UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Jhúlia Graziella de Souza Rodrigues

Validação de Dependências Funcionais em Grafos

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Jhúlia Graziella de Souza Rodrigues

Validação de Dependências Funcionais em Grafos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Maria Adriana Vidigal de Lima

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2018

Jhúlia Graziella de Souza Rodrigues

Validação de Dependências Funcionais em Grafos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 18 de dezembro de 2018:

Maria Adriana Vidigal de Lima
Orientador

Luiz Cláudio Theodoro
Convidado 1

Paulo Henrique Ribeiro Gabriel
Convidado 2

Uberlândia, Brasil
2018

Resumo

Dependências funcionais (DFs) são utilizadas no projeto de bancos de dados relacionais para definir que o valor de um conjunto de atributos deve depender do valor de outro conjunto de atributos. As DFs integram a teoria das bases de dados e fundamentam o projeto conceitual de dados, a otimização de consultas e a prevenção de inconsistências na atualização de dados. Este trabalho pretende: (i) estudar as dependências de dados no contexto dos bancos de dados em grafos, considerando uma classe de dependências denominada *Graph Entity Dependencies* (GEDs) e (ii) criar uma API Java para a escrita e validação das GEDs utilizando a linguagem Cypher e o SGBDG NoSQL Neo4j.

Palavras-chave: dependências funcionais, banco de dados, grafos, validação.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
DF	Dependência Funcional
GDC	Graph Denial Constraints
GED	Graph Entity Dependency
ID	Identificador
IDE	Integrated Development Enviornment
SGBD	Sistema de Gerenciamento de Banco de Dados
SGBDG	Sistema de Gerenciamento de Banco de Dados em Grafo
SGBDR	Sistema de Gerenciamento de Banco de Dados Relacional Durabilidade
SQL	Structured Query Language
UML	Unified Modeling Language

Lista de símbolos

$\emptyset$	Conjunto vazio
$\asymp$	Correspondência
$\neq$	Diferença
$\in$	Pertence
$false$	Valor falso do tipo de dado primitivo Booleano
Γ	Letra grega Gama
$=$	Igualdade
$\rightarrow$	Implicação
$\mathbb{N}$	Conjunto dos números naturais
φ	Letra grega minúscula Phi
$\models$	Satisfação
σ	Letra grega Sigma
$\forall$	Para todo
Υ	Letra grega Upsilon

Sumário

1 Introdução

Um grafo é uma abstração muito útil na representação de problemas computacionais e na sua solução, pois permite estabalecer relações de interdependência entre elementos de um conjunto. Um banco de dados para grafos é um sistema de armazenamento que utiliza estruturas com vértices e arestas para representar e armazenar dados. O modelo mais comumente utilizado de grafos no contexto de bancos de dados em grafos é o grafo de propriedades (ROBINSON; WEBBER; EIFREM, 2015; MARGITUS; TAUER; SUDIT, 2015). Um grafo de propriedadees contém entidades conectadas, sendo que cada entidade pode possuir um número de propriedades (ou atributos) expressos em pares chave-valor. Vértices e arestas podem ser etiquetados com rótulos que representam os diferentes papéis (ou tipos) no domínio da aplicação.

Os bancos de dados em grafos e suas tecnologias, e a análise baseada em grafos aplicada à grandes conjuntos de dados não estruturados, foram consideradas por Pokornỳ (2015) como umas das mais interessantes áreas de pesquisa da atualidade. Como exemplo de *big graph* pode-se citar o Facebook com 1 bilhão de nós e 140 bilhões de ligações, necessitando de armazenamento eficiente e algoritmos de processamento especiais (MADAN; SAXENA, 2014 apud POKORNỲ, 2015).

Os bancos de dados em grafos têm foco no processamento de dados altamente conectados, na flexibilidade dos modelos de dados e na performance da recuperação de informação e são frequentemente incluídos entre as bases de dados NoSQL. As bases de dados em grafos têm então a responsabilidade de processar de forma eficiente densos conjuntos de dados e de utilizar os relacionamentos entre os dados para prover análises de correlações e padrões de dados. .

Fan e Lu (2017) propõem uma classe de dependência de dados para grafos, denominada *Graph Entity Dependency* ou GED. Uma GED combina um padrão de grafo com uma dependência de propriedade (ou atributo). Utilizando um formato uniforme, as GEDs expressam as dependências funcionais usando literais constantes (*strings*) que são úteis para capturar inconsistências nos dados. Além disso, pode-se definir dependências de dados utilizando atributos identificadores (*ids*) para distinguir entidades em um grafo.

Neste contexto, o presente trabalho propõe o estudo das dependências de dados GEDs nos bancos de dados em grafos e a implementação, em linguagem Java, de uma Application Programming Interface (API) para definição e validaçao de GEDs. O ambiente de trabalho proposto utiliza APIs do Sistema de Gerenciamento de Banco de Dados em Grafos (SGBDG) Neo4j e a sua linguagem de consulta correspondente, denominada Cypher (NEO4J, 2018).

1.1 Objetivos

O objetivo geral deste trabalho é utilizar a teoria de GEDs proposta em Fan e Lu (2017) para implementar a validação de dependências funcionais em um banco de dados em grafo. Para a especificação das dependências funcionais, serão definidas classes específicas numa API de forma que se possa, em conjunto com o SGBDG Neo4j e a linguagem Cypher, especificar dependências baseadas em padrões de grafos e utilizá-las para analisar a coerência e a qualidade dos dados.

1.2 Organização do Trabalho

Para uma melhor separação e compreensão do conteúdo, os próximos capítulos deste trabalho estão organizados da seguinte maneira:

- O Capítulo 2 apresenta os conceitos básicos necessários para compreensão do trabalho, bem como uma breve análise de trabalhos correlatos;
- O Capítulo 3 descreve as GEDs em mais detalhes e apresenta os conceitos específicos necessários para entendê-las;
- O Capítulo 4 mostra como o trabalho foi desenvolvido desde a etapa da configuração do ambiente até a parte de implementação do projeto. Nele estão presentes o diagrama de classes e todos os códigos-fonte do projeto. Na última sessão deste capítulo são apresentados os testes realizados afim de verificar a corretude do algoritmo de validação;
- O Capítulo 5 expõe as conlusões e considerações finais do trabalho, além de propostas para continuação do mesmo.

2 Fundamentação Teórica

2.1 Grafos

Grafo é uma estrutura de dados composta por um conjunto de vértices interligados por um conjunto de arestas. Dependendo da aplicação, as arestas podem ou não ser direcionadas.

Definição 2.1.1. Grafo - Um grafo G é um par (V, E), onde

- V é um conjunto finito de vértices; e
- E é um conjunto finito de arestas, onde cada par não-ordenado (v_1, v_2) representa uma aresta entre v_1 e v_2, $v1 \neq v2$ e $(v_1, v_2 \in V)$.

Definição 2.1.2. Digrafo - Um digrafo (ou grafo direcionado) G é um par (V, E) onde

- V é um conjunto finito de vértices; e
- E é um conjunto finito de arcos (ou arestas direcionadas), onde cada par ordenado (v_1, v_2) representa um arco que parte de v_1 em direção a v_2, $v1 \neq v2$ e $(v_1, v_2 \in V)$.

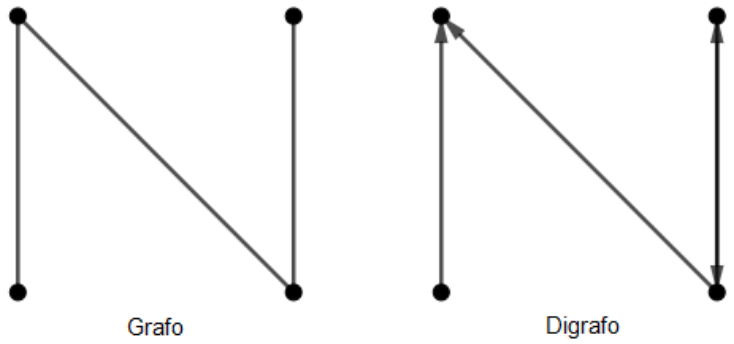

Figura 1 – Exemplo de grafo e digrafo.

Em teoria dos grafos, um grafo ou digrafo é simples se ele não tem laços e não possui mais de uma aresta ligando dois vértices. Um multigrafo é um grafo que não é simples e um multidigrafo é um digrafo que não é simples, ou seja, em multigrafos e multidigrafos é permitida a existência de laços e de arestas múltiplas.

Definição 2.1.3. Multigrafo - Um multigrafo G é um par (V, E), onde

- V é um conjunto finito de vértices; e

- E é um multiconjunto finito de arestas, onde cada par não-ordenado (v_1, v_2) representa uma aresta entre v_1 e v_2, $(v_1, v_2 \in V)$.

Definição 2.1.4. Multidigrafo - Um multidigrafo G é um par (V, E) onde

- V é um conjunto finito de vértices; e
- E é um multiconjunto finito de arcos (ou arestas direcionadas), onde cada par ordenado (v_1, v_2) representa um arco que parte de v_1 em direção a v_2, $(v_1, v_2 \in V)$.

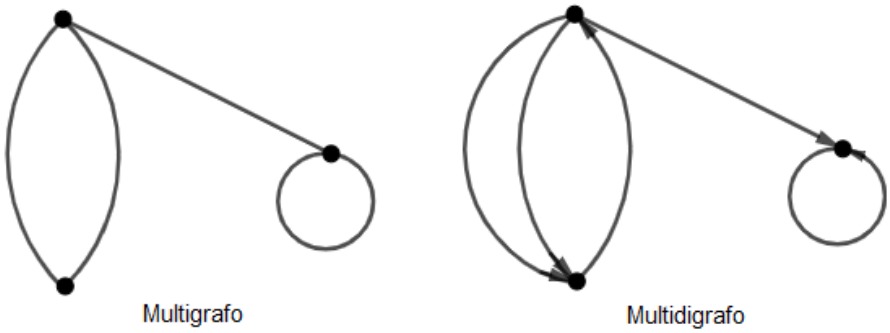

Figura 2 – Exemplo de multigrafo e multidigrafo.

Um grafo etiquetado é um grafo em que seus vértices ou arestas, ou ambos, possuem etiquetas. Multigrafos e Multidigrafos também podem ser etiquetados, de modo similar.

Definição 2.1.5. Grafo Etiquetado - Um grafo etiquetado G é uma sêxtupla $(V, E, \Sigma_V, \Sigma_E, L_V, L_E)$ onde

- V é um conjunto de vértices;
- E é um conjunto finito de arestas, onde cada par não-ordenado (v_1, v_2) representa uma aresta entre v_1 e v_2, $v1 \neq v2$ e $(v_1, v_2 \in V)$;
- Σ_V e Σ_E são alfabetos finitos de etiquetas de vértices e arestas, respectivamente; e
- L_V e L_E são funções que descrevem a etiquetagem de vértices e aretas, respectivamente.

Um grafo com atributos é um grafo que possui atributos associados aos vértices ou arestas, ou ambos. De acordo com Margitus, Tauer e Sudit (2015), um grafo com atributos onde os atributos são representados no formato chave-valor também é chamado de Grafo de Propriedades.

Definição 2.1.6. Grafo com Atributos (MARGITUS; TAUER; SUDIT, 2015) - Um grafo com atributos G é uma quádrupla (V, E, A_V, A_E) onde

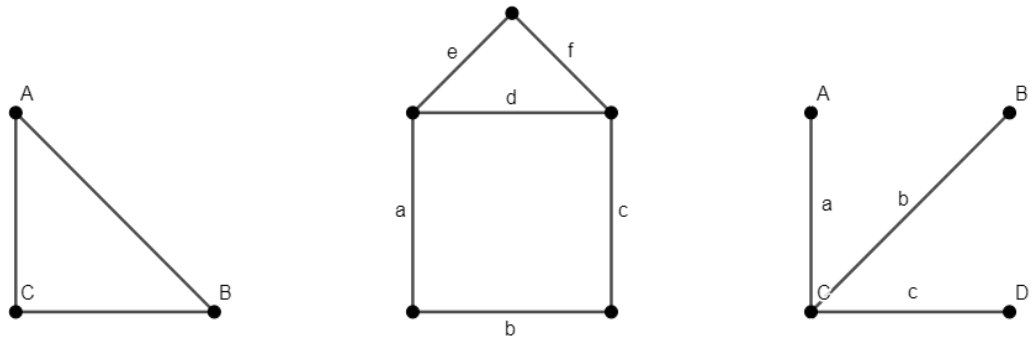

Figura 3 – Exemplo de grafos etiquetados.

- V é um conjunto de vértices;
- E é um conjunto finito de arestas, onde cada par não-ordenado (v_1, v_2) representa uma aresta entre v_1 e v_2, $v1 \neq v2$ e $(v_1, v_2 \in V)$;
- A_V e A_E são conjuntos de atributos sobre os vértices e sobre as arestas, respectivamente.

Um homomorfismo de um grafo G em um grafo G' é um mapeamento entre os dois grafos que preserva suas arestas, ou seja, toda aresta que existe em G deve ter uma aresta correspondente em G'.

Definição 2.1.7. Homomorfismo - Um homomorfismo de um grafo $G = (V, E)$ em um grafo $G' = (V', E')$ é um mapeamento $f : V \to V'$ do conjunto de vértices de G para o conjunto de vértices de G', tal que se existe uma aresta $e = (v_1, v_2) \in E$ então deve existir obrigatoriamente uma aresta $e' = (f(v_1), f(v_2)) \in E'$.

De acordo com Rodriguez e Neubauer (2010) um grafo de propriedades é um multigrafo direcionado, etiquetado e com atributos. Ou seja, num grafo de propriedades as arestas são direcionadas, podem haver laços e múltiplas arestas entre dois vértices, vértices e arestas possuem etiquetas e pares de atributos chave-valor associados, conforme grafo ilustrado na Figura 4.

Neste trabalho utilizou-se a definição de grafo encontrada em Fan e Lu (2017): um grafo de propriedades é um grafo em que vértices e arestas possuem etiquetas, porém, apenas vértices possuem atributos. Além dos conjuntos V e E (para vértices e arestas) estão presentes também os Γ, Υ e U, que são conjuntos infinitos contáveis de etiquetas, atributos e constantes, respectivamente.

Definição 2.1.8. Grafo (FAN; LU, 2017) - Um grafo G é uma quádrupla (V, E, L, F_A), onde

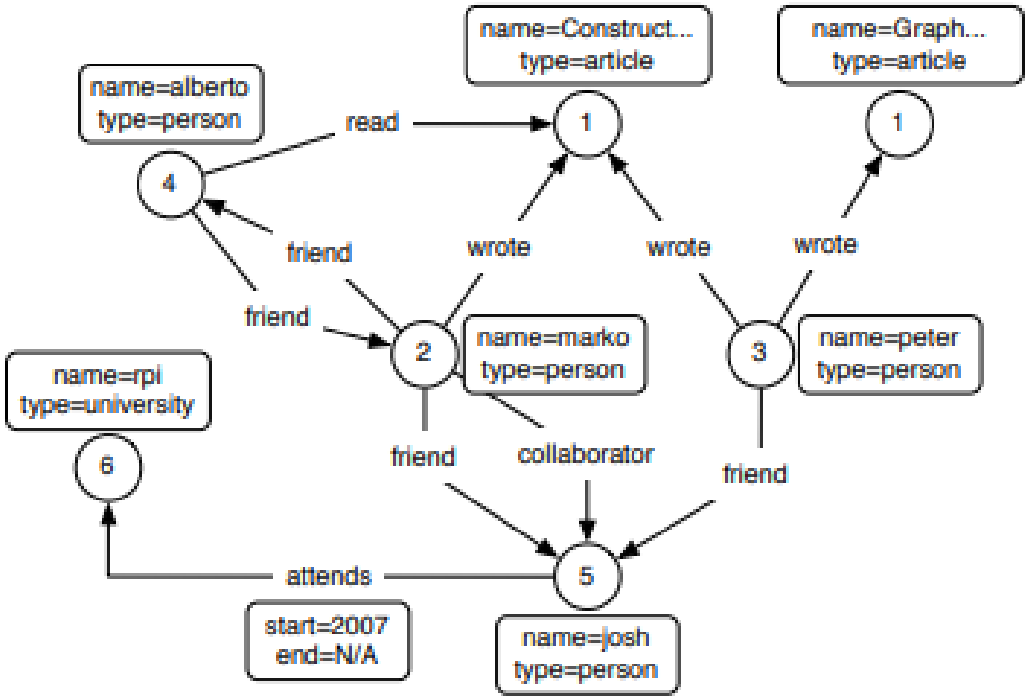

Figura 4 – Exemplo de grafo de propriedades Rodriguez e Neubauer (2010).

- V é um conjunto finito de vértices;
- E é um conjunto finito de arestas, onde (v_1, l, v_2) representa uma aresta direcionada de v_1 até v_2, $(v_1, v_2 \in V)$ e que possui uma etiqueta $l \in \Gamma$;
- Cada vértice $v \in V$ possui uma etiqueta $L(v) \in \Gamma$; e
- Cada vértice $v \in V$ possui uma tupla $F_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$ de atributos finita, onde $A_i \in \Upsilon$ e $a_i \in U$, escrita como $v.A_i = a_i$, e $A_i \neq A_j$ se $i \neq j$. Cada v obrigatoriamente possui um atributo especial id.

2.2 Banco de dados

Um banco de dados é basicamente um conjunto de informações organizadas. No modelo relacional, os dados são armazenados em uma ou mais tabelas que se relacionam entre si. A Tabela 1 apresenta parte de um banco de dados relacional que representa um conjunto de cinemas e filmes.

Segundo Penteado et al. (2014), os Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDRs) dominaram o meio empresarial e acadêmico durante décadas porque a modelagem de dados no modelo relacional é intuitiva, há uma linguagem padronizada de consulta e manipulação de dados, e as propriedades ACID (Atomicidade, Consistên-

Filmes

Título	Diretor
Os Guardiões da Galáxia	James Gunn
Seu Nome	Makoto Shinkai
Pantera Negra	Ryan Coogler

Horários

Cinema	Tela	Título
Cinépolis	1	Os Guardiões da Galáxia
Cinépolis	2	Os Guardiões da Galáxia
Cinépolis	3	Pantera Negra
Cinépolis	4	Pantera Negra
Cinemark	1	Seu Nome
Cinemark	2	Seu Nome
Cinemark	3	Pantera Negra
Cinemark	4	Os Guardiões da Galáxia

Tabela 1 – Exemplo de Banco de Dados Relacional

cia, Isolamento e Durabilidade) são garantidas em diversas aplicações. Mas apesar desses benefícios, aplicações baseadas em modelos de dados complexos podem ter problemas.

Ainda segundo Penteado et al. (2014), o banco de dados em grafos surgiu como uma alternativa ao banco de dados relacional parar dar suporte a sistemas cuja interconectividade de dados é importante. De acordo com Pokornỳ (2015), o modelo de banco de dados relacional foi inicialmente projetado para representar formulários de papel e estruturas tabulares e funciona muito bem nesses cenários, mas tem muita dificuldade para representar os relacionamentos específicos, irregulares e excepcionais que aparecem no mundo real.

Formalmente, um grafo é apenas uma coleção de vértices e arestas. Os grafos representam entidades como vértices e os relacionamentos como arestas. Esta estrutura expressiva e de propósito geral permite modelar todo e qualquer tipo de cenário.

Por exemplo, as informações de redes sociais como Twitter e Facebook podem ser representadas facilmente com o uso de grafos. Na Figura 5 é representada uma pequena rede de usuários do Twitter. Os vértices representam usuários e possuem a propriedade "nome" e a etiqueta "usuário", as arestas representam relacionamentos e possuem a etiqueta "segue".

Uma base de dados pode ser representada de diferentes formas dependendo do modelo de grafo escolhido, e o modelo mais utilizado entre os SGBDGs atuais é o grafo de propriedades (PENTEADO et al., 2014), descrito na presente seção.

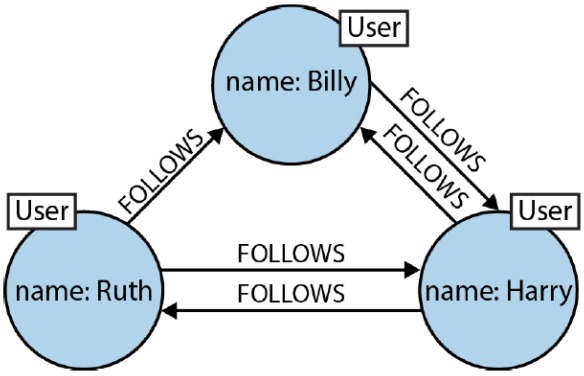

Figura 5 – Uma pequena rede de usuários do Twitter (POKORNỲ, 2015).

2.3 Restrições de Integridade

Em diversas aplicações, a qualidade dos dados armazenados é de grande importância para que resultados precisos e corretos possam ser obtidos através de consultas. O uso de restrições de integridade sobre os dados tem o objetivo de melhorar a qualidade dos dados e são aplicadas no momento da inserção dos dados e da modificação dos mesmos. Em qualquer estado do banco de dados, todas as restrições devem ser satisfeitas para que os dados estejam de acordo com a qualidade desejada. As restrições de integridade comumente suportadas e utilizadas em bancos de dados convencionais são as de domínio, integridade de entidade, estrutura de atributo e integridade referencial (ELMASRI; NAVATHE, 2010).

As restrições de integridade garantem consistência dos dados, mas não a corretude. Por exemplo, uma restrição de integridade pode ser criada para garantir que um atributo "idade" seja sempre um numero natural: isso garante que não haverá idade negativa, mas não garante que a idade inserida esteja correta.

O problema de restrições de integridade está bem consolidado na área de bancos de dados relacionais. Porém, no contexto do armazenamento de dados em grafos, um novo desafio foi estabelecido para o campo das restrições de integridade considerando a necessidade de se estabelecer regras capazes de tratar as especificidades dos dados em grafos e seus relacionamentos (ROBINSON; WEBBER; EIFREM, 2015; ŠESTAK; RABUZIN; NOVAK, 2016; FAN; LU, 2017). Em Šestak, Rabuzin e Novak (2016) as questões de implementação de restrições de integridade em grafos são discutidas e duas abordagens são apresentadas: implementação integrada ao SGBD e implementação em uma camada separada. Neste trabalho, optou-se por desenvolver uma API em uma camada separada, em que pudessem ser utilizados os plugins do SGBDG Neo4j e da linguagem Cypher.

Angles e Gutierrez (2008 apud ŠESTAK; RABUZIN; NOVAK, 2016) identificaram

vários exemplos de restrições de integridade importantes para bancos de dados em grafo:

- Consistência esquema-instância: Previne que informações incompletas ou não existentes sejam inseridas no banco de dados e implica que a instância deve conter apenas as entidades e relacionamentos previamente definidos no esquema.
- Redundância de dados: Reduz a quantidade de informações redundantes armazenadas no banco de dados.
- Integridade de identidade: Similarmente à restrição de chave primária do modelo relacional, garante que cada nó do banco de dados represente uma entidade única do mundo real que possa ser identificada por um ID ou um conjunto de valores de atributos.
- Integridade referencial: De forma similar à restrição de chave secundária do modelo relacional, garante que apenas entidades existentes no banco de dados possam ser referenciadas.
- Dependências Funcionais: Permitem testar se o fato de alguma entidade determinar o valor de outra é respeitado no conjunto dos dados.

2.3.1 Dependências Funcionais

Uma dependência funcional é uma restrição entre dois subconjuntos de atributos de um banco de dados. Seja R o conjunto de atributos $R = \{A_1, A_2, A_3, \dots A_n\}$ de um banco de dados relacional e sejam X e Y dois subconjuntos de R.

Define-se que X determina funcionalmente Y (ou que Y depende funcionalmente de X) se, e somente se, $\forall\ t_1, t_2 \in R : t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$. Denota-se que X determina funcionalmente Y por $X \rightarrow Y$. Por exemplo, na Tabela 1 tem-se que `Título` $\rightarrow$ `Diretor`, pois para cada valor de "Diretor" há apenas um valor de "Título" correspondente.

Os SGBDGs atuais para grafos não possuem suporte para a definição explícita e o uso de dependências funcionais. Segundo Šestak, Rabuzin e Novak (2016), para as duas linguagens populares utilizadas em bancos de dados em grafos, Cypher e Gremlin, o suporte para restrições de integridade de ambas é mínimo. Fan e Lu (2017) apresentam formalmente uma classe de dependências funcionais para grafos, chamadas de *Graph Entity Dependencies*, que são capazes de expressar dependências funcionais em bancos de dados em grafos.

2.4 SGBDG Neo4j

Neo4j é um banco de dados em grafo nativo NoSQL de código aberto implementado em Java e Scala. Ele começou a ser implementado em 2003 mas só se tornou disponível publicamente a partir de 2007.

Neo4j é considerado um banco de dados em grafo nativo porque ele implementa eficientemente o modelo de grafo de propriedades até o nível de armazenamento. Isso quer dizer que os dados são armazenados exatamente como podem ser representados num quadro branco, e o banco de dados usa ponteiros para navegar e percorrer o grafo. O Neo4j também fornece características completas de banco de dados, como conformidade às transações ACID, suporte à *clusters* e tolerância a falhas, tornando adequado usar grafos de dados em cenários de produção.

Segundo Neo4j (2018), alguns dos seguintes recursos tornam o Neo4j popular entre desenvolvedores, arquitetos e administradores de bancos de dados:

- Cypher, uma linguagem de consulta declarativa similar ao SQL, mas otimizada para grafos. Também utilizada por outros bancos de dados como SAP HANA Graph e Redis Graph através do projeto openCypher (MARTON; SZÁRNYAS; VARRÓ, 2017).
- Percurso em tempo constante em grafos grandes tanto para percurso em profundidade quanto em largura, devido à representação eficiente de nós e relacionamentos. Permite escalar para bilhões de nós em hardware moderado.
- Esquema de grafo de propriedades flexível que pode se adaptar ao longo do tempo, possibilitando materializar e adicionar novos relacionamentos mais tarde, de modo a acelerar os dados do domínio quando as necessidades do negócio mudarem.
- Drivers para linguagens de programação populares, como Java, JavaScript, .NET, Python e várias outras.

2.4.1 Linguagem Cypher

Cypher é a linguagem de consulta aberta do Neo4j. A sintaxe do Cypher fornece uma maneira familiar de combinar padrões de nós e relacionamentos no grafo. É uma linguagem declarativa de consulta construída sobre os conceitos básicos e cláusulas do SQL, mas com funcionalidades específicas de grafo adicionais, tornando-a simples de se trabalhar junto a um modelo rico de grafo mas sem ser verbosa demais.

A Cypher foi projetada para ser facilmente lida e compreendida. Ela é simples porque corresponde à maneira como são descritos intuitivamente os grafos usando diagramas. A noção básica é permitir que o usuário encontre dados que correspondam a um padrão

específico, e a maneira que esse padrão é descrito se parece com um desenho usando arte em ASCII (NEO4J, 2016).

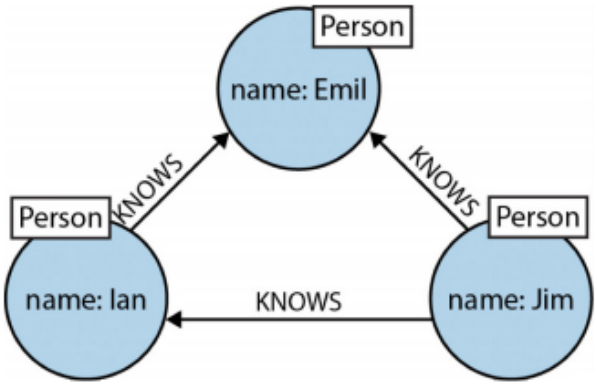

Figura 6 – Um grafo descrevendo os relacionamentos entre três amigos (NEO4J, 2016).

Por exemplo, para expressar o padrão do grafo da Figura 6 no Cypher, usa-se a consulta `(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)`.

As restrições de integridade que podem ser atualmente suportadas na linguagem Cypher sobre os vértices são:

1. Atributo identificador: é possível definir um (ou mais atributos) como sendo *chave*, com a seguinte sintaxe:

   ```
   CREATE CONSTRAINT ON (E:rotulo_vertice)
   ASSERT (E.nome_atributo) IS NODE KEY
   ```

   Define-se o rótulo do vértice que receberá a restrição, associando-o à uma variável (`E`) e define-se o atributo de `E` que receberá a restrição `NODE KEY`.

2. Atributo único: pode-se definir um atributo de um rótulo de vértice com valor único a partir da sintaxe:

   ```
   CREATE CONSTRAINT ON (E:rotulo_vertice)
   ASSERT E.nome_atributo IS UNIQUE
   ```

   sendo que `E` é uma variável representando o rótulo e após o termo `ASSERT` é definido o atributo que receberá a restrição `UNIQUE`.

3. Atributo fixo: pode-se definir um atributo com existência obrigatória em um vértice através da sintaxe:

   ```
   CREATE CONSTRAINT ON (E:rotulo_vertice)
   ASSERT exists(E.nome_atributo)
   ```

sendo que `E` é uma variável representando o rótulo para simplificar o comando e define-se o atributo de `E` que deverá existir e possuir um valor neste determinado campo.

Mais informações sobre a linguagem Cypher podem ser encontradas no seu manual de referência[1].

2.5 Trabalhos correlatos

Šestak, Rabuzin e Novak (2016) discutem o suporte para restrições de integridade de bancos de dados em grafo, a partir das linguagens Cypher e Gremlim, as mais populares para SGBDs em grafo, e demonstram que é mínimo. O trabalho apresenta os desafios de implementação técnica para restrições de integridade em grafos, considerando as abordagens em camada e integrada à um SGBD para que restrições de integridade possam ser definidas e validadas sobre os dados, garantindo mais qualidade e consistência às bases de dados. Foi proposta a implementação de uma aplicação web, construída utilizando-se o framework Spark para Java, com acesso à uma base de dados Neo4j e a linguagem Gremlim para a execução de consultas aos dados. Foi implementada uma restrição do tipo UNIQUE, ainda não suportada pela linguagem Gremlim.

Em Rabuzin, Konecki e Šestak (2016), uma nova restrição de integridade é proposta para bases de dados em grafos, denominada *check integrity constraint*, que previne que usuários entrem com valores fora de um intervalo pré-definido para um atributo, associado à um vértice. Esta restrição é implementada como uma camada adicional ao SGBDG Neo4j.

O trabalho de Fan e Lu (2017) propõe a classe GED de dependências para grafos, que é definida como a combinação de um padrão de grafo e uma dependência de atributos, com o objetivo de representar dependências funcionais em grafos. O autor aborda os problemas de satisfação, implicação e validação de GEDs e estabelece a complexidade de cada um. As GEDs foram utilizadas como base para a presente proposta.

[1] https://neo4j.com/docs/cypher-manual/current/

3 Graph Entity Dependencies

A proposta deste trabalho é desenvolver a classe GED em Java afim de criar e validar dependências funcionais em grafos. Neste capítulo os conceitos de GED serão apresentados em mais detalhes.

Uma Graph Entity Dependency (GED) é uma combinação de um padrão de grafo Q como uma restrição topológica e uma dependência funcional $X \rightarrow Y$ com conjuntos X e Y de literais de igualdade. O padrão Q identifica um conjunto de entidades no grafo e a dependência funcional $X \rightarrow Y$ é aplicada sobre essas entidades (FAN; LU, 2017).

Definição 3.0.1. GEDs (FAN; LU, 2017) Uma GED φ é definida como $Q[\overline{x}](X \rightarrow Y)$, onde:

- $Q[\overline{x}]$ é um padrão de grafo; e
- $X \rightarrow Y$ é uma dependência funcional onde X e Y são dois conjuntos (possivelmente vazios) de literais de $\overline{x}$.

Como mencionado na Seção 2.1, neste trabalho é utilizada a Definição 2.1.8 de grafo e existem existem três conjuntos infinitos contáveis, Γ, Υ e U que representam respectivamente etiquetas, atributos e constantes.

3.1 Padrão de Grafo

Um padrão de grafo é basicamente um conjunto de vértices e arestas etiquetados, que são enumerados como um conjunto de variáveis. Na Figura 7 há alguns exemplos de padrões de grafo.

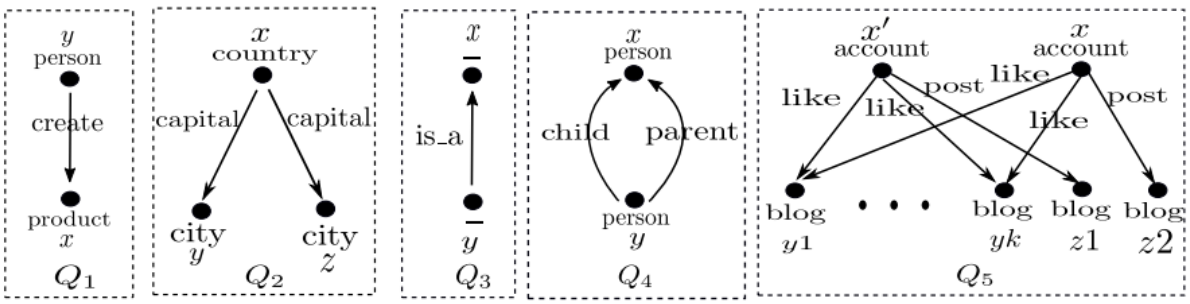

Figura 7 – Padrões de grafo (FAN; LU, 2017).

Definição 3.1.1. Padrão de Grafo (FAN; LU, 2017) Um padrão de grafo é um grafo direcionado $Q[\overline{x}] = (V_Q, E_Q, L_Q)$, onde

- V_Q é um conjunto finito de vértices do padrão;
- E_Q é um conjunto finito de arestas do padrão;
- L_Q é uma função que atribui uma etiqueta $L_Q(v)$ para cada nó $v \in V_Q$; e
- $\overline{x}$ denota os vértices em V_Q como uma lista de variáveis.

As etiquetas dos vértices e arestas do padrão são as etiquetas do conjunto Γ, além da etiqueta coringa "_" que é uma etiqueta especial de Q que corresponde a qualquer etiqueta em Γ.

Definição 3.1.2. Correspondência (FAN; LU, 2017) Dizemos que uma etiqueta l_1 *corresponde* a l_2, denotado por $l_1 \asymp l_2$, se $l_1, l_2 \in \Gamma$ e $l_1 = l_2$, ou se $l_2 \in \Gamma$ e $l_1 =$ "_".

Uma *correspondência* do padrão $Q[\overline{x}]$ no grafo G é um homomorfismo h de Q para G, tal que para cada nó $v \in V_Q$, $L_Q(v) \asymp L(h(u))$; e para cada aresta $e = (v_1, l, v_2)$ em Q, existe uma aresta $e' = (h(v_1), l', h(v_2))$ em G tal que $l \asymp l'$. Pode-se observar que quando l_1 é a etiqueta coringa "_" pode existir mais de uma aresta e' tal que $l \asymp l'$. A correspondência escolhe uma dessas arestas e a denota como $h(l^{v_1}_{v_2})$.

Quando é claro pelo contexto, a correspondência também pode ser denotada como um vetor $h(\overline{x})$ de entidades identificadas pelo padrão Q no grafo G, onde $h(\overline{x})$ consiste de $h(x)$ para todas as variáveis $x \in \overline{x}$.

Por exemplo, considerando o padrão de grafo Q_2 da Figura 7 e o grafo da Figura 8, onde o vértice verde representa um país e os vértices amarelos representam cidades que são capitais deste país, as correspondências no formato $h(\overline{x}) = (x, y, z)$ do padrão de grafo Q_2 para o grafo da imagem são $h(\overline{x}) = (country, city1, city2)$ e $h(\overline{x}) = (country, city2, city1)$.

3.2 Dependência Funcional

Dentro da definição de GEDs, uma dependência funcional $X \rightarrow Y$ é composta por dois conjuntos (possivelmente vazios) X e Y de literais de $\overline{x}$.

Um literal de $\overline{x}$, para $x, y \in \overline{x}$ pode ser (FAN; LU, 2017):

1. *Literal Constante* $x.A = c$, onde c é uma constante de U e A é um atributo de Υ diferente de id;
2. *Literal de Variável* $x.A = y.B$, onde A e B são atributos de Υ diferentes de id; e
3. *Literal de Id* $x.id = y.id$.

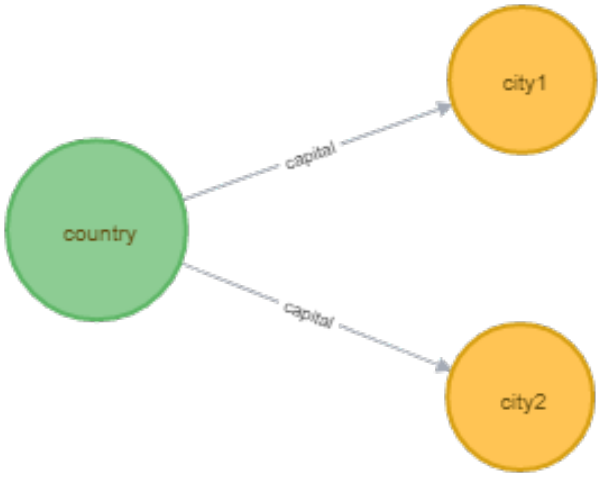

Figura 8 – Exemplo de grafo.

3.3 Exemplos de GEDs

Fan e Lu (2017) apresentam exemplos de GEDs utilizando os padrões de grafo definidos na Figura 7 e suas aplicações, alguns são listados abaixo:

1. GED $\varphi_1 = Q_1[x, y](x.type = \text{“}videogame\text{”} \rightarrow y.type = \text{“}programmer\text{”})$ Essa GED impõe que um videogame só pode ser criado por um programador. Isso impede a atribuição da criação de um videogame para uma pessoa de outra profissão, que coincidentemente tenha alguns atributos semelhantes ao programador que realmente criou.

2. GED $\varphi_2 = Q_2[x, y, z](\emptyset \rightarrow y.name = z.name)$ Essa GED impõe que, se existir mais do que um vértice etiquetado como “Cidade” representando a capital de um país, todos esses vértices devem possuir o mesmo nome. Isso impede que um país possua mais de uma capital, enquanto permite redundância de informações.

3. GED $\varphi_3 = Q_3[x, y](x.A = x.A \rightarrow y.A = x.A)$ Essa GED diz que, se y “é um” x e x tem a propriedade A, então y herda essa propriedade e possui o mesmo valor.

4. GED $\varphi_4 = Q_4[x, y](\emptyset \rightarrow \texttt{false})$ Essa GED impõe que o padrão de grafo Q_4 é inválido, por ser absurdo. Nenhuma “pessoa” pode ser simultaneamente mãe e filha de outra “pessoa”. Essa GED pode ser utilizada para detectar essa falha.

5. GED $\varphi_5 = Q_5[x, x', z_1, z_2, y_1, \ldots, y_k](X_5 \rightarrow Y_5)$, onde $X_5 = \{x'.is\_fake = 1, z_1.keyword = c, z_2.keyword = c\}$ e $Y_5 = \{x.is\_fake = 1\}$ e c é uma constante. Essa GED pode ser utilizada pra detectar contas falsas. Para contas e blogs que correspondam à Q_5, se a conta x' for confirmada como falsa e ambos os blogs $z1$ e $z2$ conterem uma keyword específica c, então x também é uma conta falsa.

3.4 Satisfatibilidade

Para interpretar a GED $\varphi = Q[\overline{x}](X \rightarrow Y)$, Fan e Lu (2017) utiliza as seguintes notações. Considerando uma correspondência $h(\overline{x})$ de Q num grafo G, e um literal l de $\overline{x}$. É dito que $h(\overline{x})$ *satisfaz* l, denotado como $h(x) \models l$, se:

1. quando l é um literal constante $x.A = c$, então o atributo A existe no nó $v = h(x)$ e $v.A = c$;
2. quando l é um literal de variável $x.A = y.B$, então o atributo A existe em $v_1 = h(x)$ e o atributo B existe em $v_2 = h(y)$ e $v_1.A = v_2.B$; e
3. quando l é um literal de id $x.id = y.id$, então $h(x)$ e $h(y)$ se referem ao mesmo vértice, ou seja, eles possuem o mesmo conjunto de atributos e arestas.

Denota-se por $h(\overline{x}) \models X$ se a correspondência $h(\overline{x})$ satisfaz *todos* os literais $l \in X$. Em particular, se $X = \emptyset$, então $h(\overline{x}) \models X$.

Escreve-se $h(\overline{x}) \models X \rightarrow Y$ se $h(\overline{x}) \models X$ *implica* em $h(\overline{x}) \models Y$. Isto significa que, quando $h(\overline{x}) \models X \rightarrow Y$, se $h(\overline{x}) \models X$ então obrigatoriamente $h(\overline{x}) \models Y$.

Um grafo G *satisfaz* a GED $\varphi = Q[\overline{x}](X \rightarrow Y)$, denotada por $G \models \varphi$, se para *todas* as correspondências $h(\overline{x})$ de Q em G, $h(\overline{x}) \models X \rightarrow Y$.

3.5 Limitação da GED

Pode-se observar que os atributos não são especificados no padrão de grafo e que são considerados grafos sem esquema. Por isso, para um literal constante $x.A = c$, o nó $h(x)$ da correspondência $h(\overline{x})$ não possui necessariamente o atributo A. Quando $x.A = c$ é um literal em X, se $h(x)$ não possui o atributo A, então $h(\overline{x})$ trivialmente satisfaz $X \rightarrow Y$ pela definição de satisfação (se $h(\overline{x})$ não satisfaz X, $h(\overline{x})$ não precisa satisfazer Y para satisfazer $X \rightarrow Y$. Mas se $x.A = c$ for um literal em Y, entao para que $h(\overline{x}) \models Y$ o vértice $h(x)$ deve ter obrigatoriamente o atributo A. O mesmo ocorre para outros tipos de literais (FAN; LU, 2017).

Assim é possível criar uma GED $Q[x](\emptyset \rightarrow x.A = x.A)$ para obrigar que todas as entidades do padrão Q tenham obrigatoriamente o atributo A. Isso é útil para, por exemplo, assegurar que todos os vértices de um grafo etiquetados como “Pessoa” possuam o atributo “nome”, definindo o padrão de grafo Q como um único vértice com etiqueta “Pessoa” e o atributo A como “nome”. Mas utilizando apenas GEDs não é possível impor que o atributo $x.A$ tenha um domínio finito, por exemplo $\mathbb{N}$, o que representa uma limitação em relação a um esquema de banco de dados.

4 Desenvolvimento

A implementação da GED foi feita na linguagem Java utilizando a IDE Eclipse, a ferramenta Apache Maven[1] para gerenciar o projeto e o SGBDG Neo4j. Um ponto positivo de trabalhar com o Neo4j é que, utilizando o aplicativo Neo4j Desktop[2], é possível visualizar o banco de dados e os resultados das consultas.

O plugin Neo4j Java Driver [3] foi utilizado para conectar e interagir com o banco de dados do Neo4j. Ele é um driver oficialmente suportado pelo Neo4j que se conecta ao banco de dados utilizando o protocolo binário. Para adicionar o Neo4j Java Driver utilizando o Maven basta adicionar as seguintes linhas de código no arquivo de configuração pom.xml:

```
<groupId>org.neo4j.driver</groupId>
<artifactId>neo4j-java-driver</artifactId>
<version>1.4.4</version>
```

O Maven é responsável por baixar e configurar as bibliotecas do driver automaticamente para que sejam utilizadas no projeto. Com esse driver é possível se conectar em um banco de dados do Neo4j ativo e executar consultas ou fazer modificações nele. Inicialmente é preciso criar um novo banco de dados no Neo4j.

4.1 Configuração e conexão ao banco de dados

4.1.1 Criação de um novo banco de dados em grafo

Utilizando o aplicativo Neo4j Desktop, cria-se um novo projeto clicando em `New` na aba Projects (Figura 9).

[1] https://maven.apache.org/
[2] https://neo4j.com/developer/neo4j-desktop/
[3] https://neo4j.com/developer/java/#neo4j-java-driver

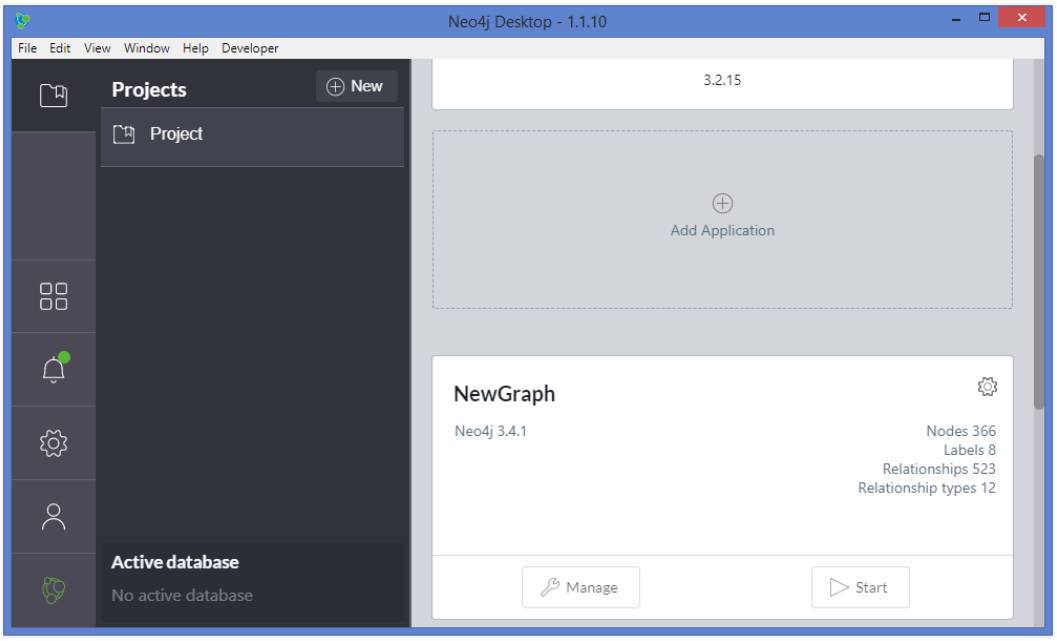

Figura 9 – Criação de um novo projeto.

Em seguida, cria-se um novo banco de dados em grafo nesse novo projeto clicando em `Add Graph` (Figura 10).

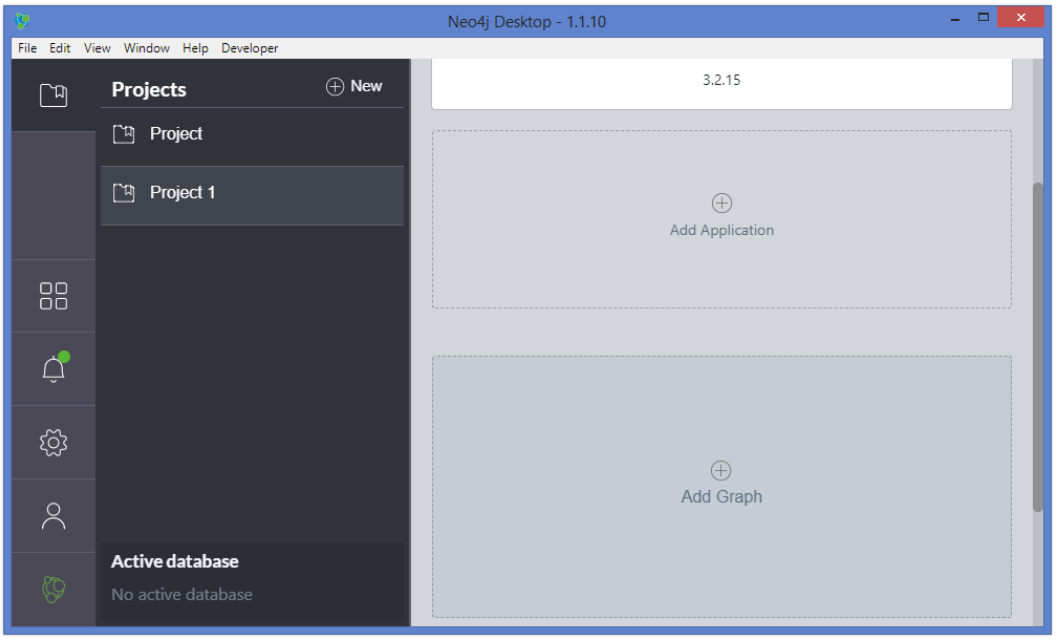

Figura 10 – Criação de um novo banco de dados em grafo.

O usuário pode escolher entre criar um grafo local ou se conectar a um grafo remoto, para a implementação deste trabalho a primeira opção foi escolhida clicando em `Create a Local Graph` (Figura 11).

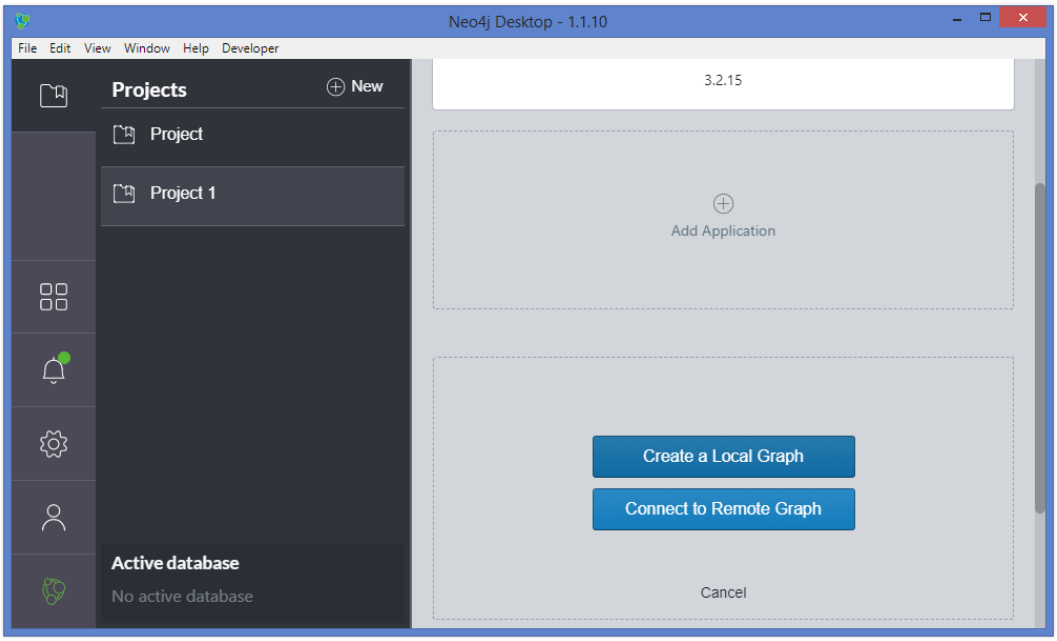

Figura 11 – Criação de um banco de dados em grafo local.

Ao criar um grafo local o usuário deve escolher um nome e uma senha para o grafo e clicar em `Create` (Figura 12). Neste trabalho foi utilizada a palavra "Graph" como nome e senha do grafo criado, mas é recomendável a escolha de uma senha mais segura se privacidade for uma preocupação.

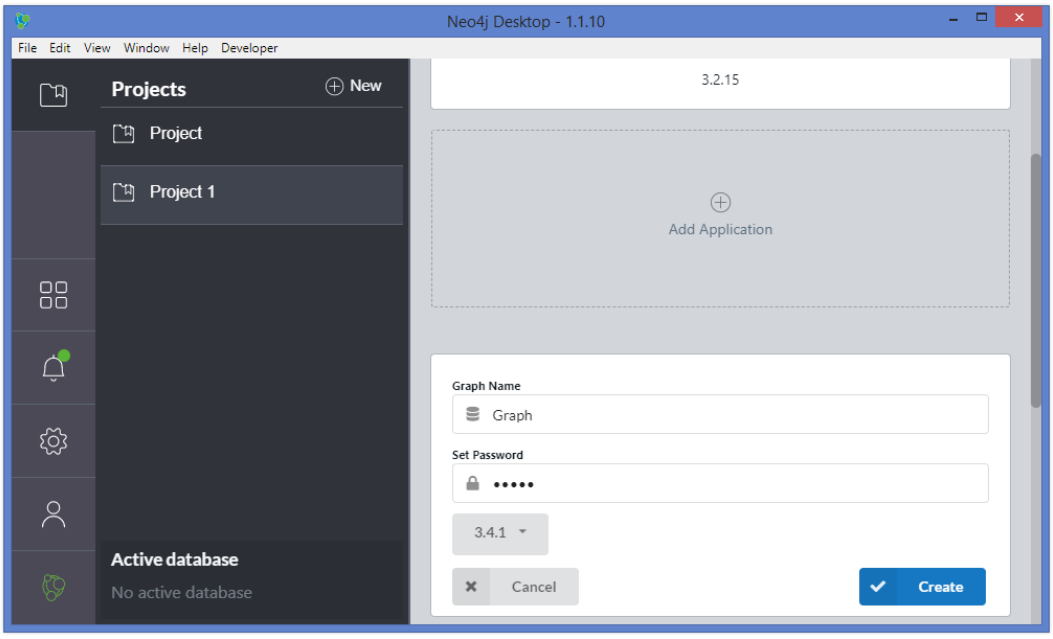

Figura 12 – Configuração de Nome e Senha do novo banco de dados em grafo.

Após a criação do banco de dados, é possível iniciá-lo clicando em `Start` (Figura 13). Só é permitido se conectar a um banco de dados que esteja ativo no momento.

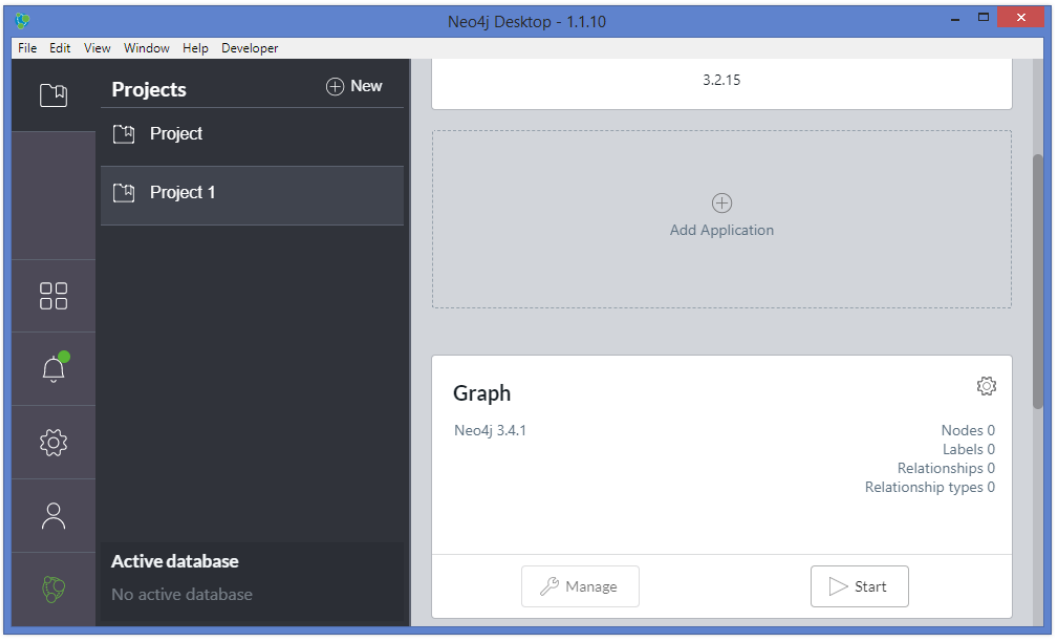

Figura 13 – Inicialização do banco de dados em grafo.

4.1.2 Conexão a um banco de dados ativo

O plugin Neo4j Java Driver foi usado para conectar a um banco de dados ativo. Todas as informações sobre as classes e métodos desse plugin podem ser encontradas em sua API [4].

Para isso deve-se criar um novo objeto da interface Driver[5], que é um acessador para um banco de dados específico do Neo4j, usando como parâmetros o número da porta Bolt e as informações de autenticação (nome de usuário e senha).

```
Driver driver = GraphDatabase.driver(
        bolt_port, AuthTokens.basic(username, password));
```

O número da porta Bolt do banco de dados pode ser visualizado clicando no botão `Manage` do grafo ativo no Neo4j Desktop, na aba `Details` (Figura 14).

Depois que a conexão é estabelecida, cria-se uma nova sessão. Uma sessão fornece um contexto de trabalho para interações no banco de dados, hospedando uma série de transações que serão realizadas nele. Para isso, basta criar um novo objeto da interface Session[6].

```
Session session = driver.session();
```

Consultas e modificações no banco de dados podem ser feitas como uma transação por meio dessa sessão, enviando um comando na linguagem Cypher para o banco. Observe

[4] https://neo4j.com/docs/api/java-driver/current/
[5] https://neo4j.com/docs/api/java-driver/current/org/neo4j/driver/v1/Driver.html
[6] https://neo4j.com/docs/api/java-driver/current/org/neo4j/driver/v1/Session.html

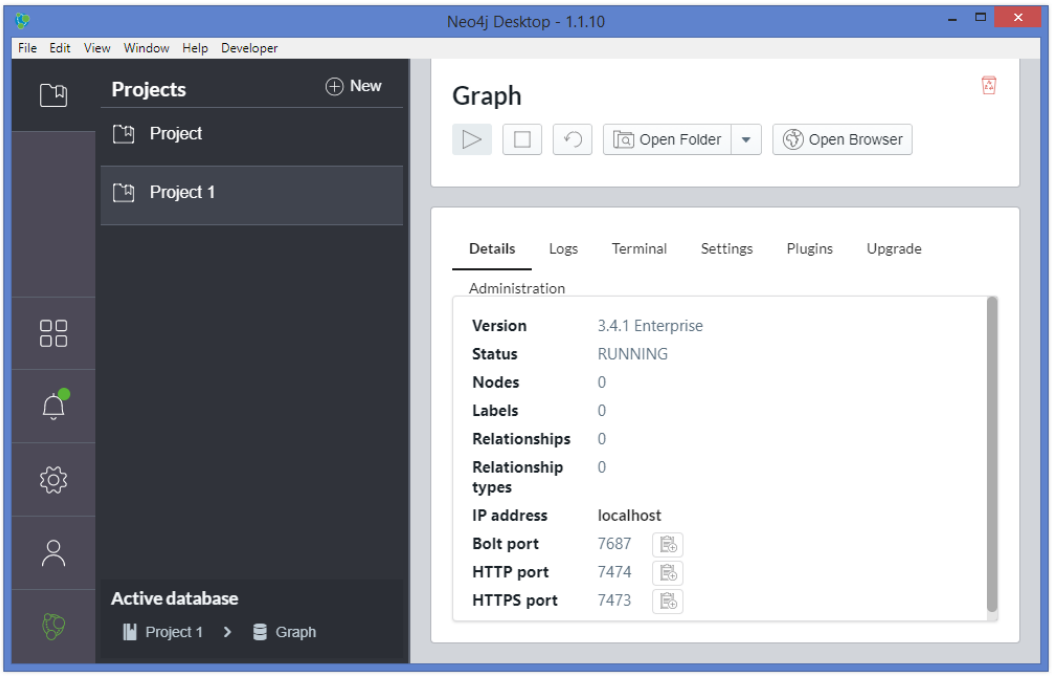

Figura 14 – Gerenciamento do banco de dados.

que também é possível consultar e modificar o banco rodando comandos diretamente no Neo4j Desktop.

4.1.3 Adição de entidades e relacionamentos ao banco

Para adicionar entidades e relacionamentos ao banco, a sessão criada anteriormente é utilizada para enviar comandos `CREATE` pela aplicação.

```
Session session = driver.session();
String query = "CREATE (a:person name: 'Jhulia', type: 'programmer')\r"
         + "CREATE (b:person name: 'Arthur', type: 'engineer')\r"
         + "CREATE (c:person name: 'Leticia', type: 'engineer')\r"
         + "CREATE (x:product name: 'Olympic', type: 'videogame')\r"
         + "CREATE (y:product name: 'Olivia', type: 'bridge')\r"
         + "CREATE (a)-[:create]->(x)\r"
         + "CREATE (b)-[:create]->(y)\r"
         + "CREATE (c)-[:create]->(y)\r";
session.run(query);
```

Após a execução do comando `run` acima o banco de dados possui cinco novas entidades e três novos relacionamentos, que podem ser visualizados no Neo4j Desktop como mostrado na Figura 15.

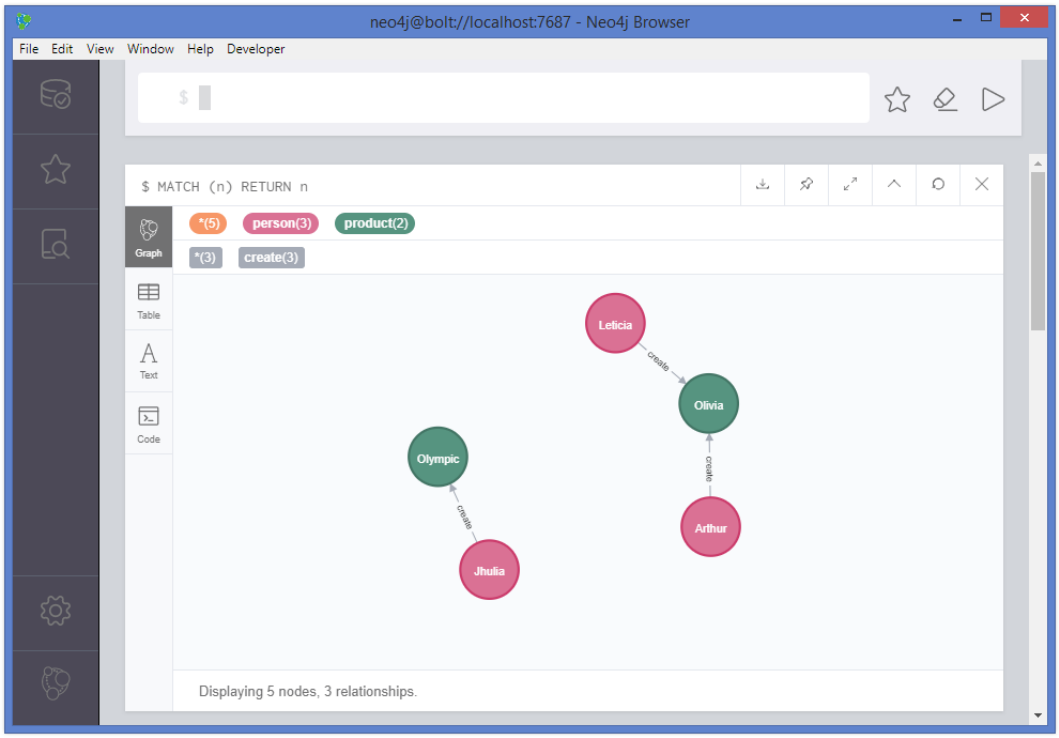

Figura 15 – Visualização dos dados do banco.

4.2 Implementação

Nesta sessão serão descritas todas as classes criadas no projeto, toda a implementação foi feita na linguagem Java. É possível ter uma visão geral da API observando o diagrama de classes UML da Figura 16.

4.2.1 Interface Literal

A interface `Literal` define um método que todas as classes que a implementarem devem definir. O método `isSatisfiedBy(Record r)` verifica se a correspondência `r` satisfaz o literal ou não.

Listing 4.1 – "Literal.java"

```
1 package pg.graph;
2
3 import org.neo4j.driver.v1.Record;
4
5 public interface Literal {
6     public boolean isSatisfiedBy(Record r);
7 }
```

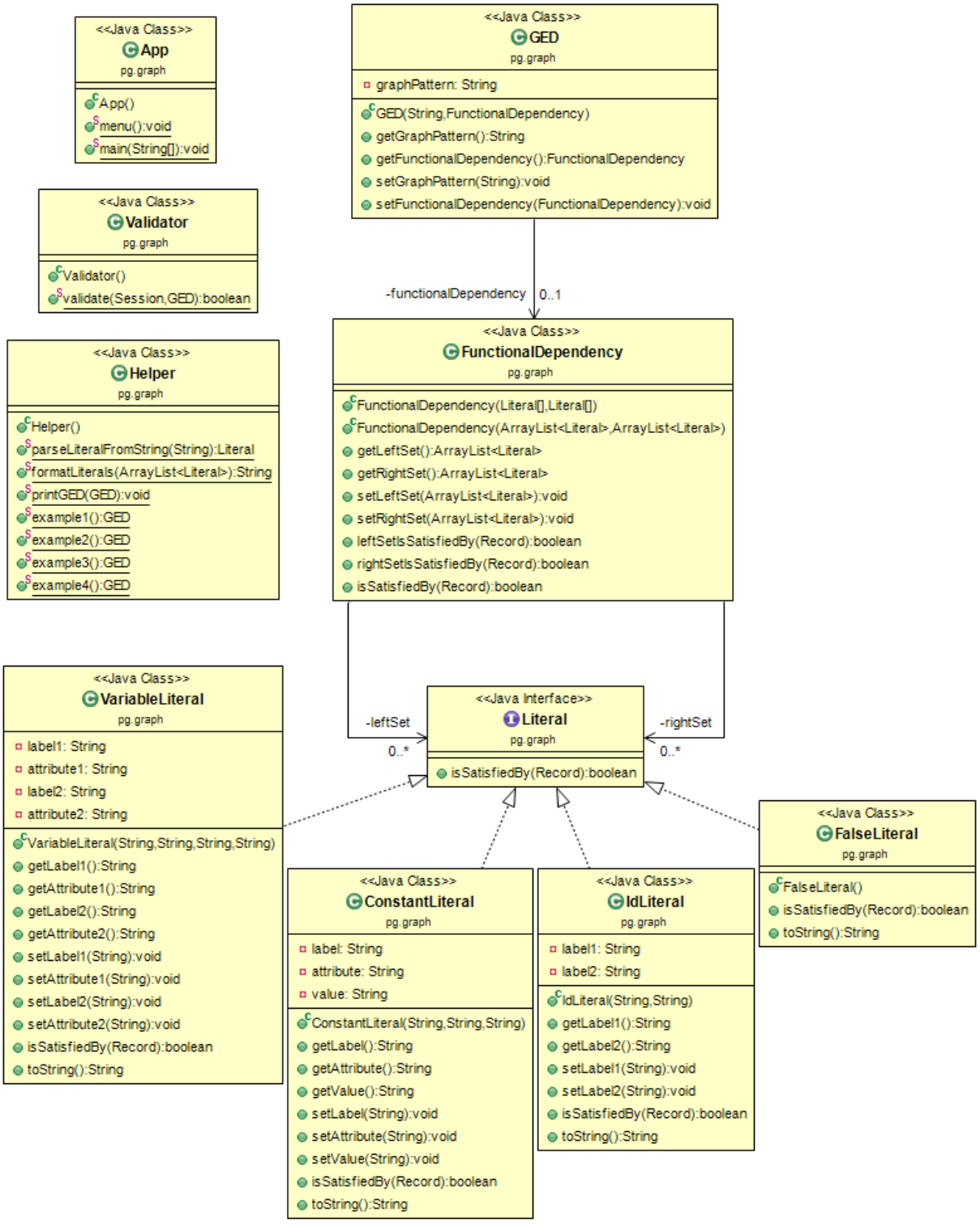

Figura 16 – Diagrama de classes do projeto.

4.2.2 Classe ConstantLiteral

A classe `ConstantLiteral` implementa a interface `Literal` e representa um literal constante $x.A = c$. Ela possui como atributos três Strings `label`, `attribute` e `value`, que representam a etiqueta, o nome do atributo e o valor da constante do literal constante.

Por exemplo, para $x.A = c$, temos *label* =“x”, *attribute* =“A” e *value* =“c”. Ela sobrescreve o método `isSatisfiedBy(Record r)` da interface `Literal` e possui um método `toString()` que retorna uma String com seus atributos no formato “label.attribute = value”.

Listing 4.2 – "ConstantLiteral.java"

```
1  package pg.graph;
2
3  import org.neo4j.driver.v1.Record;
4  import org.neo4j.driver.v1.Value;
5
6  public class ConstantLiteral implements Literal {
7      private String label;
8      private String attribute;
9      private String value;
10
11     public ConstantLiteral(String label, String attribute, String value) {
12         this.label = label;
13         this.attribute = attribute;
14         this.value = value;
15     }
16     public String getLabel() {
17         return this.label;
18     }
19     public String getAttribute() {
20         return this.attribute;
21     }
22     public String getValue() {
23         return this.value;
24     }
25     public void setLabel(String label) {
26         this.label = label;
27     }
28     public void setAttribute(String attribute) {
29         this.attribute = attribute;
30     }
31     public void setValue(String value) {
32         this.value = value;
33     }
34     public boolean isSatisfiedBy(Record r) {
35         Value v = r.get(this.label);
36         if (v.isNull()) return false;
37
38         Value a = v.get(this.attribute);
39         if (a.isNull()) return false;
40         return a.toString().equals(this.value);
41     }
```

```
42      public String toString() {
43          return this.label + "." + this.attribute + " = " + this.value;
44      }
45  }
```

4.2.3 Classe VariableLiteral

A classe `VariableLiteral` implementa a interface `Literal` e representa um literal de variável $x.A = y.B$. Ela possui como atributos quatro Strings `label1`, `label2`, `attribute1` e `attribute2`, que representam as etiquetas e os nomes dos atributos do literal de variável. Por exemplo, para $x.A = y.B$, temos $label1$ =“x”, $label2$ =“y”, $attribute1$ =“A” e $attribute2$ =“B”. Ela sobrescreve o método `isSatisfiedBy(Record r)` da interface `Literal` e possui um método `toString()` que retorna uma String com seus atributos no formato “label1.attribute1 = label2.attribute2”.

Listing 4.3 – "VariableLiteral"

```
1  package pg.graph;
2
3  import org.neo4j.driver.v1.Record;
4  import org.neo4j.driver.v1.Value;
5
6  public class VariableLiteral implements Literal {
7      private String label1;
8      private String attribute1;
9      private String label2;
10     private String attribute2;
11
12     public VariableLiteral (String label1, String attribute1,
13                             String label2, String attribute2) {
14         this.label1 = label1;
15         this.attribute1 = attribute1;
16         this.label2 = label2;
17         this.attribute2 = attribute2;
18     }
19     public String getLabel1() {
20         return this.label1;
21     }
22     public String getAttribute1() {
23         return this.attribute1;
24     }
25     public String getLabel2() {
26         return this.label2;
27     }
28     public String getAttribute2() {
29         return this.attribute2;
30     }
```

```
31      public void setLabel1(String label) {
32          this.label1 = label;
33      }
34      public void setAttribute1(String attribute) {
35          this.attribute1 = attribute;
36      }
37      public void setLabel2(String label) {
38          this.label2 = label;
39      }
40      public void setAttribute2(String attribute) {
41          this.attribute2 = attribute;
42      }
43      public boolean isSatisfiedBy(Record r) {
44          Value v1 = r.get(this.label1);
45          Value v2 = r.get(this.label2);
46          if (v1.isNull() || v2.isNull()) return false;
47
48          Value a1 = v1.get(this.attribute1);
49          Value a2 = v2.get(this.attribute2);
50          if (a1.isNull() || a2.isNull()) return false;
51          return a1.equals(a2);
52      }
53      public String toString() {
54          return this.label1 + "." + this.attribute1 + " = "
55                  + this.label2 + "." + this.attribute2;
56      }
57  }
```

4.2.4 Classe IdLiteral

A classe `IdLiteral` implementa a interface `Literal` e representa um literal de id $x.id = y.id$. Ela possui como atributos duas Strings `label1` e `label2`, que representam as etiquetas do literal de id. Por exemplo, para $x.id = y.id$, temos $label1 =$"x", $label2 =$"y". Ela sobrescreve o método `isSatisfiedBy(Record r)` da interface `Literal` e possui um método `toString()` que retorna uma String com seus atributos no formato "label1.id = label2.id".

Listing 4.4 – "IdLiteral.java"

```
1  package pg.graph;
2
3  import org.neo4j.driver.v1.Record;
4  import org.neo4j.driver.v1.Value;
5
6  public class IdLiteral implements Literal {
7      private String label1;
8      private String label2;
```

```
9
10      public IdLiteral(String label1, String label2) {
11          this.label1 = label1;
12          this.label2 = label2;
13      }
14      public String getLabel1() {
15          return this.label1;
16      }
17      public String getLabel2() {
18          return this.label2;
19      }
20      public void setLabel1(String label) {
21          this.label1 = label;
22      }
23      public void setLabel2(String label) {
24          this.label2 = label;
25      }
26      public boolean isSatisfiedBy(Record r) {
27          Value v1 = r.get(this.label1);
28          Value v2 = r.get(this.label2);
29          if (v1.isNull() || v2.isNull()) return false;
30
31          Value a1 = v1.get("id");
32          Value a2 = v2.get("id");
33          if (a1.isNull() || a2.isNull()) return false;
34          return a1.equals(a2);
35      }
36      public String toString() {
37          return this.label1 + ".id = " + this.label2 + ".id";
38      }
39  }
```

4.2.5 Classe FalseLiteral

A classe `FalseLiteral` implementa a interface `Literal` e representa um literal especial que não é satisfeito por nenhuma correspondência. Ele é útil para invalidar GEDs que tenham padrões de grafo absurdos, como no exemplo da GED φ_4 na Seção 3.3. Ela sobrescreve o método `isSatisfiedBy(Record r)` da interface `Literal` e possui um método `toString()` que retorna a String "false".

Listing 4.5 – "FalseLiteral.java"

```
1  package pg.graph;
2
3  import org.neo4j.driver.v1.Record;
4
5  public class FalseLiteral implements Literal{
```

```
6       public boolean isSatisfiedBy(Record r) {
7           return false;
8       }
9       public String toString() {
10          return "false";
11      }
12  }
```

4.2.6 Classe FunctionalDependency

Uma dependência funcional $X \rightarrow Y$ é composta por dois conjuntos de literais X e Y. A classe `FunctionalDependency` possui como atributos dois conjuntos de literais, `leftSet` e `rightSet`.

Além dos construtores, *getters* e *setters*, a classe possui os métodos `leftSetIsSatisfiedBy(Record)` e `rightSetIsSatisfiedBy(Record r)`, que verificam se a correspondência em `r` satisfaz o conjunto de literais `leftSet` e `rightSet`, respectivamente, e o método `isSatisfiedBy(Record r)` que verifica se a correspondência `r` satisfaz a dependência funcional.

Uma correspondência satisfaz a dependência funcional quando se o conjunto de literais em `leftSet` for satisfeito, o conjunto de literais em `rightSet` também for satisfeito. Isso significa que se o conjunto de literais em `leftSet` não for satisfeito, a dependência funcional é automaticamente satisfeita.

Listing 4.6 – "FunctionalDependency.java"

```
1  package pg.graph;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5
6  import org.neo4j.driver.v1.Record;
7
8  public class FunctionalDependency {
9      private ArrayList<Literal> leftSet;
10     private ArrayList<Literal> rightSet;
11
12     public FunctionalDependency (Literal[] left, Literal[] right) {
13         this.leftSet = new ArrayList<Literal>(Arrays.asList(left));
14         this.rightSet = new ArrayList<Literal>(Arrays.asList(right));
15     }
16     public FunctionalDependency (ArrayList<Literal> left,
17                                  ArrayList<Literal> right) {
18         this.leftSet = left;
19         this.rightSet = right;
20     }
```

```
21      public ArrayList<Literal> getLeftSet() {
22          return this.leftSet;
23      }
24      public ArrayList<Literal> getRightSet() {
25          return this.rightSet;
26      }
27      public void setLeftSet(ArrayList<Literal> l) {
28          this.leftSet = l;
29      }
30      public void setRightSet(ArrayList<Literal> r) {
31          this.rightSet = r;
32      }
33      public boolean leftSetIsSatisfiedBy(Record r) {
34          for (Literal l : this.leftSet) {
35              if (!l.isSatisfiedBy(r)) {
36                  return false;
37              }
38          }
39          return true;
40      }
41      public boolean rightSetIsSatisfiedBy(Record r) {
42          for (Literal l : this.rightSet) {
43              if (!l.isSatisfiedBy(r)) {
44                  return false;
45              }
46          }
47          return true;
48      }
49      public boolean isSatisfiedBy(Record r) {
50          return (!leftSetIsSatisfiedBy(r) || rightSetIsSatisfiedBy(r));
51      }
52  }
```

4.2.7 Classe GED

Uma GED é uma combinação de um padrão de grafo Q e uma dependência funcional $X \rightarrow Y$, então naturalmente a classe GED possui como atributos esses dois itens. A classe possui como métodos apenas um construtor e os *getters* e *setters* padrões.

O padrão de grafo `graphPattern` é uma `String` simples de Java, representando uma consulta em Cypher que retorna as correspondências ao padrão desejado. A dependência funcional `functionalDependency` é um objeto da classe `FunctionalDependency`.

Listing 4.7 – "GED.java"

```
1  package pg.graph;
2
```

```
3  public class GED {
4      private String graphPattern;
5      private FunctionalDependency functionalDependency;
6
7      public GED(String pattern, FunctionalDependency fd) {
8          this.graphPattern = pattern;
9          this.functionalDependency = fd;
10     }
11     public String getGraphPattern() {
12         return this.graphPattern;
13     }
14     public FunctionalDependency getFunctionalDependency() {
15         return this.functionalDependency;
16     }
17     public void setGraphPattern(String graphPattern) {
18         this.graphPattern = graphPattern;
19     }
20     public void setFunctionalDependency(FunctionalDependency fd) {
21         this.functionalDependency = fd;
22     }
23 }
```

4.2.8 Classe Validator

A classe `Validator` serve apenas como um validador de GEDs. Ela possui um único método estático `validate(Session s, GED ged)` que verifica se a GED é válida para o grafo conectado na sessão ativa.

Listing 4.8 – "Validator.java"

```
1  package pg.graph;
2
3  import java.util.List;
4
5  import org.neo4j.driver.v1.Record;
6  import org.neo4j.driver.v1.Session;
7  import org.neo4j.driver.v1.StatementResult;
8
9  public class Validator {
10     public static boolean validate(Session s, GED ged) {
11         StatementResult res = s.run(ged.getGraphPattern());
12         List<Record> records = res.list();
13         for (Record r : records) {
14             if (!ged.getFunctionalDependency().isSatisfiedBy(r))
15                 return false;
16         }
17         return true;
```

```
18     }
19 }
```

4.2.9 Aplicação

A classe `Helper` foi criada para agrupar um conjunto de métodos estáticos de utilidade para auxiliar na criação da aplicação. Nesta classe há métodos para imprimir GEDs de maneira organizada, um parser para criar um `Literal` a partir de uma String e um conjunto de métodos que criam as GEDs $\varphi_1 - \varphi_4$ listadas na Seção 3.3.

Listing 4.9 – "Helper.java"

```
1  package pg.graph;
2
3  import java.util.ArrayList;
4
5  public class Helper {
6      public static Literal parseLiteralFromString(String s) {
7          s = s.replaceAll("\\s+","");  // remove empty spaces
8          String symbols = "";
9          for (int i=0; i<s.length(); i++) {
10             if (s.charAt(i) == '.' || s.charAt(i) == '=') {
11                 symbols += s.charAt(i);
12             }
13         }
14         if (symbols.equals("")) {  // this must be a FalseLiteral
15             if (s.equals("False") || s.equals("false"))
16                 return new FalseLiteral();
17         } else if (symbols.equals(".=")) {  // this must be a
       ConstantLiteral
18             int i = 0;
19             String label = "";
20             String attribute = "";
21             String value = "";
22             while (i < s.length() && s.charAt(i) != '.')
23                 label += s.charAt(i++);
24             i++;
25             while (i < s.length() && s.charAt(i) != '=')
26                 attribute += s.charAt(i++);
27             i++;
28             while (i < s.length())
29                 value += s.charAt(i++);
30
31             if (!label.isEmpty() && !attribute.isEmpty())
32                 return new ConstantLiteral(label, attribute, value);
33         } else if (symbols.equals(".=.")) {  // this may be a
       VariableLiteral
```

```
34              int i = 0;                      // or an IdLiteral
35              String label1 = "";
36              String attribute1 = "";
37              String label2 = "";
38              String attribute2 = "";
39              while (i < s.length() && s.charAt(i) != '.')
40                  label1 += s.charAt(i++);
41              i++;
42              while (i < s.length() && s.charAt(i) != '=')
43                  attribute1 += s.charAt(i++);
44              i++;
45              while (i < s.length() && s.charAt(i) != '.')
46                  label2 += s.charAt(i++);
47              i++;
48              while (i < s.length())
49                  attribute2 += s.charAt(i++);
50
51              if (!label1.isEmpty() && !attribute1.isEmpty()
52                      && !label2.isEmpty() && !attribute2.isEmpty()) {
53                  if (attribute1.equals("id") && attribute2.equals("id"))
54                      return new IdLiteral(label1, label2);
55                  if (!attribute1.equals("id") && !attribute2.equals("id"))
56                      return new VariableLiteral(label1, attribute1,
57                                                 label2, attribute2);
58              }
59          }
60          return null;  // couldn't parse String s
61      }
62      public static String formatLiterals(ArrayList<Literal> literals) {
63          if (literals.isEmpty()) {
64              return "{}";
65          }
66          String formatedLiterals = literals.get(0).toString();
67          for (int i=1; i<literals.size(); i++) {
68              formatedLiterals += ", " + literals.get(1).toString();
69          }
70          return "{" + formatedLiterals + "}";
71      }
72      public static void printGED(GED ged) {
73          System.out.println("Graph pattern: " + ged.getGraphPattern());
74          System.out.println("Functional dependency: ");
75          System.out.println("Left side: " + formatLiterals(
76                  ged.getFunctionalDependency().getLeftSet()));
77          System.out.println("Right side: " + formatLiterals(
78                  ged.getFunctionalDependency().getRightSet()));
79      }
80      public static GED example1() {
```

```
81          String graphPattern =
82                  "MATCH (x:product)<-[:create]-(y:person) RETURN x, y";
83          FunctionalDependency fd = new FunctionalDependency(
84                  new Literal[]{new
85                          ConstantLiteral("x", "type", "videogame")},
86                  new Literal[]{new
87                          ConstantLiteral("y", "type", "programmer")});
88          return new GED(graphPattern, fd);
89      }
90      public static GED example2() {
91          String graphPattern = "MATCH (y:city)<-[:capital]-(x:country)"
92                  + "-[:capital]->(z:city) RETURN x, y, z";
93          FunctionalDependency fd = new FunctionalDependency(
94                  new Literal[] {},
95                  new Literal[] {new VariableLiteral("y", "name",
96                                                     "z", "name")});
97          return new GED(graphPattern, fd);
98      }
99      public static GED example3() {
100         String graphPattern =  "MATCH (x)<-[:is_a]-(y) RETURN x, y";
101         FunctionalDependency fd = new FunctionalDependency(
102                 new Literal[] {new VariableLiteral("x", "can_fly",
103                                                    "x", "can_fly")},
104                 new Literal[] {new VariableLiteral("y", "can_fly",
105                                                    "x", "can_fly")});
106         return new GED(graphPattern, fd);
107     }
108     public static GED example4() {
109         String graphPattern =  "MATCH (x:person)<-[:child]-(y:person)"
110                 + " WHERE (x)<-[:parent]-(y) RETURN x, y";
111         FunctionalDependency fd = new FunctionalDependency(
112                 new Literal[] {},
113                 new Literal[] {new FalseLiteral()});
114         return new GED(graphPattern, fd);
115     }
116 }
```

A classe `App` que possui o método `main()` do projeto representa uma aplicação interativa simples para testar as classes. Ela se conecta a um banco de dados específico do Neo4j e permite a criação e validação de novas GEDs pelo usuário, a execução de comandos Cypher no banco e a validação dos exemplos de GEDs criados por métodos da classe `Helper`.

Listing 4.10 – "App.java"

```
1 package pg.graph;
2
3 import java.util.ArrayList;
```

```
4  import java.util.Scanner;
5
6  import org.neo4j.driver.v1.AuthTokens;
7  import org.neo4j.driver.v1.Driver;
8  import org.neo4j.driver.v1.GraphDatabase;
9  import org.neo4j.driver.v1.Session;
10
11 public class App
12 {
13     public static void menu() {
14         System.out.println(" ---------------------------------------- ");
15         System.out.println("| 1) Avaliar GED exemplo                 |");
16         System.out.println("| 2) Avaliar GED criada pelo usuário     |");
17         System.out.println("| 3) Listar GEDs exemplo                 |");
18         System.out.println("| 4) Listar GEDs criadas pelo usuário    |");
19         System.out.println("| 5) Criar uma nova GED                  |");
20         System.out.println("| 6) Executar um comando no banco        |");
21         System.out.println("| 0) Encerrar                            |");
22         System.out.println(" ---------------------------------------- ");
23         System.out.print("Escolha uma opção: ");
24     }
25     public static void main(String[] args)
26     {
27         try{
28             Driver driver = GraphDatabase.driver("bolt://localhost:7687",
29                     AuthTokens.basic("neo4j", "Graph"));
30             Session session = driver.session();
31             Scanner sc = new Scanner(System.in);
32             int op;
33
34             GED[] example_geds = new GED[] {Helper.example1(),
35                     Helper.example2(), Helper.example3(),
36                     Helper.example4()};
37             ArrayList<GED> user_created_geds = new ArrayList<GED>();
38
39             do {
40                 menu();
41                 op = sc.nextInt(); sc.nextLine();
42                 switch(op) {
43                     case 1:
44                         System.out.println("Digite o id [1, 4] da GED"
45                                 + " exemplo que deseja avaliar: ");
46                         int id = sc.nextInt(); sc.nextLine();
47                         if (id >= 1 && id <= 4) {
48                             Helper.printGED(example_geds[id-1]);
49                             System.out.println(Validator.validate(
50                                     session, example_geds[id-1]) ?
```

```
51                                  "GED válida." : "GED inválida.");
52                          } else {
53                              System.out.println("Índice inválido.");
54                          }
55                          break;
56                      case 2:
57                          System.out.print("Digite o id [0, "
58                                  + user_created_geds.size() + "] da GED"
59                                  + " criada por usuário que deseja"
60                                  + " avaliar: ");
61                          id = sc.nextInt(); sc.nextLine();
62                          if (id >= 0 && id < user_created_geds.size()) {
63                              Helper.printGED(user_created_geds.get(id));
64                              System.out.println(Validator.validate(
65                                      session, user_created_geds.get(id)) ?
66                                      "GED válida." : "GED inválida.");
67                          } else {
68                              System.out.println("Índice inválido.");
69                          }
70                          break;
71                      case 3:
72                          System.out.println("Lista de GEDs exemplo:");
73                          for (int i=1; i<=4; i++) {
74                              System.out.println("GED exemplo " + i + ":");
75                              Helper.printGED(example_geds[i-1]);
76                          }
77                          break;
78                      case 4:
79                          System.out.println("Lista de GEDs criadas pelo"
80                                  + "usuário:");
81                          for (int i=0; i<user_created_geds.size(); i++) {
82                              System.out.println("GED criada pelo usuário "
83                                      + i + ":");
84                              Helper.printGED(user_created_geds.get(i));
85                          }
86                          break;
87                      case 5:
88                          System.out.print("Digite a consulta em Cypher que"
89                                  + " retorna o padrão de grafo da GED: ");
90                          String graphPattern = sc.nextLine();
91
92                          System.out.print("Digite a quantidade de literais"
93                                  + " no conjunto esquerdo: ");
94                          int n = sc.nextInt(); sc.nextLine();
95                          ArrayList<Literal> leftSet =
96                                  new ArrayList<Literal>();
97                          for (int i=0; i<n; i++) {
```

```
98                          System.out.print("Digite o literal " + i
99                                  + ": ");
100                         Literal literal = Helper.
101                                 parseLiteralFromString(sc.nextLine());
102                         if (literal == null) {
103                             System.out.println("Não foi possível"
104                                     + " parsear a string para um"
105                                     + " literal.");
106                         } else {
107                             leftSet.add(literal);
108                         }
109                     }
110
111                     System.out.print("Digite a quantidade de literais"
112                             + " no conjunto direito: ");
113                     n = sc.nextInt(); sc.nextLine();
114                     ArrayList<Literal> rightSet =
115                             new ArrayList<Literal>();
116                     for (int i=0; i<n; i++) {
117                         System.out.print("Digite o literal " + i
118                                 + ": ");
119                         Literal literal = Helper.
120                                 parseLiteralFromString(sc.nextLine());
121                         if (literal == null) {
122                             System.out.println("Não foi possível"
123                                     + " parsear a string para um"
124                                     + " literal.");
125                         } else {
126                             rightSet.add(literal);
127                         }
128                     }
129                     user_created_geds.add(new GED(graphPattern, new
130                             FunctionalDependency(leftSet, rightSet)));
131                     break;
132                 case 6:
133                     System.out.print("Digite o comando que deseja"
134                             + " executar: ");
135                     String query = sc.nextLine();
136                     try {
137                         session.run(query);
138                         System.out.println("Comando executado com"
139                                 + " sucesso.");
140                     } catch(Exception e) {
141                         System.out.println(e.getMessage());
142                         System.out.println("Falha na execução.");
143                     }
144                     break;
```

```
145                     case 0:
146                         System.out.println("O programa será encerrado.");
147                         session.close();
148                         driver.close();
149                         sc.close();
150                         break;
151                     default:
152                         System.out.println("Opção inválida.");
153                 }
154             } while(op != 0);
155         } catch(Exception e) {
156             System.out.println(e.getMessage());
157             e.getStackTrace();
158         }
159     }
160 }
```

4.3 Testes de GEDs

Foi criada uma base de dados consistente, de modo a propositalmente satisfazer todas as GEDs inicialmente, para testar a validação das GEDs $\varphi_1 - \varphi_4$ listadas na Seção 3.3. A base de dados foi criada executando o comando Cypher abaixo no Neo4j Desktop e pode ser visualizada na Figura 17.

```
 // Sample data for example 1:
CREATE (a:person {name:‘Jhulia’, type:‘programmer’})
CREATE (b:person {name:‘Leticia’, type:‘engineer’})
CREATE (c:person {name:‘Arthur’, type:‘engineer’})
CREATE (d:person {name:‘Marcos’, type:‘programmer’})
CREATE (e:person {name:‘Felipe’, type:‘engineer’})
CREATE (f:product {name:‘Olympic’, type:‘videogame’})
CREATE (g:product {name:‘Olivia’, type:‘bridge’})
CREATE (h:product {name:‘Armor System Rework Mod’, type:‘videogame mod’})
CREATE (a)-[:create]->(f)
CREATE (b)-[:create]->(g)
CREATE (c)-[:create]->(g)
CREATE (d)-[:create]->(f)
CREATE (e)-[:create]->(h)

// Sample data for example 2:
CREATE (i:country {name:‘Brasil’})
CREATE (j:city {name:‘Brasilia’})
```

```
CREATE (k:city {name:‘Brasilia’})
CREATE (l:city {name:‘Brasilia’})
CREATE (i)-[:capital]->(j)
CREATE (i)-[:capital]->(k)
CREATE (i)-[:capital]->(l)
CREATE (m:country {name:‘Some Country’})
CREATE (n:city {name:‘Some Country\’s Capital’})
CREATE (o:city {name:‘Some Country\’s Capital’})
CREATE (m)-[:capital]->(n)
CREATE (m)-[:capital]->(o)

// Sample data for example 3:
CREATE (p:bird {name:‘Bird’, can_fly:‘true’})
CREATE (q:eagle {name:‘Eagle’, can_fly:‘true’})
CREATE (q)-[:is_a]->(p)
CREATE (r:toy {name:‘Toy’, playable:true})
CREATE (s:doll {name:‘Woody’, playable:true})
CREATE (t:doll {name:‘Buzz Lightyear’, playable:true})
CREATE (u:lego {name:‘Batman Lego’, playable:true})
CREATE (s)-[:is_a]->(r)
CREATE (t)-[:is_a]->(r)
CREATE (u)-[:is_a]->(r)

// Sample data for example 4:
CREATE (v:person {name:‘Helena’})
CREATE (w:person {name:‘Domingas’})
CREATE (x:person {name:‘Fulano’})
CREATE (y:person {name:‘Ciclano’})
CREATE (v)-[:parent]->(a)
CREATE (v)-[:parent]->(c)
CREATE (w)-[:parent]->(v)
CREATE (x)-[:parent]->(y)
CREATE (a)-[:child]->(v)
CREATE (c)-[:child]->(v)
CREATE (v)-[:child]->(w)
CREATE (y)-[:child]->(x)
```

Para verificar a corretude do algoritmo, foi verificado se o grafo inicial validou todas as GEDs e foram feitas pequenas alterações no grafo, sempre verificando se o resultado das

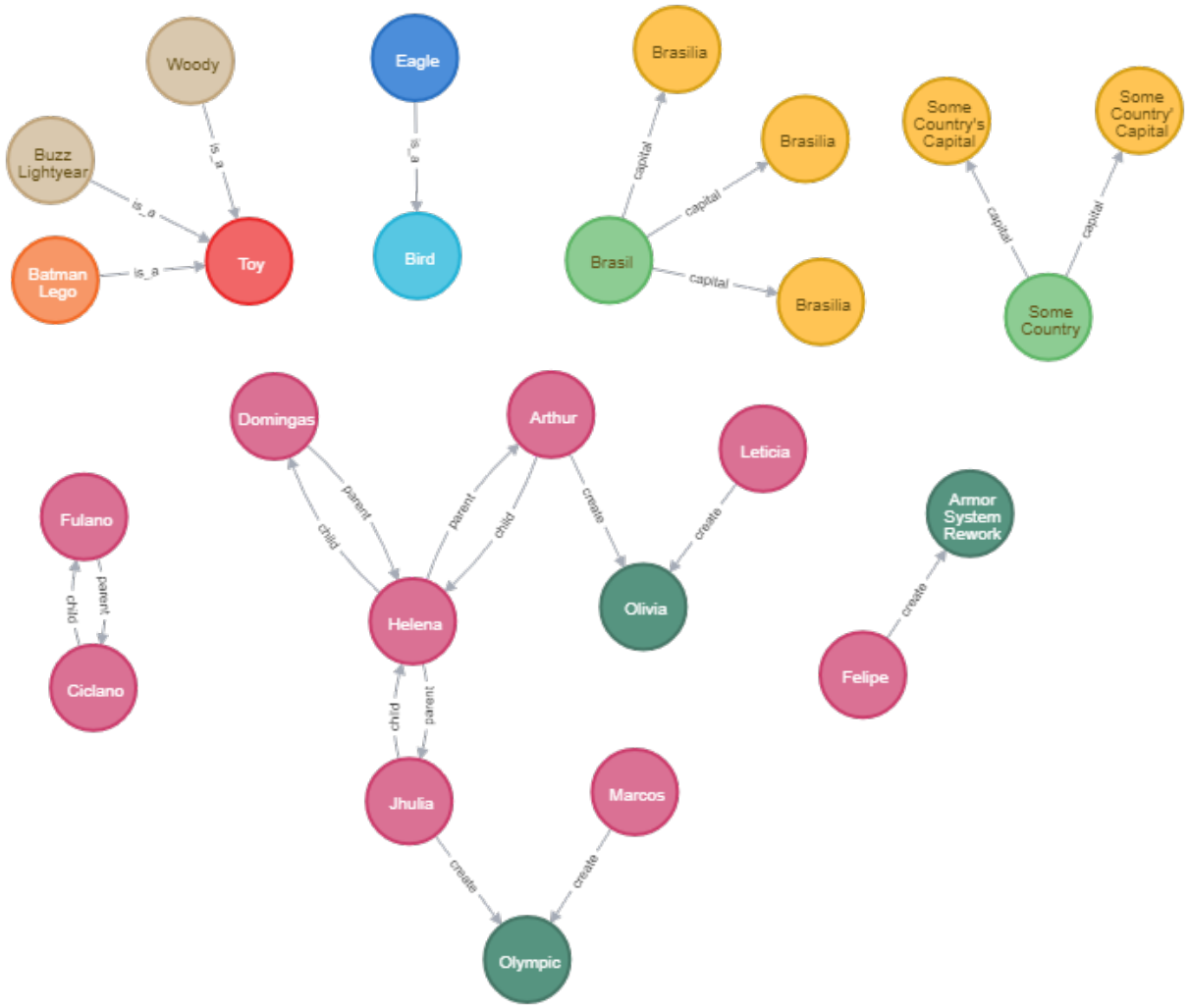

Figura 17 – Base de dados utilizada.

validações estava coerente com o esperado. A seguir estão listados alguns testes realizados:

1. O grafo inicial validou corretamente todas as GEDs.

2. O atributo `type` da entidade `person` com nome "Jhulia" foi alterado de "programmer" para "student" com o comando:

   ```
   MATCH (x:person {name:'Jhulia'}) SET x.type = 'student'
   ```

   A GED φ_1 se tornou inválida (como esperado) e a alteração foi desfeita com o comando:

   ```
   MATCH (x:person {name:'Jhulia'}) SET x.type = 'programmer'
   ```

3. O atributo `type` da entidade `product` com nome "Olivia" foi alterada de "bridge" para "videogame" com o comando:

   ```
   MATCH (x:product {name:'Olivia'}) SET x.type = 'videogame'
   ```

A GED φ_1 se tornou inválida. Então o atributo `type` de uma das criadoras da ponte "Olívia" foi alterado para "programmer" com o comando:

```
MATCH (x:person {name:'Leticia'}) SET x.type = 'programmer'
```

A GED $\$varphi_1$ continuou inválida, pois ainda havia outro criador que não era do tipo `programmer`. Então o atributo `type` do outro criador também foi alterado:

```
MATCH (x:person {name:'Arthur'}) SET x.type = 'programmer'
```

Como ambos os criadores do "videogame" chamado "Olivia" se tornaram programadores, a GED φ_1 voltou a ser válida. No final as alterações foram desfeitas:

```
MATCH (x:product {name:'Olivia'}) SET x.type = 'bridge'
MATCH (x:person {name:'Leticia'}) SET x.type = 'engineer'
MATCH (x:person {name:'Arthur'}) SET x.type = 'engineer'
```

4. Foi criada uma nova entidade `city` com nome "Other Country's Capital" e um novo relacionamento dizendo que esta nova entidade é a capital do país de nome "Some Country" com o comando:

   ```
   MATCH (x:country {name:'Some Country'})
   CREATE (y:city {name:'Other Country\'s Capital'})
   CREATE (x)-[:capital]->(y)
   ```

   E então a GED φ_2 se tornou inválida, pois existiam duas capitais de um mesmo país com nomes diferentes. Em seguida o atributo `name` da nova entidade criada foi deletado com o comando:

   ```
   MATCH (x:city {name:'Other Country\'s Capital'}) REMOVE x.name
   ```

   E a GED continuou inválida, porque nessa GED a existência do atributo `name` é obrigatório nas entidades etiquetadas como `city`. A GED voltou a ser válida quando o atributo `name` da nova entidade foi criado novamente, com o valor "Some Country's Capital", usando o comando:

   ```
   MATCH (x:city) WHERE NOT EXISTS (x.name)
   SET x.name = 'Some Country\'s Capital'
   ```

5. O atributo `can_fly` foi deletado das entidade de tipo `bird` com o comando:

   ```
   MATCH (x:bird) REMOVE x.can_fly
   ```

   A GED φ_3 continuou válida, pois nessa situação a existência do atributo `can_fly` não é obrigatória na entidade que corresponde ao vértice x do padrão. Em seguida essa alteração foi revertida e o atributo `can_fly` foi removido das entidades de tipo `eagle` com os comandos:

   ```
   MATCH (x:bird) SET x.can_fly = true
   MATCH (x:eagle) REMOVE x.can_fly
   ```

E então a GED φ_3 se tornou inválida, pois a existência do atributo `can_fly` é obrigatória na entidade que corresponde ao vértice y do padrão. A GED voltou a ser válida quando a alteração foi desfeita com o comando:

```
MATCH (x:eagle) SET x.can_fly = true
```

6. Foi criada uma nova GED φ_6 semelhante à GED φ_3 utilizando a aplicação. A GED φ_6 possui o mesmo padrão de grafo `MATCH (x)<-[:is_a]-(y) RETURN x, y` que a GED φ_3, mas sua dependência funcional é diferente ($\{x.playable = x.playable\} \rightarrow \{y.playable = x.playable\}$). A GED φ_6 é válida inicialmente, mas deixa de ser quando o valor do atributo `playable` das entidades com etiqueta `doll` é alterado para "false" com o comando:

   ```
   MATCH (x:doll) SET x.playable = false
   ```

   Em seguida o valor do atributo `playable` das entidades com etiqueta `toy` também foi alterado para "false" com o comando: `MATCH (x:toy) SET x.playable = false`

   A GED φ_6 continuou inválida pois ainda existia uma correspondência que não a satifazia: a entidade `lego` ainda possuía o atributo `playable` com valor verdadeiro. Após alterar o valor do atributo `playable` das entidades com etiqueta `lego` para falso a GED voltou a ser válida. No final as alterações foram revertidas com o comando:

   ```
   MATCH (x {playable:false}) SET x.playable = true
   ```

7. A GED φ_4 possui um padrão de grafo absurdo, onde uma pessoa é simultaneamente pai e filha de outra, e é inválida caso alguma correspondência exista. A pessoa de nome "Fulano" é pai da pessoa de nome "Ciclano", então cria-se um novo relacionamento entre os dois dizendo que "Fulano" é filho de "Ciclano" com o comando:

   ```
   MATCH (x:person {name:'Fulano'}), (y:person {name:'Ciclano'})
   CREATE (x)-[:child]->(y)
   ```

   E a GED φ_4 tornou-se inválida. Então as alterações foram revertidas com o comando:

   ```
   MATCH (x:person {name:'Fulano'})-[a:child]->(y:person) DELETE a
   ```

Como todos os testes acima corresponderam ao comportamento esperado, conclui-se que o código produzido pode ser utilizado para validar alguns tipos de dependências funcionais em grafo. O código também pode ser estendido para outros tipos de FDs, como por exemplo as Graph Denial Contraints (GDCs), também propostas em Fan e Lu (2017).

5 Conclusão

O armazenamento de dados em bancos de dados em grafo é uma boa alternativa quando se trabalha com uma base de dados densa e interconectada. Por sua estrutura expressiva e de propósito geral, a estrutura de grafo permite a modelagem de qualquer tipo de cenário.

A inserção e modificação de dados em uma base de dados são processos importantes que devem ser gerenciados e controlados para garantir que os dados estejam consistentes e para que as restrições de integridade estejam sempre satisfeitas.

Porém o suporte para restrições de integridade em bancos de dados em grafo, considerando grafos de propriedades, ainda é pequeno. Restrições de integridade são muito importantes por prevenirem que informações inconsistentes ou de baixa qualidade sejam armazenadas no grafo, e o baixo suporte a elas caracteriza um problema.

A implementação de restrições de integridade pode ser feita de forma integrada ou por meio de uma nova camada de aplicação. Neste trabalho, foi implementado um projeto, utilizado como uma nova camada de aplicação, que permite a criação de algumas restrições de integridade (FDs que podem ser representadas como GEDs) e sua validação em relação à um grafo.

Como parte de trabalhos futuros pode-se vislumbrar a criação de uma aplicação que permita a definição de GEDs em mais alto nível. Da maneira como foi implementada, o usuário deve conhecer sobre a linguagem Cypher e digitar uma consulta que corresponda perfeitamente ao padrão de grafo desejado. É desejável que usuários sem muito conhecimento de Cypher consigam criar GEDs. Além disso é proposta uma extensão do trabalho que propicie a inclusão das *Graph Denial Constraints* (GDCs), também propostas por Fan e Lu (2017), que representam restrições ainda mais gerais, permitindo por exemplo criar restrições sobre o domínio de valores dos atributos, dentre outros.

Referências

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 40, n. 1, p. 1:1–1:39, fev. 2008. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1322432.1322433>. Citado na página 17.

ELMASRI, R.; NAVATHE, S. *Fundamentals of database systems*. [S.l.]: Addison-Wesley Publishing Company, 2010. Citado na página 17.

FAN, W.; LU, P. Dependencies for graphs. In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. New York, NY, USA: ACM, 2017. (PODS '17), p. 403–416. ISBN 978-1-4503-4198-1. Disponível em: <http://doi.acm.org/10.1145/3034786.3056114>. Citado 13 vezes nas páginas 4, 10, 11, 14, 17, 18, 21, 22, 23, 24, 25, 50 e 51.

MADAN, P.; SAXENA, A. Graph databases. *International Journal*, v. 4, n. 5, p. 195–200, 2014. Citado na página 10.

MARGITUS, M. R.; TAUER, G.; SUDIT, M. RDF versus attributed graphs: The war for the best graph representation. In: *18th International Conference on Information Fusion, FUSION 2015, Washington, DC, USA, July 6-9, 2015*. [s.n.], 2015. p. 200–206. Disponível em: <http://ieeexplore.ieee.org/document/7266563/>. Citado 2 vezes nas páginas 10 e 13.

MARTON, J.; SZÁRNYAS, G.; VARRÓ, D. Formalising opencypher graph queries in relational algebra. In: KIRIKOVA, M.; NØRVÅG, K.; PAPADOPOULOS, G. A. (Ed.). *Advances in Databases and Information Systems*. Cham: Springer International Publishing, 2017. p. 182–196. ISBN 978-3-319-66917-5. Citado na página 19.

NEO4J. The definitive guide to graph databases for the rdbms developer. In: _____. [s.n.], 2016. cap. Query Languages: SQL vs. Cypher. Disponível em: <https://go.neo4j.com/rs/710-RRC-335/images/Definitive-Guide-Graph-Databases-for-RDBMS-Developer.pdf>. Citado 2 vezes nas páginas 4 e 20.

NEO4J. *What is Neo4j?* 2018. <https://neo4j.com/developer/graph-database/>. [Online; accessed 29-November-2018]. Citado 2 vezes nas páginas 10 e 19.

PENTEADO, R. R. M. et al. Um estudo sobre bancos de dados em grafos nativos. *X ERBD - Escola Regional de Banco de Dados*, 2014. Disponível em: <http://www.inf.ufpr.br/carmem/pub/erbd2014-artigo.pdf>. Citado 2 vezes nas páginas 15 e 16.

POKORNÝ, J. Graph databases: their power and limitations. In: SPRINGER. *IFIP International Conference on Computer Information Systems and Industrial Management*. [S.l.], 2015. p. 58–69. Citado 4 vezes nas páginas 4, 10, 16 e 17.

RABUZIN, K.; KONECKI, M.; ŠESTAK, M. Implementing check integrity constraint in graph databases. In: *IIER 105th International Conference on Recent Innovations in Engineering and Technology*. [S.l.: s.n.], 2016. Citado na página 21.

ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, 2015. ISBN 9781491930861. Disponível em: <https://books.google.com.br/books?id=RTvcCQAAQBAJ>. Citado 2 vezes nas páginas 10 e 17.

RODRIGUEZ, M. A.; NEUBAUER, P. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010. Disponível em: <http://arxiv.org/abs/1006.2361>. Citado 3 vezes nas páginas 4, 14 e 15.

ŠESTAK, M.; RABUZIN, K.; NOVAK, M. Integrity constraints in graph databases - implementation challenges. 2016. Disponível em: <https://bib.irb.hr/datoteka/833711.Integrity_constraints_in_graph_databases_implementation_challenges.pdf>. Citado 3 vezes nas páginas 17, 18 e 21.