a *"$ref"*. Dessa forma, é possível estabelecer uma estrutura similiar à herança presente nas linguagens de programação orientadas a objetos. Na Figura 9, por exemplo, o esquema, além de possuir a propriedade *job*, também conterá o esquema definido para *person.json*.

```
{
    "type": "object",
    "allOf": [
        {
            "\$ref": person.json
        }
    ]
    "properties": {
        "job": {
            "type": "string"
        },
    }
}
```

Figura 9 – Exemplo do uso da chave *"allOf"* para estruturar uma herança

O tipo *{"type": "array"}* pode ter seus itens descritos de duas formas: uma com {} e outra com [ ]. A definição com chaves se refere à uma lista, enquanto a definição com colchetes à tuplas. A chave *"minItems"* ainda pode ser utilizada para especificar um tamanho mínimo, para, por exemplo, não permitir listas vazias.

Na Figura 10 é criado um *array* com nome *relacionamentos*, que precisa ter no

mínimo um item. Portanto, não pode ser vazio. Com a chave *oneOf* é especificado que os itens deste *array* devem ser ou do tipo *number* ou uma tupla $(number, string)$.

```
{
    "relacionamentos": {
        "type": "array",
        "items": {
            "oneOf": [
                {
                    "type": "array",
                    "items": [
                        {
                            "type": "number",
                        },
                        {
                            "type": "string",
                        }
                    ]
                },
                {
                    "type": "number",
                }
            ]
        },
        "minItems": 1
    }
}
```

Figura 10 – Exemplo do tipo *{"type": "array"}*

No momento, o JSON *Schema* está em seu oitavo *draft*, também chamado de 2019-09, publicado em Setembro de 2019.

3 Trabalhos Relacionados

Este capítulo detalha trabalhos que possuem algum vínculo com a proposta deste trabalho. Inicialmente, apresenta-se uma proposta de extração de esquemas de BDs NoSQL do tipo documento, que serviu de inspiração para este trabalho de conclusão de curso [Costa 2017]. Na sequência, descreve-se um trabalho que mapeia um modelo entidade-relacionamento para um BD NoSQL do tipo grafo a fim de verificar restrições nos vértices e arestas [Sousa e Cura 2018]. Ainda, demonstram-se os trabalhos de Roy-Hubara et al. [Roy-Hubara et al. 2017], que, similar ao anterior, cria um esquema a partir de um diagrama entidade-relaciomento, e de Bonifati et al. [Bonifati et al. 2019], que propõe uma forma de validar e evoluir um esquema de BD do tipo grafo a partir de uma DDL. Por fim, descreve-se os recursos que os principais SGBDs orientados a grafos (OrientDB[1] e Neo4j) possuem quanto à geração de esquemas.

Até o momento da escrita deste documento não foram encontrados trabalhos sobre extração de esquemas especificamente para BDs do tipo grafo. Apenas, de uma forma geral, trabalhos que extraem esquemas de dados semiestruturados, como em Nestorov et al. [Nestorov, Abiteboul e Motwani 1998].

Para a pesquisa por trabalhos relacionados foi utilizado o motor de busca DBLP[2], que é um dos principais indexadores de conferências e periódicos na área de Ciência da Computação. A *string* de busca utilizada foi *graph$ database | design | extract | schema*. O símbolo $ foi utilizado para requerer a palavra *graph* sem os seus derivados, como *graphical*. Os caracteres de espaço simbolizam o operador lógico *AND* e os caracteres | simbolizam o operador lógico *OR*.

A pesquisa resultou em 109 artigos que foram submetidos a 3 etapas de filtragem. A primeira etapa excluiu os artigos publicados antes de 2017, restando 44 artigos. A segunda etapa consistiu na leitura do título e do resumo e foram filtrados 12 artigos pertinentes e, dentre eles, selecionados os 3 artigos descritos nas próximas seções, com exceção do trabalho de Costa [Costa 2017], que foi desenvolvido como trabalho de conclusão de curso no âmbito do Grupo de Banco de Dados da UFSC[3].

[1] https://orientdb.com/
[2] https://dblp.uni-trier.de/
[3] http://lisa.inf.ufsc.br/wiki/index.php/Main

3.1 Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL Orientados a Documentos

Este trabalho propõe um algoritmo para extrair um esquema no formato JSON *Schema* a partir de um BD NoSQL do tipo documento [Costa 2017]. Seu propósito é similar ao do trabalho, porém o foco é diferente, uma vez que neste a essência consiste em BDs do tipo grafo.

O autor utiliza JSON *Schema* como formato de saída, por JSON ter se tornado um formato padrão para intercâmbio de dados e por ter sido utilizado também em seus trabalhos relacionados. O trabalho propõe uma aplicação *Web* que, a partir de uma coleção de documentos JSON, extrai o seu esquema. Para isso, seu algoritmo de extração percorre todos os documentos JSON armazenados e analisa suas propriedades para identificar o esquema bruto de cada documento e, então, unifica os esquemas brutos e gera um JSON *Schema* único que representa a coleção de documentos como um todo.

O algoritmo de extração pode ser dividido em quatro etapas:

1. A obtenção do esquema bruto dos documentos;
2. O agrupamento dos esquemas brutos iguais;
3. A unificação dos esquemas brutos;
4. A obtenção do JSON *Schema*.

Na primeira etapa é criado um esquema bruto, onde são percorridos todos os documentos JSON, mantendo sua estrutura em relação à campos, objetos aninhados e *arrays*, sendo cada valor primitivo substituído pelo seu tipo de dado JSON. Já na segunda etapa, são realizadas operações de agregação para extrair um número mínimo de objetos necessários para executar o processo de unificação de esquemas brutos, agrupando os documentos que possuem um mesmo esquema bruto.

Por sua vez, a terceira etapa do algoritmo produz uma estrutura temporária denominada estrutura unificada de esquema bruto (RSUS). Ela armazena a estrutura hierárquica de cada esquema bruto, contendo informações a respeito dos atributos dos objetos, seus tipos de dados e o caminho da raiz até a propriedade ou item dentro do documento. A RSUS é dividida em cinco propriedades: *field* que representa um atributo, *primitiveType* que representa um tipo primitivo JSON, *extendedType* representa um tipo de dado JSON estendido, *objectType* que representa um objeto JSON contendo os atributos e *arrayType* que representa uma lista. Por fim, na quarta etapa, cada propriedade da RSUS é mapeada para o seu equivalente em JSON *Schema*.

Algumas ideias desta abordagem contribuíram para o presente trabalho, como o uso de JSON *Schema* como formato de saída do processo de extração e a divisão em etapas do processo de extração desenvolvido.

3.2 Logical Design of Graph Databases from an Entity-Relationship Conceptual Model

Este trabalho apresenta um mapeamento de um modelo conceitual entidade-relacionamento para um modelo lógico estendido para BDs do tipo grafo. O projeto de BDs tradicionais geralmente segue três etapas de modelagem (conceitual, lógica e física), sendo que o modelo entidade-relacionamento é utilizado na primeira etapa e o objetivo do trabalho é produzir um esquema lógico para BDs NoSQL de grafo.

O modelo entidade-relacionamento considerado é uma forma simplificada com três componentes: as entidades, as relações e os atributos. Já o modelo lógico de grafo utilizado consiste em seis componentes: os vértices rotulados, as arestas rotuladas, um conjunto de propriedades para o vértice, um conjunto de propriedades para a aresta, um cojunto de restrições para os atributos dos vértices e um conjunto de restrições para as arestas. Por sua vez, as restrições consideradas são do tipo única, obrigatória, existente ou restrita, tanto para vértices quanto para arestas.

O algoritmo possui uma função geral que chama outras funções para verificar as restrições de vértices, arestas e restrições de cardinalidade, sendo dividido em três etapas. Na primeira delas, é criado um vértice para cada entidade e uma aresta para cada relação entre entidades. Na segunda, é analisada a cardinalidade dessas relações com a finalidade de verificar se esta aresta deve ser restrita ou obrigatória. E, na terceira etapa, são verificadas as restrições para cada uma das propriedades dos vértices e das arestas.

O trabalho mostra, como caso de uso, um sistema de *streaming* de música, onde a partir de um modelo entidade-relacionamento foi produzido um design lógico exemplificando as restrições citadas.

Ideias desta abordagem que auxiliaram este trabalho:

- as restrições utilizadas para modelar os vértices e arestas serviram de base para a análise dos dados físicos do BD NoSQL do tipo grafo;

- a utilização de engenharia reversa para, a partir do modelo lógico já existente, obter um modelo entidade-relacionamento.

3.3 *Modeling Graph Database Schema*

Assim como o trabalho da seção anterior, este propõe um processo que, a partir de um modelo entidade-relacionamento, obtém um esquema para um BD do tipo grafo. Seu diferencial, como o próprio autor descreve, é que o modelo conceitual considerado é mais completo, contemplando entidades fracas, relações ternárias e generalizações, entre outras características que não são abordadas em trabalhos similares.

O trabalho se preocupa em manter as restrições nas propriedades e nas relações existentes através de restrições. Desse modo, o esquema gerado possui restrições de cardinalidade, algo que não é suportado pelos BDs do tipo grafo atuais.

O processo de criação do esquema é dividido em duas etapas, sendo uma delas onde as estruturas mais complexas do modelo conceitual são transformadas em outras mais simples, e outra onde cada entidade é transformada em um vértice, cada relação em uma aresta e as restrições vão sendo implementadas.

Algumas ideias do trabalho proposto que contribuíram para este são:

- A forma como foram tratadas as estruturas mais complexas, como relacionamentos ternários;
- A proposta de definir restrições na cardinalidade das arestas do grafo.

3.4 *Schema Validation and Evolution for Graph Databases*

Os dois últimos trabalhos apresentados visam criar um esquema lógico com restrições do domínio. Já este trabalho afirma que BDs NoSQL no geral são atraentes por não possuírem esquema, não fazendo sentido incluir restrições, tampouco perder a flexibilidade.

O esquema é dividido em dois espectros extremos: o *descritivo* e o *prescritivo*. No primeiro deles, o esquema consiste nos dados em si, apenas refletindo os dados que estão ali inseridos, não havendo nenhum tipo de restrição imposta sobre eles. Por sua vez, o segundo é onde existe toda uma série de restrições e validações em cima do grafo, de modo que qualquer modificação nos dados do BD siga as regras do esquema.

Dessa maneira, o trabalho propõe:

- Um modelo de esquema onde é suportado os dois extremos do espectro de forma flexível, podendo transitar de um para o outro ou então utilizá-lo de uma forma hibrída;
- Uma DDL para este modelo de esquema;

- Um framework matemático para validação de uma instância do BD por meio de um homomorfismo com um grafo de esquema;

- Especificações matemáticas para evolução do esquema e como propagar essas mudanças.

O modelo de grafo é definido como uma tupla (N, E, n, P, v, M), onde N é um conjunto de tipos de vértices, E é um conjunto de tipos de arestas, n é um conjunto de funções que ligam um vértice origem a um vértice destino através de uma aresta $E \rightarrow N \times N$, P é um conjunto de propriedades, v é uma relação finita que atribui um conjunto de valores às propriedades e M é um conjunto de propriedades obrigatórias.

Assim sendo, a saída produzida pela ferramenta proposta neste trabalho se assemelha bastante ao modelo definido por esta abordagem de Bonifati et al. [Bonifati et al. 2019]. Por exemplo: a saída da ferramenta define também um conjunto de propriedades e um conjunto de funções que ligam dois vértices.

3.5 Esquemas Neo4j e OrientDB

O Neo4j e OrientDB são os principais SGBDs para BDs do tipo grafo. Ambos possuem alguma forma de respresentação de seu esquema mas que possuem alguma lacuna.

O Neo4j contém a função *db.schema()* mas esse esquema apenas apresenta os rótulos dos vértices e seus possíveis tipos de relacionamentos, não tem nenhuma informação referente as propriedades de seus vértices e relacionamentos nem quanto a sua obrigatoriedade ou ao seus tipos.

No OrientDB é possível realizar algumas consultas em seus metadados com a *query select expand(classes) from metadata:schema* e por conseguinte *select expand(properties) from { select expand(classes) from metadata:schema } where name = 'className'* para trazer informações sobre os rótulos de vértices e suas propriedades, mas sem nenhuma informação sobre os relacionamentos existentes.

Sendo assim este presente trabalho busca trazer o melhor de cada uma dessas duas formas de extração de esquema.

4 Ferramenta Para Extração de Esquemas de Bancos de Dados NoSQL de Grafo

A partir dos estudos realizados acerca dos trabalhos relacionados, onde foram apresentadas abordagens que modelam esquemas para BDs do tipo grafo e um trabalho que extrai o esquema de um BD do tipo documento, que foi inspiração para este trabalho, uma ferramenta para extração de esquemas de BDs de grafo foi desenvolvida.

Ao longo deste capítulo apresenta-se o projeto da ferramenta e o processo de extração de esquemas.

4.1 Projeto

A ferramenta é uma aplicação local para extração de esquemas, no formato JSON *Schema*, de um conjunto de vértices e arestas armazenados em um BD do tipo grafo. O projeto da ferramenta iniciou-se pelo levantamento de requisitos, os quais representam a descrição das necessidades dos usuários perante um *software*. Os requisitos de um *software* podem ser classificados em Requisitos Funcionais (RF) e Requisitos Não-Funcionais (RNF) [Sommerville 2011].

Os RFs são responsáveis por definir o comportamento do *software*, como o sistema deve reagir a entradas específicas e se comportar em determinadas situações. Por sua vez, os RNFs são relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenção e tecnologias envolvidas. Os referidos requisitos dizem respeito a como as funcionalidades dos RFs serão entregues ao usuário do *software*.

Logo, foram definidos os seguintes RFs e RNFs:

- RF01: Selecionar o BD de grafo;
- RF02: Extrair esquema;
- RF03: Salvar o esquema extraído;
- RNF01: Utilizar linguagem de programação *Python*[1], devido ao seu fácil trabalho com dicionários, mapas, listas e com o formato JSON;

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

- RNF02: Usar o SGBD Neo4j, por sua ampla comunidade e documentação, e também pelo fato de ser o BD do tipo grafo mais utilizado no mundo, segundo o site *db-engines*[2];

- RNF03: Extrair o esquema para um formato canônico em JSON *Schema*[3] para fins de padronização e integração com outros trabalhos.

Por mais que a estrutura de grafo não se assemelhe a um documento JSON, como utilizado no trabalho do Costa [Costa 2017], foi decidido o formato JSON *Schema* não para fins de validação, mas no sentido de informar o esquema de dados implícito dentro de um BD de grafo. Desta forma, no futuro pode-se utilizar este esquema como um intermediário em um processo de consulta sobre dados no formato JSON e seu posterior mapeamento para uma consulta específica sobre um BDs de grafo.

A linguagem de programação empregada foi a Python, que se trata de uma linguagem multiparadigma: imperativo, procedural, funcional e orientado a objetos. Para o desenvolvimento da ferramenta foi utilizado principalmente o paradigma funcional, nele o resultado das funções dependem apenas de seus argumentos, ou seja, chamar uma função com os mesmos argumentos sempre produz o mesmo resultado [Grover 2019].

Foi ainda usada a noção de classes, que vem do paradigma de orientação a objetos, para organizar as funções e redirecionar a saída de uma função como entrada de outra. A organização das classes é mostrada na Figura 11. O diagrama não representa nenhum padrão da UML (*Unified Modeling Language*)[4]. Ele é apenas uma representação de quais classes possuem acesso às funções de outras classes.

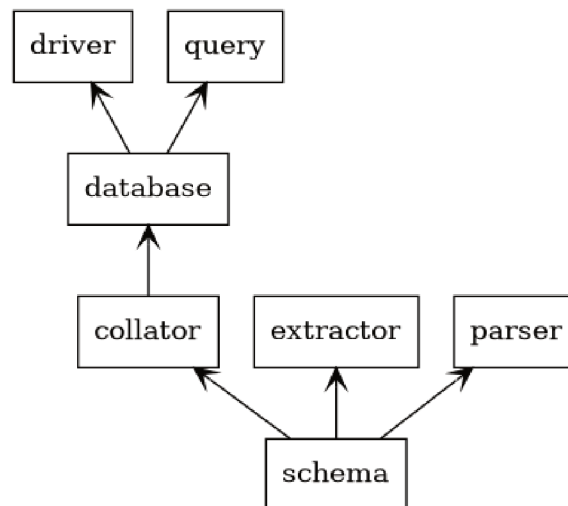

Figura 11 – Organização das funções da ferramenta

[2] https://db-engines.com/
[3] http://json-schema.org/
[4] https://www.uml.org/

Enquanto a classe *driver* é responsável por executar as transações no BD, a classe *query* armazena todas as *queries* que serão necessárias no processo de extração. Já a classe *database* funciona como um intermediador, fazendo com que o *driver* execute uma determinada *query* e, posteriormente, retornando o resultado para classe *collator*.

A classe *collator* possui os algoritmos descritos no capítulo 4.2.1, onde são reunidas as propriedades e relacionamentos dos vértices. A classe *extractor* possui os algoritmos mencionados no capítulo 4.2.2, os quais são responsáveis por tratar os casos de vértices com mútiplos rótulos. A classe *parser* é descrita no capítulo 4.2.3 e é encarregada de mapear os dados intermediários dos algoritmos anteriores na saída final em JSON *Schema*. Por fim, a classe *schema* é responsável por gerenciar e redirecionar as saídas do *collator* para o *extractor* e dele para o *parser*.

Deste modo, a ferramenta recebe como entrada o IP, porta, usuário e senha do BD Neo4j, bem como um diretório para onde será gerada a saída. Por não possuir interface gráfica, essas entradas são passadas a ela por meio de uma interface de linha de comando. A partir disso, a ferramenta produz como saída o(s) arquivo(s) JSON *Schema* correspondente(s) no diretório especificado. O código produzido para a ferramenta pode ser encontrado no Apêndice A ou no repositório do GitHub[5].

4.2 Processo de Extração de Esquemas

A extração de esquemas de BDs do tipo grafo envolve a descoberta de tipos de vértices e de arestas, bem como suas propriedades. Assim sendo, o processo proposto é dividido em três etapas, conforme mostra a Figura 12. O processo recebe como entrada um grafo de dados de um BD Neo4j e produz um esquema para esse grafo de dados no formato JSON *Schema*.

A primeira etapa realiza um *agrupamento* (*Collator*). Todos os vértices são percorridos, analisando e agrupando suas propriedades e relacionamentos de acordo com o rótulo do vértice. Na segunda etapa (*Extractor*), todas as arestas são percorridas, verificando suas propriedades e agrupando de acordo com o tipo de aresta. Na terceira etapa (*Parser*), um conjunto de documentos JSON *Schema* é gerado para essas informações agrupadas (sumarizadas) do esquema dos dados do BD Neo4j.

O BD Neo4j apresenta uma peculiaridade, se comparado com outros BDs de grafo: ele permite definir *múltiplos rótulos* por vértice. Assim sendo, é realizada a segunda etapa de *extração*, onde sobre o agrupamento de vértices é realizada uma extração a fim de obter apenas vértices de rótulo único. Esta extração é efetuada com a finalidade de identificar a qual rótulo específico uma determinada propriedade ou relacionamento está atrelado.

[5] https://github.com/MaoRodriguesJ/neo4j-schema-extract-code

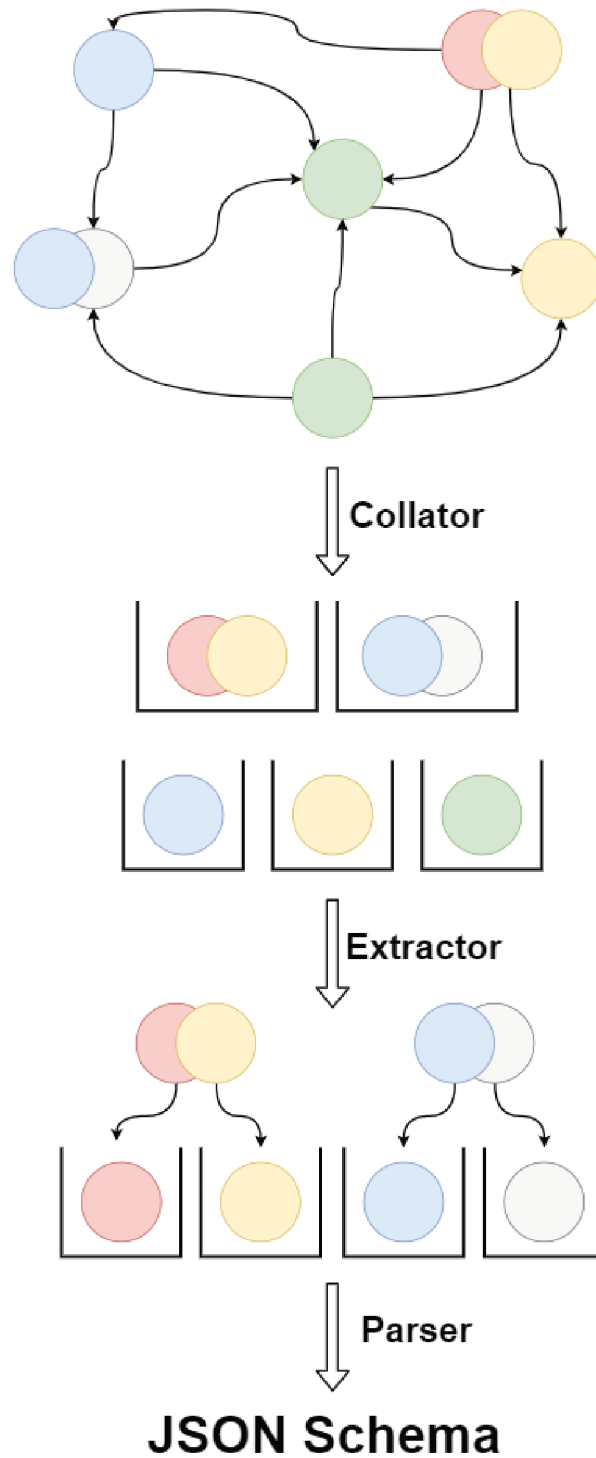

Figura 12 – Visão geral da ferramenta

Ela é executada através de operações de intersecção e de diferença entre conjuntos, sendo detalhada na Seção 4.2.2.

Por sua vez, a terceira e última etapa realiza o *mapeamento* do agrupamento já extraído para o formato JSON *Schema*. Os tópicos seguintes detalham as três etapas.

4.2.1 Agrupamento

Esta primeira etapa do processo de extração gera dois dicionários distintos. Um deles descreve os vértices, suas propriedades e relacionamentos (*Dicionário de Vértices*), e o outro os relacionamentos e suas propriedades (*Dicionário de Relacionamentos*). A Figura 13 mostra a estrutura do primeiro dicionário e a Figura 14 a do segundo.

```
"rótulos do vértice": {
    "propriedades": [
        {
            "(nome da propriedade, tipo da propriedade)":
            "obrigatoriedade"
        }
    ],
    "relacionamentos": [
        {
            "(tipo do relacionamento, rótulo do vértice destino)":
            "obrigatoriedade"
        }
    ]
}
```

Figura 13 – Dicionário de vértices

```
"tipo do relacionamento": {
    "propriedades": [
        {
            "(nome da propriedade, tipo da propriedade)":
            "obrigatoriedade"
        }
    ]
}
```

Figura 14 – Dicionário de relacionamentos

O Algoritmo 1 apresenta a função que realiza o agrupamento dos vértices e gera a estrutura do *Dicionário de Vértices*. Para isso, todos os vértices são percorridos a fim de agrupar suas propriedades e relacionamentos, definindo seus tipos e obrigatoriedades. Tal estrutura é armazenada na variável *grouping* da linha 4.

Para exemplificar essa etapa será utilizado como entrada um grafo como o da figura 15.

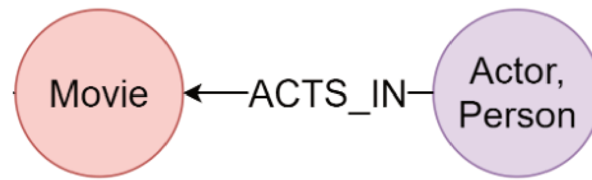

Figura 15 – Grafo exemplo

O referido algoritmo aplica a função *Powerset* sobre uma lista de rótulos de vértices (linha 3). A função mencionada produz um conjunto com todos os possíveis subconjuntos dos rótulos de vértices da lista, como por exemplo:

$Powerset([Person, Actor, Movie]) \longrightarrow \{(),\ (Person),\ (Actor),\ (Movie),$
$(Person, Actor),\ (Person, Movie),\ (Actor, Movie),\ (Person, Actor, Movie)\}$

A função *Powerset* é utilizada como um facilitador que permite definir *queries* mais específicas sem a necessidade de posteriormente verificar o(s) rótulo(s) do vértice analisado para identificar a qual agrupamento o vértice pertence, visto que um vértice pertence sempre a um único conjunto de rótulos no powerset.

Algoritmo 1: Agrupamento de Vértices

Function *grouping_nodes*:

Output: Dictionary with grouped nodes, their properties and relationships

```
1  begin
2      labels ⟵ list of nodes types;
3      labels_powerset ⟵ Powerset(labels);
4      grouping ⟵ {};
5      for label_combination ∈ labels_powerset do
6          records ← list of nodes with label combination;
7          key ← (label_combination, records size);
8          grouping[key] ⟵ {};
9          for record ∈ records do
10             Process_properties(grouping, key, record);
11             Populate_relationships(grouping, key, record id);
12         end
13     end
14     Check_mandatory_properties(grouping);
15     return grouping
16 end
```

Dentro do primeiro laço de iteração do Algoritmo 1 (linha 5) é feita uma consulta no BD para buscar todos os vértices que possuem uma das combinações de rótulo do *Powerset*, sendo esses vértices armazenados na lista *records* (linha 6). A váriavel *key* (linha

7) armazena uma tupla com a combinação de rótulos que está sendo iterada e a quantidade de vértices que possuem aquela combinação, utilizando o tamanho da lista *records*. Essa quantidade de vértices é usada para determinar se uma propriedade é obrigatória.

Na linha 8 é inicializado um novo dicionário vazio dentro de *grouping* para a chave definida. Ele manterá as propriedades e relacionamentos dos vértices de uma determinada combinação de rótulos, para esta etapa será considerada a combinação *{Actor, Person}*. Antes do próximo laço de iteração (linha 9), o dicionário *grouping* possui uma estrutura similar ao exemplo a seguir, onde existem 200 vértices que apresentam os rótulos *Actor* e *Person*:

```
{ ({Actor, Person}, 200) : {} }
```

Cada vértice da lista *records* é iterado e passa por duas funções. A função *Process_properties* (linha 10) calcula quantos vértices possui determinada propriedade e a função *Populate_relationships* (linha 11) contabiliza quais relacionamentos partem dos vértices com aquela combinação de rótulos.

Ao final (linha 14), a função *Check_mandatory_properties* verifica se uma propriedade é obrigatória. Para inferir essa obrigatoriedade, a função compara o número de ocorrências de uma determinada propriedade, que foi registrado anteriormente pelo Algoritmo 2, com o valor de *records size* armazenado na chave definida na linha 7 do Algoritmo 1. Se os valores forem iguais, a propriedade é considerada obrigatória, pois está presente em todos os vértices.

Essas funções são utilizadas também pelo processo de agrupamento de relacionamentos (Algoritmo 4), que possui entedimento similar ao do Algoritmo 1, diferindo apenas pelo fato que a iteração é feita sobre relacionamentos e não vértices, e a sua saída segue a estrutura do *Dicionário de Relacionamentos*.

No Algoritmo 2, para contabilizar as propriedades, é criada na linha 3 uma nova entrada no dicionário com a chave *"properties"*, dentro da chave passada como parâmetro *key*, que representa o(s) rótulo(s) do vértice. Posteriormente, itera-se sobre as propriedades do vértice, que é um dos parâmetros de *record* (linha 5). Dentro do laço, na linha 6, é criada uma tupla com o nome da propriedade a ser iterada e o seu tipo. Na estrutura condicional (linhas 7 e 11), verifica-se se esta propriedade já foi encontrada em outro vértice com a mesma combinação de rótulos para ou aumentar sua quantidade de ocorrências ou inicializá-la em 1.

A váriavel *grouping* do Algoritmo 1, após passar pelo processamento do Algoritmo 2 para todos os vértices de uma determinada combinação de rótulos, terá uma estrutura similar à apresentada a seguir. Uma das propriedades dos vértices de rótulo múltiplo *Actor* e *Person* é *name*, do tipo *str*, que ocorre em 200 dos 200 vértices com esses rótulos:

```
{
    ({Actor, Person}, 200) : {
        "properties" : {
            ('name', <class 'str'>): 200
        }
    }
}
```

No Algoritmo 2 é contado quantas ocorrências uma determinada propriedade teve para verificar se ela é obrigatória ou não. O mesmo não pode ser feito para averiguar se um relacionamento é obrigatório ou não, pois um vértice pode ter infinitos relacionamentos.

Assim sendo, para realizar essa contabilidade (Algoritmo 3), é assumido, para o primeiro vértice analisado, que todos os seus relacionamentos são obrigatórios e, a partir de então, vai se ajustando esta restrição à medida que outros vértices são analisados. O Algoritmo 3 inicialmente consulta no BD (linha 2) por uma lista de relacionamentos que partem de vértices de um determinado identificador único passado como parâmetro (*id*) e armazenado na váriavel *relationships*.

Caso seja o primeiro vértice analisado, entra-se na condição da linha 3 e executa-se as linhas 4 a 8. Uma nova entrada definida no dicionário com a chave *"relationships"*, dentro da chave passada como parâmetro *key*. Para cada relacionamento é criada uma chave (linha 6) com o tipo do relacionamento e quais são os rótulos do vértice destino. Esta chave possui o valor *True*, para indicar que o relacionamento é obrigatório.

Por consequência, os próximos vértices analisados irão para a linha 10 da estrutura condicional, onde serão armazenadas as chaves que já existiam em uma váriavel auxiliar

Algoritmo 2: Contabilizar propriedades

Function *Process_properties(grouping, key, record)*:

Result: Number of occurrences of each property from a type of node

1 **begin**
2   **if** $"properties" \notin grouping[key]$ **then**
3     $grouping[key]["properties"] \longleftarrow \{\}$;
4   **end**
5   **for** $property \in record.properties$ **do**
6     $prop_key \leftarrow$ (property name, property type);
7   **if** $prop_key \in grouping[key]["properties"]$ **then**
8       $grouping[key]["properties"][prop_key] + +$;
9   **else**
10       $grouping[key]["properties"][prop_key] \leftarrow 1$;
11   **end**
12   **end**
13 **end**

old_keys (linha 11). Dentro do laço da linha 12 é verificado se o relacionamento analisado atualmente já existia. Caso ele ainda não tenha sido analisado, isso quer dizer que existem outros vértices com aquele rótulo que não possuem aquele relacionamento. Assim sendo, sua obrigatoriedade passa a ser *False* (linha 16). É necessário ainda examinar se os relacionamentos anteriormente descobertos existem nesse vértice a ser analisado, caso não existam, também se tornam não obrigatório (linhas 19 a 23).

Algoritmo 3: Popular relacionamentos

```
   Function Populate_relationships(grouping, key, id):
   Result: Populate with relationships and if they are mandatory or not
1  begin
2      relationships ⟵ list of relationships that flow out from the node with a given
        id;
3      if "relationships" ∉ grouping[key] then
4          grouping[key]["relationships"] ⟵ {};
5          for relationship ∈ relationships do
6              relationship_key ← (relationship type, relationship reached node types);
7              grouping[key]["relationships"][relationship_key] ← True;
8          end
9      else
10         relationships_keys ← {};
11         old_keys ← grouping[key]["relationships"].keys;
12         for relationship ∈ relationships do
13             relationship_key ← (relationship type, relationship reached node types);
14             relationships_keys ← relationships_key.add(relationship_key);
15             if relationship_key ∉ old_keys then
16                 grouping[key]["relationships"][relationship_key] ← False;
17             end
18         end
19         for key ∈ old_keys do
20             if key ∉ relationships_keys then
21                 grouping[key]["relationships"][key] ← False;
22             end
23         end
24     end
25 end
```

Ao final da execução do Algoritmo 1 obtém-se um dicionário de vértices, conforme o seguinte exemplo:

```
{
    {Actor, Person} : {
        "properties" : {
            ('name', <class 'str'>) : True
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        }
    }
}
```

Algoritmo 4: Agrupamento de relacionamentos

```
Function grouping_relationships:
Output: Dictionary with grouped relationships and their properties
1  begin
2      types ⟵ list of relationships types ;
3      grouping ⟵ {};
4      for type ∈ types do
5          records ← list of relationships of type;
6          key ← (type);
7          grouping[key] ⟵ {};
8          for record ∈ records do
9              Process_properties(grouping, key, record);
10         end
11     end
12     Check_mandatory_properties(grouping);
13     return grouping_relationships
14 end
```

4.2.2 Extração

A etapa de extração utiliza como entrada o resultado do Algoritmo 1 *(Dicionário de Vértices)* e de forma iterativa vai convertendo os vértices com múltiplos rótulos em vértices de rótulos únicos (Algoritmo 5). Esse processo de extração permite representar características comuns uma únicas vez, de modo similar ao que é feito com herança em liguagens de programação orientadas a objetos, onde as características comuns ficam em uma classe pai.

Para um melhor entendimento de como o Algoritmo 5 funciona, utiliza-se como exemplo a entrada armazenada na variável *grouping_nodes* com a seguinte estrutura:

```
{
    {Actor, Person} : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('oscar', <class 'bool'>) : False
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        }
    },
    {User, Person} : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('lastLogin', <class 'str'>) : True
        },
        "relationships" : {
            ('RATED', {'Movie'}) : False
        }
    }
    Movie : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('genre', <class 'str'>) : True
        },
    }
}
```

Na linha 2 do Algoritmo 5 é inicializado um novo dicionário vazio *unique_labels*, que contém a mesma estrutura do *Dicionário de Vértices* da Figura 13, e que será populado à medida que o *grouping_nodes* vai sendo consumido.

Para controlar quando a extração já foi finalizada, uma váriavel booleana é inicializada como verdadeira. Assim, em todo início de uma nova iteração essa variável recebe o estado falso na linha 5 e sempre é colocada novamente no estado verdadeiro quando existem novos vértices a serem processados.

Na linha 6 são armazenados os conjuntos de rótulos presentes na entrada. Para o exemplo sendo utilizado, tem-se: *{Actor, Person}*, *{User, Person}* e *Movie*. O laço de iteração que segue verifica se existe algum conjunto de rótulo único e, caso exista, o retira do dicionário *grouping_nodes* e o coloca tanto em uma lista *labels_length_1* quanto na

Algoritmo 5: Extração de vértices com rótulo único

```
   Function extract(grouping_nodes):
   Output: Dictionary with unique type nodes, their properties and relationships
 1 begin
 2     unique_labels ⟵ {};
 3     labels_to_process ⟵ True;
 4     while labels_to_process do
 5         labels_to_process ⟵ False;
 6         labels_combinations ← grouping_nodes.keys;
 7         labels_length_1 ← {};
 8         for label ∈ labels_combinations do
 9             if label.length == 1 then
10                 unique_labels[label] ⟵ grouping_nodes.pop(label);
11                 labels_to_process ← True;
12                 labels_length_1.add(label);
13             end
14         end
15         if labels_to_process == True then
16             for label ∈ labels_length_1 do
17                 grouping_nodes ← Process_intersection(grouping_nodes,
18                                                       label,
19                                                       unique_labels[label]);
20             end
21         else
22             intersections ⟵ Labels_intersections(labels_combinations);
23             if intersection.length > 0 then
24                 labels_to_process ← True;
25                 key_most_intersected ←node type with most intersections;
26                 unique_labels[key_most_intersected] ⟵
                     Intersect_properties(grouping_nodes,
27                                        key_most_intersected,
28                                        intersections[key_most_intersected]);
29                 grouping_nodes ←
30                   Process_intersection(grouping_nodes,
31                                        key_most_intersected,
32                                        unique_labels[key_most_intersected]);
33             end
34         end
35     end
36     if grouping_nodes.length > 0 then
37         for key ∈ grouping_nodes do
38             unique_labels[key] ⟵ grouping_nodes[key];
39         end
40     end
41     return unique_labels
42 end
```

saída final *unique_labels*.

No exemplo, *Movie* possui rótulo único. Ele então é retirado do *grouping_nodes* e colocado nessa lista. Além disso, é sinalizado que existem rótulos a serem processados. Na linha 16, para cada rótulo a ser processado, é feita então uma reatribuição à váriavel *grouping_nodes*, que recebe o resultado do Algoritmo 7 *Process_intersection* (linha 17). Para esse cenário atual, a chamada de *Process_intersection* não altera o conteúdo de *grouping_nodes*, mas veremos o funcionamento deste algoritmo na linha 30.

Ao final desta primeira iteração, apenas movemos o rótulo *Movie* do dicionário *grouping_nodes* para o *unique_labels*. Já na segunda iteração, como não existem mais vértices de rótulo único em *grouping_nodes*, a execução do algoritmo é desviada para a linha 22. Nesta linha armazenam-se as intersecções existentes entre os conjuntos de rótulos (execução do Algoritmo 6). Para o exemplo sendo mostrado, a saída produzida é a seguinte:

```
{ Person : [{Actor, Person}, {User, Person}] }
```

Caso existissem, por exemplo, outros conjuntos de rótulos que contivessem o rótulo *Actor*, também seria inserida uma chave *Actor* com suas devidas intersecções. Como foram encontradas intersecções, isso é sinalizado na linha 24 e, na linha 25, é escolhido o rótulo que possui o maior número de intersecções. Para o nosso exemplo, temos apenas o rótulo *Person* com duas intersecções.

Na linha 26 é inserido na estrutura de saída final (*unique_labels*) o rótulo *Person* e seu conteúdo, cujo resultado é produzido pelo Algoritmo 8. A função *Intersect_properties* verifica o que existe em comum dentre propriedades e relacionamentos, na estrutura inicial *grouping_nodes*, no que diz respeito ao rótulo *Person*, produzindo o seguinte resultado:

```
{
    Person : {
        "properties" : {
            ('name', <class 'str'>) : True
        }
    }
}
```

Este resultado também serve como entrada para a reatribuição de *grouping_nodes* por meio da função *Process_intersection* da linha 29. Serão extraídos esse rótulo e seus devidos atributos dos possíveis conjuntos de rótulos que existam dentro do antigo *grouping_nodes*. Isso faz com que essa váriavel receba o seguinte valor:

```
{
    Actor : {
        "properties" : {
             ('oscar', <class 'bool'>) : False
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        },
        "allOf" : {
            Person
        }
    },
    User : {
        "properties" : {
             ('lastLogin', <class 'str'>) : True
        },
        "relationships" : {
            ('RATED', {'Movie'}) : False
        },
        "allOf" : {
            Person
        }
    }
}
```

A chave *"allOf"* é inserida na linha 9 do Algortimo 7 e é utilizada pelo processo de mapeamento da seção 4.2.3 para especificar que esses vértices são especializações daquele rótulo, ou seja, também possuem todos os atributos e relacionamentos dele.

Por fim, é feita uma nova e última iteração, onde esse nova estrutura *grouping_nodes* passa pelo mesmo processo para o rótulo *Movie* e entre as linhas 36 a 40. Após todos os rótulos serem processados é verificado se ainda restou algum conjunto de rótulos que não foi possível inferir se as propriedades e relacionamentos eram comuns com outros vértices.

A saída final para o exemplo utilizado é:

```
{
    Person : {
        "properties" : {
            ('name', <class 'str'>) : True
        }
    },
    Actor : {
        "properties" : {
            ('oscar', <class 'bool'>) : False
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        },
        "allOf" : {
            Person
        }
    },
    User : {
        "properties" : {
            ('lastLogin', <class 'str'>) : True
        },
        "relationships" : {
            ('RATED', {'Movie'}) : False
        },
        "allOf" : {
            Person
        }
    },
    Movie : {
        "properties" : {
            ('name', <class 'str'>) : True,
            ('genre', <class 'str'>) : True
        },
    }
}
```

Algoritmo 6: Encontrar intersecções

```
Function Labels_intersections(labels_combinations):
Output: Dictionary with intersected label and its combinations
1  begin
2  | intersections ⟵ {};
3  | for l1 ∈ labels_combinations do
4  | | for l2 ∈ labels_combinations do
5  | | | l3 ← l1 ∩ l2;
6  | | | if l3.length == 1 then
7  | | | | if l3 ∉ intersections then
8  | | | | | intersections[l3] ← {l1, l2};
9  | | | | else
10 | | | | | intersections[l3].add(l1, l2);
11 | | | | end
12 | | | end
13 | | end
14 | end
15 | return intersections
16 end
```

Algoritmo 7: Processar intersecções

```
Function Process_intersection(grouping, label, properties):
Output: Grouping with separated label from nodes if able
1  begin
2  | new_grouping ⟵ {};
3  | for key ∈ grouping do
4  | | if key ∩ label then
5  | | | new_key ← key \ label;
6  | | | new_properties ←
   | | |   grouping[key]["properties"] \ properties["properties"];
7  | | | new_relationships ←
   | | |   grouping[key]["relationships"] \ properties["relationships"];
8  | | | Process_relationships(new_relationships, label);
9  | | | new_grouping[new_key] ⟵ {"properties" :
   | | |   new_properties, "relationships" : new_relationships, "allOf" :
   | | |   label};
10 | | else
11 | | | new_grouping[key] ← grouping[key];
12 | | end
13 | end
14 | return new_grouping
15 end
```

Algoritmo 8: Intersectar propriedades

```
   Function Intersect_properties(grouping, key, intersections):
   Output: Dictionary with properties and relationships intersected
1  begin
2  |   aux_key ⟵ intersections[0];
3  |   new_properties ⟵ grouping[aux_key]["properties"];
4  |   new_relationships ⟵ grouping[aux_key]["relationships"];
5  |   for label ∈ intersections do
6  |   |   new_properties ← new_properties ∩ grouping[label]["properties"];
7  |   |   new_relationships ←
   |   |     new_relationships ∩ grouping[label]["relationships"];
8  |   end
9  |   Process_relationships(new_relationships, key);
10 |   return {"properties": new_properties, "relationships": new_relationships}
11 end
```

4.2.3 Mapeamento

Uma decisão de projeto para a ferramenta proposta neste trabalho foi separar cada rótulo de vértice e cada tipo de relacionamento em um arquivo JSON *Schema* separado, para melhorar a organização e permitir o reuso de partes do esquema completo do BD de grafo que mantêm diferentes representações dos dados.

As etapas de *Agrupamento* e *Extração* geram resultados na forma de dicionários, como mostram as Figuras 13 e 14, uma vez que este formato tem bastante semelhança com o formato JSON, o que facilita o mapeamento para o esquema final do BD de grafo. No entanto, para se adequar ao formato de saída da ferramenta (JSON *Schema*) foram necessários alguns ajustes.

Primeiramente, os tipos de dados das propriedades seguem o mapeamento das Tabelas 1 e 2, visto que as consultas realizadas no BD seguem o padrão de tipos do Neo4j, a linguagem de programação utilizada para o desenvolvimento da ferramenta foi Python, e a saída produzida está no formato JSON *Schema*.

Neo4j	**Python**
Null	None
Integer	int
Float	float
String	str
List	list
Map	dict
Boolean	bool

Tabela 1 – Mapeamento de tipos Neo4j para Python

Python	**Json *Schema***
None	null
int/float	number
str	string
list	array
dict	object
bool	boolean

Tabela 2 – Mapeamento de tipos Python para JSON *Schema*

Além dessa conversão de tipos, outros ajustes necessários foram os seguintes:

- caso existam referências a outros esquemas, estes devem ser insridos em uma chave *"definitions"*.
- propriedades obrigatórias devem ser inseridas em uma chave *"required"*.

- a chave *"allOf"* é usada para definir propriedades e relacionamentos em comum entre vértices. Além disso, deve ser colocado como referência o vértice que possui essas informações, como se fosse uma superclasse em um projeto orientado a objetos.

- A fim de comportar os relacionamentos entre os vértices foi criada uma propriedade *"relationships"*, que é uma lista de tuplas, onde cada uma delas representa o tipo do relacionamento e os rótulos do vértice destino. Isto é feito através da palavra-chave *oneOf*, que exige que os elementos da lista pertençam a alguma das tuplas descritas.

- a palavra-chave *"minItems"* com valor 1 é definida quando um relacionamento é obrigatório.

A Figura 16 mostra um exemplo de *JSON Schema* final para um dos vértices de um BD de atores e filmes, similar aos exemplos utilizados nas etapas anteriores. É possível ver que existem referências a outros arquivos *JSON Schema* que definem o relacionamento do tipo *acts_in* e o vértice de rótulo *movie*. Este relacionamento é obrigatório para todo ator, ou seja, todo ator deve ter atuado em pelo menos um filme como mostra a palavra-chave *"minItems"* com o valor 1. Por fim, a lista definida pela palavra-chave *"required"* demonstra que todas as propriedades são obrigatórias, exceto a propriedade *profileImageUrl*.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Actor",
    "description": "Actor node",
    "\$id": "actor.json",
    "definitions": {
        "acts_in": {
            "type": "object",
            "\$ref": "acts_in.json"
        },
        "movie": {
            "type": "object",
            "\$ref": "movie.json"
        }
    },
    "type": "object",
    "allOf": [
        {
            "\$ref": "person.json"
        }
    ],
    "properties": {
        "id": {
            "type": "number"
        },
        "biography": {
            "type": "string"
        },
        "profileImageUrl": {
            "type": "string"
        },
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
                                "type": "object",
                                "\$ref": "#/definitions/acts_in"
                            },
                            {
                                "type": "object",
                                "\$ref": "#/definitions/movie"
                            }
                        ]
                    }
                ]
            },
            "minItems": 1
        }
    },
    "required": [
        "id",
        "biography",
        "relationships"
    ],
    "additionalProperties": false
}
```

Figura 16 – Exemplo para um vértice de rótulo ator

5 Avaliação da Ferramenta

Este capítulo apresenta a avaliação da ferramenta desenvolvida considerando dois aspectos: *qualitativo* e *tempo de processamento*. Enquanto o primeiro analisa a qualidade do mapeamento BD grafo $\longrightarrow$ JSON *Schema*, o segundo verifica a complexidade e tempo de processamento da ferramenta na execução de processos de extração de esquemas.

5.1 Avaliação Qualitativa

Dois BDs de grafos no Neo4J, que simulam serviços existentes, foram utilizados para avaliar a ferramenta de forma qualitativa. O primeiro mantém dados que simulam o site *Airbnb*[1], um serviço para o anúncio e reserva de acomodações e meios de hospedagem. O segundo é baseado nos dados do *IMDB*[2], uma base de dados que contém filmes, séries, atores, diretores e usuários que podem realizar avaliações. As seções a seguir detalham essas avaliações.

5.1.1 Airbnb

O Neo4j possui uma função *call db.schema()*, que produz um grafo mostrando como seus vértices e relacionamentos estão organizados. O esquema do BD do Airbnb, segundo essa função, pode ser visto na Figura 17.

Para realizar a avaliação do esquema extraído com a ferramenta desenvolvida foi necessário fazer uma análise manual dos dados pertencentes ao BD, uma vez que este tipo de BD não possui um esquema explícito. A avaliaçâo foi realizada sobre o vértice de rótulo *Host* e do relacionamento *HOSTS*, podendo ser estendida aos demais.

Ao inferir manualmente a estrutura dos vértices de rótulo *Host*, verificamos que ele contém as propriedades *name, superhost, image, location* e *host_id*, sendo todas obrigatórias. Também verificou-se que todo vértice de rótulo *Host* deve possuir ao menos um relacionamento do tipo *HOSTS*, que liga este vértice a um vértice de rótulo *Listing*.

As Figuras 18 e 19 mostram os esquemas gerados pela ferramenta proposta para o relacionamento *HOSTS* e para o vértice *Host*, respectivamente. Pode-se constatar que todas as propriedades de um *Host* são obrigatórias, dado que todas estão declaradas como *required*. Ainda, todo vértice de rótulo *Host* deve conter pelo menos um relacionamento do tipo *HOSTS* que liga um Host a um vértice com rótulo *Listing*, pois seu esquema possui o valor 1 para a restrição *"minItems"*.

[1] https://www.airbnb.com
[2] https://www.imdb.com/

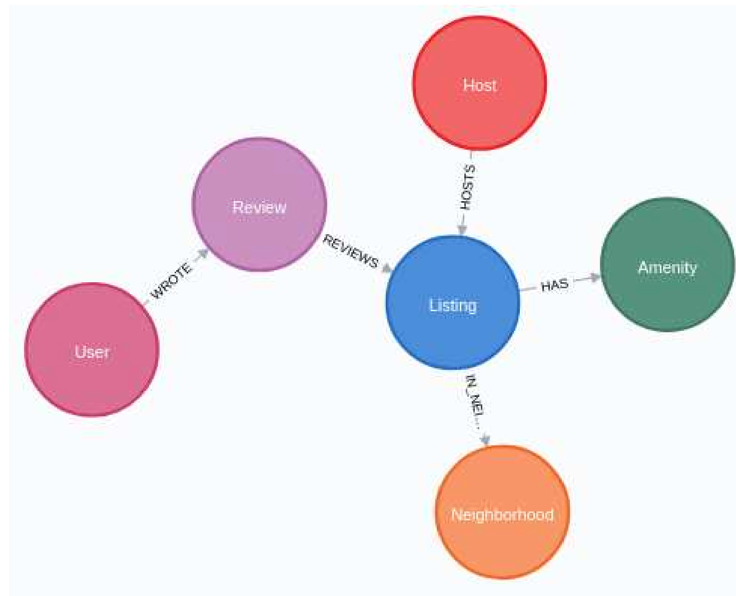

Figura 17 – Esquema Airbnb no BD Neo4j

Assim, quanto à qualidade do esquema gerado, foi obtido um resultado com 100% de acurácia, uma vez que foram extraídos com sucesso as propriedades e o relacionamento obrigatórios, bem como os seus tipos de dados. Os demais arquivos *JSON schema* produzidos podem ser encontrados no Apêndice B. Eles também obtiveram uma acurácia de 100% nos resultados gerados.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "HOSTS",
    "description": "HOSTS relationship",
    "\$id": "hosts.json",
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        }
    },
    "required": [
        "<id>"
    ],
    "additionalProperties": false
}
```

Figura 18 – JSON *Schema* para o relacionamento *HOSTS*

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Host",
    "description": "Host node",
    "\$id": "host.json",
    "definitions": {
        "hosts": {
            "type": "object",
            "\$ref": "hosts.json"
        },
        "listing": {
            "type": "object",
            "\$ref": "listing.json"
        }
    },
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "name": {
            "type": "string"
        },
        "superhost": {
            "type": "boolean"
        },
        "image": {
            "type": "string"
        },
        "location": {
            "type": "string"
        },
        "host_id": {
            "type": "string"
        },
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
                                "type": "object",
                                "\$ref": "#/definitions/hosts"
                            },
                            {
                                "type": "object",
                                "\$ref": "#/definitions/listing"
                            }
                        ]
                    }
                ]
            },
            "minItems": 1
        }
    },
    "required": [
        "<id>",
        "name",
        "superhost",
        "image",
        "location",
        "host_id",
        "relationships"
    ],
    "additionalProperties": false
}
```

Figura 19 – JSON *Schema* para o vértice *Host*

5.1.2 IMDB

A função *call db.schema()* do Neo4j aplicada sobre o BD do IMDB gera o resultado apresentado na Figura 20.

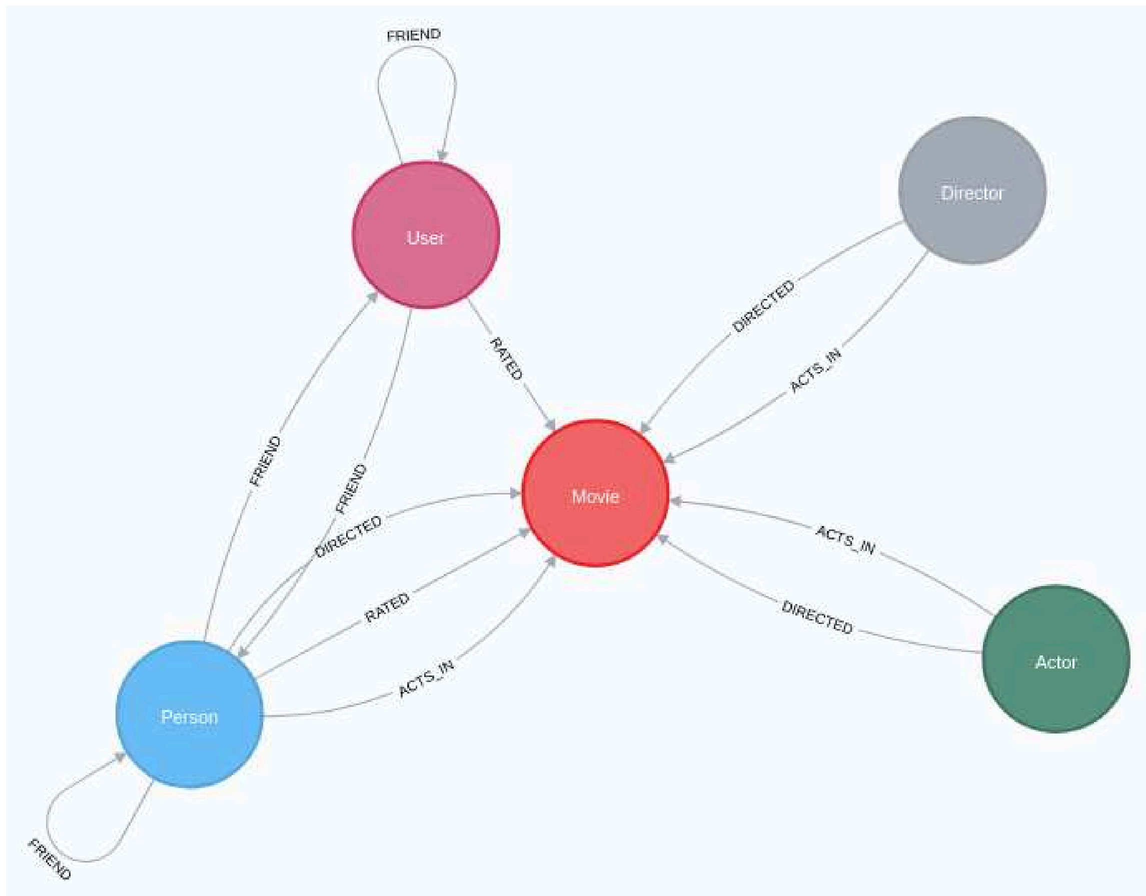

Figura 20 – Esquema IMDB no BD Neo4j

Com base em uma análise manual, como realizado na seção anterior, verificou-se que a Figura 20 não é uma boa representação de como os dados estão realmente estruturados. Foi possível averiguar que não existem vértices que possuem apenas o rótulo *Person* ou *Actor*, por exemplo. Na verdade, todo vértice presente no BD contém alguma das combinações de rótulo definidas a seguir:

- *Movie*
- *User, Person*
- *Actor, Person*
- *Director, Person*
- *Actor, Director, Person*

Este BD justifica a escolha por utilizar uma estratégia de intersecção de conjuntos para realizar a extração. Ela produz, como efeito, uma saída que se assemelha a uma hierarquia de *herança* em uma modelagem orientada a objetos. Neste caso, o rótulo *Person* pode conter propriedades e relacionamentos comuns aos rótulos *Actor*, *User* e *Director*, como se fosse uma superclasse deles.

Assim sendo, o JSON *Schema* resultante difere do esquema da Figura 20, pois o vértice de rótulo *Person* não possui nenhum relacionamento, já que estes relacionamentos apenas tem significado semântico nas suas especializações *Actor*, *Director* e *User*. Esta estratégia de geração de esquemas utilizada pela ferramenta proposta reduz o volume do esquema de saída gerado, uma vez que fatora, e representa uma única vez, as características comuns. É possível verificar isso nos esquemas produzidos nas Figuras 21, 22 e 23.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Person",
    "description": "Person node",
    "\$id": "person.json",
    "definitions": {},
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "name": {
            "type": "string"
        }
    },
    "required": [
        "<id>",
        "name"
    ],
}
```

Figura 21 – JSON *Schema* para o vértice *Person*

Portanto, após uma comparação entre a análise manual das propriedades e relacionamentos, e o resultado gerado pela ferramenta proposta, foi obtida uma acurácia de 100% no que diz respeito a qualidade do esquema gerado, uma vez que tamb em foram identificadas todas as propriedades e relacionamentos, bem como os seus tipos e restrições de obrigatoriedade. Além disso, foi produzido um esquema mais enxuto e com melhor valor semântico que o disponibilizado pela função *call db.schema()* do Neo4j.

Os demais arquivos do esquema produzido para o *IMDB* podem ser encontrados no Apêndice C.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Actor",
    "description": "Actor node",
    "\$id": "actor.json",
    "definitions": {
        "acts_in": {
            "type": "object",
            "\$ref": "acts_in.json"
        },
        "movie": {
            "type": "object",
            "\$ref": "movie.json"
        }
    },
    "type": "object",
    "allOf": [
        {
            "\$ref": "person.json"
        }
    ],
    "properties": {
        "<id>": {
            "type": "number"
        },
        "id": {
            "type": "string"
        },
        "birthplace": {
            "type": "string"
        },
        "version": {
            "type": "number"
        },
        "profileImageUrl": {
            "type": "string"
        },
        "biography": {
            "type": "string"
        },
        "lastModified": {
            "type": "string"
        },
        "birthday": {
            "type": "string"
        },
```

Figura 22 – JSON *Schema* 1/2 para o vértice *Actor*

```
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
                                "type": "object",
                                "\$ref": "#/definitions/acts_in"
                            },
                            {
                                "type": "object",
                                "\$ref": "#/definitions/movie"
                            }
                        ]
                    }
                ]
            },
            "minItems": 1
        }
    },
    "required": [
        "<id>",
        "id",
        "birthplace",
        "version",
        "profileImageUrl",
        "biography",
        "lastModified",
        "birthday",
        "relationships"
    ],
    "additionalProperties": false
}
```


Figura 23 – JSON *Schema* 2/2 para o vértice *Actor*

5.2 Análise de Tempo de Processamento

Esta seção demonstra os resultados da análise de tempo de processamento na extração de esquemas realizada pela ferramenta. Esta avaliação tem como objetivo verificar a complexidade do algoritmo de extração. Para isso, foram escolhidos 4 BDs de grafo com número de vértices e de relacionamentos diferentes: os dois já utilizados na seção 5.1, um BD menor referente ao universo da série televisiva *Dr WHo* e um BD grande extraído da API[3] do Stackoverflow[4]. O *Stackoverflow* é um site que reúne perguntas e respostas sobre programação, que podem ser avaliadas e comentadas por seus usuários. É possível visualizar o número de vértices e de relacionamentos de cada BD na Tabela 3.

BD	Vértices	Relacionamentos	V + R
Dr Who	1.060	2.286	3.346
IMDB	63.042	106.651	169.693
Airbnb	129.444	220.183	349.627
Stackoverflow	690.656	1.408.205	2.098.861

Tabela 3 – Tamanhos das Bases de Dados

Os experimentos foram executados em um sistema operacional Arch Linux, em uma máquina com processador Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, com 16GB de memória RAM e um SSD como memória não volátil. Cada uma dos quatro BDs utilizados no experimento estava dentro de um container Docker[5] do Neo4j na versão 3.5.12 e a ferramenta foi executada com Python versão 3.7.4.

Após a execução da ferramenta para os BDs considerados, foi possível notar que o processo de geração de esquemas cresce linearmente de acordo com o número de vértices e de relacionamentos, visto que os agrupamentos realizados pela função *powerset* evitam que um mesmo vértice seja percorrido mais de uma vez, para cada rótulo identificado no BD. Assim sendo, graças a esta estratégia de agrupamentos, a complexidade do processo de geração do esquema proposto é $O(m + n)$, onde m é o número de vértices e n o número de relacionamentos. Se cada vértice tivesse que ser analisado para descobrir o esquema de cada rótulo existente no BD, a complexidade de pior caso seria $O(r.m + n)$, onde r é o número de rótulos, ou seja, uma complexidade maior.

A referida complexidade pode ser verificada de forma prática. Uma média de tempo de processamento foi gerada sobre 10 execuções de extração de esquemas para os quatro BDs. A Tabela 4 mostra as cinco primeiras execuções, a Tabela 5 as cinco últimas, e a Tabela 6 as médias por BD. Esses dados foram plotados em um gráfico (Figura 24), onde é possível perceber uma função linear com relação ao número de vértices mais o número

[3] https://api.stackexchange.com/docs
[4] https://stackoverflow.com/
[5] https://www.docker.com/

de arestas.

BD	Tempo(s)				
Dr Who	2.914	1.591	1.234	1.246	1.053
IMDB	60.185	53.746	51.165	53.447	52.720
Airbnb	108.935	108.316	109.874	106.27	101.092
Stackoverflow	523.249	556.428	597.009	584.939	530.543

Tabela 4 – Tempos de Processamento 1/3

BD	Tempo(s)				
Dr Who	1.075	1.037	1.009	0.976	1.000
IMDB	51.783	55.508	52.633	61.232	51.950
Airbnb	98.708	142.959	116.541	122.886	108.766
Stackoverflow	550.322	666.126	600.355	556.883	552.429

Tabela 5 – Tempos de Processamento 2/3

BD	Tempo Médio (s)
Dr Who	1.064
IMDB	53.084
Airbnb	108.935
Stackoverflow	556.656

Tabela 6 – Média de Tempo de Processamento

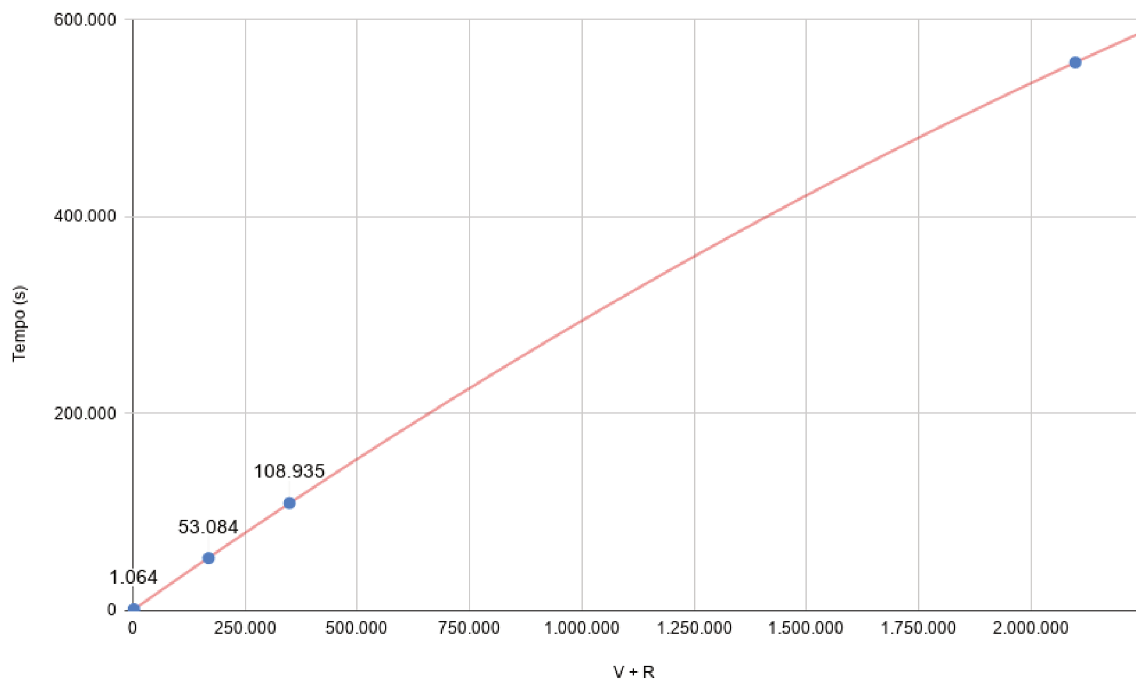

Figura 24 – Tempo de processamento X Tamanho do grafo

6 Conclusão

Este trabalho de conclusão de curso teve como objetivo principal o desenvolvimento de uma ferramenta para extração de esquemas de um BD NoSQL do tipo grafo. Os esquemas são gerados no formato JSON *Schema*, que é uma recomendação atual para a representação de esquemas de dados complexos.

A principal motivação para a realização deste trabalho é contribuir com pesquisas em desenvolvimento no *Grupo de BD da UFSC (GBD/UFSC)*. Trabalhos anteriores realizaram a extração de esquemas de BDs NoSQL do tipo Documento e Colunar para o formato JSON *Schema*. Este trabalho permite agora a extração de esquemas também de BDs NoSQL do tipo grafo, que é uma lacuna de pesquisa na literatura. Do ponto de vista de contribuição científica e técnica, a geração de esquemas para BDs *schemaless*, como é o caso de BDs NoSQL, facilita as tarefas de manipulação e integração de dados, pois permite a aquisição de conhecimento sobre a representação dos dados presentes nesses BDs.

Os objetivos específicos (seção 1.2.2) podem ser considerados como atingidos. No caso do objetivo específico 1, que tratou do mapeamento do grafo para um formato intermediário reduzido, este pode ser visto em detalhes nas seções 4.2.1 e 4.2.2. Já o objetivo 2, que tratou do mapeamento deste formato intermediário para o formato padrão JSON *Schema*, é descrito na seção 4.2.3. Por sua vez, o objetivo 3, que se refere a avaliação da ferramenta proposta, é descrito no capítulo 5, apresentando bons resultados. Por fim, com relação ao objetivo 4, todo o código produzido neste trabalho encontra-se no GitHub[1] ou no Apêndice A deste documento. Assim, pode-se concluir que o objetivo geral do projeto também foi atingido, visto que a ferramenta foi projetada e realiza o que foi proposto.

Os temas e tecnologias aqui abordados muito acrescentaram para a formação do autor, incluindo BDs NoSQL, JSON *Schema* e desenvolvimento de software. Vale observar também o conhecimento adquirido da ferramenta Neo4j e de estruturas de dados como listas, conjuntos e dicionários.

Por fim, algumas sugestões de trabalhos futuros relacionados a este assunto são:

- Melhorar o desempenho da primeira etapa do processo proposto utilizando técnicas estatísticas, considerando a grande redundância em termos de esquemas dos dados;
- Gerar esquemas para outros BDs NoSQL de grafo;
- Integrar diferentes esquemas de BDs NoSQL.

[1] https://github.com/MaoRodriguesJ/neo4j-schema-extract-code

Referências

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Comput. Surv*, ACM, Universidad de Chile, v. 40, 2008. Citado 2 vezes nas páginas 33 e 34.

BONIFATI, A. et al. Schema validation and evolution for graph databases. *CoRR*, abs/1902.06427, 2019. Disponível em: <http://arxiv.org/abs/1902.06427>. Citado 3 vezes nas páginas 28, 43 e 47.

COSTA, F. D. S. D. *Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL Orientados a Documentos*. Florianópolis: Universidade Federal de Santa Catarina, 2017. Citado 7 vezes nas páginas 27, 28, 32, 33, 43, 44 e 50.

CYPHER, The Graph Query Language. 2018. Disponível em: <https://neo4j.com/cypher-graph-query-language/>. Citado na página 37.

DEFREYN, E. D. *HBASI - HBASE SCHEMA INFERENCE TOOL: UMAFERRAMENTA PARA EXTRAÇÃO DE ESQUEMAS DEBANCOS DE DADOS NOSQL COLUNARES*. Florianópolis: Universidade Federal de Santa Catarina, 2019. Citado na página 28.

FROZZA, A. A.; MELLO, R. dos S. A process for reverse engineering of aggregate-oriented nosql databases with emphasis on geographic data. In: *XXXIII Simpósio Brasileiro de Banco de Dados: Demos e WTDBD, SBBD 2018 Companion, Rio de Janeiro, RJ, Brazil, August 25-26, 2018*. [S.l.: s.n.], 2018. p. 109–115. Citado na página 28.

GESSERT, F. et al. Nosql database systems: a survey and decision guidance. *Comput Sci Res Dev*, Springer, v. 32, p. 353–365, 2017. ISSN 1865-2042. Citado 2 vezes nas páginas 27 e 33.

GROVER, J. *Perceiving Python programming paradigms*. 2019. Acessado em 30 de Outubro de 2019. Disponível em: <https://opensource.com/article/19/10/python-programming-paradigms>. Citado na página 50.

HAN, J. et al. Survey on nosql database. In: *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference*. Port Elizabeth, South Africa: IEEE, 2011. ISBN 978-1-4577-0208-2. Citado 3 vezes nas páginas 27, 31 e 32.

HECHT, R.; JABLONSKI, S. Nosql evaluation: A use case oriented survey. In: *Cloud and Service Computing (CSC), 2011 International Conference*. Hong Kong, China: IEEE, 2011. ISBN 978-1-4577-1637-9. Citado 3 vezes nas páginas 27, 32 e 33.

MARIANI, A. C. *Teoria de Grafos*. 2018. Disponível em: <https://www.inf.ufsc.br/grafos/livro.html>. Citado na página 34.

MARR, B. *Big Data: The 5 Vs Everyone Must Know*. 2014. Disponível em: <https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know/>. Citado na página 31.

NEO4J. *Neo4j Basics*. 2018. Disponível em: <https://neo4j.com/product/#basics>. Citado 2 vezes nas páginas 17 e 36.

NEO4J. *Why Graph Databases*. 2018. Disponível em: <https://neo4j.com/why-graph-databases/>. Citado na página 37.

NESTOROV, S.; ABITEBOUL, S.; MOTWANI, R. Extracting schema from semistructured data. In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1998. (SIGMOD '98), p. 295–306. ISBN 0-89791-995-5. Disponível em: <http://doi.acm.org/10.1145/276304.276331>. Citado 2 vezes nas páginas 28 e 43.

PEZOA, F. et al. Foundations of json schema. In: *WWW '16 Proceedings of the 25th International Conference on World Wide Web*. Montréal, Québec, Canada: ACM, 2016. ISBN 978-1-4503-4143-1/16/04. Citado 2 vezes nas páginas 38 e 39.

ROY-HUBARA, N. et al. Modeling graph database schema. *IT Professional*, IEEE, v. 19, p. 34–43, 11 2017. Citado 2 vezes nas páginas 28 e 43.

SADALAGE, P. J.; FOWLER, M. *NoSQL Distilled*. United States: Pearson, 2012. ISBN 978-0-321-82662-6. Citado 2 vezes nas páginas 28 e 32.

SOMMERVILLE, I. *Engenharia de software*. [S.l.]: PEARSON BRASIL, 2011. ISBN 9788579361081. Citado na página 49.

SOUSA, V. M. D.; CURA, L. M. del V. Logical design of graph databases from an entity-relationship conceptual model. In: *The 20th International Conference on Information Integration and Web-based Applications & Services*. Vienna, Austria: ACM, 2018. Citado na página 43.

Apêndices

APÊNDICE A – Código

```
1  from neo4j.v1 import GraphDatabase
2
3  class Driver():
4
5      def __init__(self, uri, user, password):
6          self._driver = GraphDatabase.driver(uri, auth=(user, password))
7
8      def close(self):
9          self._driver.close()
10
11     def read_transaction(self, query, parameters=None):
12         with self._driver.session() as session:
13             if parameters:
14                 return session.read_transaction(query, parameters)
15
16             return session.read_transaction(query)
17
18     def write_transaction(self, query, parameters=None):
19         with self._driver.session() as session:
20             if parameters:
21                 return session.write_transaction(query, parameters)
22
23             return session.write_transaction(query)
```

code/driver.py

```
1  from driver import Driver
2  from query import Query
3
4  class Database():
5
6      def __init__(self, driver):
7          self._driver = driver
8
9      def get_labels(self):
10         return [l['label'] for l in self._driver.read_transaction(Query.
     ↪ _get_labels)]
11
12     def get_relationship_types(self):
13         return [t['relationshipType'] for t in self._driver.
     ↪ read_transaction(Query._get_types)]
14
15     def get_nodes_by_labels(self, labels):
```

```
16          params = {}
17          for index, value in enumerate(labels):
18              params['label' + str(index)] = value
19
20          return self._driver.read_transaction(Query._get_nodes_by_label,
    ↪ params)
21
22      def get_relationships_types_by_id(self, node_id):
23          params = {'id' : node_id}
24          return self._driver.read_transaction(Query.
    ↪ _get_relationships_types_by_id, params)
25
26      def get_relationships_by_type(self, typed):
27          params = {'type': typed}
28          return self._driver.read_transaction(Query.
    ↪ _get_relationships_by_type, params)
```

code/database.py

```
1 class Query():
2
3     @staticmethod
4     def _get_types(tx):
5         return tx.run("call db.relationshipTypes")
6
7     @staticmethod
8     def _get_labels(tx):
9         return tx.run("call db.labels")
10
11    @staticmethod
12    def _get_nodes_by_label(tx, params):
13        query = "match (node) where "
14        for key in params:
15            query = query + "$" + key + " in labels(node) and "
16
17        query = query + "length(labels(node))=" + str(len(params)) + "
    ↪ return node"
18        return tx.run(query, params)
19
20    @staticmethod
21    def _get_relationships_types_by_id(tx, params):
22        return tx.run("match (node)-[relationship]->(end_node) where id(
    ↪ node)=$id "+
23                      "return type(relationship) as relationship, labels(
    ↪ end_node) as labels", params)
24
25    @staticmethod
26    def _get_relationships_by_type(tx, params):
```

```
27          # Get a list of relationships by its type
28          return tx.run("match␣()-[relationship]-()␣where␣type(relationship)=
    ↪ $type␣return␣relationship", params)
```

code/query.py

```
1 from database import Database
2 from itertools import chain, combinations
3
4 class Collator():
5
6     def __init__(self, database):
7         self._database = database
8
9     def grouping_nodes(self):
10        labels_powerset = self._get_labels_combination(self._database.
    ↪ get_labels())
11        grouping = dict()
12        for label_combination in labels_powerset:
13            records = [r for r in self._database.get_nodes_by_labels(
    ↪ label_combination)]
14            if len(records) == 0:
15                continue
16
17            key = (label_combination, len(records))
18            grouping[key] = dict()
19            for record in records:
20                self._process_props_grouping(grouping, key, record, 'node')
21                self._process_relationships_grouping(grouping, key, record[
    ↪ 'node'].id)
22
23        self._check_mandatory_props(grouping)
24        return self._process_keys(grouping, 'node')
25
26    def grouping_relationships(self):
27        types = self._database.get_relationship_types()
28        grouping = dict()
29        for typed in types:
30            records = [r for r in self._database.get_relationships_by_type(
    ↪ typed)]
31            if len(records) == 0:
32                continue
33
34            key = (typed, len(records))
35            grouping[key] = dict()
36            for record in records:
37                self._process_props_grouping(grouping, key, record, '
    ↪ relationship')
```

```
38
39          self._check_mandatory_props(grouping)
40          return self._process_keys(grouping, 'relationship')
41
42      def _process_props_grouping(self, grouping, key, record, record_name):
43          if 'props' not in grouping[key].keys():
44              grouping[key]['props'] = dict()
45
46          for prop in record[record_name].keys():
47              typed = type(record[record_name].get(prop))
48              prop_key = (prop, typed)
49              if prop_key in grouping[key]['props'].keys():
50                  grouping[key]['props'][prop_key] = grouping[key]['props'][
     ↪ prop_key] + 1
51              else:
52                  grouping[key]['props'][prop_key] = 1
53
54      def _process_relationships_grouping(self, grouping, key, node_id):
55          relationships = self._database.get_relationships_types_by_id(
     ↪ node_id)
56          if 'relationships' not in grouping[key].keys():
57              grouping[key]['relationships'] = dict()
58              for r in relationships:
59                  relationship_key = (r['relationship'], frozenset(r['labels'
     ↪ ]))
60                  grouping[key]['relationships'][relationship_key] = True
61
62          else:
63              relationships_keys = set()
64              old_keys = grouping[key]['relationships'].keys()
65              for r in relationships:
66                  relationship_key = (r['relationship'], frozenset(r['labels'
     ↪ ]))
67                  relationships_keys.add(relationship_key)
68                  if relationship_key not in old_keys:
69                      grouping[key]['relationships'][relationship_key] =
     ↪ False
70
71              for k in old_keys:
72                  if k not in relationships_keys:
73                      grouping[key]['relationships'][k] = False
74
75
76      def _check_mandatory_props(self, grouping):
77          for key in grouping:
78              size = key[1]
79              for prop in grouping[key]['props']:
```

```
80                  if grouping[key]['props'][prop] == size:
81                      grouping[key]['props'][prop] = True
82                  else:
83                      grouping[key]['props'][prop] = False
84
85      def _process_keys(self, grouping, new_key_name):
86          processed_grouping = dict()
87          for key in grouping:
88              processed_grouping[(key[0], new_key_name)] = grouping[key]
89
90          return processed_grouping
91
92      def _get_labels_combination(self, labels):
93          return self._powerset(labels)
94
95      def _powerset(self, iterable):
96          "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
97          s = list(iterable)
98          tuples = list(chain.from_iterable(combinations(s, r) for r in range
        ↪ (len(s)+1)))
99          return filter(None, map(frozenset, tuples))
```

code/collator.py

```
1
2  class Extractor():
3
4      def __init__(self):
5          pass
6
7      def extract(self, grouping):
8          grouping_nodes = grouping
9
10         unique_labels_processed = dict()
11         labels_to_process = True
12
13         while labels_to_process:
14             labels_to_process = False
15             grouping_nodes.pop((frozenset(), 'node'), None)
16
17             labels_combinations = {k[0] for k in grouping_nodes.keys()}
18
19             labels_len_1 = set()
20             for label in labels_combinations:
21                 if len(label) == 1:
22                     unique_labels_processed[label] = grouping_nodes.pop((
        ↪ label, 'node'))
23                     labels_to_process = True
```

```
24                      labels_len_1.add(label)
25
26              if labels_to_process:
27                  for l in labels_len_1:
28                      grouping_nodes = self._process_intersection(
    ↪ grouping_nodes, l, unique_labels_processed[l])
29
30              else:
31                  intersections = self._get_labels_intersections(
    ↪ labels_combinations)
32                  if len(intersections) > 0:
33                      labels_to_process = True
34                      key_with_most_intersections = self.
    ↪ _get_key_with_most_intersections(intersections)
35
36                      unique_labels_processed[key_with_most_intersections] =
    ↪ self._intersect_props_relationships(
37
    ↪  grouping_nodes,
38
    ↪  intersections[key_with_most_intersections],
39
    ↪  key_with_most_intersections)
40
41                      grouping_nodes = self._process_intersection(
    ↪ grouping_nodes,
42
    ↪ key_with_most_intersections,
43
    ↪ unique_labels_processed[key_with_most_intersections])
44
45          if len(grouping_nodes) > 0:
46              for k in grouping_nodes:
47                  unique_labels_processed[k] = grouping_nodes[k]
48
49          return unique_labels_processed
50
51      def _process_intersection(self, grouping, label, props):
52          new_grouping = dict()
53          for key in grouping:
54              if key[0].intersection(label):
55                  new_key = (key[0].difference(label), 'node')
56                  new_props = {k : grouping[key]['props'][k]
57                               for k in set(grouping[key]['props']).
    ↪ difference(set(props['props']))}
58
59                  new_relationships = {k : grouping[key]['relationships'][k]
```

```
60                                  for k in set(grouping[key]['
   ↪ relationships']).difference(set(props['relationships']))}
61
62                  self._process_new_relationships(new_relationships, label)
63
64                  new_grouping[new_key] = {'props' : new_props, '
   ↪ relationships' : new_relationships, 'allOf' : label}
65              else:
66                  new_grouping[key] = grouping[key]
67
68          return new_grouping
69
70      def _intersect_props_relationships(self, grouping, intersections, key):
71          aux_key = (next(iter(intersections)), 'node')
72          new_props = grouping[aux_key]['props']
73          new_relationships = grouping[aux_key]['relationships']
74          for label in intersections:
75              new_props = {k : grouping[aux_key]['props'][k]
76                           for k in set(new_props).intersection(set(grouping
   ↪ [(label, 'node')]['props']))}
77
78              new_relationships = {k : grouping[aux_key]['relationships'][k]
79                                   for k in set(new_relationships).
   ↪ intersection(set(grouping[(label, 'node')]['relationships']))}
80
81          self._process_new_relationships(new_relationships, key)
82
83          return {'props' : new_props, 'relationships' : new_relationships}
84
85      def _process_new_relationships(self, new_relationships, key):
86          old_relationships = list()
87          for relationship in new_relationships:
88              if len(relationship[1]) > 1 and next(iter(key)) in relationship
   ↪ [1]:
89                  old_relationships.append({relationship : new_relationships[
   ↪ relationship]})
90
91          for relationship in old_relationships:
92              for r in relationship:
93                  new_relationships.pop(r)
94                  new_relationships[(r[0], key)] = relationship[r]
95                  new_relationships[(r[0], r[1].difference(key))] =
   ↪ relationship[r]
96
97      def _get_labels_intersections(self, labels_combinations):
98          intersections = dict()
99          for l1 in labels_combinations:
```

```
100             for l2 in labels_combinations:
101                 l3 = l1.intersection(l2)
102                 if len(l3) == 1:
103                     if l3 not in intersections.keys():
104                         intersections[l3] = {l1, l2}
105                     else:
106                         intersections[l3].add(l1)
107                         intersections[l3].add(l2)
108 
109         return intersections
110 
111     def _get_key_with_most_intersections(self, intersections):
112         key_with_most_intersections = next(iter(intersections))
113         for k in intersections:
114             if len(intersections[k]) > len(intersections[
    ↪ key_with_most_intersections]):
115                 key_with_most_intersections = k
116 
117         return key_with_most_intersections
```

code/extractor.py

```
1 
2 class Parser():
3 
4     def __init__(self):
5         pass
6 
7     def parse(self, grouping_nodes, grouping_relationships):
8         parsed_list = list()
9         for key in grouping_relationships.keys():
10             parsed_list.append(self._parse_relationship(key,
    ↪ grouping_relationships[key]['props']))
11 
12         for key in grouping_nodes.keys():
13             parsed_list.append(self._parse_node(key, grouping_nodes[key]))
14 
15         return parsed_list
16 
17     def _parse_node(self, key, node):
18         parsed_node = dict()
19         parsed_node["$schema"] = "https://json-schema.org/draft/2019-09/
    ↪ schema#"
20         parsed_node["title"] = self._build_node_id(key).capitalize()
21         parsed_node["description"] = self._build_node_id(key).capitalize()
    ↪ + " node"
22         parsed_node["$id"] = self._build_node_id(key) + '.json'
23         parsed_node["definitions"] = dict()
```

```
24          parsed_node["type"] = "object"
25          parsed_node["properties"] = dict()
26          if 'allOf' in node.keys():
27              parsed_node['allOf'] = [{"$ref" : next(iter(node['allOf'])) + '
    ↪ .json'}]
28          parsed_node["required"] = list()
29          parsed_node["required"].append("<id>")
30          parsed_node["properties"]["<id>"] = {"type" : "number"}
31
32          for key in node['props']:
33              typed = self._check_type(key[1])
34              parsed_node["properties"][key[0]] = {"type" : typed}
35              if node['props']:
36                  parsed_node["required"].append(key[0])
37
38          if len(node['relationships']) > 0:
39              parsed_node["properties"]["relationships"] = dict()
40              parsed_node["properties"]["relationships"]["type"] = "array"
41              parsed_node["properties"]["relationships"]["items"] = dict()
42              parsed_node["properties"]["relationships"]["items"]["oneOf"] =
    ↪ list()
43
44              relationship_required = False
45              for key in node['relationships']:
46                  new_relationship = dict()
47                  new_relationship["type"] = "array"
48                  new_relationship["items"] = list()
49
50                  relationship_id = str(key[0]).lower()
51                  node_id = self._build_node_id(key[1])
52
53                  relationship_type = dict()
54                  relationship_type["type"] = "object"
55                  relationship_type["$ref"] = "#/definitions/" +
    ↪ relationship_id
56                  new_relationship["items"].append(relationship_type)
57
58                  node_type = dict()
59                  node_type["type"] = "object"
60                  node_type["$ref"] = "#/definitions/" + node_id
61                  new_relationship["items"].append(node_type)
62
63                  parsed_node["definitions"][relationship_id] = dict()
64                  parsed_node["definitions"][relationship_id]["type"] = "
    ↪ object"
65                  parsed_node["definitions"][relationship_id]["$ref"] =
    ↪ relationship_id + '.json'
```

```
66
67                 parsed_node["definitions"][node_id] = dict()
68                 parsed_node["definitions"][node_id]["type"] = "object"
69                 parsed_node["definitions"][node_id]["$ref"] = node_id + '.
    ↪ json'
70
71                 parsed_node["properties"]["relationships"]["items"]["oneOf"
    ↪ ].append(new_relationship)
72
73                 if not relationship_required and node['relationships'][key
    ↪ ]:
74                     relationship_required = True
75                     parsed_node["properties"]["relationships"]["minItems"]
    ↪ = 1
76                     parsed_node["required"].append("relationships")
77
78         return parsed_node
79
80     def _parse_relationship(self, key, props):
81         parsed_relationship = dict()
82         parsed_relationship["$schema"] = "https://json-schema.org/draft
    ↪ /2019-09/schema#"
83         parsed_relationship["title"] = str(key[0])
84         parsed_relationship["description"] = str(key[0]) + " relationship"
85         parsed_relationship["$id"] = str(key[0]).lower() + '.json'
86         parsed_relationship["type"] = "object"
87         parsed_relationship["properties"] = dict()
88         parsed_relationship["required"] = list()
89         parsed_relationship["required"].append("<id>")
90         parsed_relationship["properties"]["<id>"] = {"type" : "number"}
91         parsed_relationship["additionalProperties"] = False
92
93         for key in props:
94             typed = self._check_type(key[1])
95             parsed_relationship["properties"][key[0]] = {"type" : typed}
96             if props[key]:
97                 parsed_relationship["required"].append(key[0])
98
99         return parsed_relationship
100
101     def _check_type(self, typed):
102         if typed is str:
103             return "string"
104
105         if typed is int or typed is float:
106             return "number"
107
```

```
108         if typed is dict:
109             return "object"
110
111         if typed is list:
112             return "array"
113
114         if typed is bool:
115             return "boolean"
116
117         else:
118             return typed
119
120     def _build_node_id(self, key):
121         id_name = ''
122         for label in key:
123             id_name += '_' + str(label).lower()
124
125         return id_name[1:]
```

code/parser.py

```
1 from collator import Collator
2 from database import Database
3 from driver import Driver
4 from extractor import Extractor
5 from parser import Parser
6 from pathlib import Path
7 import sys, os, json
8
9 class Schema():
10
11     def __init__(self, database):
12         self._collator = Collator(database)
13         self._extractor = Extractor()
14         self._parser = Parser()
15
16     def generate(self, path):
17         grouping_nodes = self._collator.grouping_nodes()
18         grouping_relationships = self._collator.grouping_relationships()
19         extracted_grouping_nodes = self._extractor.extract(grouping_nodes)
20         parsed_list = self._parser.parse(extracted_grouping_nodes,
    ↪ grouping_relationships)
21
22         self._save(path, parsed_list)
23
24     def _save(self, path, parsed_list):
25         if not os.path.exists(path):
26             os.makedirs(path)
```

```
27
28          data_folder = Path(path)
29          for item in parsed_list:
30              with open(data_folder / item['$id'], 'w') as parsed_file:
31                  json.dump(item, parsed_file, indent=4)
32
33
34  if __name__ == '__main__':
35      url = "bolt://0.0.0.0:7687"
36      login = "neo4j"
37      password = "neo4jadmin"
38      path = "generated_schema"
39
40      if len(sys.argv) == 5:
41          url = sys.argv[1]
42          login = sys.argv[2]
43          password = sys.argv[3]
44          path = sys.argv[4]
45
46      schema = Schema(Database(Driver(url, login, password)))
47      schema.generate(path)
```

code/schema.py

APÊNDICE B – Esquema Airbnb

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Host",
    "description": "Host node",
    "$id": "host.json",
    "definitions": {
        "hosts": {
            "type": "object",
            "$ref": "hosts.json"
        },
        "listing": {
            "type": "object",
            "$ref": "listing.json"
        }
    },
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "name": {
            "type": "string"
        },
        "superhost": {
            "type": "boolean"
        },
        "image": {
            "type": "string"
        },
        "location": {
            "type": "string"
        },
        "host_id": {
            "type": "string"
        },
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
```

```
44                                  "type": "object",
45                                  "$ref": "#/definitions/hosts"
46                              },
47                              {
48                                  "type": "object",
49                                  "$ref": "#/definitions/listing"
50                              }
51                          ]
52                      }
53                  ]
54              },
55              "minItems": 1
56          }
57      },
58      "required": [
59          "<id>",
60          "name",
61          "superhost",
62          "image",
63          "location",
64          "host_id",
65          "relationships"
66      ],
67      "additionalProperties": false
68  }
```

airbnb_schema/host.json

```
1   {
2       "$schema": "https://json-schema.org/draft/2019-09/schema#",
3       "title": "Listing",
4       "description": "Listing␣node",
5       "$id": "listing.json",
6       "definitions": {
7           "in_neighborhood": {
8               "type": "object",
9               "$ref": "in_neighborhood.json"
10          },
11          "neighborhood": {
12              "type": "object",
13              "$ref": "neighborhood.json"
14          },
15          "has": {
16              "type": "object",
17              "$ref": "has.json"
18          },
19          "amenity": {
20              "type": "object",
```

```
21                "$ref": "amenity.json"
22            }
23        },
24        "type": "object",
25        "properties": {
26            "<id>": {
27                "type": "number"
28            },
29            "bedrooms": {
30                "type": "number"
31            },
32            "listing_id": {
33                "type": "string"
34            },
35            "price": {
36                "type": "number"
37            },
38            "accommodates": {
39                "type": "number"
40            },
41            "name": {
42                "type": "string"
43            },
44            "property_type": {
45                "type": "string"
46            },
47            "bathrooms": {
48                "type": "number"
49            },
50            "availabiliity_365": {
51                "type": "number"
52            },
53            "cleaning_fee": {
54                "type": "number"
55            },
56            "weekly_price": {
57                "type": "number"
58            },
59            "relationships": {
60                "type": "array",
61                "items": {
62                    "oneOf": [
63                        {
64                            "type": "array",
65                            "items": [
66                                {
67                                    "type": "object",
```

```
68                                  "$ref": "#/definitions/in_neighborhood"
69                              },
70                              {
71                                  "type": "object",
72                                  "$ref": "#/definitions/neighborhood"
73                              }
74                          ]
75                      },
76                      {
77                          "type": "array",
78                          "items": [
79                              {
80                                  "type": "object",
81                                  "$ref": "#/definitions/has"
82                              },
83                              {
84                                  "type": "object",
85                                  "$ref": "#/definitions/amenity"
86                              }
87                          ]
88                      }
89                  ]
90              },
91              "minItems": 1
92          }
93      },
94      "required": [
95          "<id>",
96          "bedrooms",
97          "listing_id",
98          "price",
99          "accommodates",
100         "name",
101         "property_type",
102         "bathrooms",
103         "availabiliity_365",
104         "cleaning_fee",
105         "weekly_price",
106         "relationships"
107     ],
108     "additionalProperties": false
109 }
```

airbnb_schema/listing.json

```
1 {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "Amenity",
```

```
4     "description": "Amenity node",
5     "$id": "amenity.json",
6     "definitions": {},
7     "type": "object",
8     "properties": {
9         "<id>": {
10            "type": "number"
11        },
12        "name": {
13            "type": "string"
14        }
15    },
16    "required": [
17        "<id>",
18        "name"
19    ],
20    "additionalProperties": false
21 }
```

airbnb_schema/amenity.json

```
1 {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "Neighborhood",
4     "description": "Neighborhood node",
5     "$id": "neighborhood.json",
6     "definitions": {},
7     "type": "object",
8     "properties": {
9         "<id>": {
10            "type": "number"
11        },
12        "name": {
13            "type": "string"
14        },
15        "neighborhood_id": {
16            "type": "string"
17        }
18    },
19    "required": [
20        "<id>",
21        "name",
22        "neighborhood_id"
23    ],
24    "additionalProperties": false
25 }
```

airbnb_schema/neighborhood.json

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Review",
    "description": "Review node",
    "$id": "review.json",
    "definitions": {
        "reviews": {
            "type": "object",
            "$ref": "reviews.json"
        },
        "listing": {
            "type": "object",
            "$ref": "listing.json"
        }
    },
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "date": {
            "type": "string"
        },
        "review_id": {
            "type": "string"
        },
        "comments": {
            "type": "string"
        },
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
                                "type": "object",
                                "$ref": "#/definitions/reviews"
                            },
                            {
                                "type": "object",
                                "$ref": "#/definitions/listing"
                            }
                        ]
                    }
                ]
```

```
48                }, 
49                "minItems": 1
50            }
51        },
52        "required": [
53            "<id>",
54            "date",
55            "review_id",
56            "comments",
57            "relationships"
58        ],
59        "additionalProperties": false
60    }
```

airbnb_schema/review.json

```
1  {
2      "$schema": "https://json-schema.org/draft/2019-09/schema#",
3      "title": "User",
4      "description": "User␣node",
5      "$id": "user.json",
6      "definitions": {
7          "wrote": {
8              "type": "object",
9              "$ref": "wrote.json"
10         },
11         "review": {
12             "type": "object",
13             "$ref": "review.json"
14         }
15     },
16     "type": "object",
17     "properties": {
18         "<id>": {
19             "type": "number"
20         },
21         "name": {
22             "type": "string"
23         },
24         "user_id": {
25             "type": "string"
26         },
27         "relationships": {
28             "type": "array",
29             "items": {
30                 "oneOf": [
31                     {
32                         "type": "array",
```

```
                         "items": [
                             {
                                 "type": "object",
                                 "$ref": "#/definitions/wrote"
                             },
                             {
                                 "type": "object",
                                 "$ref": "#/definitions/review"
                             }
                         ]
                     }
                 ]
             },
             "minItems": 1
         }
     },
     "required": [
         "<id>",
         "name",
         "user_id",
         "relationships"
     ],
     "additionalProperties": false
}
```

airbnb_schema/user.json

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "WROTE",
    "description": "WROTE relationship",
    "$id": "wrote.json",
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        }
    },
    "required": [
        "<id>"
    ],
    "additionalProperties": false
}
```

airbnb_schema/wrote.json

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
```

```
3     "title": "REVIEWS",
4     "description": "REVIEWS relationship",
5     "$id": "reviews.json",
6     "type": "object",
7     "properties": {
8         "<id>": {
9             "type": "number"
10        }
11    },
12    "required": [
13        "<id>"
14    ],
15    "additionalProperties": false
16 }
```

airbnb_schema/reviews.json

```
1  {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "IN_NEIGHBORHOOD",
4     "description": "IN_NEIGHBORHOOD relationship",
5     "$id": "in_neighborhood.json",
6     "type": "object",
7     "properties": {
8         "<id>": {
9             "type": "number"
10        }
11    },
12    "required": [
13        "<id>"
14    ],
15    "additionalProperties": false
16 }
```

airbnb_schema/in_neighborhood.json

```
1  {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "HAS",
4     "description": "HAS relationship",
5     "$id": "has.json",
6     "type": "object",
7     "properties": {
8         "<id>": {
9             "type": "number"
10        }
11    },
12    "required": [
```

```
13          "<id>"
14      ],
15      "additionalProperties": false
16  }
```

airbnb_schema/has.json

```
1   {
2       "$schema": "https://json-schema.org/draft/2019-09/schema#",
3       "title": "HOSTS",
4       "description": "HOSTS relationship",
5       "$id": "hosts.json",
6       "type": "object",
7       "properties": {
8           "<id>": {
9               "type": "number"
10          }
11      },
12      "required": [
13          "<id>"
14      ],
15      "additionalProperties": false
16  }
```

airbnb_schema/hosts.json

APÊNDICE C – Esquema IMDB

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Person",
    "description": "Person node",
    "$id": "person.json",
    "definitions": {},
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "name": {
            "type": "string"
        }
    },
    "required": [
        "<id>",
        "name"
    ]
}
```

imdb_schema/person.json

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "User",
    "description": "User node",
    "$id": "user.json",
    "definitions": {
        "rated": {
            "type": "object",
            "$ref": "rated.json"
        },
        "movie": {
            "type": "object",
            "$ref": "movie.json"
        },
        "friend": {
            "type": "object",
            "$ref": "friend.json"
        },
        "person": {
            "type": "object",
```

```
21              "$ref": "person.json"
22          },
23          "user": {
24              "type": "object",
25              "$ref": "user.json"
26          }
27      },
28      "type": "object",
29      "allOf": [
30          {
31              "$ref": "person.json"
32          }
33      ],
34      "properties": {
35          "<id>": {
36              "type": "number"
37          },
38          "roles": {
39              "type": "string"
40          },
41          "login": {
42              "type": "string"
43          },
44          "password": {
45              "type": "string"
46          },
47          "relationships": {
48              "type": "array",
49              "items": {
50                  "oneOf": [
51                      {
52                          "type": "array",
53                          "items": [
54                              {
55                                  "type": "object",
56                                  "$ref": "#/definitions/rated"
57                              },
58                              {
59                                  "type": "object",
60                                  "$ref": "#/definitions/movie"
61                              }
62                          ]
63                      },
64                      {
65                          "type": "array",
66                          "items": [
67                              {
```

```
68                                 "type": "object",
69                                 "$ref": "#/definitions/friend"
70                             },
71                             {
72                                 "type": "object",
73                                 "$ref": "#/definitions/person"
74                             }
75                         ]
76                     },
77                     {
78                         "type": "array",
79                         "items": [
80                             {
81                                 "type": "object",
82                                 "$ref": "#/definitions/friend"
83                             },
84                             {
85                                 "type": "object",
86                                 "$ref": "#/definitions/user"
87                             }
88                         ]
89                     }
90                 ]
91             }
92         }
93     },
94     "required": [
95         "<id>",
96         "roles",
97         "login",
98         "password"
99     ]
100 }
```

imdb_schema/user.json

```
1 {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "Movie",
4     "description": "Movie␣node",
5     "$id": "movie.json",
6     "definitions": {},
7     "type": "object",
8     "properties": {
9         "<id>": {
10            "type": "number"
11        },
12        "studio": {
```

```
13                 "type": "string"
14             },
15             "releaseDate": {
16                 "type": "string"
17             },
18             "imdbId": {
19                 "type": "string"
20             },
21             "runtime": {
22                 "type": "number"
23             },
24             "description": {
25                 "type": "string"
26             },
27             "language": {
28                 "type": "string"
29             },
30             "title": {
31                 "type": "string"
32             },
33             "version": {
34                 "type": "number"
35             },
36             "trailer": {
37                 "type": "string"
38             },
39             "imageUrl": {
40                 "type": "string"
41             },
42             "genre": {
43                 "type": "string"
44             },
45             "tagline": {
46                 "type": "string"
47             },
48             "lastModified": {
49                 "type": "string"
50             },
51             "id": {
52                 "type": "string"
53             },
54             "homepage": {
55                 "type": "string"
56             }
57         },
58         "required": [
59             "<id>",
```

```
        "studio",
        "releaseDate",
        "imdbId",
        "runtime",
        "description",
        "language",
        "title",
        "version",
        "trailer",
        "imageUrl",
        "genre",
        "tagline",
        "lastModified",
        "id",
        "homepage"
    ]
}
```

imdb_schema/movie.json

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Actor",
    "description": "Actor node",
    "$id": "actor.json",
    "definitions": {
        "acts_in": {
            "type": "object",
            "$ref": "acts_in.json"
        },
        "movie": {
            "type": "object",
            "$ref": "movie.json"
        }
    },
    "type": "object",
    "allOf": [
        {
            "$ref": "person.json"
        }
    ],
    "properties": {
        "<id>": {
            "type": "number"
        },
        "biography": {
            "type": "string"
        },
```

```
29          "id": {
30              "type": "string"
31          },
32          "lastModified": {
33              "type": "string"
34          },
35          "version": {
36              "type": "number"
37          },
38          "profileImageUrl": {
39              "type": "string"
40          },
41          "birthday": {
42              "type": "string"
43          },
44          "birthplace": {
45              "type": "string"
46          },
47          "relationships": {
48              "type": "array",
49              "items": {
50                  "oneOf": [
51                      {
52                          "type": "array",
53                          "items": [
54                              {
55                                  "type": "object",
56                                  "$ref": "#/definitions/acts_in"
57                              },
58                              {
59                                  "type": "object",
60                                  "$ref": "#/definitions/movie"
61                              }
62                          ]
63                      }
64                  ]
65              },
66              "minItems": 1
67          }
68      },
69      "required": [
70          "<id>",
71          "biography",
72          "id",
73          "lastModified",
74          "version",
75          "profileImageUrl",
```

```
76          "birthday",
77          "birthplace",
78          "relationships"
79      ]
80  }
```

imdb_schema/actor.json

```
1   {
2       "$schema": "https://json-schema.org/draft/2019-09/schema#",
3       "title": "Director",
4       "description": "Director node",
5       "$id": "director.json",
6       "definitions": {
7           "directed": {
8               "type": "object",
9               "$ref": "directed.json"
10          },
11          "movie": {
12              "type": "object",
13              "$ref": "movie.json"
14          }
15      },
16      "type": "object",
17      "allOf": [
18          {
19              "$ref": "person.json"
20          }
21      ],
22      "properties": {
23          "<id>": {
24              "type": "number"
25          },
26          "biography": {
27              "type": "string"
28          },
29          "id": {
30              "type": "string"
31          },
32          "lastModified": {
33              "type": "string"
34          },
35          "version": {
36              "type": "number"
37          },
38          "profileImageUrl": {
39              "type": "string"
40          },
```

```
41          "birthday": {
42              "type": "string"
43          },
44          "birthplace": {
45              "type": "string"
46          },
47          "relationships": {
48              "type": "array",
49              "items": {
50                  "oneOf": [
51                      {
52                          "type": "array",
53                          "items": [
54                              {
55                                  "type": "object",
56                                  "$ref": "#/definitions/directed"
57                              },
58                              {
59                                  "type": "object",
60                                  "$ref": "#/definitions/movie"
61                              }
62                          ]
63                      }
64                  ]
65              },
66              "minItems": 1
67          }
68      },
69      "required": [
70          "<id>",
71          "biography",
72          "id",
73          "lastModified",
74          "version",
75          "profileImageUrl",
76          "birthday",
77          "birthplace",
78          "relationships"
79      ]
80  }
```

imdb_schema/director.json

```
1  {
2      "$schema": "https://json-schema.org/draft/2019-09/schema#",
3      "title": "ACTS_IN",
4      "description": "ACTS_IN relationship",
5      "$id": "acts_in.json",
```

```
6     "type": "object",
7     "properties": {
8         "<id>": {
9             "type": "number"
10        },
11        "name": {
12            "type": "string"
13        }
14    },
15    "required": [
16        "<id>",
17        "name"
18    ],
19    "additionalProperties": false
20 }
```

imdb_schema/acts_in.json

```
1 {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "DIRECTED",
4     "description": "DIRECTED relationship",
5     "$id": "directed.json",
6     "type": "object",
7     "properties": {
8         "<id>": {
9             "type": "number"
10        }
11    },
12    "required": [
13        "<id>"
14    ],
15    "additionalProperties": false
16 }
```

imdb_schema/directed.json

```
1 {
2     "$schema": "https://json-schema.org/draft/2019-09/schema#",
3     "title": "FRIEND",
4     "description": "FRIEND relationship",
5     "$id": "friend.json",
6     "type": "object",
7     "properties": {
8         "<id>": {
9             "type": "number"
10        }
11    },
```

```
12      "required": [
13          "<id>"
14      ],
15      "additionalProperties": false
16  }
```

imdb_schema/friend.json

```
1   {
2       "$schema": "https://json-schema.org/draft/2019-09/schema#",
3       "title": "RATED",
4       "description": "RATED relationship",
5       "$id": "rated.json",
6       "type": "object",
7       "properties": {
8           "<id>": {
9               "type": "number"
10          },
11          "comment": {
12              "type": "string"
13          },
14          "stars": {
15              "type": "number"
16          }
17      },
18      "required": [
19          "<id>",
20          "stars"
21      ],
22      "additionalProperties": false
23  }
```

imdb_schema/rated.json

APÊNDICE D – Artigo Formato SBC

Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL do Tipo Grafos

Salomão Rodrigues Jacinto[1]

[1]Universidade Federal de Santa Catarina

Abstract. *Currently, a large volume of heterogeneous data is generated and consumed on the network in an unprecendented scale which led to the creation of database models named NoSQL. These databases are capable of handling a large volume of data and are schemaless, in other words, they do not have an implicit schema such as relational databases. But the knowledge of how data is structurally stored is of great importance for the development of an application or an data analysis. There are works in the literature that extract the schema from a semistructured data in general and works that propose a theoretical schema model for graph databases. Diffferent from them, this work aims to develop a tool to extract a schema from an existing graph NoSQL database to a JSON Schema format, as well as the elaboration of a document containing the studies and tests carried out on top of the implemented application. Experimental evaluations show that the proposed tool generates a suitable schema representation with a linear complexity.*

Resumo. *Atualmente, uma grande quantidade de dados heterogêneos são gerados e consumidos em uma escala sem precedentes, o que motivou a criação de sistemas gerenciadores de bancos de dados que levam o nome de NoSQL. Esses bancos de dados possuem capacidade para lidar com um grande volume de dados e não necessariamente possuem um esquema implícito como os bancos de dados relacionais. Mesmo assim, o conhecimento de como os dados estão sendo armazenados estruturalmente é de suma importância para diversas tarefas, como integração ou análise de dados. Existem trabalhos na literatura que extraem o esquema de dados semiestruturados de forma geral e trabalhos que propõem um modelo teórico de esquema para bancos de dados do tipo grafo. Como diferencial, o presente trabalho visa o desenvolvimento de uma ferramenta para extração de um esquema de um banco de dados NoSQL do tipo grafo para um formato do tipo JSON Schema, assim como a elaboração de um documento contendo os estudos e testes realizados sobre a ferramenta implementada. Avaliações experimentais demonstram que a ferramenta produz uma representação adequada de um esquema com uma complexidade linear.*

1. Introdução

Com o crescimento contínuo da Internet e de aplicações lidando com um grande volume de dados, se tornou necessário desenvolver Bancos de Dados (BDs) que fossem capazes de armazenar e processar esses dados de forma efetiva, ou seja, com um alto desempenho em termos de operações de leitura e escrita [Han et al. 2011].

BDs tradicionais (BDs relacionais) oferecem mecanismos para gerenciar dados com forte consistência, tendo alcançado um nível de estabilidade e confiabilidade ao

longo de muitos anos de desenvolvimento. Entretanto, dados gerados por redes sociais ou redes de sensores, por exemplo, são naturalmente volumosos e heterogêneos, além de não apresentarem, em geral, um esquema associado, o que faz com que não consigam ser gerenciados de forma eficiente por BDs relacionais [Gessert et al. 2017].

Logo, escalar verticalmente (estendendo as capacidades computacionais de um servidor de dados, por exemplo) não é mais viável econômica e fisicamente. Assim, para solucionar esses problemas percebeu-se a oportunidade de escalar horizontalmente por intermédio de BDs distribuídos, isto é, incluindo mais nodos servidores de dados à infraestrutura computacional. Porém, como BDs tradicionais não foram projetados para funcionar de forma distribuída, não se adequando bem ao gerenciamento de particionamento e fragmentação de dados sob demanda, surgiu uma nova família de BDs denominada *NoSQL*, um acrônimo para *Not only SQL* [Han et al. 2011].

Os BDs NoSQL surgem, então, para suprir essa necessidade de desempenho de leitura e escrita sobre uma grande massa de dados de forma escalável, concorrente e eficiente. Para atingir esses objetivos, esses BDs dispensam as tradicionais propriedades ACID (*Atomicity, Consistency, Isolation, Durability*) dos BDs relacionais e adotam as propriedades BASE (*Basically Available, Soft State, Eventual Consistency*), relaxando a consistência para garantir alta disponibilidade [Da Costa 2017].

Dentre uma das famílias de BDs NoSQL encontra-se BDs do tipo grafo. Em contraste com BDs relacionais, BDs do tipo grafo são especializados em gerenciar, de modo eficiente, dados extremamente conectados, pois o custo de operações como *joins* recursivos podem ser substituídos por operações de travessias no grafo. Os principais BDs do tipo grafo são baseados em grafos direcionados com múltiplos relacionamentos e propriedades. Neste caso, vértices e arestas são objetos com propriedades do tipo chave-valor [Hecht and Jablonski 2011].

Em BDs relacionais é necessário ter uma estrutura de tabelas pré-definidas antes que se possam armazenar os dados, também denominada de *esquema*. Já em BDs NoSQL, não é preciso existir uma estrutura prévia para começar a armazenar os dados, sendo possível que dados com representações diferentes possam coexistir no BD. Essa característica dos BDs NoSQL é chamada *schemaless* [Sadalage and Fowler 2012].

A ausência de controle de esquemas dos dados permite uma flexibilidade de representação dos dados, evitando, por exemplo, que existam diversas colunas sem valor, como no caso de BDs relacionais que possuem esquemas rígidos. Essa característica trouxe grande popularidade aos BDs NoSQL e, apesar de ser um ponto positivo, muitas vezes a estrutura dos dados pode estar implicitamente agregada ao código da aplicação, dificultando o crescimento e manutenção da mesma [Sadalage and Fowler 2012].

Os esquemas que BDs do tipo grafo providenciam são tipicamente *descritivos*, ou seja, apenas refletem os dados que estão ali inseridos, mas não criam nenhum tipo de restrição. Este esquema pode ser alterado simplesmente mudando os dados, isto é, ao inserir um vértice que seja completamente diferente de todos que já existem na base, o esquema dos dados do BD muda naturalmente. Essa flexibilidade inicialmente é percebida como uma ótima característica, principalmente nos estágios iniciais de desenvolvimento, porém na medida em que a aplicação vai maturando, essas mudanças precisam ser feitas com maior cuidado, criando uma demanda por restrições de esquemas, melhor dizendo,

um esquema mais *prescritivo* [Bonifati et al. 2019].

1.1. Justificativa

O conhecimento de como os dados estão estruturados e armazenados é de suma importância para o desenvolvimento e manutenção de uma aplicação. Contudo, quando isto está implícito no código da aplicação pode se tornar um grande problema, porquanto dificulta a integração entre múltiplos aplicativos, em um mundo onde a necessidade desse conhecimento é essencial [Sadalage and Fowler 2012].

Trabalhos já desenvolvidos no Grupo de BD da UFSC (GBD UFSC) realizaram a extração de esquemas de BDs NoSQL do tipo *Documento* [Da Costa 2017], do tipo *Colunar* [Dias Defreyn 2019] e *Orientado a Agregados* [Frozza and dos Santos Mello 2018] para o formato canônico JSON *Schema*[1]. Esse formato foi escolhido devido à grande popularidade do padrão JSON para representação e intercâmbio de dados.

A intenção deste trabalho é contribuir com uma pesquisa em nível de pós-graduação, que visa extrair esquemas de todos os modelos de dados NoSQL para uma posterior integração de dados através da extração de esquemas de BDs do tipo grafo.

Trabalhos relacionados na literatura propõem um modelo teórico de esquema para bancos de dados do tipo grafo [Roy-Hubara et al. 2017] ou extraem algum tipo de esquema a partir de dados semiestruturados de forma geral [Nestorov et al. 1998]. No entanto, nenhum deles extrai o esquema especificamente de um BD do tipo grafo.

Assim sendo, a ferramenta proposta por este artigo tem como objetivo extrair o esquema de um BD NoSQL do tipo grafo, em particular o SGBD Neo4j[2], para um formato JSON *Schema*, com o intuito de auxiliar no desenvolvimento de aplicações, integração entre serviços e validação de dados. O Neo4j foi escolhido por ser o SGBD do tipo grafo mais utilizado no mundo, segundo o site *db-engines*[3].

1.2. Objetivos

1.3. Objetivo Geral

O objetivo geral do presente artigo consiste no desenvolvimento de uma ferramenta para extração de esquemas de um BD NoSQL do tipo grafo para um formato JSON *Schema*.

1.4. Objetivos Específicos

Os objetivos específicos são os seguintes:

- Mapear o conjunto de tipos e atributos dos vértices e arestas de um BD do tipo grafo para uma versão reduzida e refinada, utilizando o SGBD Neo4j e operações de conjuntos;
- Extrair o mapeamento realizado para um esquema no padrão JSON *Schema*;
- Avaliar a ferramenta proposta através de estudos de caso e testes de desempenho;
- Disponibilizar todo código produzido de forma livre.

[1] http://json-schema.org/
[2] https://neo4j.com/
[3] https://db-engines.com/

1.5. Metodologia

A metodologia de pesquisa e desenvolvimento deste trabalho dividiu-se em três etapas:

- **Etapa 1:** consistiu em uma pesquisa bibliográfica dos assuntos necessários para desenvolver a aplicação, investigando o estado da arte em *"esquemas de BDs NoSQL do tipo grafo"*. A pesquisa foi realizada em artigos, documentação das ferramentas, *surveys* e outras publicações relacionadas ao assunto.
- **Etapa 2:** Análise de requisitos e desenvolvimento da aplicação para extração de esquemas de BDs do tipo grafos. Também foram consideradas pesquisas acerca das tecnologias a serem utilizadas e dos algoritmos a serem usados e/ou criados.
- **Etapa 3:** Documentação, teste e avaliação da ferramenta desenvolvida para análise dos resultados, verificando a qualidade dos esquemas extraídos de grandes *datasets*, bem como simular cenários de serviços como IMDB[4] e Airbnb[5].

2. Fundamentação Teórica

Esta seção apresenta os conceitos utilizados no entendimento e desenvolvimento do trabalho, quais sejam, *Big Data*, BDs NoSQL e seus diferentes modelos de dados, além da teoria de grafos, o BD Neo4J e o formato JSON *Schema*.

2.1. *Big Data*

Big data é um termo usado para se referir a um grande volume de dados, tanto estruturados quanto não estruturados, de difícil processamento, gerenciamento e armazenamento [Han et al. 2011]. O fenômeno *Big Data* também é frequentemente descrito por meio dos chamados 5 Vs: Volume, Velocidade, Variedade, Veracidade e Valor [Marr 2014].

O *volume* se refere a vasta quantidade de dados gerada por pessoas ou máquinas, como e-mails, mensagens e fotos que são produzidos e compartilhados. Se for comparado todo o volume de dados gerado desde o início da Internet até o ano de 2008, a mesma quantidade é gerada a todo minuto hoje em dia. Isso torna os conjuntos atuais de dados grandes e de difícil armazenamento e análise. Com a tecnologia *Big Data* é possível armazenar e manipular esses conjuntos de dados com a ajuda de sistemas distribuídos e métodos de processamento paralelo massivo.

Por sua vez, a *velocidade* diz respeito a velocidade com que novos dados são gerados, transmitidos e utilizados por diversas fontes. A tecnologia *Big Data* permite analisar esses dados à medida que são gerados, ajudando na tomada de decisões.

Já a *variedade*, consiste nos diferentes tipos de dados utilizados atualmente. No passado só eram considerados dados estruturados que se encaixassem no formato relacional, mas, atualmente, com a tecnologia da *Big Data* é possível guardar dados de sensores, gravações de áudio e vídeo, além de manipular todos de forma conjunta.

Quanto à *veracidade*, trata-se da acurácia dos dados, que são menos controláveis e podem ser irrelevantes. É necessário que se saiba como trabalhar com esses dados através de filtros que garantam dados com maior qualidade.

---

[4]https://www.imdb.com/
[5]https://www.airbnb.com

Por último, mas não menos importante, o *valor* indica que não adianta nada ter um volume massivo de dados se não for possível gerar informação e conhecimento útil deles.

2.2. Bancos de Dados NoSQL

Os BDs NoSQL são uma família de gerenciadores de dados que não seguem o modelo de dados relacional [Sadalage and Fowler 2012]. Eles visam melhor desempenho de acesso, armazenamento de um grande volume de dados, escalabilidade e disponibilidade, tudo isso ao custo de abdicar as tradicionais propriedades ACID (*Atomicity, Consistency, Isolation, Durability*)[Han et al. 2011].

Em vez de as propriedades ACID, os BDs NoSQL se baseiam nas propriedades BASE. Um sistema BASE se preocupa em garantir alta disponibilidade (*Basically Available*) e não precisa estar continuamente consistente (*Soft State*), uma vez que eventualmente um sistema estará no referido estado em momento futuro (*Eventually Consistent*). Destaca-se que geralmente as atualizações são propagadas para todos os nodos de um BD distribuídos conforme o dado é requisitado por este nodo, evitando, assim, um baixo desempenho com operações de atualização e necessidade de atualização síncrona [Da Costa 2017].

Conforme comentando anteriormente, BDs NoSQL são uma família de gerenciadores de dados, não existindo apenas um único modelo de dados utilizado. Eles são organizados em quatro categorias, de acordo com o modelo de dados adotado, podendo ser chave-valor, colunar, documento e grafo [Hecht and Jablonski 2011], que serão detalhadas nas seções seguintes.

2.2.1. Modelo Chave-Valor

BDs do tipo chave-valor são similares a mapas e dicionários, onde o dado é endereçado por uma chave única e não é interpretável pelo sistema. Como não há controle sobre o conteúdo dos dados, questões como relacionamentos e restrições de integridade devem ser tratadas na lógica da aplicação.

Além de suportar o armazenamento em massa, os BDs chave-valor oferecem um alto desempenho em operações de leitura e escrita concorrentes, estando entre os mais populares o Voldemort, DynamoDB e Redis [Hecht and Jablonski 2011].

2.2.2. Modelo Colunar

Os BDs colunares, também chamados de BDs de famílias de colunas, oferecem uma arquitetura altamente escalável. Possuem uma similaridade com BDs chave-valor, porém com um modelo de representação mais complexo. Diferentemente dos BDs relacionais, que são orientados a tuplas, os BDs colunares são orientados a atributos (colunas).

Dados em um BD colunar são acessados por chaves que mapeiam para grupos de valores mantidas em famílias de colunas. Uma família de colunas, como o próprio nome indica, mantém um conjunto de atributos, não sendo obrigatório todas as chaves conservarem o mesmo conjunto fixo de atributos. Logo, cada registro de dados pode ter

seu próprio conjunto de colunas, oferecendo flexibilidade de representação para registros armazenados no BD. Entre os BDs colunares mais conhecidos estão o Cassandra e o HBase [Gessert et al. 2017].

2.2.3. Modelo de Documento

BDs do tipo documento encapsulam pares chave-valor no formato JSON[6] (documento JSON), ou formatos similares a JSON, como XML. Dentro de cada documento as chaves de cada atributo (seus nomes) devem ser únicas, e cada documento possui uma chave especial que também é única no conjunto de documentos.

Diferente do modelo chave-valor, aqui a estrutura dos documentos é conhecida pelo SGBD, ou seja, dados podem ser manipulados pelo SGBD, com a vantagem de suportar tipos de dados.

O número de atributos de um documento não é limitado e novos campos podem ser adicionados dinamicamente a um documento. Um documento pode conter subdocumentos ou mesmo listas de subdocumentos, gerando uma estrutura na forma de árvore. Os BDs populares nessa categoria são o CouchDB e o MongoDB [Da Costa 2017].

Um ponto em comum entre BDs chave-valor, documento e colunar é que todos podem manter dados não normalizados, isto é, dados aninhados dentro de outros dados mais complexos com intuito de facilitar o acesso a dados relacionados sem precisar executar operações de junção. Entretanto, isso pode gerar alta redundância de dados e, como eles não controlam integridade referencial, não há garantias de que os relacionamentos estejam corretos [Hecht and Jablonski 2011].

2.2.4. Modelo de Grafo

Os BDs do tipo grafo podem ser definidos como estruturas onde o esquema e suas instâncias são modeladas como grafos (ver Seção 2.3) ou generalizações de grafos, sendo a manipulação de dados expressa em operações que percorrem grafos. Estão entre os principais BDs deste tipo o Neo4j, GraphDB e FlockDB.

Convém mencionar que esse modelo surgiu juntamente com BDs do tipo orientado a objetos nos anos 80, mas foi deixado de lado com o surgimento de BDs semiestruturados [Angles and Gutierrez 2008]. Contudo, recentemente, a necessidade em gerenciar *Big Data* na forma de grafos fez com que BDs do tipo grafo se tornassem relevantes novamente.

BDs nesta categoria geralmente mantêm grafos direcionados e rotulados, melhor dizendo, dados são organizados em vértices e arestas (relacionamentos) rotuladas. Além disso, tanto os vértices quanto as arestas podem ter propriedadas (atributos) [Angles and Gutierrez 2008]. Ainda, a manipulação de dados é realizada por meio de típicas operações sobre grafos, como navegação através de arestas e noções de vizinhança, subgrafos e conectividade, operações essas que serão descritas na seção seguinte (Seção 2.3).

[6] http://json.org/

Os BDs do tipo grafo se aplicam a áreas onde a informação sobre conexões entre dados e sua topologia é bastante importante ou tão importante quanto os dados em si [Angles and Gutierrez 2008].

2.3. Grafos

Algumas definições acerca de uma estrutura de dados em grafo são necessárias para o desenvolvimento deste trabalho. As definições a seguir foram retiradas do livro eletrônico sobre Teoria de Grafos do Professor Antônio Carlos Mariani, da Universidade Federal de Santa Catarina [Mariani 2018].

Um *Grafo* pode ser definido por dois conjuntos V e A, onde V são os vértices do grafo e A um conjunto de pares pertencentes a V que definem suas arestas. Este grafo também pode ser um dígrafo, que consiste em um grafo orientado onde suas arestas possuem direção.

Outros conceitos associados a uma estrutura de grafo são os seguintes:

- *Adjacência*: dois vértices são ditos adjacentes (vizinhos) se existe uma aresta entre eles.
- *Grau*: o grau de um vértice é dado pelo número de arestas que lhe são incidentes. Um grafo orientado pode ser dividido em grau de emissão e grau de recepção.
- *Cadeia*: é uma sequência qualquer de arestas adjacentes que ligam dois vértices. O conceito de cadeia vale também para grafos orientados, bastando que se ignore o sentido da orientação. Uma cadeia é dita *elementar* se não passa duas vezes pelo mesmo vértice. Por outro lado, é *simples* se não passa duas vezes pela mesma aresta.
- *Caminho*: é uma cadeia na qual todos os arcos possuem a mesma orientação, sendo aplicada, portanto, somente a grafos orientados.
- *Grafo conexo*: quando há pelo menos uma cadeia ligando cada par de vértices deste grafo.

Buscas (ou percorrimentos) podem ser realizadas sobre um grafo. Dentre as principais formas de busca estão a *busca em profundidade* e a *busca em largura*, sendo que ambas partem de um vértice e vão construindo uma árvore de busca. Em uma busca em largura, cada iteração varre todos os nodos adjacentes, e depois os adjacentes dos adjacentes, e assim sucessivamente. Já em uma busca em profundidade, procura-se uma cadeia o máximo que for possível antes de realizar um *backtracking*.

As Figuras 2 e 3 representam possíveis resultados para uma busca em profundidade e uma busca em largura, respectivamente, sobre o grafo da Figura 1.

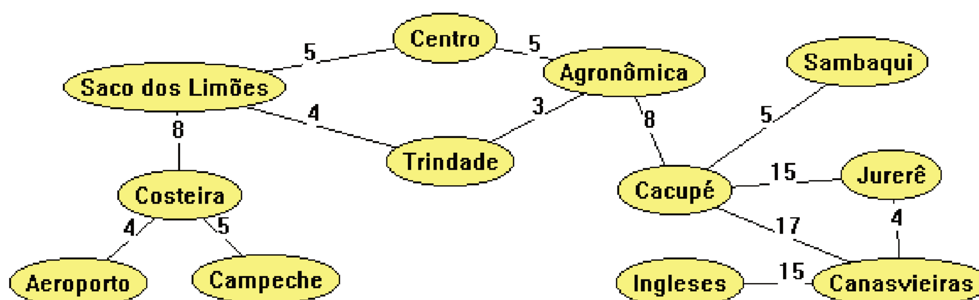

Figura 1. Grafo exemplo com localidades de Florianópolis

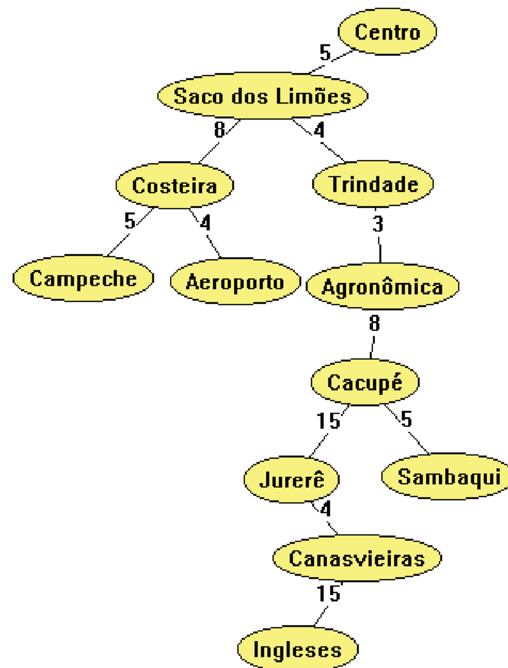

Figura 2. Possível resultado para busca em profundidade a partir do vértice Centro

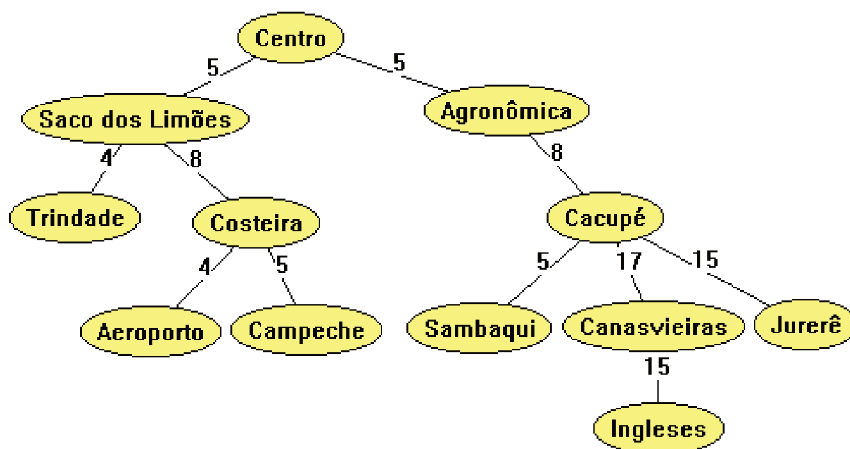

Figura 3. Possível resultado para busca em largura a partir do vértice Centro

2.4. Neo4j

O BD NoSQL do tipo grafo Neo4J foi o escolhido para o desenvolvimento da ferramenta proposta neste trabalho. A escolha se justifica pelo fato deste BD ser altualmente um dos principais representantes desta categoria, bem como por possuir uma vasta gama de artigos e documentação. Além disso, o Neo4J apresenta um modelo de dados mais robusto e flexível que outros BDs similares. A nomenclatura do Neo4j é descrita na Figura 4.

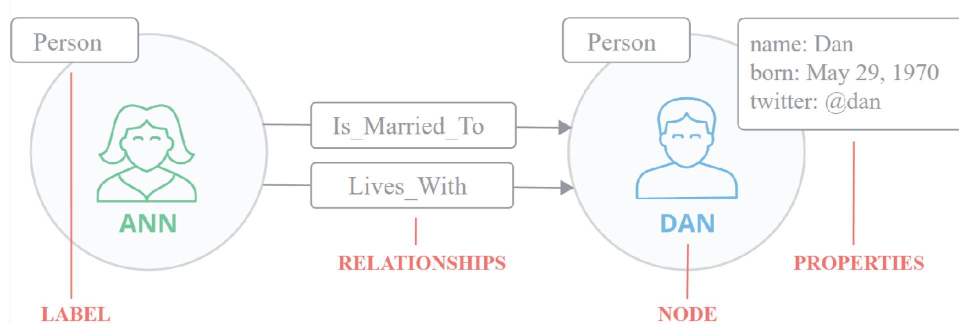

Figura 4. Nomenclatura Neo4j [Neo4j 2018a]

Os conceitos principais do modelo de dados do Neo4j são os seguintes:

- *Vértices*: consiste no elemento principal do modelo de dados do Neo4j. Eles podem ser conectados por meio de *relacionamentos*, ter uma ou mais *propriedades* (atributos guardados como um par chave-valor) e ter um ou múltiplos *rótulos* (identificador do tipo do vértice).
- *Relacionamentos*: conectam dois vértices e são orientados, ou seja, possuem direção. Eles podem apresentar uma ou mais *propriedades*, e também possuem um *tipo*, que tem a mesma função dos rótulos dos vértices, porém no caso de relacionamentos são limitados a um único tipo.
- *Propriedades*: são valores nomeados, em outras palavras, pares chave-valor. A chave sempre é uma *String* e seu valor pode ser um número (*Integer* ou *Float*), uma *String*, um *Boolean*, um tipo espacial *Point*, uma gama de tipos temporais (*Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* e *Duration*). Listas e mapas de tipos simples também são permitidos.

Ainda, o Neo4j utiliza um armazenamento nativo em grafo. É também chamado de *index-free adjacency*, onde cada vértice aponta fisicamente para sua localização na memória, melhorando o desempenho de acesso [Neo4j 2018b].

A linguagem de consulta do Neo4j é a *Cypher*, a qual é inspirada na linguagem SQL e adota o conceito de combinação de padrões da linguagem SPARQL[7]. Cypher descreve os vértices, os relacionamentos e as propriedades como se formassem um desenho utilizando caracteres ASCII, tornando, assim, as consultas mais fáceis de ler e entender [Cyp 2018].

2.5. JSON *Schema*

Antes de apresentar JSON *Schema* é necessário se ter o conhecimento do que é JSON. JSON (*JavaScript Object Notation*) é um formato leve para troca de dados, sendo de fácil entendimento e leitura por humanos e também de simples manipulação por máquinas. Trata-se de um formato de texto completamente independente de linguagem, mas que usa convenções familiares com diversas linguagens de programação como C, C++, C#, *Java*, *JavaScript*, *Perl* e *Python*. Essas propriedades fazem com que JSON seja ideal para troca de informações[8].

[7] https://www.w3.org/TR/sparql11-query/
[8] https://www.json.org/

JSON é projetado sobre duas estruturas universais, ou seja, estruturas suportadas por todas as linguagens de programação modernas. São elas:

- Uma *coleção de pares chave-valor*, também conhecida nas linguagens de programação como *record*, *struct*, *hash table* ou *dictionary*.
- Uma *lista ordenada de valores*, também denominada nas linguagens de programação como *array*, *vector* ou *list*.

Com base nessas estruturas, uma representação no formato JSON apresenta os seguintes conceitos básicos, conforme ilustram as Figuras 5 e 6:

- Um *objeto (Object)*, que é uma sequência não ordenada de chave-valor. É definido entre chaves ('{', '}'), sendo os pares chave-valor internos separados por vírgula.
- Um *valor*, que pode ser do tipo *String*, *Number*, *Object*, *Array*, *Boolean* ou *Null*.

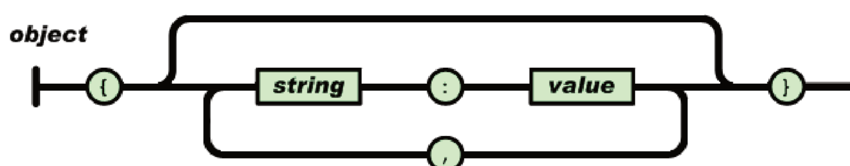

Figura 5. Especificação de um objeto JSON

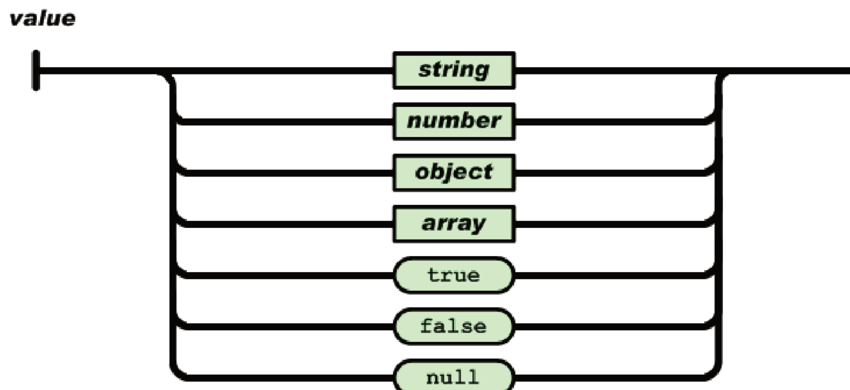

Figura 6. Especificação de um valor JSON

Atualmente, JSON tem um papel chave no desenvolvimento de aplicações. Um programa que execute funções de forma remota precisa estabelecer um protocolo preciso de comunicação para receber dados e respondê-los, o que é chamado de API (*Application Programming Interface*). Como JSON é uma linguagem facilmente compreendida por humanos e máquinas, acabou se tornando o formato mais popular para se enviar pedidos a APIs por intermédio do protocolo HTTP [Pezoa et al. 2016].

Por exemplo, considere a requisição para um *web service* que deseja saber o clima de um determinado local. Uma chamada hipotética a essa API conteria os seguintes dados em formato JSON:

```
{"Country": "Chile", "City": "Santiago"}
```

A requisição está solicitando o clima para o país *Chile*, na cidade de *Santiago*. Assim sendo, a API poderia responder com o seguinte JSON:

```
{"timestamp": "14/10/2015 11:59:07",
 "temperature": 25, "Country": "Chile",
 "City": "Santiago", "description": "Sunny"}
```

A API então responde indicando que a temperatura para aquela data é de 25 graus Celsius e que o dia está ensolarado. O exemplo mostra a simplicidade e legibilidade do formato JSON.

Em diversos cenários deste tipo é possível se beneficiar de uma forma declarativa de especificar um esquema, isto é, uma forma de definir como a requisição deve ser formatada para evitar chamadas mal formadas para a API. Desta maneira, utiliza-se o JSON *Schema* para validar expressões no formato JSON [Pezoa et al. 2016].

JSON *Schema* é uma linguagem que permite ao usuário restringir a estrutura de documentos JSON e prover um *framework* com a finalidade de verificar a integridade das requisições de acordo com a API.

```
{
    "type": "object",
    "properties": {
        "Country": {"type": "string"},
        "City": {"type": "string"},
    },
    "required": ["Country", "City"]
    "additionalProperties": false
}
```

Figura 7. Exemplo de especificação em JSON *Schema*

A Figura 7 mostra como seria um JSON *Schema* para averiguar a requisição de clima. Ela indica que a requisição deve conter um objeto com as chaves *"Country"* e *"City"*, do tipo *String*, pois estão dentro da chave *"properties"*. Além disso, ambas são obrigatórias, visto que estão compreendidas na lista da chave *"required"*. Também indica que não é possível adicionar propriedades adicionais.

Na recomendação JSON *Schema*, além das chaves *"properties"* e *"required"*, também são utilizadas neste trabalho as chaves *"definitions"* e *"allOf"*, assim como o tipo {*"type": "array"*}, que possuem algumas construções diferentes.

A chave *"definitions"* permite o reuso de uma estrutura complexa dentro do esquema. Ela pode ser utilizada aliada à chave *"$ref"*, que permite referenciar esquemas ou estruturas presentes em outros arquivos. Quando utilizado com o sufixo #, a chave *"$ref"* referencia dentro do próprio arquivo.

Na Figura 8 é utilizado o *"$ref"* dentro de *"definitions"* para criar uma definição local do tipo *"host"*, que aponta para um arquivo externo. Na seção *"properties"* é definido, então, uma propriedade do tipo *"host"* que indica a definição criada em *"definitions"*, com o uso do sufixo #.

```
{
    "definitions": {
        "host": {
            "type": "object",
            "\$ref": "hosts.json"
        },
    },
    "type": "object",
    "properties": {
        "host": {
            "type": "object",
            "\$ref": "#/definitions/host"
        },
    }
}
```

Figura 8. Exemplo do uso das chaves *"definitions"* e *"$ref"*

A chave *"allOf"* é usada aliada também a *"$ref"*. Dessa forma, é possível estabelecer uma estrutura similiar à herança presente nas linguagens de programação orientadas a objetos. Na Figura 9, por exemplo, o esquema, além de possuir a propriedade *job*, também conterá o esquema definido para *person.json*.

```
{
    "type": "object",
    "allOf": [
        {
            "\$ref": person.json
        }
    ]
    "properties": {
        "job": {
            "type": "string"
        },
    }
}
```

Figura 9. Exemplo do uso da chave *"allOf"* para estruturar uma herança

O tipo {*"type": "array"*} pode ter seus itens descritos de duas formas: uma com {} e outra com [ ]. A definição com chaves se refere à uma lista, enquanto a definição com colchetes à tuplas. A chave *"minItems"* ainda pode ser utilizada para especificar um tamanho mínimo, para, por exemplo, não permitir listas vazias.

Na Figura 10 é criado um *array* com nome *relacionamentos*, que precisa ter no mínimo um item. Portanto, não pode ser vazio. Com a chave *oneOf* é especificado que os itens deste *array* devem ser ou do tipo *number* ou uma tupla $(number, string)$.

```
{
    "relacionamentos": {
        "type": "array",
        "items": {
            "oneOf": [
                {
                    "type": "array",
                    "items": [
                        {
                            "type": "number",
                        },
                        {
                            "type": "string",
                        }
                    ]
                },
                {
                    "type": "number",
                }
            ]
        },
        "minItems": 1
    }
}
```

Figura 10. Exemplo do tipo {*"type": "array"*}

No momento, o JSON *Schema* está em seu oitavo *draft*, também chamado de 2019-09, publicado em Setembro de 2019.

3. Trabalhos Relacionados

Esta seção detalha trabalhos que possuem algum vínculo com a proposta deste trabalho. Inicialmente, apresenta-se uma proposta de extração de esquemas de BDs NoSQL do tipo documento, que serviu de inspiração para este artigo [Da Costa 2017]. Na sequência, descreve-se um trabalho que mapeia um modelo entidade-relacionamento para um BD NoSQL do tipo grafo a fim de verificar restrições nos vértices e arestas [Sousa and del Val Cura 2018]. Ainda, demonstram-se os trabalhos de Roy-Hubara et al. [Roy-Hubara et al. 2017], que, similar ao anterior, cria um esquema a partir de um diagrama entidade-relaciomento, e de Bonifati et al. [Bonifati et al. 2019], que propõe uma forma de validar e evoluir um esquema de BD do tipo grafo a partir de uma DDL. Por fim, descreve-se os recursos que os principais SGBDs orientados a grafos (OrientDB[9] e Neo4j) possuem quanto à geração de esquemas.

Até o momento da escrita deste documento não foram encontrados trabalhos sobre extração de esquemas especificamente para BDs do tipo grafo. Apenas, de uma forma

[9]https://orientdb.com/

geral, trabalhos que extraem esquemas de dados semiestruturados, como em Nestorov et al. [Nestorov et al. 1998].

Para a pesquisa por trabalhos relacionados foi utilizado o motor de busca DBLP[10], que é um dos principais indexadores de conferências e periódicos na área de Ciência da Computação. A *string* de busca utilizada foi *graph$ database — design — extract — schema*. O símbolo $ foi utilizado para requerer a palavra *graph* sem os seus derivados, como *graphical*. Os caracteres de espaço simbolizam o operador lógico *AND* e os caracteres — simbolizam o operador lógico *OR*.

A pesquisa resultou em 109 artigos que foram submetidos a 3 etapas de filtragem. A primeira etapa excluiu os artigos publicados antes de 2017, restando 44 artigos. A segunda etapa consistiu na leitura do título e do resumo e foram filtrados 12 artigos pertinentes e, dentre eles, selecionados os 3 artigos descritos nas próximas seções, com exceção do trabalho de Costa [Da Costa 2017], que foi desenvolvido como trabalho de conclusão de curso no âmbito do Grupo de Banco de Dados da UFSC[11].

3.1. Uma Ferramenta para Extração de Esquemas de Bancos de Dados NoSQL Orientados a Documentos

Este trabalho propõe um algoritmo para extrair um esquema no formato JSON *Schema* a partir de um BD NoSQL do tipo documento [Da Costa 2017]. Seu propósito é similar ao do trabalho, porém o foco é diferente, uma vez que neste a essência consiste em BDs do tipo grafo.

O autor utiliza JSON *Schema* como formato de saída, por JSON ter se tornado um formato padrão para intercâmbio de dados e por ter sido utilizado também em seus trabalhos relacionados. O trabalho propõe uma aplicação *Web* que, a partir de uma coleção de documentos JSON, extrai o seu esquema. Para isso, seu algoritmo de extração percorre todos os documentos JSON armazenados e analisa suas propriedades para identificar o esquema bruto de cada documento e, então, unifica os esquemas brutos e gera um JSON *Schema* único que representa a coleção de documentos como um todo.

O algoritmo de extração pode ser dividido em quatro etapas:

1. A obtenção do esquema bruto dos documentos;
2. O agrupamento dos esquemas brutos iguais;
3. A unificação dos esquemas brutos;
4. A obtenção do JSON *Schema*.

Na primeira etapa é criado um esquema bruto, onde são percorridos todos os documentos JSON, mantendo sua estrutura em relação à campos, objetos aninhados e *arrays*, sendo cada valor primitivo substituído pelo seu tipo de dado JSON. Já na segunda etapa, são realizadas operações de agregação para extrair um número mínimo de objetos necessários para executar o processo de unificação de esquemas brutos, agrupando os documentos que possuem um mesmo esquema bruto.

Por sua vez, a terceira etapa do algoritmo produz uma estrutura temporária denominada estrutura unificada de esquema bruto (RSUS). Ela armazena a estrutura

[10]https://dblp.uni-trier.de/

[11]http://lisa.inf.ufsc.br/wiki/index.php/Main

hierárquica de cada esquema bruto, contendo informações a respeito dos atributos dos objetos, seus tipos de dados e o caminho da raiz até a propriedade ou item dentro do documento. A RSUS é dividida em cinco propriedades: *field* que representa um atributo, *primitiveType* que representa um tipo primitivo JSON, *extendedType* representa um tipo de dado JSON estendido, *objectType* que representa um objeto JSON contendo os atributos e *arrayType* que representa uma lista. Por fim, na quarta etapa, cada propriedade da RSUS é mapeada para o seu equivalente em JSON *Schema*.

Algumas ideias desta abordagem contribuíram para o presente trabalho, como o uso de JSON *Schema* como formato de saída do processo de extração e a divisão em etapas do processo de extração desenvolvido.

3.2. *Logical Design of Graph Databases from an Entity-Relationship Conceptual Model*

Este trabalho apresenta um mapeamento de um modelo conceitual entidade-relacionamento para um modelo lógico estendido para BDs do tipo grafo. O projeto de BDs tradicionais geralmente segue três etapas de modelagem (conceitual, lógica e física), sendo que o modelo entidade-relacionamento é utilizado na primeira etapa e o objetivo do trabalho é produzir um esquema lógico para BDs NoSQL de grafo.

O modelo entidade-relacionamento considerado é uma forma simplificada com três componentes: as entidades, as relações e os atributos. Já o modelo lógico de grafo utilizado consiste em seis componentes: os vértices rotulados, as arestas rotuladas, um conjunto de propriedades para o vértice, um conjunto de propriedades para a aresta, um cojunto de restrições para os atributos dos vértices e um conjunto de restrições para as arestas. Por sua vez, as restrições consideradas são do tipo única, obrigatória, existente ou restrita, tanto para vértices quanto para arestas.

O algoritmo possui uma função geral que chama outras funções para verificar as restrições de vértices, arestas e restrições de cardinalidade, sendo dividido em três etapas. Na primeira delas, é criado um vértice para cada entidade e uma aresta para cada relação entre entidades. Na segunda, é analisada a cardinalidade dessas relações com a finalidade de verificar se esta aresta deve ser restrita ou obrigatória. E, na terceira etapa, são verificadas as restrições para cada uma das propriedades dos vértices e das arestas.

O trabalho mostra, como caso de uso, um sistema de *streaming* de música, onde a partir de um modelo entidade-relacionamento foi produzido um design lógico exemplificando as restrições citadas.

Ideias desta abordagem que auxiliaram este trabalho:

- as restrições utilizadas para modelar os vértices e arestas serviram de base para a análise dos dados físicos do BD NoSQL do tipo grafo;
- a utilização de engenharia reversa para, a partir do modelo lógico já existente, obter um modelo entidade-relacionamento.

3.3. *Modeling Graph Database Schema*

Assim como o trabalho da seção anterior, este propõe um processo que, a partir de um modelo entidade-relacionamento, obtém um esquema para um BD do tipo grafo. Seu diferencial, como o próprio autor descreve, é que o modelo conceitual considerado é mais

completo, contemplando entidades fracas, relações ternárias e generalizações, entre outras características que não são abordadas em trabalhos similares.

O trabalho se preocupa em manter as restrições nas propriedades e nas relações existentes através de restrições. Desse modo, o esquema gerado possui restrições de cardinalidade, algo que não é suportado pelos BDs do tipo grafo atuais.

O processo de criação do esquema é dividido em duas etapas, sendo uma delas onde as estruturas mais complexas do modelo conceitual são transformadas em outras mais simples, e outra onde cada entidade é transformada em um vértice, cada relação em uma aresta e as restrições vão sendo implementadas.

Algumas ideias do trabalho proposto que contribuíram para este são:

- A forma como foram tratadas as estruturas mais complexas, como relacionamentos ternários;
- A proposta de definir restrições na cardinalidade das arestas do grafo.

3.4. *Schema Validation and Evolution for Graph Databases*

Os dois últimos trabalhos apresentados visam criar um esquema lógico com restrições do domínio. Já este trabalho afirma que BDs NoSQL no geral são atraentes por não possuírem esquema, não fazendo sentido incluir restrições, tampouco perder a flexibilidade.

O esquema é dividido em dois espectros extremos: o *descritivo* e o *prescritivo*. No primeiro deles, o esquema consiste nos dados em si, apenas refletindo os dados que estão ali inseridos, não havendo nenhum tipo de restrição imposta sobre eles. Por sua vez, o segundo é onde existe toda uma série de restrições e validações em cima do grafo, de modo que qualquer modificação nos dados do BD siga as regras do esquema.

Dessa maneira, o trabalho propõe:

- Um modelo de esquema onde é suportado os dois extremos do espectro de forma flexível, podendo transitar de um para o outro ou então utilizá-lo de uma forma hibrída;
- Uma DDL para este modelo de esquema;
- Um framework matemático para validação de uma instância do BD por meio de um homomorfismo com um grafo de esquema;
- Especificações matemáticas para evolução do esquema e como propagar essas mudanças.

O modelo de grafo é definido como uma tupla (N, E, n, P, v, M), onde N é um conjunto de tipos de vértices, E é um conjunto de tipos de arestas, n é um conjunto de funções que ligam um vértice origem a um vértice destino através de uma aresta $E \to N \times N$, P é um conjunto de propriedades, v é uma relação finita que atribui um conjunto de valores às propriedades e M é um conjunto de propriedades obrigatórias.

Assim sendo, a saída produzida pela ferramenta proposta neste trabalho se assemelha bastante ao modelo definido por esta abordagem de Bonifati et al. [Bonifati et al. 2019]. Por exemplo: a saída da ferramenta define também um conjunto de propriedades e um conjunto de funções que ligam dois vértices.

3.5. Esquemas Neo4j e OrientDB

O Neo4j e OrientDB são os principais SGBDs para BDs do tipo grafo. Ambos possuem alguma forma de respresentação de seu esquema mas que possuem alguma lacuna.

O Neo4j contém a função *db.schema()* mas esse esquema apenas apresenta os rótulos dos vértices e seus possíveis tipos de relacionamentos, não tem nenhuma informação referente as propriedades de seus vértices e relacionamentos nem quanto a sua obrigatoriedade ou ao seus tipos.

No OrientDB é possível realizar algumas consultas em seus metadados com a *query select expand(classes) from metadata:schema* e por conseguinte *select expand(properties) from* { *select expand(classes) from metadata:schema* } *where name = 'className'* para trazer informações sobre os rótulos de vértices e suas propriedades, mas sem nenhuma informação sobre os relacionamentos existentes.

Sendo assim este presente trabalho busca trazer o melhor de cada uma dessas duas formas de extração de esquema.

4. Ferramenta Para Extração de Esquemas de Bancos de Dados NoSQL de Grafo

A partir dos estudos realizados acerca dos trabalhos relacionados, onde foram apresentadas abordagens que modelam esquemas para BDs do tipo grafo e um trabalho que extrai o esquema de um BD do tipo documento, que foi inspiração para este trabalho, uma ferramenta para extração de esquemas de BDs de grafo foi desenvolvida.

Ao longo deste capítulo apresenta-se o projeto da ferramenta e o processo de extração de esquemas.

4.1. Projeto

A ferramenta é uma aplicação local para extração de esquemas, no formato JSON *Schema*, de um conjunto de vértices e arestas armazenados em um BD do tipo grafo. O projeto da ferramenta iniciou-se pelo levantamento de requisitos, os quais representam a descrição das necessidades dos usuários perante um *software*. Os requisitos de um *software* podem ser classificados em Requisitos Funcionais (RF) e Requisitos Não-Funcionais (RNF) [Sommerville 2011].

Os RFs são responsáveis por definir o comportamento do *software*, como o sistema deve reagir a entradas específicas e se comportar em determinadas situações. Por sua vez, os RNFs são relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenção e tecnologias envolvidas. Os referidos requisitos dizem respeito a como as funcionalidades dos RFs serão entregues ao usuário do *software*.

Logo, foram definidos os seguintes RFs e RNFs:

- RF01: Selecionar o BD de grafo;
- RF02: Extrair esquema;
- RF03: Salvar o esquema extraído;
- RNF01: Utilizar linguagem de programação *Python*[12], devido ao seu fácil trabalho com dicionários, mapas, listas e com o formato JSON;

[12]https://www.python.org/

- RNF02: Usar o SGBD Neo4j, por sua ampla comunidade e documentação, e também pelo fato de ser o BD do tipo grafo mais utilizado no mundo, segundo o site *db-engines*[13];
- RNF03: Extrair o esquema para um formato canônico em JSON *Schema*[14] para fins de padronização e integração com outros trabalhos.

Por mais que a estrutura de grafo não se assemelhe a um documento JSON, como utilizado no trabalho do Costa [Da Costa 2017], foi decidido o formato JSON *Schema* não para fins de validação, mas no sentido de informar o esquema de dados implícito dentro de um BD de grafo. Desta forma, no futuro pode-se utilizar este esquema como um intermediário em um processo de consulta sobre dados no formato JSON e seu posterior mapeamento para uma consulta específica sobre um BDs de grafo.

A linguagem de programação empregada foi a Python, que se trata de uma linguagem multiparadigma: imperativo, procedural, funcional e orientado a objetos. Para o desenvolvimento da ferramenta foi utilizado principalmente o paradigma funcional, nele o resultado das funções dependem apenas de seus argumentos, ou seja, chamar uma função com os mesmos argumentos sempre produz o mesmo resultado [Grover 2019].

Foi ainda usada a noção de classes, que vem do paradigma de orientação a objetos, para organizar as funções e redirecionar a saída de uma função como entrada de outra. A organização das classes é mostrada na Figura 11. O diagrama não representa nenhum padrão da UML (*Unified Modeling Language*)[15]. Ele é apenas uma representação de quais classes possuem acesso às funções de outras classes.

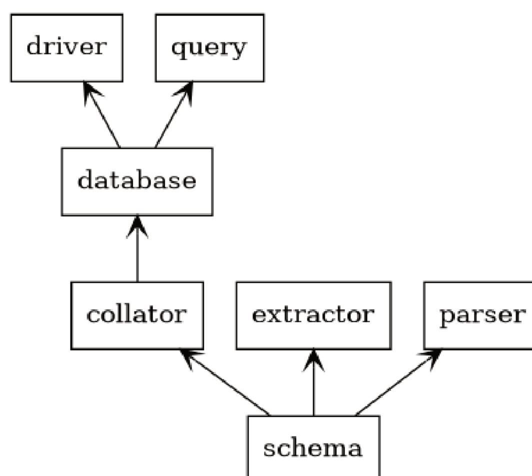

Figura 11. Organização das funções da ferramenta

Enquanto a classe *driver* é responsável por executar as transações no BD, a classe *query* armazena todas as *queries* que serão necessárias no processo de extração. Já a classe *database* funciona como um intermediador, fazendo com que o *driver* execute uma determinada *query* e, posteriormente, retornando o resultado para classe *collator*.

[13]https://db-engines.com/
[14]http://json-schema.org/
[15]https://www.uml.org/

A classe *collator* possui os algoritmos descritos no capítulo 4.2.1, onde são reunidas as propriedades e relacionamentos dos vértices. A classe *extractor* possui os algoritmos mencionados no capítulo 4.2.2, os quais são responsáveis por tratar os casos de vértices com mútiplos rótulos. A classe *parser* é descrita no capítulo 4.2.3 e é encarregada de mapear os dados intermediários dos algoritmos anteriores na saída final em JSON *Schema*. Por fim, a classe *schema* é responsável por gerenciar e redirecionar as saídas do *collator* para o *extractor* e dele para o *parser*.

Deste modo, a ferramenta recebe como entrada o IP, porta, usuário e senha do BD Neo4j, bem como um diretório para onde será gerada a saída. Por não possuir interface gráfica, essas entradas são passadas a ela por meio de uma interface de linha de comando. A partir disso, a ferramenta produz como saída o(s) arquivo(s) JSON *Schema* correspondente(s) no diretório especificado. O código produzido para a ferramenta pode ser encontrado no Apêndice A ou no repositório do GitHub[16].

4.2. Processo de Extração de Esquemas

A extração de esquemas de BDs do tipo grafo envolve a descoberta de tipos de vértices e de arestas, bem como suas propriedades. Assim sendo, o processo proposto é dividido em três etapas, conforme mostra a Figura 12. O processo recebe como entrada um grafo de dados de um BD Neo4j e produz um esquema para esse grafo de dados no formato JSON *Schema*.

A primeira etapa realiza um *agrupamento* (*Collator*). Todos os vértices são percorridos, analisando e agrupando suas propriedades e relacionamentos de acordo com o rótulo do vértice. Na segunda etapa (*Extractor*), todas as arestas são percorridas, verificando suas propriedades e agrupando de acordo com o tipo de aresta. Na terceira etapa (*Parser*), um conjunto de documentos JSON *Schema* é gerado para essas informações agrupadas (sumarizadas) do esquema dos dados do BD Neo4j.

O BD Neo4j apresenta uma peculiaridade, se comparado com outros BDs de grafo: ele permite definir *múltiplos rótulos* por vértice. Assim sendo, é realizada a segunda etapa de *extração*, onde sobre o agrupamento de vértices é realizada uma extração a fim de obter apenas vértices de rótulo único. Esta extração é efetuada com a finalidade de identificar a qual rótulo específico uma determinada propriedade ou relacionamento está atrelado. Ela é executada através de operações de intersecção e de diferença entre conjuntos, sendo detalhada na Seção 4.2.2.

Por sua vez, a terceira e última etapa realiza o *mapeamento* do agrupamento já extraído para o formato JSON *Schema*. Os tópicos seguintes detalham as três etapas.

[16]https://github.com/MaoRodriguesJ/neo4j-schema-extract-code

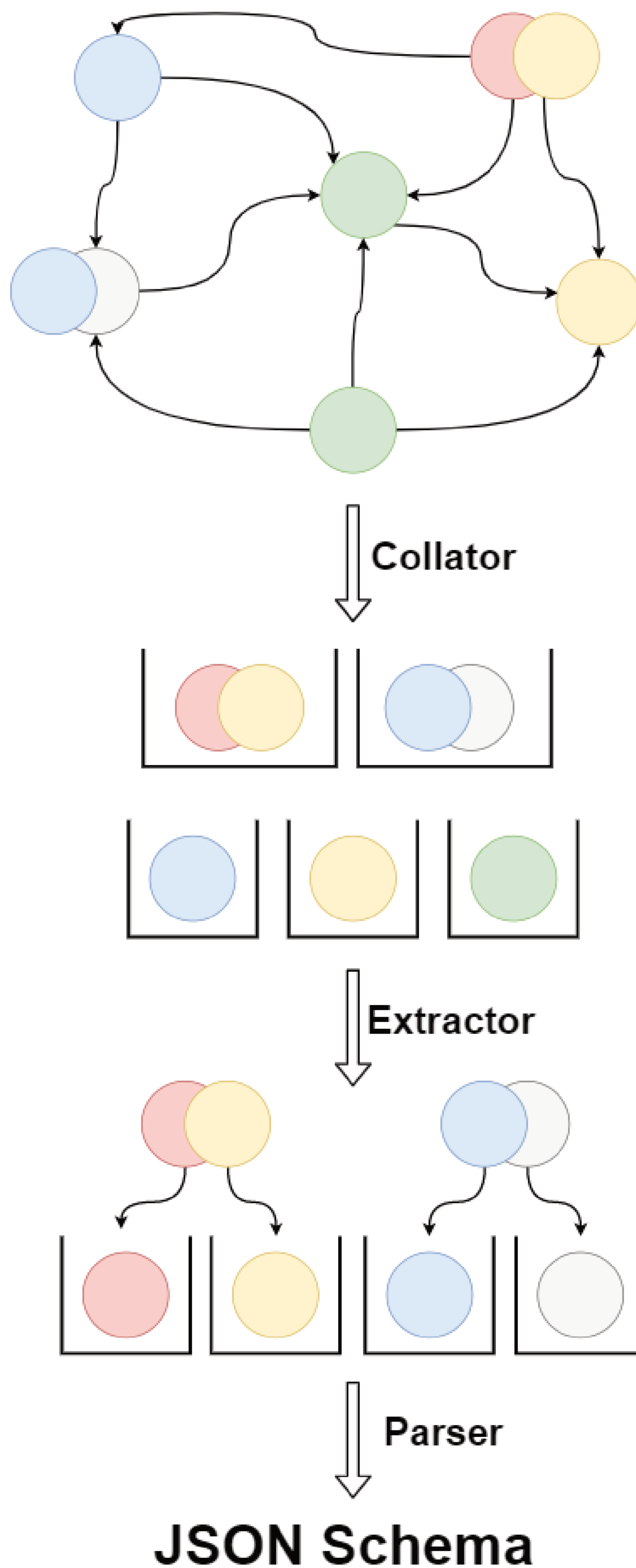

Figura 12. Visão geral da ferramenta

4.2.1. Agrupamento

Esta primeira etapa do processo de extração gera dois dicionários distintos. Um deles descreve os vértices, suas propriedades e relacionamentos (*Dicionário de Vértices*), e o outro os relacionamentos e suas propriedades (*Dicionário de Relacionamentos*). A Figura 13 mostra a estrutura do primeiro dicionário e a Figura 14 a do segundo.

```
"rótulos do vértice": {
    "propriedades": [
        {
            "(nome da propriedade, tipo da propriedade)":
            "obrigatoriedade"
        }
    ],
    "relacionamentos": [
        {
            "(tipo do relacionamento, rótulo do vértice destino)":
            "obrigatoriedade"
        }
    ]
}
```

Figura 13. Dicionário de vértices

```
"tipo do relacionamento": {
    "propriedades": [
        {
            "(nome da propriedade, tipo da propriedade)":
            "obrigatoriedade"
        }
    ]
}
```

Figura 14. Dicionário de relacionamentos

O Algoritmo 1 apresenta a função que realiza o agrupamento dos vértices e gera a estrutura do *Dicionário de Vértices*. Para isso, todos os vértices são percorridos a fim de agrupar suas propriedades e relacionamentos, definindo seus tipos e obrigatoriedades. Tal estrutura é armazenada na variável *grouping* da linha 4.

Para exemplificar essa etapa será utilizado como entrada um grafo como o da figura 15.

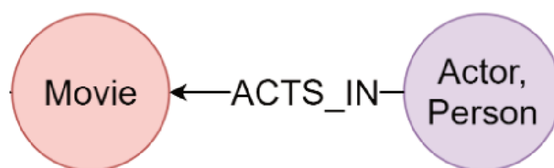

Figura 15. Grafo exemplo

O referido algoritmo aplica a função *Powerset* sobre uma lista de rótulos de vértices (linha 3). A função mencionada produz um conjunto com todos os possíveis subconjuntos dos rótulos de vértices da lista, como por exemplo:

$Powerset([Person, Actor, Movie]) \longrightarrow \{(),\ (Person),\ (Actor),\ (Movie),$ $(Person, Actor),\ (Person, Movie),\ (Actor, Movie),\ (Person, Actor, Movie)\}$

A função *Powerset* é utilizada como um facilitador que permite definir *queries* mais específicas sem a necessidade de posteriormente verificar o(s) rótulo(s) do vértice analisado para identificar a qual agrupamento o vértice pertence, visto que um vértice pertence sempre a um único conjunto de rótulos no powerset.

Algorithm 1: Agrupamento de Vértices

Function *grouping_nodes*:

Output: Dictionary with grouped nodes, their properties and relationships

```
1  begin
2      labels ⟵ list of nodes types;
3      labels_powerset ⟵ Powerset(labels);
4      grouping ⟵ {};
5      for label_combination ∈ labels_powerset do
6          records ← list of nodes with label combination;
7          key ← (label_combination, records size);
8          grouping[key] ⟵ {};
9          for record ∈ records do
10             Process_properties(grouping, key, record);
11             Populate_relationships(grouping, key, record id);
12         end
13     end
14     Check_mandatory_properties(grouping);
15     return grouping
16 end
```

Dentro do primeiro laço de iteração do Algoritmo 1 (linha 5) é feita uma consulta no BD para buscar todos os vértices que possuem uma das combinações de rótulo do *Powerset*, sendo esses vértices armazenados na lista *records* (linha 6). A váriavel *key* (linha 7) armazena uma tupla com a combinação de rótulos que está sendo iterada e a quantidade de vértices que possuem aquela combinação, utilizando o tamanho da lista

records. Essa quantidade de vértices é usada para determinar se uma propriedade é obrigatória.

Na linha 8 é inicializado um novo dicionário vazio dentro de *grouping* para a chave definida. Ele manterá as propriedades e relacionamentos dos vértices de uma determinada combinação de rótulos, para esta etapa será considerada a combinação {*Actor, Person*}. Antes do próximo laço de iteração (linha 9), o dicionário *grouping* possui uma estrutura similar ao exemplo a seguir, onde existem 200 vértices que apresentam os rótulos *Actor* e *Person*:

```
{ ({Actor, Person}, 200) : {} }
```

Cada vértice da lista *records* é iterado e passa por duas funções. A função *Process_properties* (linha 10) calcula quantos vértices possui determinada propriedade e a função *Populate_relationships* (linha 11) contabiliza quais relacionamentos partem dos vértices com aquela combinação de rótulos.

Ao final (linha 14), a função *Check_mandatory_properties* verifica se uma propriedade é obrigatória. Para inferir essa obrigatoriedade, a função compara o número de ocorrências de uma determinada propriedade, que foi registrado anteriormente pelo Algoritmo 2, com o valor de *records size* armazenado na chave definida na linha 7 do Algoritmo 1. Se os valores forem iguais, a propriedade é considerada obrigatória, pois está presente em todos os vértices.

Essas funções são utilizadas também pelo processo de agrupamento de relacionamentos (Algoritmo 4), que possui entedimento similar ao do Algoritmo 1, diferindo apenas pelo fato que a iteração é feita sobre relacionamentos e não vértices, e a sua saída segue a estrutura do *Dicionário de Relacionamentos*.

Algorithm 2: Contabilizar propriedades

Function *Process_properties(grouping, key, record)*:

Result: Number of occurrences of each property from a type of node

1 **begin**
2 | **if** $"properties" \notin grouping[key]$ **then**
3 | | $grouping[key]["properties"] \longleftarrow \{\}$;
4 | **end**
5 | **for** $property \in record.properties$ **do**
6 | | $prop_key \leftarrow$ (property name, property type);
7 | | **if** $prop_key \in grouping[key]["properties"]$ **then**
8 | | | $grouping[key]["properties"][prop_key] + +$;
9 | | **else**
10 | | | $grouping[key]["properties"][prop_key] \leftarrow 1$;
11 | | **end**
12 | **end**
13 **end**

No Algoritmo 2, para contabilizar as propriedades, é criada na linha 3 uma nova entrada no dicionário com a chave *"properties"*, dentro da chave passada como parâmetro

key, que representa o(s) rótulo(s) do vértice. Posteriormente, itera-se sobre as propriedades do vértice, que é um dos parâmetros de *record* (linha 5). Dentro do laço, na linha 6, é criada uma tupla com o nome da propriedade a ser iterada e o seu tipo. Na estrutura condicional (linhas 7 e 11), verifica-se se esta propriedade já foi encontrada em outro vértice com a mesma combinação de rótulos para ou aumentar sua quantidade de ocorrências ou inicializá-la em 1.

A váriavel *grouping* do Algoritmo 1, após passar pelo processamento do Algoritmo 2 para todos os vértices de uma determinada combinação de rótulos, terá uma estrutura similar à apresentada a seguir. Uma das propriedades dos vértices de rótulo múltiplo *Actor* e *Person* é *name*, do tipo *str*, que ocorre em 200 dos 200 vértices com esses rótulos:

```
{
    ({Actor, Person}, 200) : {
        "properties" : {
             ('name', <class 'str'>): 200
        }
    }
}
```

No Algoritmo 2 é contado quantas ocorrências uma determinada propriedade teve para verificar se ela é obrigatória ou não. O mesmo não pode ser feito para averiguar se um relacionamento é obrigatório ou não, pois um vértice pode ter infinitos relacionamentos.

Assim sendo, para realizar essa contabilidade (Algoritmo 3), é assumido, para o primeiro vértice analisado, que todos os seus relacionamentos são obrigatórios e, a partir de então, vai se ajustando esta restrição à medida que outros vértices são analisados. O Algoritmo 3 inicialmente consulta no BD (linha 2) por uma lista de relacionamentos que partem de vértices de um determinado identificador único passado como parâmetro (*id*) e armazenado na váriavel *relationships*.

Caso seja o primeiro vértice analisado, entra-se na condição da linha 3 e executa-se as linhas 4 a 8. Uma nova entrada definida no dicionário com a chave *"relationships"*, dentro da chave passada como parâmetro *key*. Para cada relacionamento é criada uma chave (linha 6) com o tipo do relacionamento e quais são os rótulos do vértice destino. Esta chave possui o valor *True*, para indicar que o relacionamento é obrigatório.

Por consequência, os próximos vértices analisados irão para a linha 10 da estrutura condicional, onde serão armazenadas as chaves que já existiam em uma váriavel auxiliar *old_keys* (linha 11). Dentro do laço da linha 12 é verificado se o relacionamento analisado atualmente já existia. Caso ele ainda não tenha sido analisado, isso quer dizer que existem outros vértices com aquele rótulo que não possuem aquele relacionamento. Assim sendo, sua obrigatoriedade passa a ser *False* (linha 16). É necessário ainda examinar se os relacionamentos anteriormente descobertos existem nesse vértice a ser analisado, caso não existam, também se tornam não obrigatório (linhas 19 a 23).

Algorithm 3: Popular relacionamentos

Function *Populate_relationships(grouping, key, id)*:

Result: Populate with relationships and if they are mandatory or not

1 **begin**
2   $relationships \longleftarrow$ list of relationships that flow out from the node with a given id;
3   **if** $"relationships" \notin grouping[key]$ **then**
4     $grouping[key]["relationships"] \longleftarrow \{\}$;
5   **for** $relationship \in relationships$ **do**
6       $relationship_key \leftarrow$ (relationship type, relationship reached node types);
7       $grouping[key]["relationships"][relationship_key] \leftarrow True$;
8   **end**
9   **else**
10     $relationships_keys \leftarrow \{\}$;
11     $old_keys \leftarrow grouping[key]["relationships"].keys$;
12   **for** $relationship \in relationships$ **do**
13       $relationship_key \leftarrow$ (relationship type, relationship reached node types);
14       $relationships_keys \leftarrow relationships_key.add(relationship_key)$;
15   **if** $relationship_key \notin old_keys$ **then**
16         $grouping[key]["relationships"][relationship_key] \leftarrow False$;
17   **end**
18   **end**
19   **for** $key \in old_keys$ **do**
20   **if** $key \notin relationships_keys$ **then**
21         $grouping[key]["relationships"][key] \leftarrow False$;
22   **end**
23   **end**
24   **end**
25 **end**

Ao final da execução do Algoritmo 1 obtém-se um dicionário de vértices, conforme o seguinte exemplo:

```
{
    {Actor, Person} : {
        "properties" : {
             ('name', <class 'str'>) : True
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        }
    }
}
```

Algorithm 4: Agrupamento de relacionamentos

Function *grouping_relationships*:

Output: Dictionary with grouped relationships and their properties

```
1  begin
2  │  types ⟵ list of relationships types ;
3  │  grouping ⟵ {};
4  │  for type ∈ types do
5  │  │  records ← list of relationships of type;
6  │  │  key ← (type);
7  │  │  grouping[key] ⟵ {};
8  │  │  for record ∈ records do
9  │  │  │  Process_properties(grouping, key, record);
10 │  │  end
11 │  end
12 │  Check_mandatory_properties(grouping);
13 │  return grouping_relationships
14 end
```

4.2.2. Extração

A etapa de extração utiliza como entrada o resultado do Algoritmo 1 *(Dicionário de Vértices)* e de forma iterativa vai convertendo os vértices com múltiplos rótulos em vértices de rótulos únicos (Algoritmo 5). Esse processo de extração permite representar características comuns uma únicas vez, de modo similar ao que é feito com herança em liguagens de programação orientadas a objetos, onde as características comuns ficam em uma classe pai.

Para um melhor entendimento de como o Algoritmo 5 funciona, utiliza-se como exemplo a entrada armazenada na variável *grouping_nodes* com a seguinte estrutura:

```
{
    {Actor, Person} : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('oscar', <class 'bool'>) : False
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        }
    },
    {User, Person} : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('lastLogin', <class 'str'>) : True
        },
        "relationships" : {
            ('RATED', {'Movie'}) : False
        }
    }
    Movie : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('genre', <class 'str'>) : True
        },
    }
}
```

Algorithm 5: Extração de vértices com rótulo único

```
   Function extract(grouping_nodes):
   Output: Dictionary with unique type nodes, their properties and
           relationships
1  begin
2      unique_labels ⟵ {};
3      labels_to_process ⟵ True;
4      while labels_to_process do
5          labels_to_process ⟵ False;
6          labels_combinations ← grouping_nodes.keys;
7          labels_length_1 ← {};
8          for label ∈ labels_combinations do
9              if label.length == 1 then
10                 unique_labels[label] ⟵ grouping_nodes.pop(label);
11                 labels_to_process ← True;
12                 labels_length_1.add(label);
13             end
14         end
15         if labels_to_process == True then
16             for label ∈ labels_length_1 do
17                 grouping_nodes ← Process_intersection(grouping_nodes,
18                                                        label,
19
                     unique_labels[label]);
20             end
21         else
22             intersections ⟵ Labels_intersections(labels_combinations);
23             if intersection.length > 0 then
24                 labels_to_process ← True;
25                 key_most_intersected ←node type with most intersections;
26                 unique_labels[key_most_intersected] ⟵
                     Intersect_properties(grouping_nodes,
27                                        key_most_intersected,
28
                     intersections[key_most_intersected]);
29                 grouping_nodes ←
30                   Process_intersection(grouping_nodes,
31                                        key_most_intersected,
32
                     unique_labels[key_most_intersected]);
33             end
34         end
35     end
36     if grouping_nodes.length > 0 then
37         for key ∈ grouping_nodes do
38             unique_labels[key] ⟵ grouping_nodes[key];
39         end
40     end
41     return unique_labels
42 end
```

Na linha 2 do Algoritmo 5 é inicializado um novo dicionário vazio *unique_labels*, que contém a mesma estrutura do *Dicionário de Vértices* da Figura 13, e que será populado à medida que o *grouping_nodes* vai sendo consumido.

Para controlar quando a extração já foi finalizada, uma váriavel booleana é inicializada como verdadeira. Assim, em todo início de uma nova iteração essa variável recebe o estado falso na linha 5 e sempre é colocada novamente no estado verdadeiro quando existem novos vértices a serem processados.

Na linha 6 são armazenados os conjuntos de rótulos presentes na entrada. Para o exemplo sendo utilizado, tem-se: {*Actor, Person*}, {*User, Person*} e *Movie*. O laço de iteração que segue verifica se existe algum conjunto de rótulo único e, caso exista, o retira do dicionário *grouping_nodes* e o coloca tanto em uma lista *labels_length_1* quanto na saída final *unique_labels*.

No exemplo, *Movie* possui rótulo único. Ele então é retirado do *grouping_nodes* e colocado nessa lista. Além disso, é sinalizado que existem rótulos a serem processados. Na linha 16, para cada rótulo a ser processado, é feita então uma reatribuição à váriavel *grouping_nodes*, que recebe o resultado do Algoritmo 7 *Process_intersection* (linha 17). Para esse cenário atual, a chamada de *Process_intersection* não altera o conteúdo de *grouping_nodes*, mas veremos o funcionamento deste algoritmo na linha 30.

Ao final desta primeira iteração, apenas movemos o rótulo *Movie* do dicionário *grouping_nodes* para o *unique_labels*. Já na segunda iteração, como não existem mais vértices de rótulo único em *grouping_nodes*, a execução do algoritmo é desviada para a linha 22. Nesta linha armazenam-se as intersecções existentes entre os conjuntos de rótulos (execução do Algoritmo 6). Para o exemplo sendo mostrado, a saída produzida é a seguinte:

```
{ Person : [{Actor, Person}, {User, Person}] }
```

Caso existissem, por exemplo, outros conjuntos de rótulos que contivessem o rótulo *Actor*, também seria inserida uma chave *Actor* com suas devidas intersecções. Como foram encontradas intersecções, isso é sinalizado na linha 24 e, na linha 25, é escolhido o rótulo que possui o maior número de intersecções. Para o nosso exemplo, temos apenas o rótulo *Person* com duas intersecções.

Na linha 26 é inserido na estrutura de saída final (*unique_labels*) o rótulo *Person* e seu conteúdo, cujo resultado é produzido pelo Algoritmo 8. A função *Intersect_properties* verifica o que existe em comum dentre propriedades e relacionamentos, na estrutura inicial *grouping_nodes*, no que diz respeito ao rótulo *Person*, produzindo o seguinte resultado:

```
{
    Person : {
        "properties" : {
            ('name', <class 'str'>) : True
        }
    }
}
```

Este resultado também serve como entrada para a reatribuição de *grouping_nodes* por meio da função *Process_intersection* da linha 29. Serão extraídos esse rótulo e seus devidos atributos dos possíveis conjuntos de rótulos que existam dentro do antigo *grouping_nodes*. Isso faz com que essa váriavel receba o seguinte valor:

```
{
    Actor : {
        "properties" : {
            ('oscar', <class 'bool'>) : False
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        },
        "allOf" : {
            Person
        }
    },
    User : {
        "properties" : {
            ('lastLogin', <class 'str'>) : True
        },
        "relationships" : {
            ('RATED', {'Movie'}) : False
        },
        "allOf" : {
            Person
        }
    }
}
```

A chave *"allOf"* é inserida na linha 9 do Algortimo 7 e é utilizada pelo processo de mapeamento da seção 4.2.3 para especificar que esses vértices são especializações daquele rótulo, ou seja, também possuem todos os atributos e relacionamentos dele.

Por fim, é feita uma nova e última iteração, onde esse nova estrutura *grouping_nodes* passa pelo mesmo processo para o rótulo *Movie* e entre as linhas 36 a 40. Após todos os rótulos serem processados é verificado se ainda restou algum conjunto de rótulos que não foi possível inferir se as propriedades e relacionamentos eram comuns com outros vértices.

A saída final para o exemplo utilizado é:

```
{
    Person : {
        "properties" : {
             ('name', <class 'str'>) : True
        }
    },
    Actor : {
        "properties" : {
             ('oscar', <class 'bool'>) : False
        },
        "relationships" : {
            ('ACTS_IN', {'Movie'}) : True
        },
        "allOf" : {
            Person
        }
    },
    User : {
        "properties" : {
             ('lastLogin', <class 'str'>) : True
        },
        "relationships" : {
            ('RATED', {'Movie'}) : False
        },
        "allOf" : {
            Person
        }
    },
    Movie : {
        "properties" : {
             ('name', <class 'str'>) : True,
             ('genre', <class 'str'>) : True
        },
    }
}
```

Algorithm 6: Encontrar intersecções

```
    Function Labels_intersections(labels_combinations):
    Output: Dictionary with intersected label and its combinations
 1  begin
 2      intersections ⟵ {};
 3      for l1 ∈ labels_combinations do
 4          for l2 ∈ labels_combinations do
 5              l3 ← l1 ∩ l2;
 6              if l3.length == 1 then
 7                  if l3 ∉ intersections then
 8                      intersections[l3] ← {l1, l2};
 9                  else
10                      intersections[l3].add(l1, l2);
11                  end
12              end
13          end
14      end
15      return intersections
16  end
```

Algorithm 7: Processar intersecções

```
    Function Process_intersection(grouping, label, properties):
    Output: Grouping with separated label from nodes if able
 1  begin
 2      new_grouping ⟵ {};
 3      for key ∈ grouping do
 4          if key ∩ label then
 5              new_key ← key \ label;
 6              new_properties ←
                  grouping[key]["properties"] \ properties["properties"];
 7              new_relationships ←
                  grouping[key]["relationships"] \ properties["relationships"];
 8              Process_relationships(new_relationships, label);
 9              new_grouping[new_key] ⟵ {"properties" :
                  new_properties, "relationships" :
                  new_relationships, "allOf" : label};
10          else
11              new_grouping[key] ← grouping[key];
12          end
13      end
14      return new_grouping
15  end
```

Algorithm 8: Intersectar propriedades

Function *Intersect_properties(grouping, key, intersections)*:

Output: Dictionary with properties and relationships intersected

1 **begin**
2 $aux_key \longleftarrow intersections[0]$;
3 $new_properties \longleftarrow grouping[aux_key]["properties"]$;
4 $new_relationships \longleftarrow grouping[aux_key]["relationships"]$;
5 **for** $label \in intersections$ **do**
6 $new_properties \leftarrow new_properties \cap grouping[label]["properties"]$;
7 $new_relationships \leftarrow new_relationships \cap grouping[label]["relationships"]$;
8 **end**
9 $Process_relationships(new_relationships, key)$;
10 **return** {*"properties": new_properties, "relationships": new_relationships*}
11 **end**

4.2.3. Mapeamento

Uma decisão de projeto para a ferramenta proposta neste trabalho foi separar cada rótulo de vértice e cada tipo de relacionamento em um arquivo JSON *Schema* separado, para melhorar a organização e permitir o reuso de partes do esquema completo do BD de grafo que mantêm diferentes representações dos dados.

As etapas de *Agrupamento* e *Extração* geram resultados na forma de dicionários, como mostram as Figuras 13 e 14, uma vez que este formato tem bastante semelhança com o formato JSON, o que facilita o mapeamento para o esquema final do BD de grafo. No entanto, para se adequar ao formato de saída da ferramenta (JSON *Schema*) foram necessários alguns ajustes.

Primeiramente, os tipos de dados das propriedades seguem o mapeamento das Tabelas 1 e 2, visto que as consultas realizadas no BD seguem o padrão de tipos do Neo4j, a linguagem de programação utilizada para o desenvolvimento da ferramenta foi Python, e a saída produzida está no formato JSON *Schema*.

Neo4j	Python
Null	None
Integer	int
Float	float
String	str
List	list
Map	dict
Boolean	bool

Tabela 1. Mapeamento de tipos Neo4j para Python

Python	**Json *Schema***
None	null
int/float	number
str	string
list	array
dict	object
bool	boolean

Tabela 2. Mapeamento de tipos Python para JSON *Schema*

Além dessa conversão de tipos, outros ajustes necessários foram os seguintes:

- caso existam referências a outros esquemas, estes devem ser insridos em uma chave *"definitions"*.
- propriedades obrigatórias devem ser inseridas em uma chave *"required"*.
- a chave *"allOf"* é usada para definir propriedades e relacionamentos em comum entre vértices. Além disso, deve ser colocado como referência o vértice que possui essas informações, como se fosse uma superclasse em um projeto orientado a objetos.
- A fim de comportar os relacionamentos entre os vértices foi criada uma propriedade *"relationships"*, que é uma lista de tuplas, onde cada uma delas representa o tipo do relacionamento e os rótulos do vértice destino. Isto é feito através da palavra-chave *oneOf*, que exige que os elementos da lista pertençam a alguma das tuplas descritas.
- a palavra-chave *"minItems"* com valor 1 é definida quando um relacionamento é obrigatório.

A Figura 16 mostra um exemplo de *JSON Schema* final para um dos vértices de um BD de atores e filmes, similar aos exemplos utilizados nas etapas anteriores. É possível ver que existem referências a outros arquivos *JSON Schema* que definem o relacionamento do tipo *acts_in* e o vértice de rótulo *movie*. Este relacionamento é obrigatório para todo ator, ou seja, todo ator deve ter atuado em pelo menos um filme como mostra a palavra-chave *"minItems"* com o valor 1. Por fim, a lista definida pela palavra-chave *"required"* demonstra que todas as propriedades são obrigatórias, exceto a propriedade *profileImageUrl*.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Actor",
    "description": "Actor node",
    "\$id": "actor.json",
    "definitions": {
        "acts_in": {
            "type": "object",
            "\$ref": "acts_in.json"
        },
        "movie": {
            "type": "object",
            "\$ref": "movie.json"
        }
    },
    "type": "object",
    "allOf": [
        {
            "\$ref": "person.json"
        }
    ],
    "properties": {
        "id": {
            "type": "number"
        },
        "biography": {
            "type": "string"
        },
        "profileImageUrl": {
            "type": "string"
        },
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
                                "type": "object",
                                "\$ref": "#/definitions/acts_in"
                            },
                            {
                                "type": "object",
                                "\$ref": "#/definitions/movie"
                            }
                        ]
                    }
                ]
            },
            "minItems": 1
        }
    },
    "required": [
        "id",
        "biography",
        "relationships"
    ],
    "additionalProperties": false
}
```

Figura 16. Exemplo para um vértice de rótulo ator

5. Avaliação da Ferramenta

Esta seção apresenta a avaliação da ferramenta desenvolvida considerando dois aspectos: *qualitativo* e *tempo de processamento*. Enquanto o primeiro analisa a qualidade do mapeamento BD grafo ⟶ JSON *Schema*, o segundo verifica a complexidade e tempo de

processamento da ferramenta na execução de processos de extração de esquemas.

5.1. Avaliação Qualitativa

Dois BDs de grafos no Neo4J, que simulam serviços existentes, foram utilizados para avaliar a ferramenta de forma qualitativa. O primeiro mantém dados que simulam o site *Airbnb*[17], um serviço para o anúncio e reserva de acomodações e meios de hospedagem. O segundo é baseado nos dados do *IMDB*[18], uma base de dados que contém filmes, séries, atores, diretores e usuários que podem realizar avaliações. As seções a seguir detalham essas avaliações.

5.1.1. Airbnb

O Neo4j possui uma função *call db.schema()*, que produz um grafo mostrando como seus vértices e relacionamentos estão organizados. O esquema do BD do Airbnb, segundo essa função, pode ser visto na Figura 17.

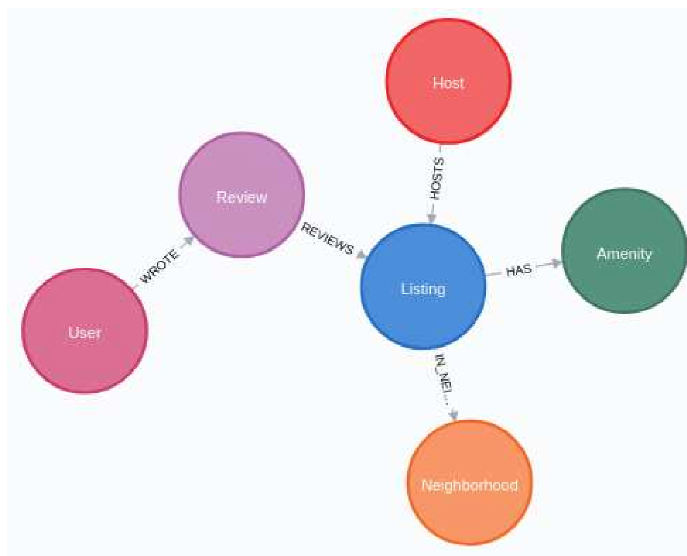

Figura 17. Esquema Airbnb no BD Neo4j

Para realizar a avaliação do esquema extraído com a ferramenta desenvolvida foi necessário fazer uma análise manual dos dados pertencentes ao BD, uma vez que este tipo de BD não possui um esquema explícito. A avaliação foi realizada sobre o vértice de rótulo *Host* e do relacionamento *HOSTS*, podendo ser estendida aos demais.

Ao inferir manualmente a estrutura dos vértices de rótulo *Host*, verificamos que ele contém as propriedades *name, superhost, image, location* e *host_id*, sendo todas obrigatórias. Também verificou-se que todo vértice de rótulo *Host* deve possuir ao menos um relacionamento do tipo *HOSTS*, que liga este vértice a um vértice de rótulo *Listing*.

As Figuras 18 e 19 mostram os esquemas gerados pela ferramenta proposta para o relacionamento *HOSTS* e para o vértice *Host*, respectivamente. Pode-se constatar que

[17]https://www.airbnb.com
[18]https://www.imdb.com/

todas as propriedades de um *Host* são obrigatórias, dado que todas estão declaradas como *required*. Ainda, todo vértice de rótulo *Host* deve conter pelo menos um relacionamento do tipo *HOSTS* que liga um Host a um vértice com rótulo *Listing*, pois seu esquema possui o valor 1 para a restrição *"minItems"*.

Assim, quanto à qualidade do esquema gerado, foi obtido um resultado com 100% de acurácia, uma vez que foram extraídos com sucesso as propriedades e o relacionamento obrigatórios, bem como os seus tipos de dados.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "HOSTS",
    "description": "HOSTS relationship",
    "\$id": "hosts.json",
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        }
    },
    "required": [
        "<id>"
    ],
    "additionalProperties": false
}
```

Figura 18. JSON *Schema* para o relacionamento *HOSTS*

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Host",
    "description": "Host node",
    "\$id": "host.json",
    "definitions": {
        "hosts": {
            "type": "object",
            "\$ref": "hosts.json"
        },
        "listing": {
            "type": "object",
            "\$ref": "listing.json"
        }
    },
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "name": {
            "type": "string"
        },
        "superhost": {
            "type": "boolean"
        },
        "image": {
            "type": "string"
        },
        "location": {
            "type": "string"
        },
        "host_id": {
            "type": "string"
        },
        "relationships": {
            "type": "array",
            "items": {
                "oneOf": [
                    {
                        "type": "array",
                        "items": [
                            {
                                "type": "object",
                                "\$ref": "#/definitions/hosts"
                            },
                            {
                                "type": "object",
                                "\$ref": "#/definitions/listing"
                            }
                        ]
                    }
                ]
            },
            "minItems": 1
        }
    },
    "required": [
        "<id>",
        "name",
        "superhost",
        "image",
        "location",
        "host_id",
        "relationships"
    ],
    "additionalProperties": false
}
```

Figura 19. JSON *Schema* para o vértice *Host*

5.1.2. IMDB

A função *call db.schema()* do Neo4j aplicada sobre o BD do IMDB gera o resultado apresentado na Figura 20.

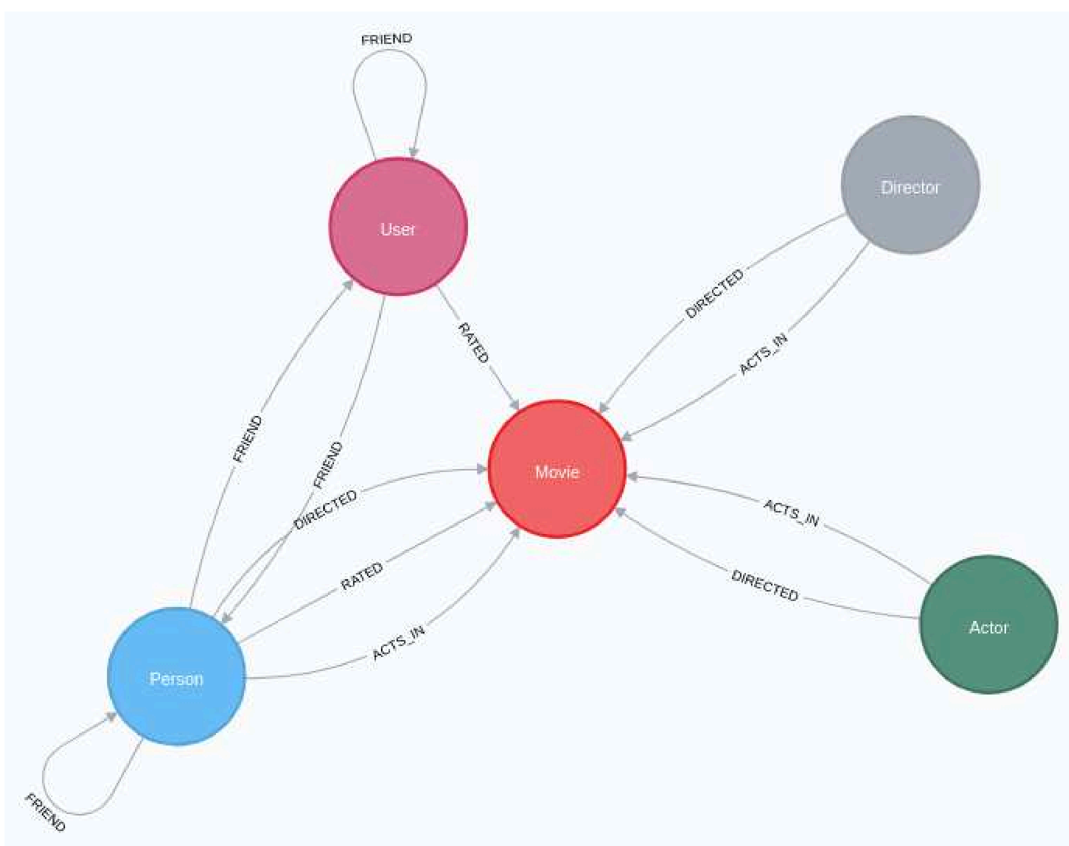

Figura 20. Esquema IMDB no BD Neo4j

Com base em uma análise manual, como realizado na seção anterior, verificou-se que a Figura 20 não é uma boa representação de como os dados estão realmente estruturados. Foi possível averiguar que não existem vértices que possuem apenas o rótulo *Person* ou *Actor*, por exemplo. Na verdade, todo vértice presente no BD contém alguma das combinações de rótulo definidas a seguir:

- *Movie*
- *User, Person*
- *Actor, Person*
- *Director, Person*
- *Actor, Director, Person*

Este BD justifica a escolha por utilizar uma estratégia de intersecção de conjuntos para realizar a extração. Ela produz, como efeito, uma saída que se assemelha a uma hierarquia de *herança* em uma modelagem orientada a objetos. Neste caso, o rótulo *Person* pode conter propriedades e relacionamentos comuns aos rótulos *Actor*, *User* e *Director*, como se fosse uma superclasse deles.

Assim sendo, o JSON *Schema* resultante difere do esquema da Figura 20, pois o vértice de rótulo *Person* não possui nenhum relacionamento, já que estes relacionamentos apenas tem significado semântico nas suas especializações *Actor*, *Director* e *User*. Esta estratégia de geração de esquemas utilizada pela ferramenta proposta reduz o volume do esquema de saída gerado, uma vez que fatora, e representa uma única vez, as características comuns. É possível verificar isso nos esquemas produzidos nas Figuras 21, 22 e 23.

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Person",
    "description": "Person node",
    "\$id": "person.json",
    "definitions": {},
    "type": "object",
    "properties": {
        "<id>": {
            "type": "number"
        },
        "name": {
            "type": "string"
        }
    },
    "required": [
        "<id>",
        "name"
    ],
}
```

Figura 21. JSON *Schema* para o vértice *Person*

```
{
    "\$schema": "https://json-schema.org/draft/2019-09/schema#",
    "title": "Actor",
    "description": "Actor node",
    "\$id": "actor.json",
    "definitions": {
        "acts_in": {
            "type": "object",
            "\$ref": "acts_in.json"
        },
        "movie": {
            "type": "object",
            "\$ref": "movie.json"
        }
    },
    "type": "object",
    "allOf": [
        {
            "\$ref": "person.json"
        }
    ],
    "properties": {
        "<id>": {
            "type": "number"
        },
        "id": {
            "type": "string"
        },
        "birthplace": {
            "type": "string"
        },
        "version": {
            "type": "number"
        },
        "profileImageUrl": {
            "type": "string"
        },
        "biography": {
            "type": "string"
        },
        "lastModified": {
            "type": "string"
        },
        "birthday": {
            "type": "string"
        },
```

Figura 22. JSON *Schema* 1/2 para o vértice *Actor*

```
            "relationships": {
                "type": "array",
                "items": {
                    "oneOf": [
                        {
                            "type": "array",
                            "items": [
                                {
                                    "type": "object",
                                    "\$ref": "#/definitions/acts_in"
                                },
                                {
                                    "type": "object",
                                    "\$ref": "#/definitions/movie"
                                }
                            ]
                        }
                    ]
                },
                "minItems": 1
            }
        },
        "required": [
            "<id>",
            "id",
            "birthplace",
            "version",
            "profileImageUrl",
            "biography",
            "lastModified",
            "birthday",
            "relationships"
        ],
        "additionalProperties": false
    }
```

Figura 23. JSON *Schema* 2/2 para o vértice *Actor*

Portanto, após uma comparação entre a análise manual das propriedades e relacionamentos, e o resultado gerado pela ferramenta proposta, foi obtida uma acurácia de 100% no que diz respeito a qualidade do esquema gerado, uma vez que tamb em foram identificadas todas as propriedades e relacionamentos, bem como os seus tipos e restrições de obrigatoriedade. Além disso, foi produzido um esquema mais enxuto e com melhor valor semântico que o disponibilizado pela função *call db.schema()* do Neo4j.

5.2. Análise de Tempo de Processamento

Esta seção demonstra os resultados da análise de tempo de processamento na extração de esquemas realizada pela ferramenta. Esta avaliação tem como objetivo verificar a complexidade do algoritmo de extração. Para isso, foram escolhidos 4 BDs de grafo com número de vértices e de relacionamentos diferentes: os dois já utilizados na seção 5.1, um BD menor referente ao universo da série televisiva *Dr WHo* e um BD grande extraído da API[19] do Stackoverflow[20]. O *Stackoverflow* é um site que reúne perguntas e respostas

[19]https://api.stackexchange.com/docs
[20]https://stackoverflow.com/

sobre programação, que podem ser avaliadas e comentadas por seus usuários. É possível visualizar o número de vértices e de relacionamentos de cada BD na Tabela 3.

BD	Vértices	Relacionamentos	V + R
Dr Who	1.060	2.286	3.346
IMDB	63.042	106.651	169.693
Airbnb	129.444	220.183	349.627
Stackoverflow	690.656	1.408.205	2.098.861

Tabela 3. Tamanhos das Bases de Dados

Os experimentos foram executados em um sistema operacional Arch Linux, em uma máquina com processador Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, com 16GB de memória RAM e um SSD como memória não volátil. Cada uma dos quatro BDs utilizados no experimento estava dentro de um container Docker[21] do Neo4j na versão 3.5.12 e a ferramenta foi executada com Python versão 3.7.4.

Após a execução da ferramenta para os BDs considerados, foi possível notar que o processo de geração de esquemas cresce linearmente de acordo com o número de vértices e de relacionamentos, visto que os agrupamentos realizados pela função *powerset* evitam que um mesmo vértice seja percorrido mais de uma vez, para cada rótulo identificado no BD. Assim sendo, graças a esta estratégia de agrupamentos, a complexidade do processo de geração do esquema proposto é $O(m+n)$, onde m é o número de vértices e n o número de relacionamentos. Se cada vértice tivesse que ser analisado para descobrir o esquema de cada rótulo existente no BD, a complexidade de pior caso seria $O(r.m + n)$, onde r é o número de rótulos, ou seja, uma complexidade maior.

A referida complexidade pode ser verificada de forma prática. Uma média de tempo de processamento foi gerada sobre 10 execuções de extração de esquemas para os quatro BDs. A Tabela 4 mostra as cinco primeiras execuções, a Tabela 5 as cinco últimas, e a Tabela 6 as médias por BD. Esses dados foram plotados em um gráfico (Figura 24), onde é possível perceber uma função linear com relação ao número de vértices mais o número de arestas.

BD	Tempo(s)				
Dr Who	2.914	1.591	1.234	1.246	1.053
IMDB	60.185	53.746	51.165	53.447	52.720
Airbnb	108.935	108.316	109.874	106.27	101.092
Stackoverflow	523.249	556.428	597.009	584.939	530.543

Tabela 4. Tempos de Processamento 1/3

[21] https://www.docker.com/

BD	Tempo(s)				
Dr Who	1.075	1.037	1.009	0.976	1.000
IMDB	51.783	55.508	52.633	61.232	51.950
Airbnb	98.708	142.959	116.541	122.886	108.766
Stackoverflow	550.322	666.126	600.355	556.883	552.429

Tabela 5. Tempos de Processamento 2/3

BD	Tempo Médio (s)
Dr Who	1.064
IMDB	53.084
Airbnb	108.935
Stackoverflow	556.656

Tabela 6. Média de Tempo de Processamento

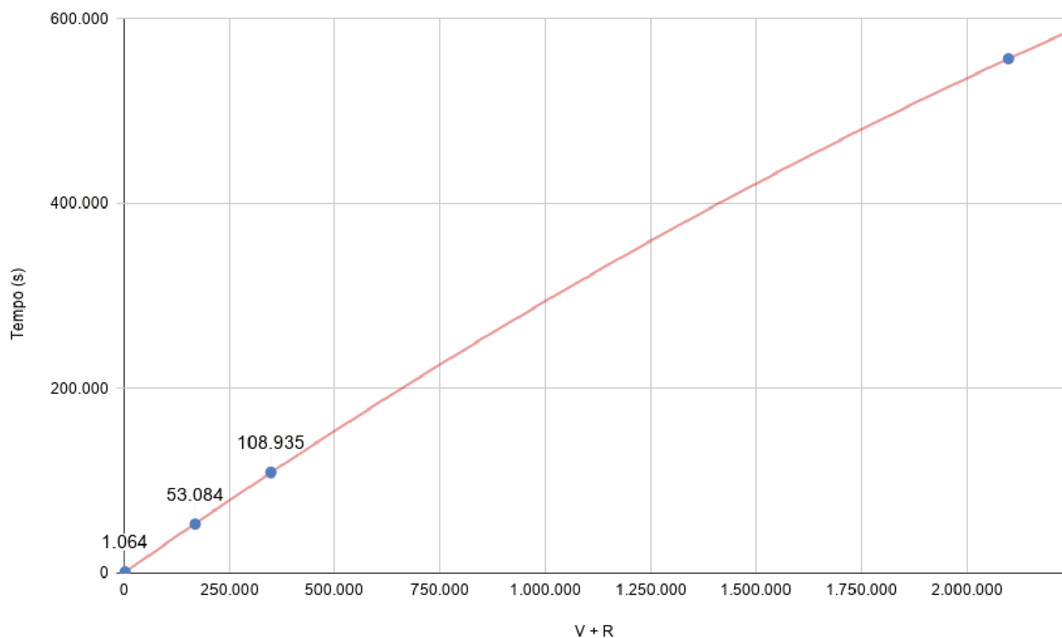

Figura 24. Tempo de processamento X Tamanho do grafo

6. Considerações Finais

Este artigo teve como objetivo principal o desenvolvimento de uma ferramenta para extração de esquemas de um BD NoSQL do tipo grafo. Os esquemas são gerados no formato JSON *Schema*, que é uma recomendação atual para a representação de esquemas de dados complexos.

A principal motivação para a realização deste trabalho é contribuir com pesquisas em desenvolvimento no *Grupo de BD da UFSC (GBD/UFSC)*. Trabalhos anteriores realizaram a extração de esquemas de BDs NoSQL do tipo Documento e Colunar para o formato JSON *Schema*. Este trabalho permite agora a extração de esquemas também de BDs NoSQL do tipo grafo, que é uma lacuna de pesquisa na literatura. Do ponto de vista de contribuição científica e técnica, a geração de esquemas para BDs *schemaless*, como é o caso de BDs NoSQL, facilita as tarefas de manipulação e integração de dados, pois permite a aquisição de conhecimento sobre a representação dos dados presentes nesses BDs.

Os objetivos específicos (seção 1.4) podem ser considerados como atingidos. No caso do objetivo específico 1, que tratou do mapeamento do grafo para um formato intermediário reduzido, este pode ser visto em detalhes nas seções 4.2.1 e 4.2.2. Já o objetivo 2, que tratou do mapeamento deste formato intermediário para o formato padrão JSON *Schema*, é descrito na seção 4.2.3. Por sua vez, o objetivo 3, que se refere a avaliação da ferramenta proposta, é descrito no capítulo 5, apresentando bons resultados. Por fim, com relação ao objetivo 4, todo o código produzido neste trabalho encontra-se no GitHub[22]. Assim, pode-se concluir que o objetivo geral do projeto também foi atingido, visto que a ferramenta foi projetada e realiza o que foi proposto.

Os temas e tecnologias aqui abordados muito acrescentaram para a formação do autor, incluindo BDs NoSQL, JSON *Schema* e desenvolvimento de software. Vale observar também o conhecimento adquirido da ferramenta Neo4j e de estruturas de dados como listas, conjuntos e dicionários.

Por fim, algumas sugestões de trabalhos futuros relacionados a este assunto são:

- Melhorar o desempenho da primeira etapa do processo proposto utilizando técnicas estatísticas, considerando a grande redundância em termos de esquemas dos dados;
- Gerar esquemas para outros BDs NoSQL de grafo;
- Integrar diferentes esquemas de BDs NoSQL.

Referências

(2018). Cypher, the graph query language.

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Comput. Surv*, 40.

Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., and Voigt, H. (2019). Schema validation and evolution for graph databases. *CoRR*, abs/1902.06427.

Da Costa, F. D. S. (2017). Uma ferramenta para extração de esquemas de bancos de dados nosql orientados a documentos.

Dias Defreyn, E. (2019). Hbasi - hbase schema inference tool: Umaferramenta para extraÇÃo de esquemas debancos de dados nosql colunares.

Frozza, A. A. and dos Santos Mello, R. (2018). A process for reverse engineering of aggregate-oriented nosql databases with emphasis on geographic data. In *XXXIII*

[22]https://github.com/MaoRodriguesJ/neo4j-schema-extract-code

Simpósio Brasileiro de Banco de Dados: Demos e WTDBD, SBBD 2018 Companion, Rio de Janeiro, RJ, Brazil, August 25-26, 2018., pages 109–115.

Gessert, F., Wingerath, W., Friedrich, S., and Ritter, N. (2017). Nosql database systems: a survey and decision guidance. *Comput Sci Res Dev*, 32:353–365.

Grover, J. (2019). Perceiving python programming paradigms. Acessado em 30 de Outubro de 2019.

Han, J., E, H., Le, G., and Du, J. (2011). Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference*, Port Elizabeth, South Africa. IEEE.

Hecht, R. and Jablonski, S. (2011). Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference*, Hong Kong, China. IEEE.

Mariani, A. C. (2018). Teoria de grafos.

Marr, B. (2014). Big data: The 5 vs everyone must know.

Neo4j (2018a). Neo4j basics.

Neo4j (2018b). Why graph databases.

Nestorov, S., Abiteboul, S., and Motwani, R. (1998). Extracting schema from semistructured data. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 295–306, New York, NY, USA. ACM.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). Foundations of json schema. In *WWW '16 Proceedings of the 25th International Conference on World Wide Web*, Montréal, Québec, Canada. ACM.

Roy-Hubara, N., Rokach, L., Shapira, B., and Shoval, P. (2017). Modeling graph database schema. *IT Professional*, 19:34–43.

Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled*. Pearson, United States.

Sommerville, I. (2011). *Engenharia de software*. PEARSON BRASIL.

Sousa, V. M. D. and del Val Cura, L. M. (2018). Logical design of graph databases from an entity-relationship conceptual model. In *The 20th International Conference on Information Integration and Web-based Applications & Services*, Vienna, Austria. ACM.