MOGWAI: UM ARCABOUÇO PARA BANCOS DE DADOS DE MÚLTIPLOS GRAFOS

Carlos Eduardo Fernandes de Padoa

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Alexandre de Assis Bento Lima

Rio de Janeiro
Setembro de 2017

MOGWAI: UM ARCABOUÇO PARA BANCOS DE DADOS DE MÚLTIPLOS GRAFOS

Carlos Eduardo Fernandes de Padoa

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Alexandre de Assis Bento Lima, D.Sc.

Profª. Marta Lima de Queirós Mattoso, D.Sc.

Prof. Fabio André Machado Porto, D.Sc.

Prof. Victor Teixeira de Almeida, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2017

Padoa, Carlos Eduardo Fernandes de

Mogwai: um arcabouço para bancos de dados de múltiplos grafos/Carlos Eduardo Fernandes de Padoa. – Rio de Janeiro: UFRJ/COPPE, 2017.

XIII, 123 p.: il.; 29, 7cm.

Orientador: Alexandre de Assis Bento Lima

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2017.

Referências Bibliográficas: p. 116 – 119.

1. Bancos de Dados de Grafos. 2. Bancos de Dados NoSQL. 3. Cypher. I. Lima, Alexandre de Assis Bento. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha família, em especial à minha esposa Stephanie, que sempre me apoiou durante esta jornada.

Agradecimentos

Gostaria de agradecer ao meu professor orientador, Alexandre de Assis Bento Lima, pelas aulas, pela cobrança, e pela orientação recebida para a realização da dissertação. Agradeço também a todos os professores do PESC pela dedicação ao curso, e pelos conhecimentos transmitidos nas disciplinas.

E agradeço à minha família, que sempre prezou pela educação.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MOGWAI: UM ARCABOUÇO PARA BANCOS DE DADOS DE MÚLTIPLOS GRAFOS

Carlos Eduardo Fernandes de Padoa

Setembro/2017

Orientador: Alexandre de Assis Bento Lima

Programa: Engenharia de Sistemas e Computação

Grafos são uma técnica poderosa de representação de informações, capaz de capturar o relacionamento entre entidades. São úteis no entendimento de uma ampla variedade de conjuntos de dados em diversas áreas como ciência, governo, e negócios. Nos últimos anos ressurgiu o interesse em armazenar e gerenciar dados representados como grafos devido à percepção de que investigar as interconexões entre entidades pode levar a descobertas interessantes em uma diversidade de problemas. Com isso, surgiram os Sistemas de Gerência de Bancos de Dados orientados a Grafos (SGBDG). A maioria dos SGBDG atuais trabalham com bases de dados constituídas por um único grafo, o que não é apropriado para certas classes de problemas. Além disto, até o momento não há consenso sobre a melhor maneira de implementar um SGBDG, e não há uma linguagem de consulta padrão. Esta dissertação descreve o Mogwai, um arcabouço de *software* desenvolvido para permitir a criação e utilização de bases de dados com múltiplos grafos, e a MogwaiQL, uma linguagem que permite a realização de consultas de forma declarativa de alto nível.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

MOGWAI: A FRAMEWORK FOR MULTIPLE GRAPHS DATABASES

Carlos Eduardo Fernandes de Padoa

September/2017

Advisor: Alexandre de Assis Bento Lima

Department: Systems Engineering and Computer Science

Graphs are a powerful representation technique, capable of capturing the relationship between entities. They are useful to understand a wide variety of data sets from many areas like science, government and business. In the last years has resurged the interest in storing and managing graph data due to the realization that investigating the interconnections between entities may lead to interesting insights in a variety of problems. Most of the available graph tools work with the single graph model, which is not suitable for certain problems. Besides that, there is no consensus on what is the best way to implement a graph database, and no standard query language has been defined to graph databases yet. This thesis describes Mogwai, a framework to allow the creation and utilization of graph databases with multiple graphs, and MogwaiQL, a descriptive query language designed to allow querying the database at a high level of abstraction.

Sumário

Lista de Figuras **xi**

Lista de Tabelas **xiii**

1 Introdução **1**

1.1 Caracterização do problema 8
1.2 Abordagem proposta 9
1.3 Organização da dissertação 11

2 Revisão Bibliográfica **12**

2.1 Grafos 12
2.1.1 Grafos multirelacionais 15
2.1.2 Grafos de propriedades 15
2.1.3 Representação de grafos em sistemas computacionais 16
2.2 Banco de dados de grafos 17
2.3 Sistemas gerenciadores de bancos de dados de grafos 17
2.3.1 Processamento nativo de grafos 18
2.3.2 Armazenamento nativo de grafos 20
2.3.3 Cache 23
2.4 Compiladores e interpretadores de consulta 23
2.5 Tipos de consulta 24
2.5.1 Adjacência 24
2.5.2 Alcance 25
2.5.3 Correspondência de padrões 25
2.5.4 Sumarização 25
2.6 Trabalhos relacionados 25
2.6.1 GOQL 27
2.6.2 GraphLog 27
2.6.3 PQL 28
2.6.4 RDF e SPARQL 29
2.6.5 GraphGrep 32

2.6.6 GraphQL . . . 33
2.6.7 AQL . . . 35
2.6.8 G-Store . . . 36
2.6.9 Gremlin . . . 38
2.6.10 Cypher . . . 38
2.7 Comparação da MogwaiQL com outras linguagens de consulta . . . 39
2.8 Arcabouços de Processamento Paralelo em Grafos . . . 44

3 Mogwai: um arcabouço para bancos de dados de múltiplos grafos 46
3.1 Visão geral . . . 46
3.2 Modelo de dados . . . 48
3.3 Linguagem . . . 51
3.3.1 Álgebra . . . 51
3.3.2 Operações . . . 52
3.4 Exemplos de consultas na álgebra . . . 54
3.4.1 Grafo exemplo . . . 55
3.4.2 Consulta 1 . . . 55
3.4.3 Consulta 2 . . . 56
3.4.4 Consulta 3 . . . 56
3.4.5 Consulta 4 . . . 57
3.4.6 Consulta 5 . . . 57
3.4.7 Consulta 6 . . . 57
3.4.8 Consulta 7 . . . 57
3.5 Sintaxe da MogwaiQL . . . 58
3.6 Descrição da Arquitetura . . . 60
3.7 Detalhes de implementação . . . 61
3.7.1 API . . . 62
3.7.2 Gerenciando um banco de dados de múltiplos grafos utilizando o Mogwai . . . 64
3.7.3 Compilador Mogwai . . . 64
3.7.4 JPA . . . 69
3.7.5 JDBC . . . 71
3.7.6 ObjectDB . . . 74
3.7.7 Neo4j . . . 75
3.7.8 Algoritmo de correspondência de padrões de grafos . . . 78
3.8 Utilitários . . . 82
3.8.1 Importação de bases de dados relacionais . . . 82
3.8.2 Importação de arquivos . . . 83
3.8.3 Interface web para consultas . . . 83

4 Resultados e Discussões **86**
4.1 Comparação com outras API 87
4.2 Análise de Desempenho 89
4.2.1 Bateria de testes 1 90
4.2.2 Bateria de testes 2 104

5 Conclusões e Trabalhos Futuros **113**

Referências Bibliográficas **116**

A Gramática **120**

Lista de Figuras

1.1 Exemplo de estrutura biomolecular representada como um grafo (figura retirada de Leser[1]) 5

2.1 as pontes de Konigsberg e seu grafo correspondente 14
2.2 índice global de adjacência 18
2.3 adjacência livre de índices 19
2.4 armazenamento em disco do Neo4j 20
2.5 armazenamento em disco do Neo4j 21
2.6 os descendentes de P2 que não são descendentes de P1 28
2.7 avaliação de consulta PQL 30
2.8 grafos 33
2.9 consulta Glide 33
2.10 exemplo de composição de padrões em GraphQL 35

3.1 modelo de dados 49
3.2 ilustração modelo de dados, exemplo de inventário de equipamentos . 50
3.3 grafo exemplo de consultas na álgebra 55
3.4 diagrama de classes 62
3.5 diagrama de arquitetura 63
3.6 diagrama de possíveis arquiteturas 63
3.7 representação dos padrões de grafo 66
3.8 diagrama de sequência para obter plano de execução de consulta . . . 68
3.9 diagrama de classes JPA 70
3.10 diagrama de classes JDBC 73
3.11 diagrama de entidades e relacionamentos 74
3.12 diagrama de classes ObjectDB 76
3.13 diagrama de classes Neo4j 78
3.14 Interface web para consultas 84

4.1 desempenho da consulta 1 em milisegundos 93
4.2 desempenho da consulta 2 em milisegundos 94
4.3 desempenho da consulta 3 em milisegundos 96

4.4 desempenho da consulta 4 em milisegundos 97
4.5 desempenho da consulta 5 em milisegundos 98
4.6 desempenho da consulta 6 em milisegundos 100
4.7 desempenho da consulta 7 em milisegundos 101
4.8 desempenho da consulta 8 em milisegundos 103
4.9 desempenho da consulta 9 104
4.10 tempo de carregamento em segundos 105
4.11 desempenho da seleção em milisegundos 106
4.12 desempenho da vizinhança em milisegundos 107
4.13 desempenho do caminho mais curto em milisegundos 108

Lista de Tabelas

4.1 tempos de execução da consulta 1 92
4.2 tempos de execução da consulta 2 94
4.3 tempos de execução da consulta 3 95
4.4 tempos de execução da consulta 4 96
4.5 tempos de execução da consulta 5 98
4.6 tempos de execução da consulta 6 99
4.7 tempos de execução da consulta 7 101
4.8 tempos de execução da consulta 8 102
4.9 tempos de execução da consulta 9 104
4.10 tempos de carregamento da base de dados 105
4.11 tempos de execução teste de seleção 106
4.12 tempos de execução teste de vizinhança 107
4.13 tempos de execução teste caminho mais curto 108
4.14 Classificação por posição na bateria de testes 1 110
4.15 Classificação por tempo total bateria de testes 1 111

Capítulo 1

Introdução

Robinson e Webber [2] comentam sobre a atenção que os grafos obtiveram neste novo milênio. Novos negócios milionários surgiram em torno de informações “conectadas”. A Google capturou o grafo da web, o Facebook capturou o grafo social, e cada vez mais as empresas reconhecem o valor da informação que pode ser obtida a partir dos relacionamentos entre os dados. Os autores acreditam que a habilidade de compreender e analisar vastos grafos de dados altamente conectados será chave para determinar que companhias se sobressaem sobre seus competidores no futuro.

Grafos são uma ferramenta poderosa de modelagem, e são úteis para resolver diversos problemas em muitas áreas do conhecimento.

Há consenso entre os cientistas [3] de que muitos fenômenos sociais, econômicos e financeiros podem ser descritos por uma rede de agentes e suas interações. Em outras palavras, podem ser modelados por grafos. Entretanto, a teoria econômica padrão raramente considera as redes econômicas em suas análises. Métodos derivados da teoria de grafos constituem uma importante inovação na teoria econômica, com um aumento no número de estudos sobre redes econômicas. Podemos citar relações de propriedade entre firmas e mesas de diretores como um exemplo de rede em economia. Relações de propriedade são instrumentos para exercer controle corporativo e diversos trabalhos estudaram relações indiretas de propriedade e padrões tais como as chamadas pirâmides. Além disto, diretores com relações entre firmas transmitem informação e poder entre elas. Outro exemplo diz respeito aos mercados de tra-

balho, onde diversos estudos empíricos demonstraram que uma fração significativa dos empregos é encontrada através das redes sociais. Uma questão central na teoria das organizações é como um problema de decisão complexo pode ser eficientemente decomposto em tarefas e distribuído entre unidades de uma organização. Uma rede pode representar os caminhos ao longo dos quais essas tarefas são distribuídas na organização.

Na própria ciência da computação [4] a Teoria dos Grafos pode ser aplicada de diversas maneiras, como, por exemplo, em mineração de dados (*data mining*), segmentação de imagens, *clusterização*, captura de imagens, e redes. Grafos podem ser utilizados para resolver problemas de planejamento eficiente de rotas para entrega de correio, e diagnóstico de falha em redes de computadores. Eles também são utilizados para representar redes de comunicação, organização de dados, dispositivos computacionais, fluxos de computação, fluxos de trabalho, fluxos científicos, entre outros.

De acordo com Balaban [5], na química, todas as fórmulas estruturais de compostos covalentemente ligados são grafos, e são, portanto, denominados grafos moleculares, ou grafos constitucionais. Outros tipos de grafos incluem os grafos de reação, grafos cinéticos e os grafos de *synthon*. A importância da teoria de grafos para a química é devida principalmente ao fenômeno do isomerismo, que é racionalizado pela teoria da estrutura química. Esta teoria descreve todos os isômeros constitucionais pelo uso de métodos da teoria dos grafos, que os primeiros químicos viam como "truques com pontos e linhas (valências)". A Teoria dos Grafos, entre outros usos, também fornece a base para a programação de computadores para auxiliar a resolução de problemas químicos.

Diversas áreas da biologia molecular[1] lidam com dados estruturados na forma de grafos, e se preocupam com seu gerenciamento, armazenamento, visualização, comparação e análise. Neste grafos, os vértices tipicamente representam entidades biológicas como enzimas, genes, ou compostos, e as arestas representam alguma forma de interação ou relacionamento. A figura 1.1 exemplifica uma entidade

biológica estruturada como grafo.

Segundo Olken [6], modelos de dados de grafos capturam naturalmente uma grande variedade de dados biológicos. Podemos citar, por exemplo, taxonomias de proteínas, redes de genes regulatórios, redes de interação de proteínas, taxonomias de compostos químicos, *clusterização* de genes, e taxonomias de organismos, entre outros. O desenvolvimento de sistemas gerenciadores de bancos de dados de grafos de propósito geral é de suma importância para o desenvolvimento de projetos que se entendem desde aplicações de sequências de DNA, grafos de estruturas químicas, até grafos de contato (de proteínas) e caminhos biológicos (metabólicos, de sinalização). Aplicações biológicas fazem uso de diversos tipos de consulta em grafos: consultas de caminho fixo, de caminho recursivo, existência de caminho, caminho mais curto, isomorfismo de subgrafos, componentes conexos, interseção de grafos, diferença de grafos, e maior subgrafo comum são alguns exemplos.

Além disto, o uso de um modelo de dados padrão e de uma linguagem de consulta padrão entre diferentes aplicações é extremamente desejável tanto do ponto de vista do usuário como de desenvolvimento de *software*. Uma linguagem padrão pode ser utilizada por diferentes SGBD (Sistema Gerenciador de Banco de Dados), reduzindo a quantidade de trabalho duplicado. Além disto, torna-se necessário que os usuários aprendam uma única linguagem, podendo utilizá-la com o SGBD de sua preferência [1]. Muita pesquisa e desenvolvimento ainda são necessários para que os sistemas atuais satisfaçam aos requisitos modernos. O desenvolvimento dos Sistemas Gerenciadores de Bancos de Dados de Grafos (SGBDG) é motivado por todos estes fatores, além de outros, como a eficiência no processamento de consultas, por exemplo.

A maioria das soluções SGBDG mais populares utiliza o modelo de grafos de propriedades. Posto de forma simples, este modelo difere do grafo "tradicional" pelo fato de que os vértices e as arestas podem conter propriedades (pares chave-valor). Além disto, vários SGBD que adotam o modelo RDF [7] são considerados como SGBDG. O modelo RDF, resumidamente, consiste em triplas formadas por

um sujeito, um predicado, e um objeto, onde o predicado qualifica o relacionamento entre o sujeito e o objeto, de tal forma que o conjunto destas triplas forma um grafo. Estes SGBD utilizam a linguagem de consulta SPARQL [8]. Uma lista de ferramentas RDF pode ser encontrada em [9].

Outra característica da maioria dos SGBDG atuais é que eles armazenam e gerenciam apenas um grafo, o que é suficiente para uma série de aplicações. Entretanto, outros diversos tipos de problemas e aplicações necessitam trabalhar com múltiplos grafos. Por exemplo, segundo El-Jaick [10], experimentos científicos apoiados por computadores realizam simulações com cadeias de programas, que são, geralmente, representadas por *workflows* científicos. Por conta da natureza exploratória dos experimentos, um mesmo *workflow* é executado diversas vezes variando-se os parâmetros de execução. Os dados intermediários e finais produzidos nos experimentos são fundamentais para que estes sejam considerados consistentes e válidos pelo cientista. Este tipo de informação é denominado proveniência. Os dados de proveniência são baseados em objetos (dados e programas) e seus relacionamentos, que são representados como grafos direcionados acíclicos. Cada execução de um *workflow* gera um grafo de proveniência. Uma base de dados de proveniência, portanto, é composta por múltiplos grafos. Segundo resultados de estudos anteriores, é possível argumentar que utilizar um SGBD de grafos é a melhor alternativa para abordar o problema das consultas de proveniência. El-Jaick salienta, entretanto, que, na maioria dos casos, os SGBDG são projetados para armazenar um único grafo, mesmo que muito volumoso, em cada base de dados, e não um conjunto com um grande número de grafos independentes em uma única base, como é o caso dos dados de proveniência. Isto faz com que a complexidade de gerenciar a repetição de consulta para cada grafo e a de agrupar os resultados obtidos recaia sobre o usuário. Em suas palavras, "utilizar um único grafo para representar conjuntos de grafos exige o emprego de certos artifícios por parte do usuário". Dependendo do artifício utilizado, pode ser o caso de as consultas se tornarem complexas, comprometendo o tempo de resposta.

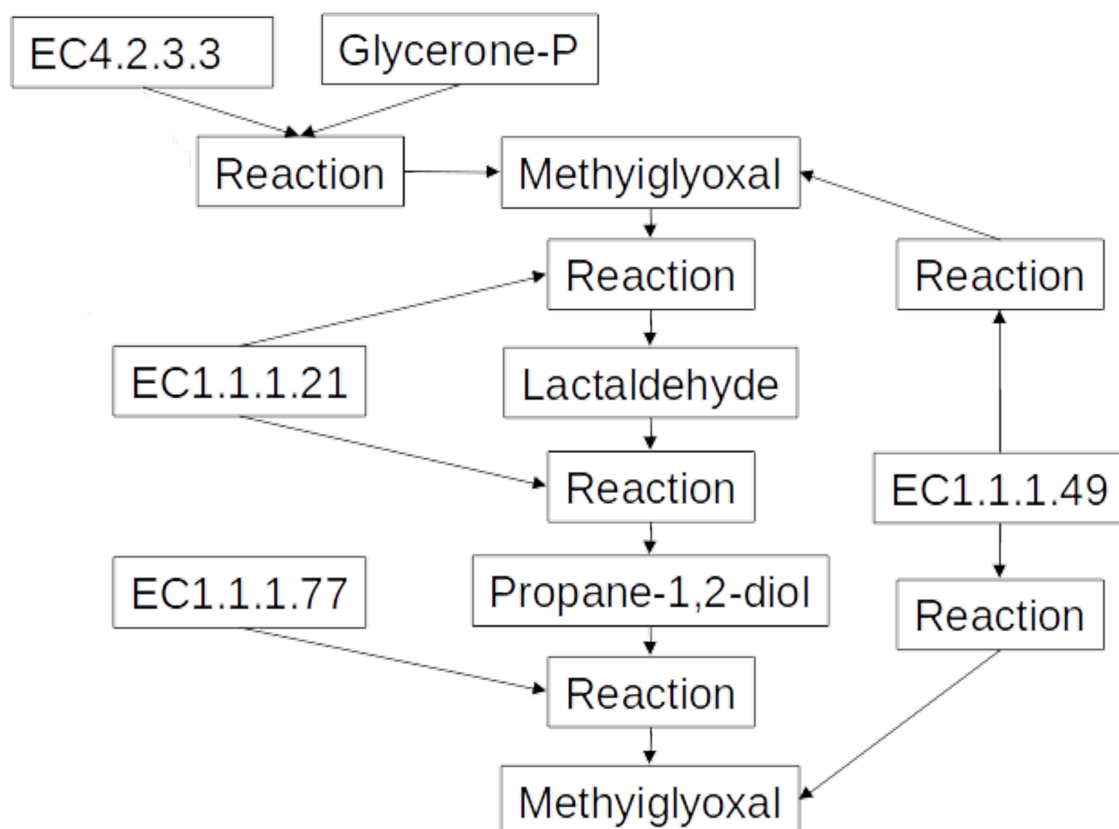


Figura 1.1: Exemplo de estrutura biomolecular representada como um grafo (figura retirada de Leser[1])

A capacidade de lidar com múltiplos grafos é importante nos problemas que a exigem pois quando ausente leva à necessidade de se adotarem estratégias complexas para simular vários grafos por meio de um só. Exemplificando, é possível diferenciar os grafos fazendo com que cada elemento contenha um identificador do grafo ao qual o elemento pertence. Outra estratégia seria fazer com que o grafo seja identificado por um determinado nó inicial. Não obstante, além das dificuldades de modelagem, surgem problemas de ordem programática, como, por exemplo, as interfaces de programação de aplicações (API, do inglês *Application Programming Interfaces*) de tais ferramentas disponibilizarem apenas uma instância do objeto grafo, de tal forma que não é possível realizar operações entre grafos, apenas entre os elementos do grafo. Por fim, como o SGBDG não está "ciente" de que há vários grafos, o usuário tem que usar subterfúgios para processar corretamente os dados, e eventualmente podem se perder oportunidades de otimização.

As dificuldades impostas pelo modelo de grafo único motivam o desenvolvimento de ferramentas capazes de manipular múltiplos grafos.

Apesar de terem surgido diversos sistemas para gerenciar bases de dados de grafos, vários deles possuíam capacidade de consulta muito limitada. Normalmente, o acesso aos dados era possível apenas programaticamente, com o uso de API, sem a possibilidade de utilizar uma linguagem de consulta [11]. E entre as ferramentas que ofereciam este recurso, não havia um padrão estabelecido[12][13]. Atualmente, as linguagens Gremlin[14] e OpenCypher[15] constituem verdadeiros esforços neste sentido. Embora não sejam nenhum padrão oficial, podemos argumentar que são padrões *de facto*. A linguagem Gremlin, por exemplo, é adotada por uma série de ferramentas[16], dentre as quais o próprio Neo4j[17], por exemplo.

Pode-se argumentar que o sucesso da Gremlin é devido ao fato de que para utilizá-la não é necessário implementar um compilador ou interpretador de linguagem. Basta apenas implementar um pequeno conjunto de interfaces e métodos definidos em sua API. Além disso, quando estes métodos são implementados, não apenas a linguagem Gremlin está disponível, como também uma série de outros recursos,

como a capacidade de executar “programas de vértice” de forma distribuída.

O sucesso da linguagem Cypher, por sua vez, podemos argumentar que é devido principalmente à sua sintaxe, que é simples e intuitiva. Mas as qualidades do SGBG Neo4j, que a implementa, também colaboram bastante para o seu sucesso e a difusão do seu uso.

A falta de uma linguagem de consulta padrão é uma desvantagem dos SGBDG atuais [11], pois linguagens de consulta sempre foram fatores-chave para o sucesso de um SGBD [18]. Elas facilitam o uso e a adoção de uma tecnologia de banco de dados.

No que tange à escolha de um SGBD, a padronização é muito importante ao se considerar a “independência de fornecedor”, em outras palavras, a qualidade de um produto poder ser substituído por outro similar ao de outro fornecedor com o menor impacto possível. Esse objetivo é possível de ser atingido, por exemplo, com os Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDR) através do uso da *Structured Query Language* (SQL) . Stonebraker e Cattell aconselham clientes a escolher um SGBDR que ofereça uma linguagem de alto nível que possa oferecer alto desempenho [18]. Linguagens de alto nível (ou declarativas) permitem que o usuário especifique em suas consultas apenas quais dados recuperar em vez de como recuperá-los. Esta característica é bastante desejável, pois, desta forma, cabe ao SGBD planejar uma estratégia de execução para a recuperação do resultado da consulta [19]. Em outras palavras, linguagens de consulta declarativas permitem que a responsabilidade sobre a otimização das consultas passe do usuário para o SGBD.

Neste contexto propomos o Mogwai, um arcabouço de *software* desenvolvido para permitir a gerência de bases de dados de múltiplos grafos, e uma nova linguagem denominada MogwaiQL, que visa a permitir ao usuário o acesso à base de múltiplos grafos de forma declarativa, de alto nível.

1.1 Caracterização do problema

A gerência de um banco de dados de múltiplos grafos pode ser feita basicamente de duas formas. A abordagem tradicional utiliza programação e arquivos. Nesta abordagem, o usuário define as estruturas dos arquivos e seus métodos de acesso, assim como os programas que os processam, para uma aplicação específica[19]. Esta solução é satisfatória para uma série de problemas, e, inclusive, conta várias ferramentas para computação de alto desempenho, como o Apache Hadoop [20], por exemplo. A segunda abordagem utiliza um banco de dados, um repositório único de dados que é definido uma única vez, e é mantido e acessado por múltiplos usuários. Este banco de dados é acessado e gerenciado unicamente através de um SGBD. Estas abordagens apresentam vantagens e desvantagens.

Segundo Elmasri[19], as vantagens da utilização de um SGBD são o controle de redundância, a restrição de acesso não autorizado, o armazenamento persistente para objetos programas, o armazenamento de estruturas para o processamento eficiente de consultas, a garantia de backup e restauração, o fornecimento de múltiplas interfaces para os usuários, a representação de relacionamentos complexos entre os dados, a imposição de restrições de integridade, e a capacidade para definir regras de dedução por inferência.

Elmasri[19] também lista como implicações adicionais do uso da abordagem com SGBD: o potencial para garantir padrões, a redução no tempo de desenvolvimento de aplicações, a flexibilidade, a disponibilidade para atualizar as informações, e as economias de escala.

No caso deste trabalho, não estamos interessados em propor uma solução para uma área de aplicação de grafos específica, tal como realizar roteamentos rodoviários, ou encontrar similaridades entre proteínas. Portanto, adotamos a abordagem que utiliza SGBD. Primeiramente por ser um software de propósito geral, e também pelas suas diversas vantagens, em especial pela redução no tempo de desenvolvimento de aplicações e pelo potencial de garantir padrões. Para melhor caracterizar o problema em questão, se faz útil ter em mente o conceito de SGBDG.

Robinson [2] define um SGBDG como um sistema *online* de gerência de bases de dados com métodos de leitura, criação, atualização e exclusão (*CRUD - Create, Read, Update, Delete*) que expõe um modelo de dados de grafo. Em outras palavras, um sistema cuja interação se dá por estruturas de dados de grafos.

O principal objetivo desta dissertação é descrever o desenvolvimento de ambas linguagem e camada de *software*. Estas ferramentas colaboram com a resolução de problemas que exigem múltiplos grafos eliminando a complexidade gerada quando se trabalha com o modelo de grafo único. Com o Mogwai torna-se possível criar bancos de dados formados por múltiplos grafos, vértices e arestas utilizando o modelo de grafo de propriedades, ao mesmo tempo em que se tem à disposição um mecanismo para explorar o relacionamento entre entidades de múltiplos grafos de forma independente de como os dados estão armazenados. Este objetivo é atingido através da definição formal de uma nova linguagem, da definição de um arcabouço para gerenciar um banco de dados de múltiplos grafos, e de suas respectivas implementações.

1.2 Abordagem proposta

A abordagem proposta para armazenar e manipular múltiplos grafos consiste em desenvolver uma camada intermediária de *software* para expor o modelo de dados de múltiplos grafos e desenvolver uma linguagem de consulta própria para o problema. Os problemas de baixo nível, como armazenamento em disco e controle de concorrência, são resolvidos com a utilização de um SGBD.

Os problemas envolvidos no desenvolvimento de um SGBD são inúmeros e possuem soluções consolidadas e satisfatórias, de forma que a abordagem proposta não consiste em implementar um SGBD em todo ou em parte, mas sim utilizá-lo como parte da solução. Ao mesmo tempo, os problemas de expor um modelo de dados de múltiplos grafos e acessar estes múltiplos grafos através de uma linguagem de consulta de alto nível ainda encontram-se em estágios iniciais, constituindo o foco desta dissertação.

Outro motivo que estimula a utilização de SGBD existentes para desenvolver um SGBDG é o fato de que ainda não há consenso sobre a superioridade de um SGBDG nativo sobre um não nativo. Se, por um lado, Robinson [2] argumenta que o processamento de grafos nativo é imperativo para um bom desempenho em SGBDG, por outro, Stonebraker [21] argumenta que um SGBDR pode ter desempenho similar ou superior a um SGBDG para resolver as mesmas consultas.

A linguagem, por sua vez, é construída a partir de uma álgebra definida nesta dissertação, e tem sintaxe similar à da linguagem Cypher [22], para se aproveitar de sua difusão de uso, e consequente facilidade de aprendizado. A linguagem é independente do SGBD utilizado. O programa que interpreta a linguagem de consulta transforma o texto em uma estrutura intermediária em memória que representa o plano de execução da consulta. Este plano é, então, executado utilizando as operações disponíveis na camada intermediária de software.

Resumidamente, a abordagem é de alto nível quando possível, e se utiliza de recursos prontos de nível mais baixo, mantendo clara separação entre a definição e a implementação, de tal forma que o trabalho pode se focar nos aspectos de "ser orientado a múltiplos grafos" e permitir a "consulta declarativa de alto nível", ao mesmo tempo em que procura não perder de vista o bom desempenho. Uma consequência desta abordagem é que o desempenho depende muito do SGBD utilizado. Isto fica claro nos resultados dos testes.

Uma outra possível abordagem seria de, ao invés de definir uma camada intermediária que opera sobre ferramentas existentes, definir os formatos dos arquivos em disco, e implementar os programas que manipulam estes arquivos de forma a oferecer todas as funcionalidades de um SGBD. Esta solução, porém, demandaria demasiado esforço para resolver problemas que hoje já estão resolvidos em troca de um possível ganho de desempenho (que poderia não ser verificado) mantendo a mesma funcionalidade.

1.3 Organização da dissertação

O presente capítulo introduziu o tema. Além deste capítulo, esta dissertação é composta por mais quatro outros capítulos. O segundo capítulo apresenta as definições teóricas necessárias para melhor compreensão do problema e da abordagem proposta, e expõe trabalhos relacionados com foco em linguagens de consulta. Trabalhos relacionados com foco em SGBDG podem ser encontrados em [23] e [11]. O terceiro capítulo apresenta a solução dada para o problema, e os seus detalhes técnicos. O quarto capítulo apresenta os resultados dos experimentos de desempenho. Por fim, o quinto capítulo conclui esta dissertação.

Capítulo 2

Revisão Bibliográfica

O presente capítulo busca fornecer embasamento teórico para melhor compreender o problema objeto desta dissertação e a abordagem adotada na sua resolução. Primeiramente são apresentadas algumas definições acerca da teoria dos grafos. De especial importância é a definição do grafo de propriedades, que é o modelo utilizado para desenvolver o arcabouço e a linguagem de consulta. Em seguida são apresentados conceitos relacionados à teoria dos sistemas de banco de dados. O conceito de SGBDG foi apresentado no capítulo introdutório e é repetido por conveniência. Os conceitos de processamento de grafos nativo e armazenamento nativo de grafos ilustram abordagens alternativas em relação ao uso de SGBD para a resolução do problema.

2.1 Grafos

Credita-se ao matemático Leonhard Euler a criação do primeiro grafo da história, em 1736, na resolução do famoso problema das sete pontes de Königsberg. A cidade de Königsberg (território da Prússia até 1945, atual Kaliningrado) é cortada pelo rio Prególia, onde na época havia duas ilhas cortadas por sete pontes. Discutia-se na cidade a possibilidade de atravessar todas as pontes sem que nenhuma fosse repetida. Euler foi o primeiro a provar que não era possível realizar tal travessia. Em seu raciocínio, Euler transformou os caminhos em linhas e sua intersecções em pontos,

e percebeu que tal travessia somente seria possível se houvesse exatamente zero ou dois pontos de onde saísse um número ímpar de caminhos. É sempre possível entrar e sair de um lugar com número par de caminhos, mas com número ímpar somente é possível entrar ou sair, não sendo possível realizar ambas ações.

A figura 2.1 ilustra a geografia da cidade de Königsberg, e o grafo correspondente.

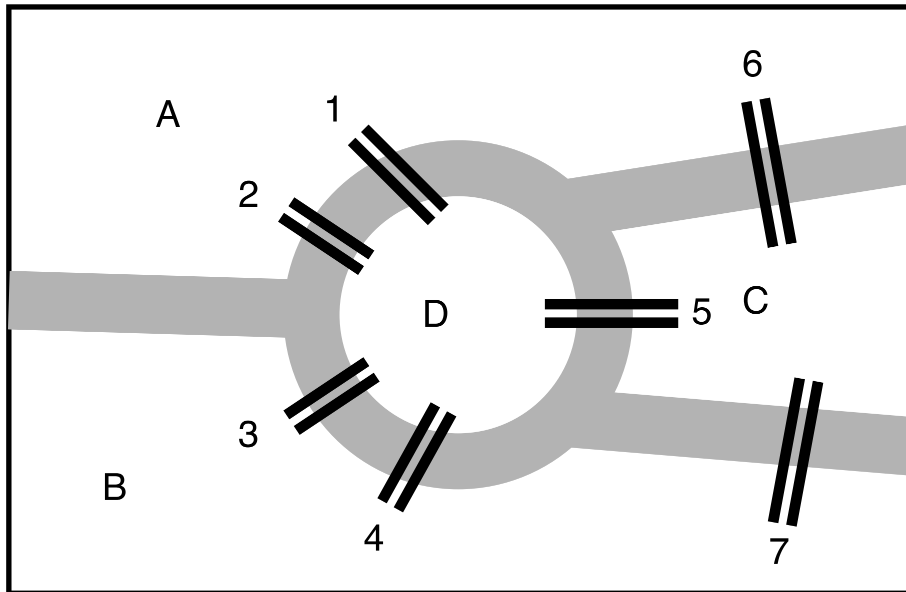
Segundo Agnarsson e Greenlaw [24], um grafo é, informalmente, uma simples coleção de vértices e arestas combinada com uma regra sobre como as arestas conectam os vértices. Os autores definem a forma geral de um grafo como sendo G = (V, E, Φ) onde V é o conjunto de vértices, E o conjunto de arestas, e Φ a função que mapeia os vértices e as arestas. Uma aresta define a adjacência entre dois vértices, ou entre um vértice e ele mesmo.

Grafos são comumente representados com diagramas, onde os pontos são os vértices, e as linhas conectando os pontos são as arestas, tal como ilustra a figura 2.1.

Existem diferentes tipos de grafos, como os digrafos e os hipergrafos, cada um com definições e características distintas. Os digrafos, por exemplo, tem arestas direcionadas, enquanto que os hipergrafos permitem que uma aresta conecte um número positivo qualquer de vértices.

De forma geral, em grafos que não permitem arestas paralelas, duas arestas se distinguem apenas pelos vértices que elas unem. Tais grafos são chamados unirelacionais. Quer dizer, entre dois vértices é estabelecida apenas um tipo de relação. As modelagens possíveis, por sua vez, admitem apenas uma semântica única para as arestas.

Numa rede social, por exemplo, poderíamos modelar as pessoas como vértices, e as arestas seriam o relacionamento de amizade entre elas. Neste caso, por termos apenas um tipo de relacão, não seria possível modelar outras relações como parentesco, casamento, entre outras.



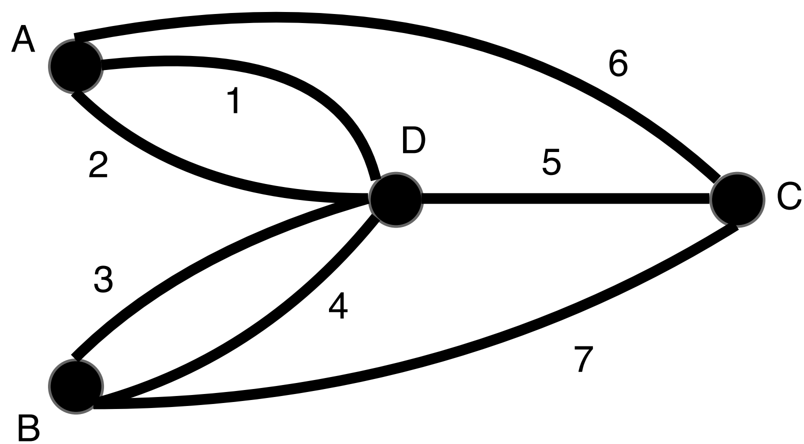


Figura 2.1: as pontes de Konigsberg e seu grafo correspondente

2.1.1 Grafos multirelacionais

Como disseram Rodriguez e Neubauer [25], quando o domínio do problema é acometido por um conjunto de relações heterogêneas, um grafo multirelacional se mostra uma construção mais adequada.

Os grafos multirelacionais permitem que mais de uma aresta seja definida entre dois vértices, mas, diferentemente das arestas paralelas, estas arestas possuem semânticas diferentes. Elas se distinguem entre si não só pelos vértices que unem, mas também por um rótulo. As arestas passam a ser uma relação ternária e = (u, v, Ω), onde Ω é o conjunto de rótulos de arestas. Com um grafo multirelacional, a rede social passa a ser capaz de representar diferentes tipos de relações entre as pessoas.

2.1.2 Grafos de propriedades

O modelo de grafo de propriedades foi adotado por diversos sistemas gerenciadores de banco de dados de grafos. Ele possui as seguintes características [2]:

- Contém vértices e arestas.
- Arestas e vértices contém propriedades (pares chave-valor).
- Arestas possuem rótulos, são direcionadas, e sempre possuem um vértice de início e outro de fim.

Os SGBDG atuais, apesar de trabalhar com o modelo de grafo de propriedades, trabalham apenas com um único grafo. A incapacidade de trabalhar com múltiplos grafos compele as aplicações que necessitam deste recurso a trabalhar com soluções de contorno, que adicionam uma complexidade desnecessária na resolução do problema. Além disto, outras preocupações fazem com que gerenciar múltiplos grafos seja uma característica desejável de um SGBDG:

- Separação de preocupações, encapsulamento, e modularidade.
- Questões de segurança e acesso à informação.

- Facilidade de instalação e configuração do SGBDG (apenas uma instalação ao invés de uma para cada banco de dados).
- Custo de licença.
- Integração de dados.
- Otimização.

2.1.3 Representação de grafos em sistemas computacionais

De forma geral [4] existem quatro maneiras de representar um grafo em um computador: por lista de incidências, por matriz de incidência, por lista de adjacências, e por matriz de adjacências. A representação por lista de incidências consiste em que cada vértice mantenha uma lista das arestas na qual ele incide. A matriz de incidência consiste em uma matriz M de m arestas por n vértices. Se um vértice v incide em uma aresta j, o valor 1 é atribuído ao elemento M_{ij}, caso contrário é atribuído o valor 0. Um grafo direcionado pode ser representado atribuindo o valor 1 ou -1 dependendo se o vértice é de origem ou destino. A lista de adjacências funciona mantendo-se para cada vértice do grafo uma lista dos seus vértices adjacentes. A matriz de adjacência consiste numa matriz quadrada de $|V|$ x $|V|$ de vértices. Um elemento V_{ij} possui o valor 1 se o vértice i é adjacente ao vértice j, e zero caso contrário.

Nota-se que nenhuma dessas representações é adequada para o modelo de grafo de propriedades, de forma que a representação utilizada pelo arcabouço desenvolvido neste trabalho é orientada a objetos. Mais especificamente, há uma classe particular para representar cada elemento do grafo, e os seus relacionamentos são mantidos através de ponteiros entre os objetos.

2.2 Banco de dados de grafos

Um banco de dados é definido como sendo uma coleção de dados relacionados[19]. E um banco de dados de grafos é definido por He e Singh[26] como sendo uma ou mais coleções de grafos.

Eles lembram que os bancos de dados podem ser, de forma geral, classificados em duas categorias: a primeira é uma grande coleção de pequenos grafos, e a outra consiste em poucos grafos muito grandes. Apesar destas características por vezes serem relevantes para determinadas aplicações particulares, esse aspecto é de certa forma ignorado no projeto do arcabouço. A única preocupação acerca deste aspecto é que o arcabouço seja capaz de lidar com grafos que não cabem inteiramente na memória.

2.3 Sistemas gerenciadores de bancos de dados de grafos

Robinson[2] define um SGBDG como sendo um sistema *online* de gerenciamento de banco de dados com métodos CRUD que expõe um modelo de dados de grafo. Além da definição, Robinson afirma que bases de dados de grafos são geralmente criadas para serem utilizadas com sistemas transacionais (*OLTP* - *Online Transaction Processing*). Estes sistemas de bancos de dados, portanto, são normalmente otimizados para desempenho transacional, e projetados considerando-se a integridade transacional dos dados e a disponibilidade operacional.

Robinson[2] discute diversos aspectos acerca dos SGBDGs, mas dedica apenas um capítulo ao funcionamento interno de um SGBDG. O fato de que não há um padrão arquitetural universal (mesmo entre os SGBDG) é uma consideração feita inicialmente no capítulo. Em outras palavras, não há uma maneira de implementar um SGBDG (um *design*) recorrente que tenha sido observada até o momento. Pode-se interpretar esta falta de padrão como uma oportunidade de pesquisar e desenvolver novas soluções, mas não é o caso desta dissertação.

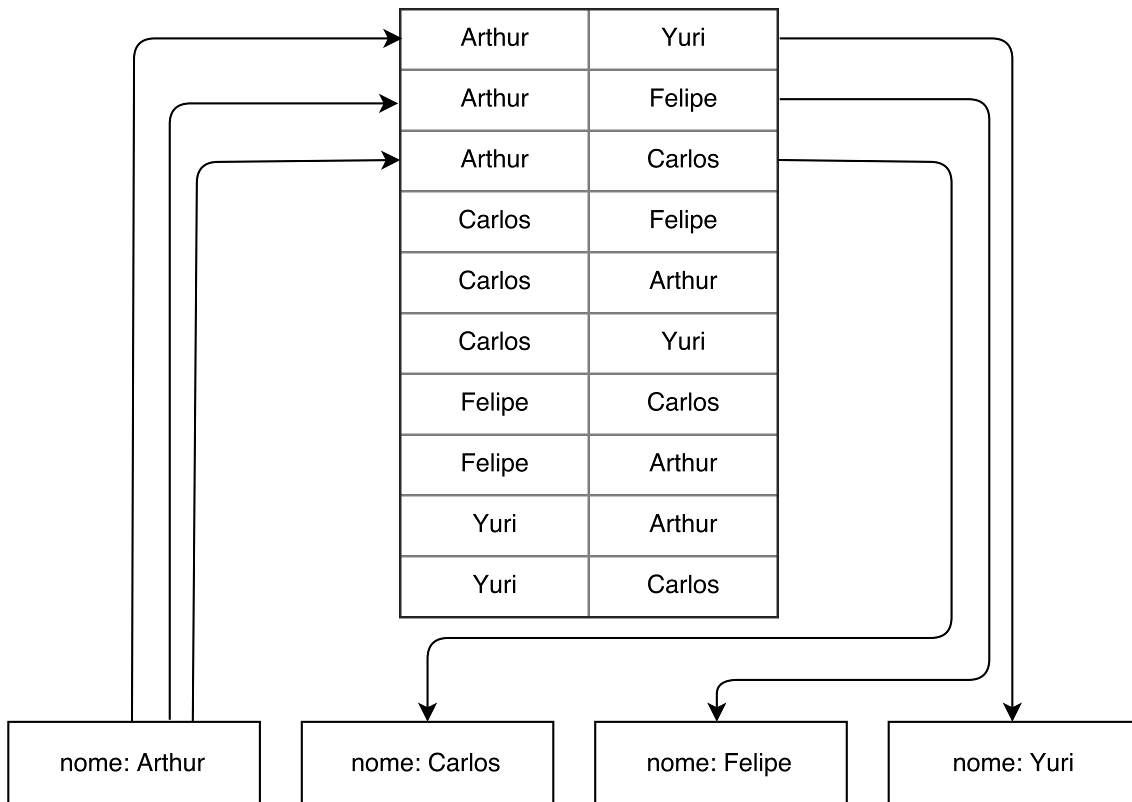


Figura 2.2: índice global de adjacência

2.3.1 Processamento nativo de grafos

Robinson[2] define que um SGBDG possui processamento nativo de grafos caso ele possua uma propriedade denominada adjacência livre de índices. Isto é, cada vértice do grafo mantém referências diretas para os vértices adjacentes, atuando como um micro índice para os vértices próximos, o que é computacionalmente menos dispendioso do que utilizar índices globais.

A utilização de adjacência livre de índices torna os tempos de execução de consultas independentes do tamanho total do grafo, e são simplesmente proporcionais à porção do grafo acessada. Esta afirmação parte do fato de que acessar os relacionamentos de um vértice tem complexidade algorítimica de O(1) devido ao acesso direto, ao passo que ao utilizar um índice para navegar pelos relacionamentos é preciso realizar uma busca no mesmo, que tem como complexidade algorítimica algo como, por exemplo, O(log n), quer dizer, dependente do tamanho do grafo[2]. Realizar uma travessia de m passos tem custos, portanto, por exemplo, de *O(m) versus*

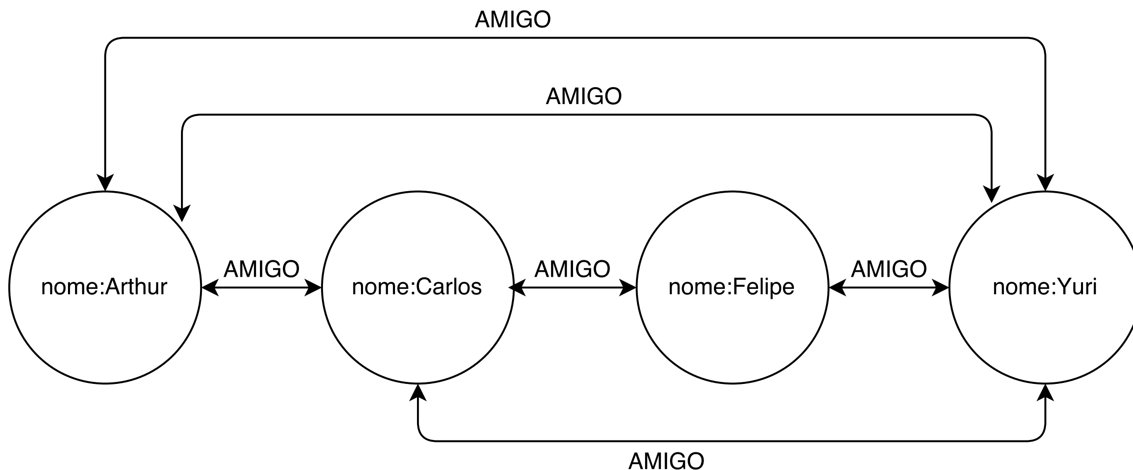


Figura 2.3: adjacência livre de índices

O(m log n).

A figura 2.2 ilustra um grafo de relacionamentos de amizades numa possível abordagem de processamento de grafos não nativo. Para encontrarmos os amigos de Arthur devemos realizar uma busca no índice global ao custo de O(log n). Este custo é aceitável em muitos casos de uso. Por outro lado, se estivermos interessados em saber de quem o Arthur é amigo, será necessário percorrer todo o índice, o que é muito mais dispendioso computacionalmente. Com a adjacência livre de índices (figura 2.3), ambas as operações possuem o mesmo custo.

Os proponentes do processamento de grafos nativo argumentam que a adjacência livre de índices é crucial para travessias rápidas e eficientes no grafo. Esta computação de alto desempenho, entretanto, somente é possível caso o SGBDG ofereça uma arquitetura projetada para este propósito.

Conforme dito anteriormente, o arcabouço Mogwai representa o grafo de forma orientada a objetos. Cada vértice mantém duas listas: uma para as arestas de entrada, e outra para as arestas de saída. Cada aresta mantém referência para os seus vértices. Portanto, de certa forma, o arcabouço utiliza adjacência livre de índices. Porém, é preciso considerar que nem toda estrutura do grafo está disponível em memória a todo momento, de forma que o benefício da adjacência livre de índices pode não ser colhido caso o acesso ao disco não seja feito adequadamente.

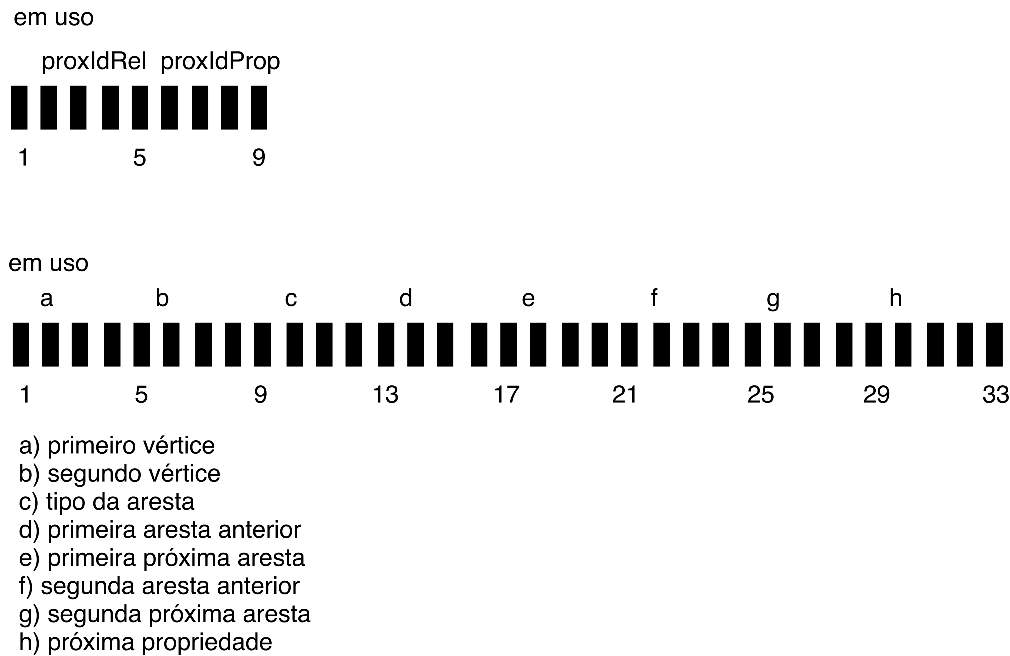


Figura 2.4: armazenamento em disco do Neo4j

2.3.2 Armazenamento nativo de grafos

Um aspecto chave da arquitetura de um SGBDG é a forma como os grafos são armazenados em disco. Idealmente, o SGBDG deve ser capaz de executar algortimos arbitrários em grafos com bom desempenho.

Robinson[2] descreve como o armazenamento em disco é feito pelo SGBDG Neo4j. As características mais notáveis do armazenamento do Neo4j são que ele divide o armazenamento em diversos arquivos, e utiliza registros de tamanho fixo sempre que possível, de forma a facilitar o endereçamento dos registros. Cada arquivo contém dados específicos de uma parte do grafo, como, por exemplo, vértices, relacionamentos e propriedades. A separação da estrutura do grafo das propriedades é importante para a realização de travessias com bom desempenho.

O registro de vértices é de tamanho fixo de 9 bytes, sendo o primeiro byte utilizado para sinalizar se o registro está em uso, os quatros bytes seguintes são o identificador do primeiro relacionamento do vértice, e os quatro últimos são o identificador da primeira propriedade do vértice. O Neo4j utiliza identificadores internos inteiros, e, como os registros possuem tamanho fixo, é fácil obter a posição de

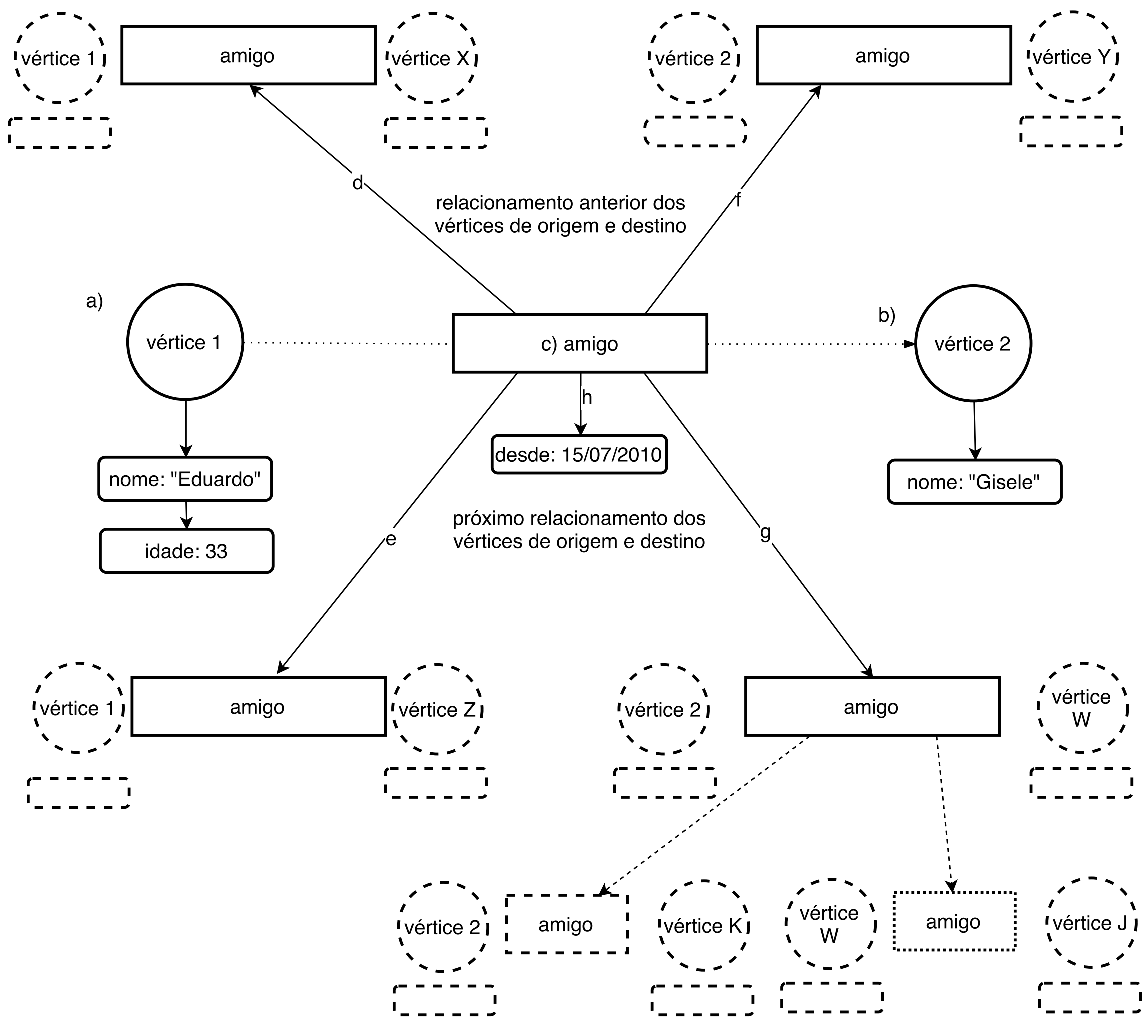


Figura 2.5: armazenamento em disco do Neo4j

qualquer registro no arquivo, bastando multiplicar o identificador do elemento pelo tamanho do registro. O vértice de identificador 100, por exemplo, está localizado no byte número 900.

O registro de arestas é de tamanho fixo de 33 bytes, sendo o primeiro byte, assim como no arquivo de vértices, destinado a indicar se o registro está em uso ou não. Os oito bytes seguintes são para os identificadores dos nós de origem e destino. Em seguida, quatro bytes são utilizados para um ponteiro para o tipo de relacionamento. Dezesseis bytes seguem para ponteiros dos relacionamentos próximos e anteriores dos nós de origem e destino. Estes últimos são denominados cadeia de relacionamentos. Por fim, quatro bytes para a primeira propriedade da aresta.

Assim como os arquivos de vértices e arestas, o arquivo de propriedade é constituído por registros de tamanho fixo. Cada registro consiste de quatro blocos de propriedade e do identificador da próxima propriedade. Cada propriedade ocupa de um a quatro blocos de propriedade, logo, um registro pode ter no máximo quatro propriedades. Cada propriedade armazena o tipo da propriedade e um ponteiro para o arquivo de índice de propriedades, onde o nome da propriedade é armazenado. Caso o valor de uma propriedade ultrapasse o tamanho disponível, esse valor é gravado em outros arquivos de armazenamento dinâmico, e o valor da propriedade passa a ser um ponteiro para este outro arquivo.

As figuras 2.4 e 2.5 ilustram como o Neo4j armazena o grafo em disco. As travessias no grafo são realizadas percorrendo-se as listas encadeadas de ponteiros de arestas, e seguindo os ponteiros de vértices e propriedades conforme necessário pela travessia.

O arcabouço Mogwai, por sua vez, não implementa diretamente o armazenamento em disco dos grafos. Esta função é delegada para as implementações particulares do arcabouço. Mais detalhes são apresentados no capítulo 3.

2.3.3 Cache

Ainda segundo Robinson[2], apesar de as máquinas de hoje em dia possuírem grande capacidade de memória de acesso aleatório, ainda há casos em que o grafo não pode ser armazenado todo em memória. E mesmo com um armazenamento em disco eficiente, ainda se faz necessário considerar o hardware para obter bom desempenho nas travessias do grafo. Os discos de estado sólido (*Solid State Discs, SSD*) oferecem um grande ganho de desempenho quando comparados com os discos rotacionais. Porém, o caminho entre a CPU (central de processamento único) e o disco é mais lento do que entre a CPU e a memória principal ou o cache L2, de tal forma que a utilização de caches é interessante para minimizar a necessidade de acesso ao disco.

O Neo4j utiliza uma arquitetura de cache de duas camadas. A camada inferior realiza cache do sistema de arquivos. A camada superior realiza o cache dos objetos em memória que representam os vértices, arestas, e propriedades. Os vértices em cache contém tanto suas propriedades como suas arestas, sendo uma representação bem mais rica do que a armazenada em disco. As arestas, por sua vez, contém suas propriedades e os vértices de origem e destino.

2.4 Compiladores e interpretadores de consulta

Nesta seção apresentam-se os conceitos relacionados com o problema de desenvolver uma linguagem de consulta.

Um interpretador de consultas é um programa que utiliza a tecnologia de compiladores, mas, diferentemente de um compilador, ele não produz um programa equivalente numa outra linguagem, mas sim traduz um predicado em comandos para percorrer um banco de dados.

Antigamente, os compiladores eram considerados programas difíceis de serem escritos. Esta dificuldade talvez seja a principal razão pela qual a maioria das linguagens de consulta é implementada apenas na própria ferramenta que a propôs. Porém, o desenvolvimento de técnicas e ferramentas tornou a tarefa de escrever os

compiladores mais fácil. Uma destas ferramentas é o *Another Tool for Language Recognition* (ANTLR) [27] .

O ANTLR é uma ferramenta geradora de *parsers* que pode ser utilizada para implementar interpretadores de linguagens, compiladores, e outros tradutores. Com ele é possível ler, processar, executar ou traduzir texto estruturado ou arquivos binários. A partir da descrição de uma linguagem formal (chamada de gramática), o ANTLR gera um *parser* para essa linguagem que constrói automaticamente as estruturas intermediárias (árvores) que representam como a gramática corresponde à entrada. O ANTLR também gera automaticamente percorredores de árvore que são utilizados para visitar os nós da árvore para executar código específico da aplicação.

2.5 Tipos de consulta

Para encerrar este capítulo, é interessante considerar no desenvolvimento de uma linguagem de consulta quais tipos de consulta ela será capaz de responder.

Diferentes modelos e estruturas de dados oferecem diferentes facilidades para responder determinadas perguntas do usuário. Os grafos são comumente utilizados para responder certos tipos de consulta, que foram classificados por Angles [11] em quatro diferentes categorias: adjacência, alcance, correspondência de padrões e sumarização. A MogwaiQL foi projetada para ser capaz de responder a todos estes tipos de consulta.

2.5.1 Adjacência

As consultas de adjacência respondem às perguntas básicas de se dois vértices são adjacentes, isto é, se há uma aresta conectando-os; e, de maneira similar, se duas arestas são adjacentes, quer dizer, se elas possuem um vértice em comum. Outra consulta comum é listar todos os vizinhos de um determinado vértice.

2.5.2 Alcance

Este tipo de consulta visa determinar se existe um caminho entre dois nós. Por ser um problema computacionalmente dispendioso, é comum que as consultas sejam feitas com uma restrição de tamanho máximo do caminho. Muitas vezes é desejável encontrar caminhos que possuam determinadas características, que são comumente expressas na forma de restrições de vértices e/ou arestas, ou utilizando expressões regulares.

Nesta categoria também encontram-se as consultas de caminho mínimo. Onde caminho mínimo pode ser o caminho com menor número de arestas, ou pode ser o caso que a distância seja medida por alguma outra grandeza, como o peso da aresta.

2.5.3 Correspondência de padrões

Este tipo de consulta consiste em encontrar todos os subgrafos isomórficos a um determinado padrão de grafo.

2.5.4 Sumarização

São as consultas que não pretendem responder perguntas sobre a estrutura do grafo. Em outras palavras, o seu retorno não são elementos do grafo (vértices, arestas, caminhos). Nesta categoria enquadram-se as consultas de agregação, contagem de elementos, média, entre outros. Também se enquadram nesta categoria consultas sobre certas propriedades do grafo e seus elementos, como a ordem do grafo, o grau de um vértice, o diâmetro do grafo, para citar alguns.

2.6 Trabalhos relacionados

Já foram propostas outras linguagens de consulta para grafos, e diferentes ferramentas para a gerência destes dados. Ocorre que a maioria das linguagens de consulta a bancos de dados de grafos está implementada em apenas uma ferramenta. Portanto,

a análise da linguagem e a análise do SGBDG praticamente andam juntas, de tal forma que será dado foco para a análise das linguagens.

Um problema na comparação dos SGBDG e suas linguagens de consulta é que nem todas as ferramentas utilizam o mesmo modelo de dados. O modelo de dados delimita os tipos de perguntas que podem ser feitas ao banco de dados, dificultando a comparação entre as linguagens quando os modelos são diferentes. Por exemplo, o modelo de dados vai dizer se os grafos são direcionados ou não, se os elementos do grafo podem conter propriedades, ou rótulos, entre outras características. Uma comparação dos diferentes modelos de bancos de dados de grafos pode ser encontrada em [11].

Ao avaliar as diferentes linguagens de consulta deve-se ter cuidado para não cair na subjetividade. Preferencialmente devem ser estabelecidos critérios de comparação objetivos. Holzschuher e Peinl [18] utilizam os seguintes critérios em sua avaliação: a facilidade de aprendizado, a legibilidade do código, a possibilidade de melhorar o desempenho manualmente, e os recursos disponíveis para realização de travessias. Destes critérios, pode-se dizer que os dois primeiros foram utilizados pelos autores de maneira mais subjetiva do que objetiva, já que os mesmos não realizaram experimentos, como, por exemplo, avaliar a facilidade de aprendizado através do tempo que uma amostra de pessoas leva para escrever um determinado número de consultas numa linguagem ou outra, mas sim avaliaram a sua própria experiência no teste que propuseram.

A maior parte dos trabalhos enumerados a seguir foi escolhida pela sua relevância acadêmica. A outra parte foi selecionada pela difusão do seu uso. O objetivo principal desta relação de trabalhos relacionados é de ilustrar o que existe no universo das linguagens de consulta a fim de obter impressões sobre a facilidade de aprendizado e legibilidade. No final do capítulo são feitas maiores considerações sobre estas linguagens.

2.6.1 GOQL

GOQL [28] utiliza um modelo de dados de grafos orientado a objetos e estende as funcionalidades da *Object Query Language* (OQL) com construções para criar, manipular e consultar objetos dos tipos grafo, caminho e aresta. A linguagem utiliza a familiar estrutura "*select...from...where*". As consultas em GOQL são traduzidas para uma linguagem baseada em operadores, O-Algebra (uma álgebra de objetos projetada para processar consultas em sistemas gerenciadores de bancos de dados orientados a objetos), também estendida com três novos operadores temporais: *next*, *connected*, e *until*. No contexto de apresentações multimídia, a consulta a seguir recupera um grafo que possui três fluxos, sendo dois de vídeo, seguidos por um de texto:

```
select g
from g in Pres_Graphs, s1, s2, s3 in g.Nodes,
  e1, e2 in g.Edges
where
  s1.tipo = 'video',
  s1.nome = 'O Periodo Romantico',
  s2.tipo = 'video',
  s2.nome = 'A ascencao do Realismo',
  s3.tipo = 'texto',
  s3.nome = 'Obras primas e seus Autores',
  e1:s1 Next s2, e2:s1 Next s3
```

2.6.2 GraphLog

GraphLog foi proposta em 1990 por Consens e Mendelzon [29] como uma linguagem visual. Ela representa tanto os dados quanto as consultas como grafos. Uma consulta pode representar caminhos com o uso de arestas com expressões regulares. As consultas são especificadas com uma ferramenta visual. A execução de uma

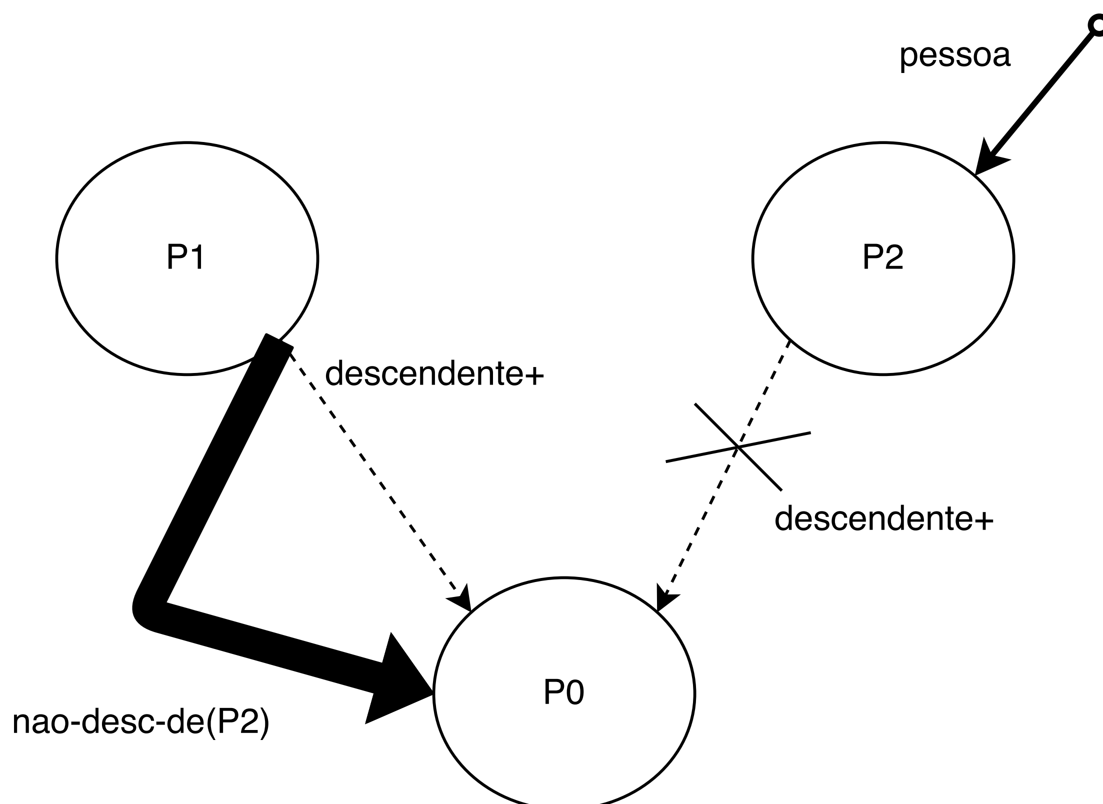


Figura 2.6: os descendentes de P2 que não são descendentes de P1

consulta utiliza uma função de tradução lógica que traduz os grafos de consulta em programas de lógica. A figura 2.6 exemplifica uma consulta em GraphLog, que é traduzida para o seguinte programa:

```
nao-desc-de(P1, P3, P2) ← desc-de(P1, P3).
not desc-de(P2, P3),
pessoa(P2).
desc-de(X, Y) ← desc(X, Y).
desc-de(X, Y) ← desc(X, Z), desc-de(Z, Y)
```

2.6.3 PQL

Para consultar bases de dados de interações ou caminhos de proteínas foi proposta a *Pathway Query Language* (PQL) [1], e apresentada uma implementação baseada no SGBDR Oracle. A sintaxe geral da PQL é da forma:

SELECT especificação-do-subgrafo

FROM variáveis-nós

WHERE conjuntos-de-condições-de-nós

A consulta a seguir, por exemplo, retorna um grafo que consiste em dois nós, um "3-isopropilmalato" e outro "EC1.1.1.85":

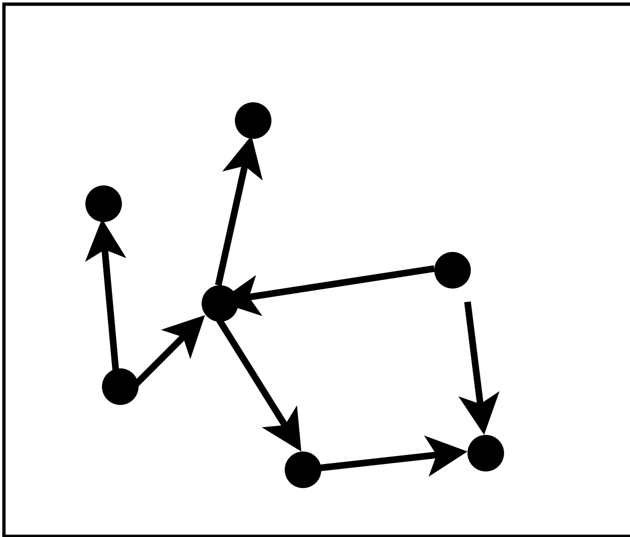
```
SELECT *
FROM A, B
WHERE A.nome = '3-isopropilmalato' AND
B.nome = 'EC1.1.1.85'
```

A sintaxe da linguagem é similar à *SQL*, e, apesar da sua simplicidade, é capaz de expressar problemas de isomorfismos de grafos. Essa característica facilita a implementação da PQL sobre um SGBDR. A avaliação de uma consulta procede da seguinte maneira: para cada uma das variáveis na cláusula *FROM* são computados todos os possíveis valores que podem ser assumidos contanto que as condições das cláusulas *WHERE* sejam avaliadas para VERDADEIRO, e em seguida é feito o produto cartesiano destas variáveis. Então são removidas todas as instâncias em que as cláusulas WHERE são avaliadas como FALSO. Em seguida, as combinações distintas são combinadas para formar o grafo resultado, que a princípio não contém nenhuma aresta. Estas são adicionadas depois dependendo do que foi especificado na cláusula *SELECT*. As consultas em PQL são traduzidas para procedimentos armazenados em PL/SQL.

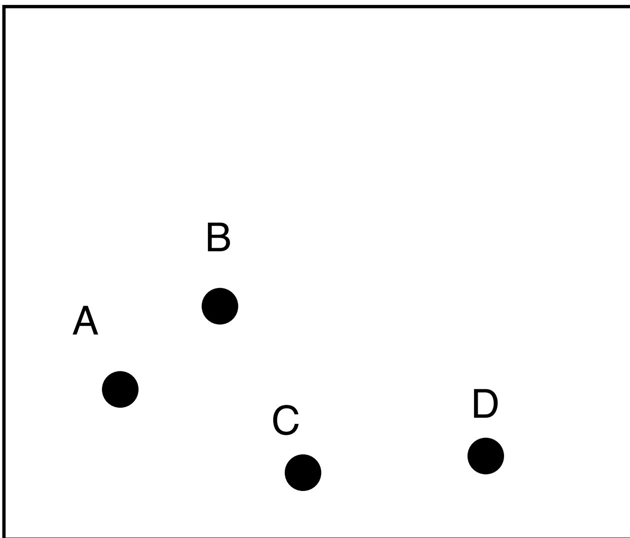
Leser [1] reconhece que a PQL não é capaz de operações muito complexas em grafos, como a computação de árvores geradoras, e que estas devem ser realizadas por aplicações com estruturas de dados otimizadas. A figura 2.7 ilustra os passos de avaliação de uma consulta PQL.

2.6.4 RDF e SPARQL

O *Resource Description Framework* (RDF) [7] é um arcabouço para representação de dados na *web*. É um modelo definido pelo consórcio W3C para intercâmbio

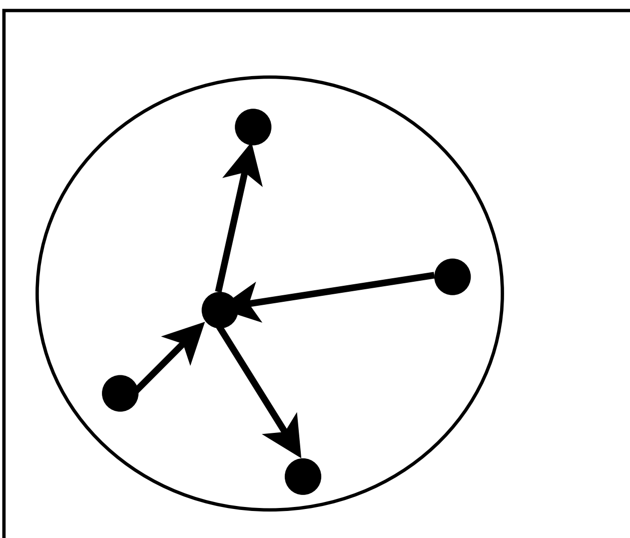


Grafo subjacente
do banco de dados



Correspondência da
consulta.

SELECT B[-1]
FROM A, B, C, D
WHERE
A[-1]B[-1]C[-1]D



Grafo resultado da
consulta. Nós são
adicionados e
removidos, arestas são
adicionadas.

Figura 2.7: avaliação de consulta PQL

de dados. Ele estende a estrutura de *links* da *web* através da utilização de *URIs* (*Uniform Resource Identifier*) para nomear o relacionamento entre objetos. Essa estrutura formada por dois objetos e um predicado é chamada de “tripla”. Um conjunto de triplas é chamado de grafo RDF, e pode ser visualizado como um diagrama onde cada tripla é representada por dois nós unidos por um arco direcionado nomeado. Em uma tripla, o nó de origem é chamado de sujeito, e o de destino, de objeto; a aresta é denominada de predicado. Os nós RDF podem ser de três tipos: literais, vazios, e *IRIs* (*Internationalized Resource Identifier*). Diversas ferramentas foram construídas para trabalhar com os grafos RDF, tais como a linguagem SPARQL e sistemas gerenciadores de bancos de dados como o GraphDB (anteriormente chamado de OWLIM).

Por exemplo, a consulta:

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>

SELECT ?titulo
WHERE  { { ?book dc10:titulo  ?titulo } UNION
         { ?book dc11:titulo  ?titulo } }
```

Com o banco de dados:

@prefix dc10: ¡http://purl.org/dc/elements/1.0/¿ .

@prefix dc11: ¡http://purl.org/dc/elements/1.1/¿ .

_:a dc10:titulo ”Tutorial da SPARQL”.

_:a dc10:criador ”Renata”.

_:b dc11:titulo ”Tutorial do Protocolo da SPARQL”.

_:b dc11:criador ”Ricardo”.

_:c dc10:titulo ”SPARQL”.

_:c dc11:titulo ”SPARQL (atualizado)”.

Retorna:

titulo

"Tutorial do Protocolo da SPARQL"

"SPARQL"

"SPARQL (atualizado)"

"Tutorial da SPARQL"

2.6.5 GraphGrep

GraphGrep[30] foi proposto como um método rápido e universal para consultar grafos e encontrar todas as ocorrências de um subgrafo num banco de dados de grafos. Ele utiliza uma linguagem de consulta chamada *Glide* que combina recursos da *XPath* e *Smart*.

A utilização do GraphGrep segue três passos: criar um arquivo de dados, pré-processar o arquivo de dados, e executar a consulta (que deve ser especificada em arquivo próprio também), desta forma gerando um arquivo de saída. As consultas em Glide são parseadas para gerar *fingerprints (hashed set of paths)*, que são utilizados para reduzir o espaço de busca da consulta. Com os *fingerprints* é possível descartar todos os grafos que claramente não contém o padrão especificado.

Os grafos são especificados utilizando uma notação linear derivada da *Smiles* onde cada nó é representado uma única vez: os nós são representados pelos seus *labels* e são separados por barras; as ramificações são agrupadas utilizando parênteses e os ciclos são quebrados através do corte de uma aresta, nomeando-a com um número inteiro. Os vértices de uma aresta de corte são representados pelo seu *label* seguido pelo número inteiro e a barra. Se um mesmo nó pertence a várias arestas de corte, o seu *label* é seguido por uma lista de inteiros.

Por exemplo, os grafos da figura 2.8 são codificados da seguinte forma:

Grafo g1: $A\%1\%2/B/C\%2/B\%1/$

Grafo g2: $A\%3/B\%1\%2\%3/(E/)C\%2/D\%1/$

Grafo g3: $B/C/A/B/C/$

São utilizados caracteres curinga para representar componentes opcionais em um grafo. O caractere "." corresponde a qualquer nó simples; "*" a zero ou mais nós,

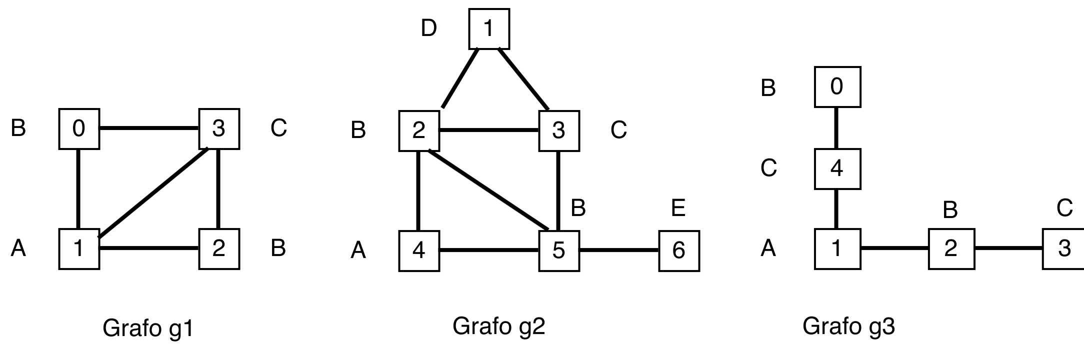


Figura 2.8: grafos

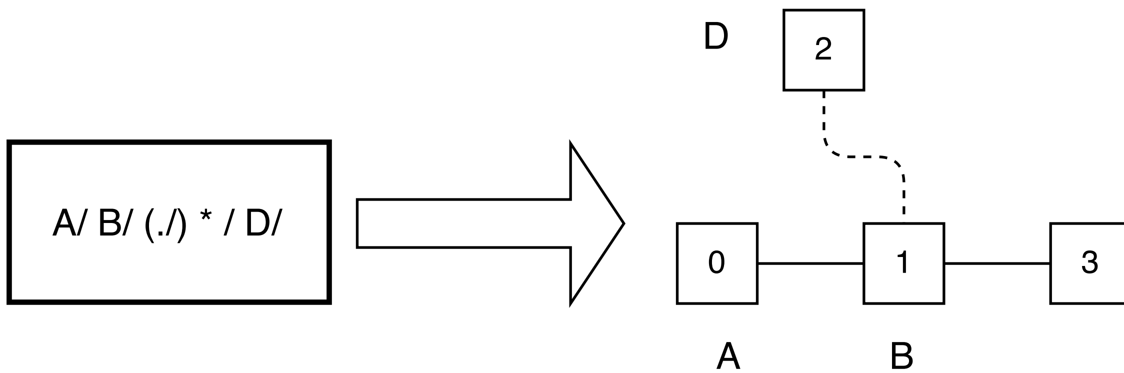


Figura 2.9: consulta Glide

"?" zero ou um nó; e "+" um ou mais nós.

A figura 2.9 ilustra uma consulta em Glide.

2.6.6 GraphQL

A GraphQL[26] é uma linguagem de consulta de grafos que utiliza um padrão de grafo como unidade operacional básica. Um padrão de grafo consiste de uma estrutura de grafo e de um predicado nos atributos do grafo.

A GraphQL introduz a ideia de linguagem formal para grafos. Em linguagens formais clássicas, uma gramática formal consiste de um conjunto finito de terminais e não terminais, e de um conjunto finito de regras de produção de caracteres. Na

GraphQL essa noção é ampliada para grafos ao invés de *strings*, de tal forma que padrões de grafos são construídos a partir de outros padrões de grafos, através de regras de produção: concatenação, disjunção, e repetição.

A GraphQL define um álgebra de grafos derivada da álgebra relacional na qual o operador de seleção é generalizado para realizar correspondência de padrões de grafos, e um operador de composição é introduzido para reescrever os resultados das consultas (este operador também expressa os operadores renomear e projetar da álgebra relacional). Além destes operadores é definido o produto cartesiano; e a junção é definida como um produto cartesiano seguido de seleção. A GraphQL utiliza a sintaxe FLWR (*For, Let, Where, and Return*) derivada da *XQuery*. A consulta a seguir gera um grafo de coautoria de um conjunto de dados DBLP.

```
graph P { node v1 <autor>;
          node v2 <autor>; };
C := graph {};
for P exhaustive in doc(''DBLP'')
let C := graph {
    graph C;
    node P.v1, P.v2;
    edge e1 (P.v1, P.v2);
    unify P.v1, C.v1 where P.v1.nome = C.v1.nome;
    unify P.v2, C.v2 where P.v2.nome = C.v2.nome; }
```

A implementação da GraphQL acelera o algortimo básico de correspondência de padrões de grafos com o uso de três técnicas que se utilizam das informações estruturais do grafo. A primeira delas gera o espaço de busca com poda local utilizando subgrafos vizinhos ou seus perfis. A segunda técnica reduz o espaço de busca geral utilizando informações estruturais globais. Por último, a ordem de busca é otimizada utilizando um modelo de custo para grafos.

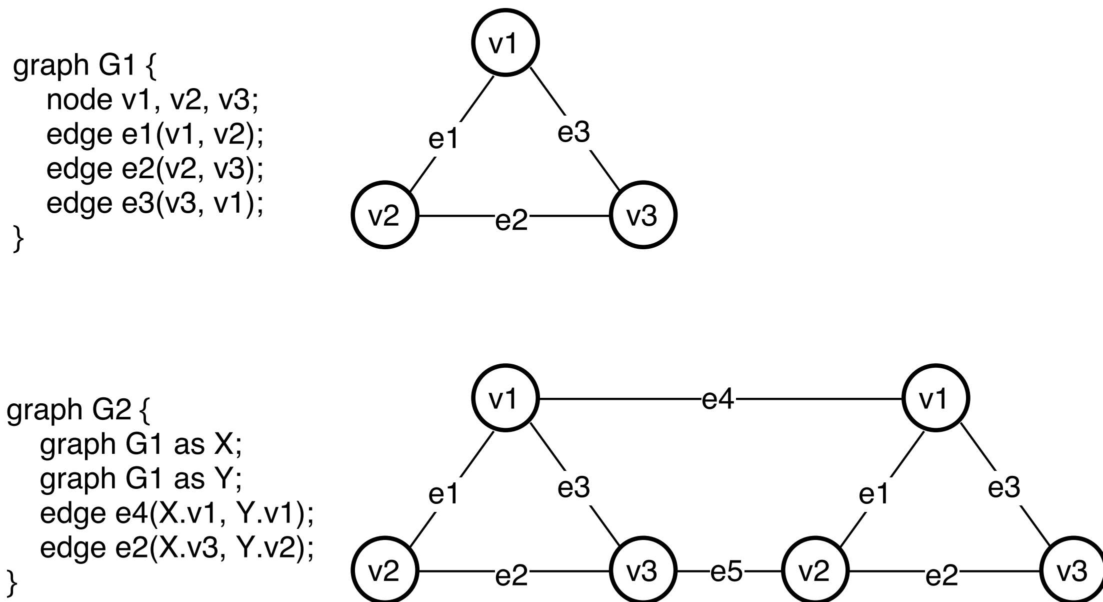


Figura 2.10: exemplo de composição de padrões em GraphQL

2.6.7 AQL

A *Arango Query Language* (AQL) [31] é a linguagem de consultas do Sistema Gerenciador de Banco de Dados (SGBD) ArangoDB. O ArangoDB é um SGBD *Not Only SQL* (NoSQL) multi-modelo: ele trabalha tanto com os modelo chave/valor, de documentos, e de grafos, sendo possível misturar esses modelos de dados em uma mesma consulta. A AQL é uma linguagem declarativa, que visa ser legível para humanos, e que, portanto, utiliza palavras-chave da língua inglesa. A AQL também visa ser independente de cliente (linguagem de programação que o cliente utilize), e ser capaz de executar padrões de consulta complexos nos diferentes modelos de dados que o ArangoDB trabalha. A linguagem é similar à SQL, e realiza apenas operações de consulta e manipulação de dados, não realizando operações de definição de dados e nem de controle de dados.

Todas as consultas em AQL retornam um *array* de elementos.

A consulta a seguir, por exemplo, recupera o documento que descreve o usuário “Eduardo”:

```
FOR doc IN users
    FILTER doc._key == "Eduardo"
    RETURN doc
```

É possível trabalhar com grafos no ArangoDB de duas maneiras. A primeira delas é utilizando grafos nomeados onde o ArangoDB gerencia as coleções envolvidas em um grafo, e a segunda é utilizando funções de grafos numa combinação de coleções de documentos e arestas.

A sintaxe geral de uma travessia em AQL é:

FOR vertex[, edge[, path]]

IN [min[..max]]

OUTBOUND—INBOUND—ANY startVertex

GRAPH graphName

[OPTIONS options]

A consulta a seguir seleciona os caminhos do grafo "traversalGraph", de tamanhos de 1 a 5, que se iniciam no vértice "circles/A", e percorrem arestas de saída.

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A'
GRAPH 'traversalGraph'
    FILTER p.edges[0].theTruth == true
    RETURN p
```

A AQL também permite realizar consultas de caminhos mínimos entre vértices, sendo que os vértices de início e fim devem ser especificados por um identificador, ou por um documento com o atributo "_id".

2.6.8 G-Store

G-Store [32] é proposto como um gerenciador leve de armazenagem de grafos baseado em disco. Ele utiliza a estrutura do grafo para determinar a melhor maneira de armazenar o grafo fisicamente. Sua estratégia é baseada num algoritmo multinível

que particiona o grafo em páginas e arranja essas páginas de forma a minimizar a distância em disco entre vértices adjacentes. A ferramenta possui um motor de consulta interno capaz de realizar travessias em profundidade, testes de alcance, buscas de caminho mínimo, e busca de árvores de caminho mínimo.

A linguagem utilizada pelo G-Store também é similar à SQL. A sintaxe geral de uma consulta é:

SELECT lista_seleção

[WHERE condição

    [cláusula_de_travessia]]

    | [cláusula_de_caminho

    [WHERE condição]]]

;

No contexto de banco de dados com os dados da Wikipedia em inglês, a consulta a seguir procura os caminhos de comprimento menor ou igual a quatro do artigo "Adam Smith" até o artigo "Bread" ou "Butter":

```
SELECT LEVEL, PATH(title,'-')
WHERE (title = 'Bread' OR title = 'Butter')
    AND LEVEL <= 4 AND ROWNUM <= 8
START WITH title = 'Adam Smith';
```

A consulta de caminho mínimo entre o artigo "Alps" e o artigo "Oxford" que não passa pelos artigo "France" e "City" pode ser expressa como:

```
SELECT title, abstract IN SHORTEST PATH
START WITH title = 'Alps'
END WITH title = 'Oxford'
THROUGH title != 'France' AND title != 'City';
```

2.6.9 Gremlin

Gremlin[14] é a linguagem de travessia do Apache TinkerPop. É uma linguagem funcional, de fluxo de dados, que visa permitir ao usuário expressar travessias e consultas a um grafo de propriedades de forma sucinta.

Uma travessia é composta por uma sequência (potencialmente aninhada) de passos, onde cada passo executa uma operação atômica no fluxo de dados. Existem três tipos de passos: um *map-step* é um passo que executa uma transformação nos objetos do fluxo de dados, um *filter-step* é um passo que remove elementos do fluxo, e, por fim, um *sideEffect-step* é um passo que realiza uma computação estatística sobre o fluxo. A biblioteca do Gremlin oferece uma ampla variedade de passos que quando combinados respondem a "qualquer pergunta concebível" do usuário sobre os dados do grafo. Por exemplo, a consulta a seguir obtém o nome das pessoas que são conhecidas pelos conhecidos de "Gremlin":

```
g.V().has("nome", "Gremlin").
    out("conhece").
    out("conhece").
    values("nome")
```

As instruções podem ser traduzidas, respectivamente para:

- 1. Obtenha o vértice de nome "gremlin".
- 2. Navegue até as pessoas que Gremlin conhece.
- 3. Navegue até as pessoas que essas pessoas conhecem.
- 4. Obtenha o nome dessas pessoas.

2.6.10 Cypher

Cypher[22] é a linguagem utilizada pelo SGBDG Neo4j, e openCypher[15] é um projeto que visa entregar uma especificação completa e aberta desta linguagem. A sintaxe do Cypher visa prover uma maneira familiar de encontrar correspondências

de padrões de grafos em um grafo. A consulta a seguir, por exemplo, obtém o elenco dos filmes que começam com “T”:

```
MATCH (ator:Pessoa)-[:ATUOU_EM]->(filme:Filme)
WHERE filme.titulo STARTS WITH "T"
RETURN filme.titulo AS titulo, collect (ator.nome) AS elenco
ORDER BY titulo ASC LIMIT 10;
```

Diferentemente da Gremlin, a Cypher é uma linguagem declarativa. Ela permite ao usuário dizer “o que” ele deseja consultar sem especificar “como”. Ela declara um padrão de grafo a ser obtido, especifica restrições neste padrão, e quais dados obter do resultado, mas não especifica como fazê-lo, não especifica os passos a serem seguidos, nem em que ordem. Linguagens declarativas se estabeleceram como amigáveis ao usuário, e apropriadas para usuários “não-programadores”, ou “leigos” [33].

Nota-se que a Cypher faz uso de “arte ASCII” (desenhos com caracteres de texto) para representar a estrutura do grafo: na primeira linha, os vértices são representados por parênteses, e a aresta pelas chaves e os outros caracteres formam o desenho de uma seta.

2.7 Comparação da MogwaiQL com outras linguagens de consulta

A linguagem MogwaiQL proposta neste trabalho, apesar de possuir menos recursos, pode-se dizer que é uma extensão da Cypher, na medida em que utiliza basicamente a mesma sintaxe, mas permite trabalhar com múltiplos grafos.

Sob o ponto de vista de ser uma ferramenta de propósito geral, a linguagem de consulta MogwaiQL se sobressai sobre a maioria das outras linguagens pelo fato de utilizar o modelo de grafo de propriedades, por trabalhar com múltiplos grafos, por ser declarativa, trabalhar com padrões de grafos e caminhos, e por possuir gramática definida, de forma que o trabalho necessário para sua adoção por outras ferramentas

é bastante reduzido. Não obstante, as outras ferramentas e linguagens possuem seus próprios pontos fortes e fracos.

Sendo uma extensão da OQL, a GOQL necessita que o *schema* do banco de dados seja definido para que os atributos dos objetos estejam disponíveis em tempo de compilação da consulta. Apesar de trabalhar com múltiplos grafos, a álgebra não possui operações entre grafos, e também não é capaz de realizar buscas por padrões de grafos. Por outro lado, há diversos operadores de caminhos, e neste ponto ela se sobressai sobre a MogwaiQL.

A GraphLog é uma linguagem visual, e, portanto, necessita de uma ferramenta visual para especificar as consultas. Ela trabalha com um único grafo, e não utiliza o modelo de grafo de propriedades. As consultas são realizadas especificando padrões de grafos visualmente. As arestas podem representar arestas simples ou caminhos. Os caminhos são especificados por expressões regulares nas arestas. As consultas são traduzidas para programas lógicos que são executadas pelo protótipo da ferramenta. Há também uma interface para executar as consultas sobre a *Hypertext Abstract Machine (HAM)*. A maneira como a consulta visual é traduzida para um programa lógico faz com que seja necessário adotar um modelo de dados específico: cada nó e aresta somente pode conter um único valor. Isto pode ser visto como desvantajoso em relação à MogwaiQL, pois este modelo de dados faz com que cada atributo de uma entidade de um determinado mini mundo seja um vértice. Consequentemente, os grafos tanto os de dados como os de consultas são maiores e mais complexos. Executar consultas mais complexas em grafos com mais elementos provavelmente exige mais poder computacional.

A PQL possui um operador interessante que é o ISA, que determina se um nó é de um determinado tipo. Os tipos são definidos por um grafo acíclico de conceitos, e a cada tipo está associado um conjunto de funções. A verificação de tipo através de uma hierarquia é uma funcionalidade não existente na MogwaiQL. Por outro lado, a PQL sofre por não ser uma linguagem de propósito geral, possuindo um modelo de dados mais restrito, que define apenas um pequeno conjunto de propriedades para

nós e arestas, de forma que não é possível, por exemplo, especificar restrições nos atributos das arestas nas consultas. As consultas são executadas sobre um único grafo.

A SPARQL é uma linguagem bastante madura, com diversos recursos interessantes. Podemos dizer que a MogwaiQL se sobressai basicamente sobre um único ponto. O modelo de dados (RDF) torna a SPARQL menos amigável como uma ferramenta de propósito geral sob o ponto de vista de modelagem. Apesar de um problema poder ser modelado de diversas formas, a modelagem de um problema em um grafo de propriedades é bastante clara e direta: as entidades são modeladas como vértices, os relacionamentos entre as entidades são modelados como arestas, e as propriedades das entidades e dos relacionamentos são modeladas como propriedades nos respectivos elementos do grafo. Já no modelo RDF, tanto as entidades quanto suas propriedades são modeladas como vértices, distinguindo-se apenas pela qualificação do relacionamento. Apesar da equivalência na resolução de problemas, o modelo RDF acaba utilizando mais elementos na sua representação, e os padrões de grafos acabam ficando mais complexos nas consultas. A SPARQL oferece diversos recursos para facilitar a especificação dos padrões de grafo, com legibilidade questionável, no entanto.

A SPARQL, além de trabalhar com múltiplos grafos, oferece diversos recursos poderosos que não estão presentes na MogwaiQL, como, por exemplo, a possibilidade de aplicar restrições de atributos com expressões regulares, a possibilidade de especificar padrões de grafos opcionais, e a possibilidade de obter os resultados em diversos formatos, como JSON, XML, CSV ou TSV, para citar alguns. SPARQL é um linguagem poderosa, de amplo uso, disponível em uma ampla variedade de ferramentas [7].

O GraphGrep é uma ferramenta que necessita de pré-processamento para permitir que as consultas Glide sejam realizadas, o que o torna inadequado para tarefas *Online Transaction Processing* (OLTP) . Por outro lado, este pré-processamento é capaz de otimizar consultas de uma forma que o Mogwai não consegue. O Mogwai

utiliza as restrições nos atributos dos elementos da consulta MogwaiQL para fazer a poda do espaço de busca, e não possui nenhum recurso de otimização quando estas restrições estão ausentes. A Glide é executada numa base de dados de múltiplos grafos, porém o seu modelo de dados é muito restrito, possuindo apenas identificadores e *labels* para os nós. Problemas que estejam interessados em dados além da estrutura do grafo são praticamente impossíveis de serem resolvidos, visto que a linguagem não permite especificar restrições em atributos de diferentes tipos de dados.

A GraphQL trabalha com múltiplos grafos, e o seu modelo de dados permite que os grafos, vértices e arestas possuam atributos arbitrários. A GraphQL possui uma notação interessante para especificar padrões de grafos nas consultas, inclusive permitindo que um padrão seja definido a partir de outros padrões, e também contém operadores interessantes para especificar o grafo resultante da consulta. Por outro lado, a GraphQL não permite definir expressões regulares para especificar caminhos.

No ArangoDB, um grafo é composto por uma ou mais coleções de vértices e arestas, de modo que a modelagem mais direta seria uma coleção para representar cada entidade do domínio do problema. Os tipos de relacionamento entre as coleções devem ser definidos individualmente. A AQL possui facilidades de travessia que a MogwaiQL não possui, como, por exemplo, garantir que nenhum caminho seja retornado com um vértice duplicado, ou garantir que cada vértice seja visitado no máximo uma vez durante uma travessia. Apesar de trabalhar com múltiplos grafos, a AQL não permite consultas com padrões de grafo, de forma que sua sintaxe é complexa mesmo nas consultas mais simples.

O G-Store pode armazenar múltiplos grafos, porém trabalha com apenas um grafo por vez, o que é uma desvantagem em relação ao Mogwai. Sua linguagem de consulta é voltada para travessias e caminhos, não sendo capaz de realizar busca por padrões de grafos.

A Gremlin é uma linguagem imperativa para realização de travessias, e na sua versão 3 incluiu uma operação para realizar buscas por padrões de grafo. Apesar

de ser uma linguagem poderosa, a Gremlin também é projetada para trabalhar com um único grafo. Assim como a MogwaiQL, a linguagem Gremlin opera sobre sua própria API, de forma que, ao implementar a API, a linguagem está igualmente disponível.

Cypher é a linguagem sobre a qual a MogwaiQL foi baseada, e inclui mais recursos que esta. Por exemplo, em Cypher é possível encadear comandos, de modo que as variáveis ficam disponíveis de um comando para o outro, enquanto que na MogwaiQL isso não é possível. Cypher também permite ao usuário criar procedimentos armazenados. A partir da versão 3 do Neo4j, a Cypher permite que um vértice possua vários rótulos, enquanto que na MogwaiQL é possível atribuir apenas um. Uma limitação da Cypher é que o resultado das consultas é sempre em tuplas, não sendo possível obter grafos como resultado. Entretanto, a Cypher trabalha com um grafo apenas, o que torna a MogwaiQL mais adequada para trabalhar com certas classes de problemas sem a necessidade de modelagens complexas.

Por fim, e ainda sob o ponto de vista de uma ferramenta de propósito geral, talvez o aspecto mais importante a ser comparado com as outras linguagens é a legibilidade. Apesar de ser um critério subjetivo, vale a pena argumentar alguns pontos. As linguagens que utilizam sintaxe similar à SQL provavelmente são facilmente compreendidas por quem já está acostumado com este tipo de sintaxe. Como os SGBDR são a solução padrão *de facto* para gerência de bancos de dados atualmente, isto pode vir a ser vantajoso. Por outro lado, esse tipo de sintaxe desfavorece a especificação da estrutura do grafo sendo pesquisado, especialmente a definição das arestas. A legibilidade das linguagens visuais é muito superior no que tange à estrutura do padrão de grafo sendo buscado. Entretanto, a especificação das restrições no grafo pode tornar as consultas um tanto confusas, com demasiadas informações. A codificação utilizada pela Glide, além de não comportar o modelo de grafo de propriedades, pode ser considerada pouco amigável ao usuário, pois derivar a estrutura do grafo a partir da codificação fica rapidamente demasiado complexo conforme a estrutura do grafo cresce.

A sintaxe da Gremlin, por ser uma linguagem imperativa, é, de forma geral, bastante clara quanto ao que está sendo executado. Por outro lado, alguns recursos da linguagem, apesar de poderosos, tornam a consulta um tanto difícil de ser compreendida. Isto não chega a ser necessariamente uma desvantagem, visto que quaisquer consultas complexas, mesmo em SQL, por vezes se tornam ilegíveis.

Devido à sua difusão de uso, a Cypher talvez seja a melhor no quesito legibilidade. Ao representar a estrutura do grafo com arte ASCII, a Cypher deixa esta parte bastante legível. As restrições nos elementos podem ser especificadas junto da estrutura do grafo, ou nas cláusulas *where*, de modo que o usuário pode escolher o que melhor lhe agrada. A linguagem MogwaiQL se aproveita da sintaxe da Cypher o tanto quanto possível, e, considerando seus demais atributos, como sua capacidade de trabalhar com múltiplos grafos e ser declarativa, pode-se argumentar que é uma linguagem bastante amigável para o tipo de problema o qual ela se propõe a resolver.

2.8 Arcabouços de Processamento Paralelo em Grafos

Uma tecnologia relacionada com os SGBDG são os arcabouços de processamento paralelo em grafos, tais como o Pregel[34], Apache Giraph[35], e Apache GraphX[36], entre outros. Eles são ferramentas especializadas na análise de dados armazenados como grafos, mas não são SGBDG. Eles não suportam transações *on-line*, controle de acesso e outras operações típicas de SGBD. O modelo de computação destas ferramentas é baseado em processamento em lote, diferentemente dos SGBDG. Estas soluções destinam-se a executar operações em grandes grafos com alto desempenho.

Pode-se argumentar, grosseiramente, que os arcabouços de processamento paralelo estão para os SGBDG assim como os *data warehouses* estão para os SGBDR. Em outras palavras, eles suprem a necessidade de analisar grandes volumes de dados com requisitos de desempenho que não são possíveis de obter com os sistemas OLTP.

O Mogwai foi projetado para ser utilizado como SGBDG, possuindo operações para abrir e fechar transações, realizar operações *on-line*, e manipular índices, por exemplo. Mas, por ser uma camada intermediária abstrata, nada impede que o Mogwai seja utilizado em conjunto com arcabouços de processamento paralelo. O Apache Gremlin, por sua vez, possui em sua biblioteca um mecanismo para computação distribuída. O Mogwai poderia, futuramente, dispor de funcionalidade semelhante.

Capítulo 3

Mogwai: um arcabouço para bancos de dados de múltiplos grafos

Este capítulo destina-se a descrever o arcabouço Mogwai e a linguagem MogwaiQL, e a apresentar os seus detalhes técnicos.

3.1 Visão geral

Este trabalho propõe um arcabouço de software para gerenciar bancos de dados e uma linguagem de consulta, ambos para múltiplos grafos. Dito de outra forma, o arcabouço de software oferece métodos *CRUD* que expõem o modelo de dados de múltiplos grafos; e a linguagem de consulta permite consultas de adjacência, alcance, correspondência de padrões, e sumarização neste modelo.

A solução proposta é formada pelos seguintes componentes:

- Especificação de uma API para trabalhar com múltiplos grafos (*CRUD* e consultas).
- Implementações da API.
- Conjunto de testes que verifica a implementação da API.

- Especificação da linguagem MogwaiQL.
- Implementação da linguagem MogwaiQL.
- Interface web para realização de consultas.
- Ferramenta de importação de dados de SGBDR.

O primeiro aspecto a ser considerado na solução proposta é que há uma separação clara entre as definições e as implementações. Apesar do protótipo ter sido desenvolvido em Java, ele poderia ter sido desenvolvido em qualquer tecnologia capaz de implementar as operações definidas. Tecnologias orientadas a objetos, entretanto, propiciam uma implementação mais direta a partir da sua especificação UML. Além disto, a própria implementação Java também faz uma separação entre a definição e a implementação. A API que expõe o *CRUD* é constituída apenas por interfaces. Consequentemente, a camada de software pode ser implementada de diferentes maneiras.

Da mesma forma, a linguagem especifica apenas uma gramática/sintaxe para reconhecer um texto. Como o texto é convertido em operações de banco de dados também está sujeito a diferentes implementações.

O segundo aspecto da solução é que ela disponibiliza quatro implementações da camada de software. Conforme exposto anteriormente, a abordagem empregada não é a de definir os próprios arquivos em disco, e os programas necessários para manipulá-los e atender os requisitos de um SGBD, mas sim a de utilizar ferramentas existentes para resolver este problema. A primeira implementação foi realizada sobre o banco de dados orientado a objetos ObjectDB [37]. A segunda implementação utiliza a API JPA para realizar a persistência dos dados. Em seguida temos a implementação realizada sobre a API JDBC, e por fim uma implementação sobre o SGBDG Neo4j [17]. Pelo fato de que essas API já são abstrações do mecanismo de persistência, o desenvolvimento e os testes do Mogwai foram realizados utilizando uma variedade de SGBD. Além dos já citados ObjectDB e Neo4j, foram utilizados os SGBDR H2 [38] e MySQL [39]. A implementação que utiliza o Neo4j possui

uma segunda forma de executar as consultas que consiste em traduzir a linguagem MogwaiQL para Cypher (que já considera a modelagem de múltiplos grafos em apenas um). A existência deste segundo compilador da MogwaiQL ilustra a separação entre a definição e a implementação da MogwaiQL.

Além do arcabouço e da linguagem de consulta propriamente ditos, a solução proposta provê uma interface web para realização de consultas, e uma ferramenta para importação de dados.

Antes de prosseguir com os detalhes de cada implementação, é necessário detalhar o modelo de dados utilizado, visto que ele delimita como os dados são estruturados e que tipos de consultas podem ser feitas com esta estrutura. Isto é feito na próxima seção. Em seguida, a seção "Linguagem" descreve a linguagem MogwaiQL, e a seção "Descrição da arquitetura" descreve a solução do ponto de vista arquitetural. Por fim, os detalhes de implementação são descritos nas demais seções deste capítulo.

3.2 Modelo de dados

O modelo de dados utilizado no Mogwai é uma extensão do modelo de grafo de propriedades e do modelo utilizado pela GraphQL [26]: cada vértice, aresta, e grafo pode possuir atributos arbitrários, além de um rótulo. A API do Mogwai reflete o seu modelo de dados diretamente, conforme ilustra a figura 3.1.

A modelagem de um mini mundo utilizando este modelo de dados pode ser feita da seguinte forma: para cada instância de uma entidade é criado um vértice. O rótulo de cada vértice é o nome da entidade da qual é sua instância. Os atributos da entidade são especificados nos atributos do vértice. Os relacionamentos entre as entidades são representados por arestas entre os vértices, com os rótulos das arestas descrevendo a semântica do relacionamento. Se um relacionamento possuir atributos, os mesmos são especificados nos atributos da aresta correspondente.

Os atributos são pares chave-valor, onde a chave identifica o atributo. O rótulo é utilizado para dar semântica do mini mundo ao elemento do grafo. Por exemplo, um vértice utilizado para representar uma pessoa numa rede social pode ter um

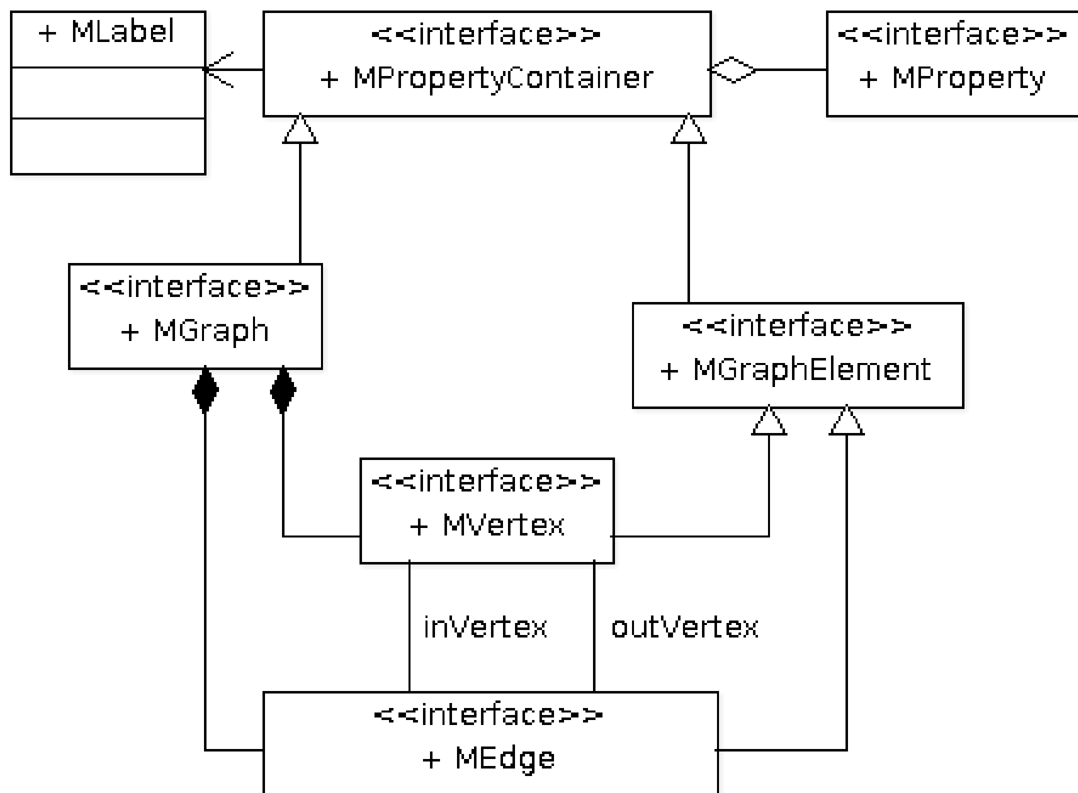


Figura 3.1: modelo de dados

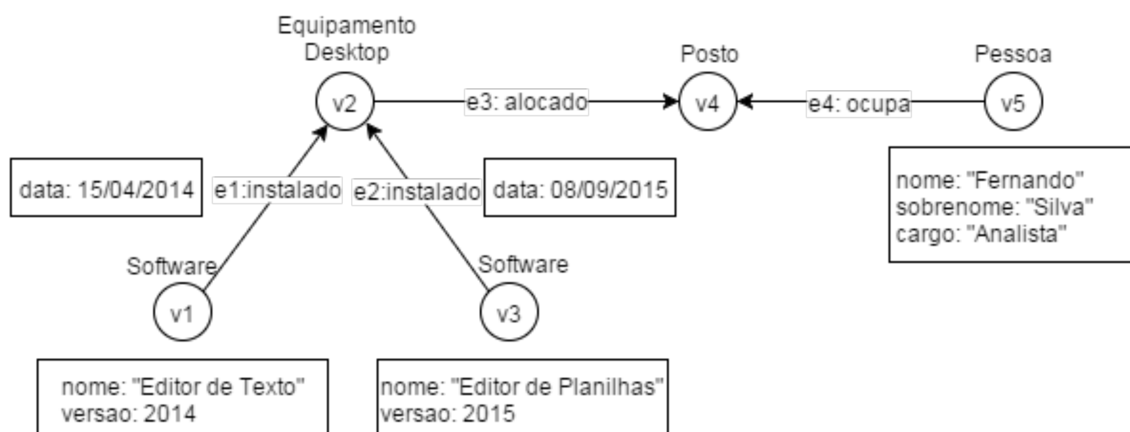


Figura 3.2: ilustração modelo de dados, exemplo de inventário de equipamentos

rótulo "pessoa" e os atributos "nome : Fernando" e "idade : 32". Em termos de implementação, as chaves e os rótulos são *strings*, e os valores dos atributos podem ser de qualquer tipo primitivo Java, além de *strings*. Os rótulos são sempre indexados, enquanto que os atributos não são necessariamente. É possível especificar quais atributos são indexados. Cada grafo e elemento do grafo deve possuir um identificador único. O identificador de um grafo deve ser único entre os grafos, e os identificadores dos elementos devem ser únicos dentro de um mesmo grafo, e podem ser *strings* ou qualquer tipo primitivo Java.

A imagem 3.2 ilustra os conceitos do modelo de dados com uma possível modelagem de um software utilizado para controlar o inventário de *softwares* e equipamentos de uma empresa. Uma pessoa ocupa um posto de trabalho, que possui equipamentos alocados a ele. Um equipamento *desktop* pode ter *softwares* instalados.

Na imagem 3.2 os vértices são representados pelos círculos, seus identificadores estão escritos na parte interior e seus rótulos na parte exterior superior. As arestas são representadas pelas setas, seus identificadores e rótulos estão descritos separados por dois pontos. As propriedades dos vértices e das arestas são exibidas dentro das caixas.

3.3 Linguagem

Nesta seção descreve-se a linguagem de consulta denominada MogwaiQL. A MogwaiQL foi desenvolvida a partir de uma álgebra projetada para permitir os tipos de consulta de adjacência, alcance, correspondência de padrões de grafos, e sumarização. Em primeiro lugar é apresentada a álgebra, e em seguida, a linguagem propriamente dita.

3.3.1 Álgebra

Para realizar solicitações básicas de recuperação de dados nos diferentes grafos de um banco de dados de múltiplos grafos, define-se uma álgebra. Antes, porém, se faz necessário definir um conceito auxiliar: o padrão de grafo.

Padrão de grafo

Um padrão de grafo é definido [26] como sendo um grafo mais um predicado nos seus atributos. Um padrão de grafo é utilizado para selecionar elementos de interesse; ele é a parte principal de uma consulta de grafos. Como o modelo de dados do Mogwai é diferente do utilizado em [26], este conceito é estendido para incluir predicados nos rótulos também.

Nas definições a seguir, sendo G um grafo, V(G) é o conjunto de vértices deste grafo, e E(G) é o conjunto de arestas do grafo. Uma aresta "e" que une dois vértices u e v é representada como e(u, v).

Definição 1: Um padrão de grafos é um par[26] P = (C, λ), onde C é um grafo, e λ é um predicado nos atributos e rótulos do grafo. O predicado λ é constituído por expressões booleanas nos atributos ou nos rótulos dos elementos do grafo.

Definição 2: Um padrão de grafo P(C, λ) tem uma (ou mais) correspondência M com o grafo D caso C seja um subgrafo isomórfico de D e as condições λ sejam respeitadas. Em outras palavras, caso exista um mapeamento injetor ϕ: V(M) $\rightarrow$ V(D) tal que para $\forall e(u, v) \in E(M), (\phi(u), \phi(v))$ é uma aresta em G, e o predicato λ é satisfeito.

Tal como na linguagem Cypher [22], na linguagem MogwaiQL um padrão de grafo é especificado utilizando arte ASCII (desenhos com caracteres da ASCII (*American Standard Code for Information Interchange*)). Os vértices são especificados entre parênteses, e as arestas com traços, colchetes e sinais de menor e maior. Ambos podem ter variáveis associadas para serem utilizadas na especificação do predicado.

Exemplo: Em relação ao exemplo da figura 3.2, a consulta aos equipamentos onde o software "Editor de Texto" está instalado poderia ser feita com o seguinte padrão: (s:Software)-[e:instalado]->(d) where s.nome = "Editor de Texto".

Este padrão especifica que sejam selecionados os vértices com rótulo "Software", que possuem uma aresta de saída de rótulo "instalado" para qualquer vértice, e cuja propriedade "nome" tenha o valor "Editor de Texto". A execução desta consulta formaria o mapeamento ϕ: s $\rightarrow$ V1, e d $\rightarrow$ V2. O vértice V1 possui o rótulo "Software", e a propriedade "nome" com valor "Editor de Texto". A aresta "e" com rótulo "instalado" possui seu correspondente em G: (V1, V2). Portanto, o predicado é atendido.

3.3.2 Operações

Esta seção descreve as operações da álgebra. Em geral é possível aplicar diversas operações da álgebra em sequência, desde que a saída de uma operação seja a entrada de outra. Mas, devido à falta de fechamento da álgebra, nem todas as operações podem ser encadeadas.

Seleção

Um operador de seleção σ é definido utilizando um padrão de grafo P. Ele toma como entrada o padrão e um conjunto de grafos $\{G_1, G_2, ...\}$, e produz um grafo composto pelos subgrafos que satisfazem o padrão de consulta.

σ_P $(G_1, G_2, ...)$: a seleção dos subgrafos isomórficos ao padrão de grafo P.

União

A união é uma operação binária da teoria dos conjuntos. O resultado desta operação é um grafo formado pela união dos vértices e arestas de ambos os grafos operandos. Por ser uma operação de conjuntos, duplicatas de vértices e arestas são eliminadas considerando os atributos dos elementos.

$G_1 \cup G_2$: a união de dois grafos.

Diferença

Assim como a união, a diferença é uma operação de conjuntos, que seleciona apenas os elementos de um grafo que não constam no outro grafo.

G_1 - G_2 : a diferença de dois grafos.

Interseção

A interseção é a última operação de conjuntos, que gera um grafo composto apenas pelos vértices e arestas comuns a ambos os grafos.

$G_1 \cap G_2$: a interseção de dois grafos.

Junção

O operador junção gera um novo grafo a partir da união de dois grafos, onde são criadas arestas entre vértices que atendem a um determinado critério.

$G_1 \nabla_{cond} G_2$: a junção de dois grafos nos vértices que atendem uma condição de junção.

Unificação

O operador unificação gera um novo grafo a partir de dois grafos, de forma que os vértices que atendem a um determinado critério são unificados. A unificação de dois vértices elimina um deles, e faz incidir no outro todas as arestas que incidiam no vértice eliminado. Após a unificação, as duplicatas são eliminadas.

$G_1 \Psi_{cond} G_2$: a unificação de dois grafos nos vértices que atendem uma condição de junção.

Correspondência

A operação de correspondência, assim como a seleção, utiliza um padrão de grafo e um grafo, mas produz uma coleção dos mapeamentos bem-sucedidos do padrão. Um mapeamento é um conjunto de pares (x, x') para todo x $\in$ P, onde x' $\in$ G é o seu elemento correspondente.

K_P (G) : os mapeamentos do padrão P no grafo G.

Projeção

A operação de projeção recebe como parâmetros uma coleção de mapeamentos e uma lista de "termos", podendo um termo designar um elemento ou um atributo de um elemento, e produz uma lista de tuplas com os valores solicitados.

Π_{termos} (M) : valores dos elementos dos mapeamentos.

Busca de caminho mínimo

Esta operação busca o caminho de menor comprimento (número de arestas) entre dois vértices. O resultado é uma coleção ordenada dos vértices que compõem o caminho do vértice de origem até o destino.

$\beta_{id1,id2}$ (G) : o caminho de menor comprimento entre o vértice origem (de identificador id1) e o destino (de identificador id2).

3.4 Exemplos de consultas na álgebra

Esta seção destina-se a comparar a álgebra proposta com a álgebra relacional, utilizando os mesmos exemplos de consulta encontrados em [19].

Para melhorar a legibilidade, os exemplos aplicam uma operação por vez, criando resultados nomeados intermediários.

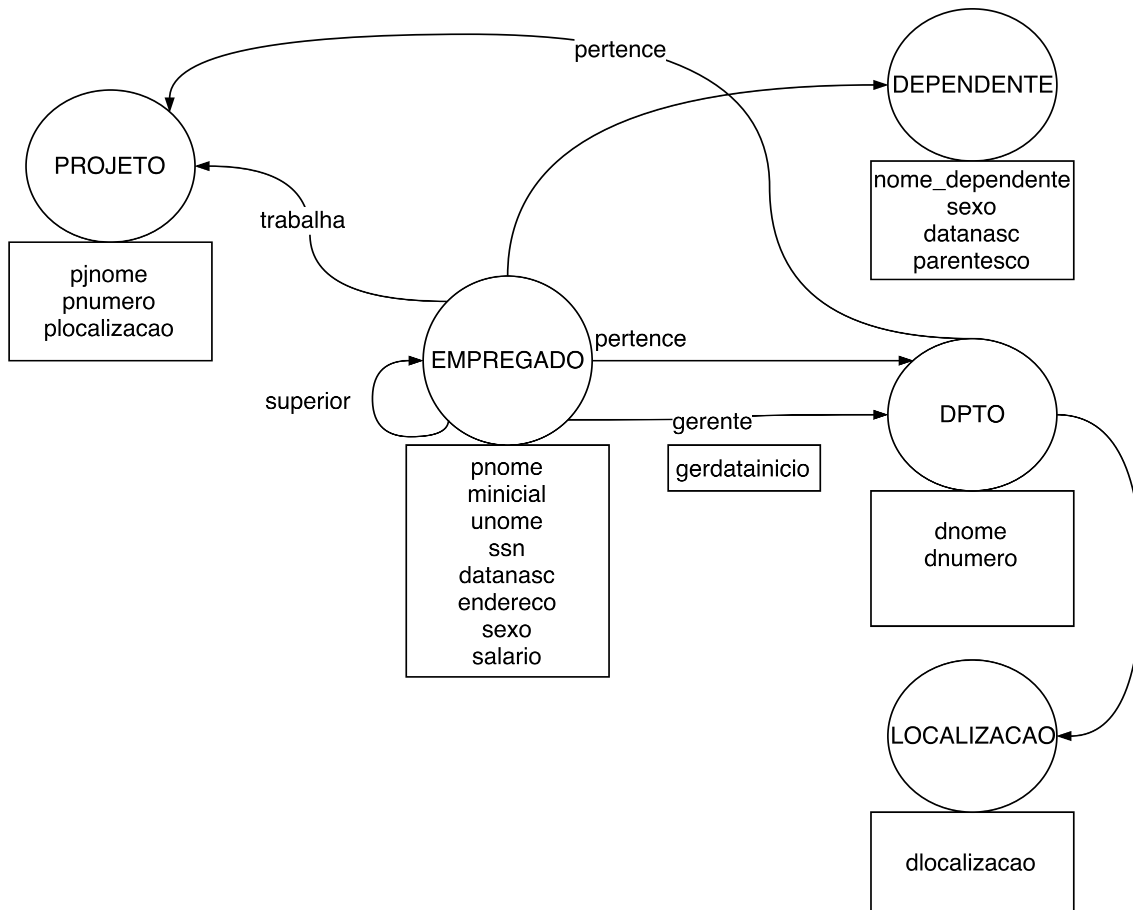


Figura 3.3: grafo exemplo de consultas na álgebra

3.4.1 Grafo exemplo

Para ilustrar a aplicação da álgebra, serão exemplificadas consultas sobre o grafo da figura 3.3. Nesta figura os vértices são representados pelos círculos, as arestas pelas setas, e as propriedades dos elementos pelos quadrados. Os rótulos dos vértices estão dentro dos círculos, e os das arestas próximos às mesmas.

As consultas são as mesmas encontradas em [19], e o grafo foi modelado a partir do banco de dados utilizado em [19] também, de forma a permitir que seja feito um paralelo com a álgebra relacional.

Os rótulos dos elementos são abreviados nas consultas por conveniência.

3.4.2 Consulta 1

Recuperar o nome e endereço dos empregados do departamento de “Pesquisa”.

$P \leftarrow e, d, p(e,$ 'pertence' $, d) | e \in EMP, d \in DPTO, d.dnome =$ 'Pesquisa'

$\Pi_{e.nome, e.endereco}\ (K_P(\text{G}))$

```
match (e:EMP)-[:pertence]->(d:DPTO)
where d.dnome = 'Pesquisa'
return e.pnome, e.endereco;
```

3.4.3 Consulta 2

Listar o número do projeto, o número do departamento de controle, e o último nome do gerente do departamento, de todos os projetos localizados em "Stafford".

$P \leftarrow g, d, p, (g,$ 'gerente' $, d), (p,$ 'pertence' $, d) | g \in EMP, d \in DPTO, p \in PRJ, p.plocalizacao =$ 'Stafford'

$\Pi_{p.pnumero, d.dnumero, g.unome}\ (K_P\ (\text{G}))$

```
match (p:PRJ)<-[:pertence]-(d:DPTO)<-[:gerente]-(g:EMP)
where p.plocalizacao = 'Stafford'
return p.pnumero, d.dnum, g.unome;
```

3.4.4 Consulta 3

Encontrar os nomes dos empregados que trabalham em todos os projetos controlados pelo departamento número 5.

$P \leftarrow e, p, d, (e,$ 'trabalha' $, p), (p,$ 'pertence' $, d) | e \in EMP, p \in PRJ, d \in DPTO, d.dnum = 5$

$\Pi_{e.pnome}\ (K_P\ (\text{G}))$

```
match (e:EMP)-[:trabalha]->(p:PRJ)<-[:pertence]-(d:DPTO)
where d.dnum = 5
return e.pnome;
```

3.4.5 Consulta 4

Fazer uma lista dos números dos projetos que envolvam um empregado cujo último nome seja “Smith”, seja ele gerente ou não.

$P \leftarrow e, p, (e, \text{'trabalha'}, p) | e \in EMP, p \in PRJ, e.unome = \text{'Smith'}$

$\Pi_{p.pnumero}\ (K_P\ (\text{G}))$

```
match (e:EMP)-[:trabalha]->(p:PRJ)
where e.unome = 'Smith'
return p.pnumero;
```

3.4.6 Consulta 5

Listar os empregados com dois ou mais dependentes.

$P \leftarrow d1, e, d2, (e, d1), (e, d2) | e \in EMP, \{d1, d2\} \in DEPEND$

$\Pi_{e.pnome}\ (K_P\ (\text{G}))$

```
match (d1:DEPEND)<--(e:EMP)-->(d2:DEPEND)
return e.pnome;
```

3.4.7 Consulta 6

Recuperar os empregados que não tenham dependentes.

dependentes $\leftarrow \sigma_{e,d,(e,d)|e \in EMP, d \in DEPEND}\ (\text{G})$

semdependentes $\leftarrow \sigma_{(e:Empregado)}\ (\text{G})$ - dependentes

$\Pi_{e.nome}(\ K_{(e:Empregado)(semdependentes)})$

```
match (e1:EMP) SUBTRACT (e2:EMP)-->(d:DEPEND)
return e1.nome;
```

3.4.8 Consulta 7

Listar os nomes dos gerentes que tenham, pelo menos, um dependente.

$P \leftarrow d, e, dpt, (e, d), (e, \text{'gerente'}, dtp) | e \in EMP, d \in DPTO, dpt \in DEPEND$

$\Pi_{e.pnome}\ (K_P)\ (\mathrm{G})$

```
match (d1:DEPEND)<--(e:EMP)-[:GERENTE]->(dpt:DPTO)
return e.pnome;
```

3.5 Sintaxe da MogwaiQL

Angles [11] argumenta que uma dificuldade no estudo das linguagens de consulta é a quase inexistência de definições formais das linguagens existentes. A linguagem MogwaiQL é definida formalmente utilizando a gramática da ferramenta ANTLR (apêndice A). Nesta seção é apresentada a sintaxe da MogwaiQL de forma mais legível.

A sintaxe básica para uma operação de seleção inicia com a palavra reservada "select" seguida por um padrão de grafo, e termina com um ponto e vírgula:

```
select padraoDeGrafo ;
```

O padrão de grafo é composto por um grafo descrito com arte ASCII e uma cláusula "*where*" onde são especificadas as restrições nos atributos dos elementos. Os elementos do grafo ASCII podem ser nomeados (para serem referenciados na cláusula *where*), e podem conter restrições de identificador, rótulo e grafo. Os vértices são especificados com parênteses, e suas restrições são especificadas entre os mesmos. As arestas são especificadas com "setas" e devem unir dois vértices. As arestas são direcionadas. Devido à restrição imposta pela forma textual, as diversas partes de um grafo são especificadas separadas por vírgula. As restrições das arestas são especificadas entre colchetes. A cláusula *where* é formada por expressões conjuntivas apenas. A primeira parte de uma expressão deve ser um atributo de um elemento nomeado. Segue-se então o operador, e a segunda parte da expressão pode ser um outro atributo, ou um valor fixo, como um número ou uma *string*.

Sintaxe dos elementos:

```
sintaxe do vértice: (nome :rótulo #idVertice $iGrafo)
sintaxe da aresta: -[nome :rótulo #idAresta $idGrafo]->
```

Sintaxe da cláusula *where*:

```
where nome.atributo operador (valor | propriedade)
    (AND nome.atributo operador (valor | propriedade))
```

A sintaxe das operações de união, subtração, interseção e unificação é similar à da operação de seleção, distinguindo-se por especificar as operações entre padrões de grafos:

```
União: select padraoDeGrafo UNION padraoDeGrafo ;

Subtração: select padraoDeGrafo SUBTRACT padraoDeGrafo ;

Interseção: select padraoDeGrafo INTERSECT padraoDeGrafo ;

Unificação: select padraoDeGrafo UNIFY padraoDeGrafo ON
    expressaoUnificacao ;
```

A operação de correspondência também é similar à seleção, distinguindo-se por utilizar a palavra reservada match:

```
Correspondência: match padraoDeGrafo ;
```

A operação de projeção é aplicada sobre uma operação de correspondência:

```
match padraoDeGrafo return nome.atributo (, nome.atributo)*
```

A sintaxe para a busca de caminho mínimo é composta por palavras reservadas e pelos identificadores dos elementos:

```
find path between idVertice1 AND idVertice2 on graph idGrafo
```

Os valores e os identificadores dos elementos são representados de forma que as *strings* são informadas entre aspas simples, e os números inteiros são informados

naturalmente. Para especificar números com outros tipos de dados, eles devem ser seguidos por uma letra qualificadora (D para *double*, L para *long*, e assim por diante).

3.6 Descrição da Arquitetura

A figura 3.5 ilustra a arquitetura básica do arcabouço. O usuário interage com o SGBDG através da linguagem de consulta ou da API. A API expõe o modelo de dados de grafo, quer dizer, ela define os métodos *CRUD* para manipular e consultar o banco de dados, como pode ser visto na figura 3.4. Além das operações sobre grafos, a API também possui métodos para fazer uso de transações e índices, para executar consultas, e inicializar e terminar o SGBDG. Na figura 3.5, o driver é o responsável por traduzir as operações da API em comandos para o SGBD. O SGBD é responsável pelo controle de concorrência e transacional, e pela persistência dos dados. O driver também traduz as operações com os índices, de forma que não necessariamente isto fica a cargo do SGBD.

O aspecto mais relevante desta arquitetura é que o Mogwai se abstrai de como suas operações são implementadas: o driver e o SGBD são elementos que podem ser trocados. Neste trabalho são desenvolvidos quatro drivers (implementações) distintos. Um driver para trabalhar com o SGBDG Neo4j, um com o SGBDO ObjectDB, um que utiliza a API JPA, e o último trabalha com a API JDBC do Java.

A arquitetura final é determinada pelas características peculiares de cada implementação, podendo assumir diferentes formas, conforme ilustra a figura 3.6. Por exemplo, a implementação Java Persistence API (JPA) pode executar em conjunto com o SGBD H2[38] tanto no modo *embedded* quanto no modo cliente/servidor, ao passo que as implementações Neo4J[17] e ObjectDB[37] podem inclusive trabalhar com configurações de alta disponibilidade, com múltiplos servidores em *cluster*.

Alguém que tenha interesse em prover uma implementação do Mogwai necessita apenas implementar as interfaces definidas na API. O conjunto de testes, um dos componentes da solução proposta, averigua se a implementação se comporta como

esperado. Um nova implementação do arcabouço deveria, a priori, interpretar e executar a MogwaiQL por conta própria. Porém, a biblioteca Java do Mogwai provê um conjunto de classes que interpreta e executa a MogwaiQL sobre a interface da API, fazendo com que este trabalho seja desnecessário. Por este motivo é que a API necessita de métodos para utilizar diretamente os índices. Os índices são utilizados na busca de correspondência de padrões de grafos. Ao mesmo tempo, a tarefa de interpretar e executar a MogwaiQL é facilitada por esta ter sido especificada com a ferramenta ANTLR e gozar das suas facilidades.

A busca por padrões utiliza os índices para acelerar o desempenho de sua execução. Primeiramente, o algoritmo verifica a cardinalidade das restrições dos elementos do padrão de grafo. Se algum elemento tiver cardinalidade zero, é possível determinar a condição de parada do algoritmo, pois se não há nenhum elemento que satisfaça a uma determinada restrição, nenhum padrão correspondente será encontrado. Com estas cardinalidades o algoritmo determina a ordem em que o padrão de grafo será percorrido, incorrendo em menos buscas desnecessárias. Por fim, o algoritmo seleciona os elementos do espaço inicial de busca a partir do elemento do padrão de grafo que possui a restrição de menor cardinalidade.

O arcabouço foi projetado para permitir que cada implementação tenha sua própria estrutura interna, utilizando suas próprias estruturas de dados e algoritmos, mas que também possa optar por utilizar os recursos já desenvolvidos.

3.7 Detalhes de implementação

O arcabouço Mogwai optou por trabalhar com algumas características que impõem determinados problemas de implementação. Em primeiro lugar, o arcabouço trabalha com o conceito de *schema-free*. Quer dizer, no Mogwai não há mecanismo para descrever a estrutura do banco de dados. A única definição possível é especificar quais propriedades dos elementos serão indexadas. Não é feito nenhum tipo de validação na estrutura do grafo, como, por exemplo, se um vértice de um determinado rótulo pode possuir determinada propriedade, ou se ele pode ter arestas de um

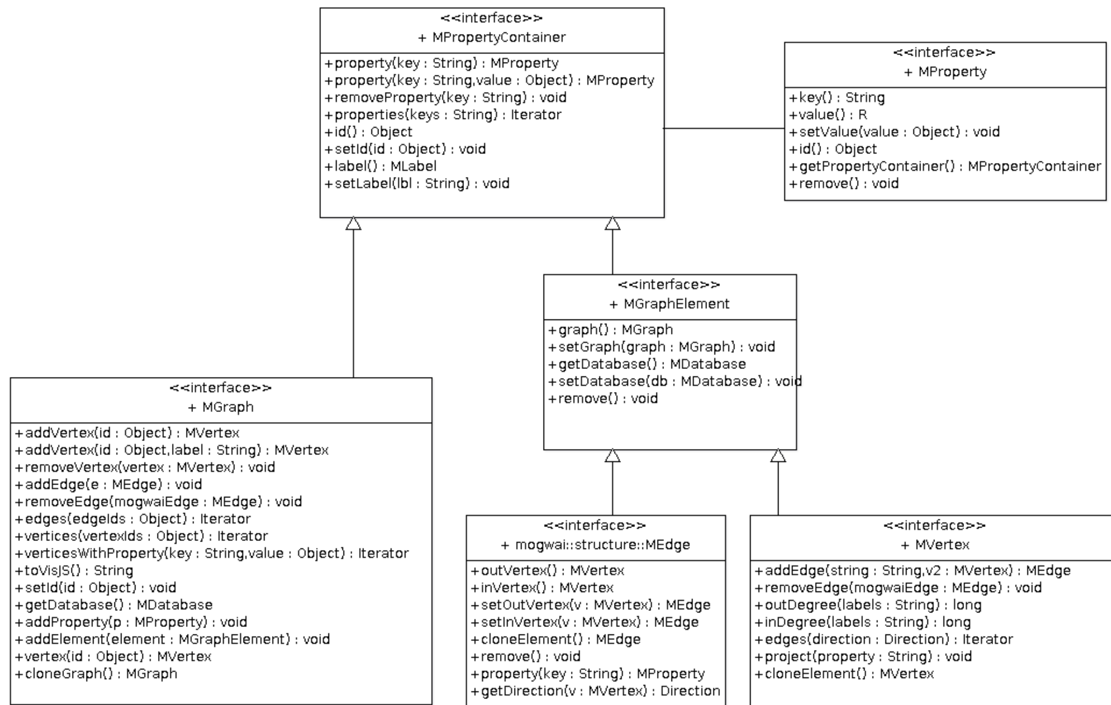


Figura 3.4: diagrama de classes

determinado rótulo com vértices de outro rótulo qualquer.

Outra característica do Mogwai é que os elementos podem ter identificadores de qualquer tipo, a única restrição é que eles sejam únicos dentro de um mesmo grafo.

3.7.1 API

O arcabouço Mogwai é composto por diversas classes e interfaces, das quais as mais importantes são aquelas que tratam da interação com o SGBDG e expõem o modelo de dados.

A interface *MDatabase* é o ponto de contato inicial com o SGBDG. Ela define os métodos para incluir e recuperar grafos, e os métodos para executar consultas, entre outros. A interação com os grafos é feita através da interface *MGraph*, que permite a adição, recuperação e remoção de vértices e arestas. Os vértices e as arestas são manipulados pelas interfaces *MVertex* e *MEdge*, respectivamente. O grafo e seus elementos estendem direta ou indiretamente a interface *MPropertyContainer*, que, além de definir operações para adicionar e remover propriedades, também define

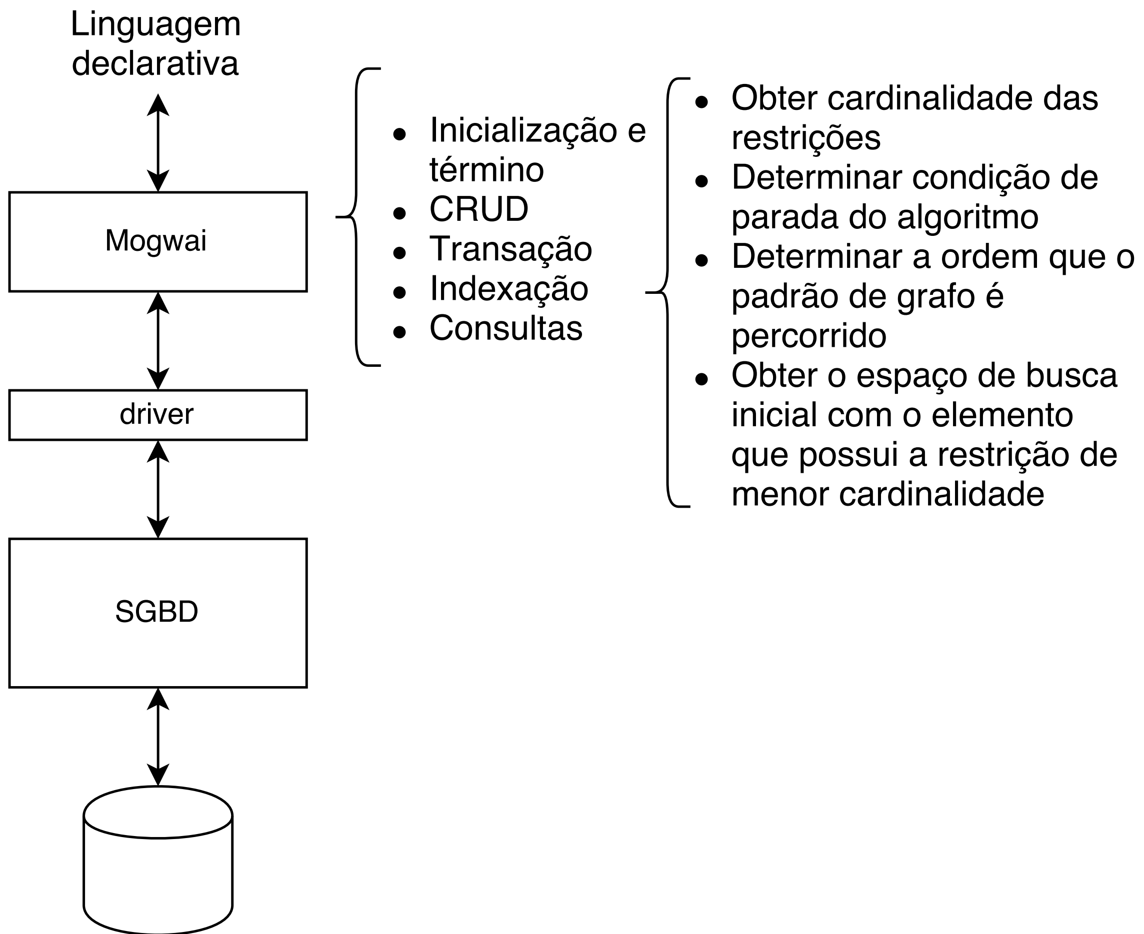


Figura 3.5: diagrama de arquitetura

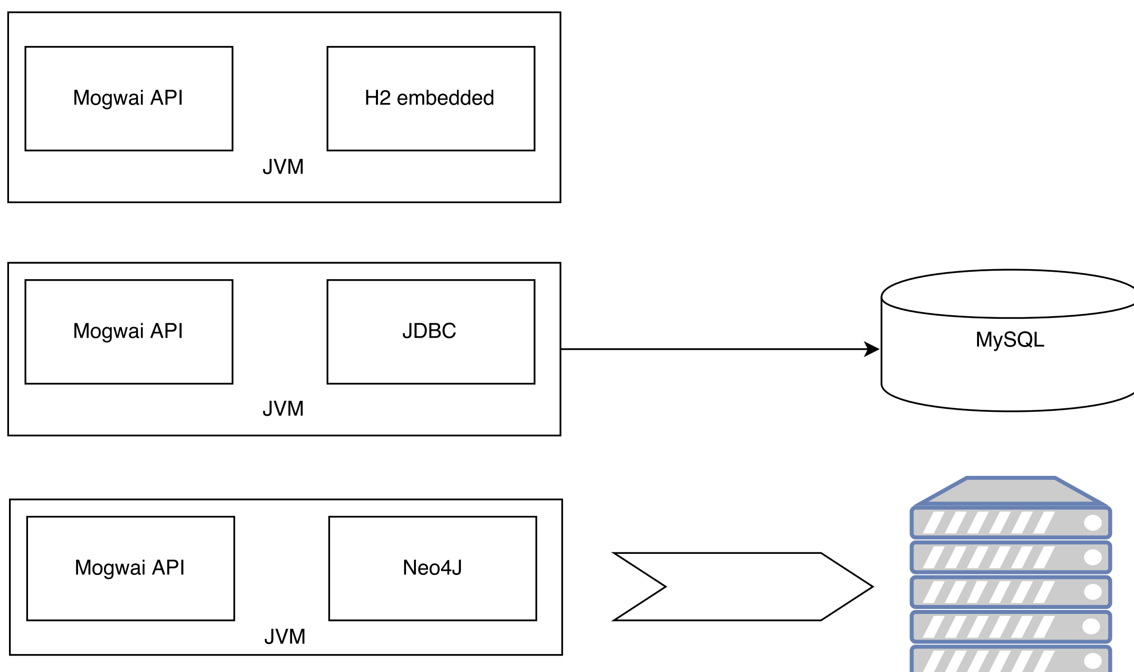


Figura 3.6: diagrama de possíveis arquiteturas

operações para definir e recuperar rótulos e identificadores.

O arcabouço foi desenhado para ser utilizado em contextos transacionais. Isto é, toda interação com a API afeta diretamente o banco de dados, e é feita *online*. A única parte que é desconectada é a execução de consultas. Além disto, para interagir com o banco de dados é necessário iniciar uma transação, e no fim da interação é necessário informar o status da transação para fechá-la. O comportamento padrão caso a aplicação cliente não informe o sucesso ou fracasso da transação é de simplesmente levantar uma exceção, o que gera problemas na gestão de recursos transacionais, mas facilita a captura de erros de programação. Este comportamento pode ser alterado pelo usuário para que seja realizado por padrão tanto o *rollback* quanto o *commit*.

3.7.2 Gerenciando um banco de dados de múltiplos grafos utilizando o Mogwai

A grosso modo, para gerenciar um banco de dados de múltiplos grafos utilizando o Mogwai basta implementar a interface *MDatabase*, pois ao implementar esta interface, naturalmente a implementação das demais classes da API se fará necessária.

Como forma de facilitar o trabalho de implementar a interface Mogwai, a biblioteca provê algumas classes abstratas que já realizam parte do trabalho necessário. Duas classes merecem atenção especial. A classe *AbstractMDatabase* fornece um esqueleto para a execução das operações de persistência de dados de tal forma a ajudar no controle de abertura e fechamento de transações. Uma limitação desta classe é que somente pode ser aberta uma conexão por *thread*. A outra classe que merece atenção é *MSession* que provê a funcionalidade de cache de grafos, arestas e vértices. O cache é realizado com escopo de *thread*.

3.7.3 Compilador Mogwai

O compilador Mogwai não traduz uma consulta diretamente em comandos para percorrer o banco de dados, mas sim cria uma representação em memória da con-

sulta a ser executada. Esta representação intermediária é o plano de execução da consulta, composta pelos seguintes operadores: *GraphOperationNode* é responsável por executar as operações entre grafos (união, intersecção, diferença, unificação e junção), *FindNode* é responsável por executar as consultas de caminhos, e por fim *MatchNode* é responsável por executar as buscas por correspondências de padrões de grafos. A implementação atual gera apenas um plano de execução de consulta.

Um padrão de grafo especificado numa consulta é traduzido para uma estrutura de dados de grafo em memória. O grafo possui a mesma estrutura que o padrão de grafo. Os identificadores dos elementos especificados na consulta são utilizados, e são atribuídos identificadores para os elementos anônimos. Os predicados nos elementos são traduzidos para objetos de classes que representam estes predicados. Cada elemento de grafo possui uma propriedade "$RESTRICTION_LIST" que mantém uma lista dos predicados sobre o elemento. No caso de expressões de operadores binários entre dois elementos, o predicado é armazenado em ambos os elementos.

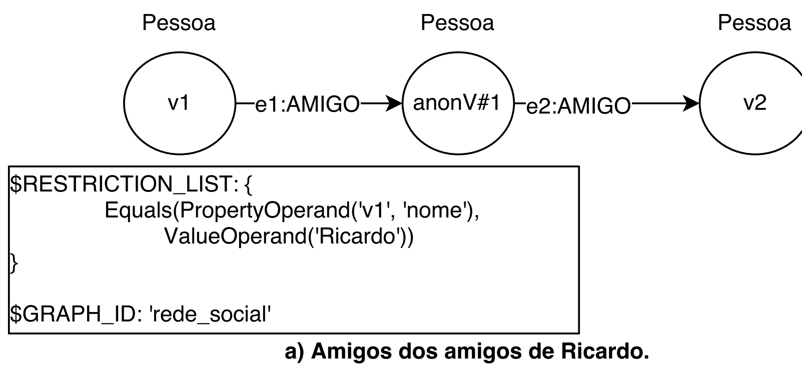
A figura 3.7 ilustra como são representados em memória os padrões de grafos das seguintes consultas:

a) match (v1:Pessoa #rede_social)-[e1:AMIGO]->(:Pessoa)-[e2:AMIGO]->(v2:Pessoa) where v1.nome = 'Ricardo';

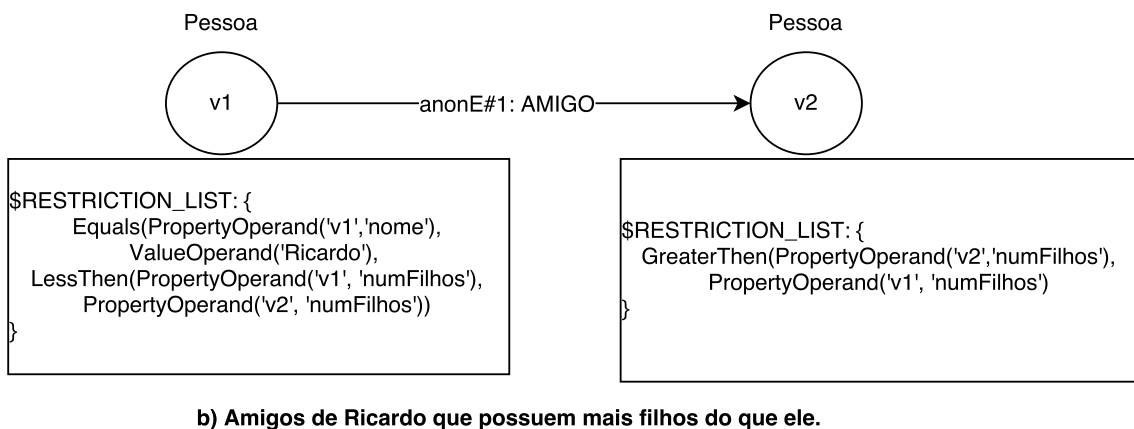
b) match (v1:Pessoa)-[:AMIGO]->(v2:Pessoa) where v1.nome = 'Ricardo'AND v1.numFilhos < v2.numFilhos;

O arcabouço utiliza a ferramenta ANTLR para gerar automaticamente um *parser* recursivo descendente a partir da gramática da linguagem MogwaiQL. Em outras palavras, o *parser* é constituído de uma coleção de métodos recursivos (um para cada regra), e o termo descendente refere-se ao fato de que o processo de *parse* começa na raiz da árvore e prossegue até os nós (*tokens*).

O *parser* gerado pelo ANTLR reconhece a linguagem MogwaiQL e gera uma árvore de sintaxe, que, ao ser percorrida, dispara eventos que são capturados por uma interface também gerada pelo ANTLR. O processo de compilação, portanto, consiste em gerar uma árvore de sintaxe e percorrê-la com um *listener* (padrão de



a) Amigos dos amigos de Ricardo.



b) Amigos de Ricardo que possuem mais filhos do que ele.

Figura 3.7: representação dos padrões de grafo

projeto *visitor*). O compilador Mogwai é uma implementação do *listener* gerado pelo ANTLR, e constrói um plano de execução de consulta de acordo com a consulta do usuário. Quando executado, o plano é traduzido para operações da API Mogwai, e estas são repassadas para o SGBD.

As classes geradas pelo ANTLR são as seguintes: a classe *MogwaiParser* contém a definição do parser que reconhece a sintaxe da linguagem MogwaiQL, ela contém um método para cada regra da gramática, e códigos de apoio; *MogwaiLexer* contém a definição do analisador léxico; *MogwaiListener* define a interface dos eventos que são disparados pelo ANTLR ao percorrer a árvore de sintaxe; *MogwaiBaseListener* provê uma implementação vazia do *MogwaiListener* para conveniência; *ANTLRExceptionListener* escuta os eventos de erro. Por fim, além dessas classes, a ferramenta gera os arquivos *Mogwai.tokens* e *MogwaiLexer.tokens*, necessários no processo de *parse*.

A classe *MQueryPlanner* estende a classe para *MogwaiBaseListener* e é quem implementa o compilador Mogwai de fato. Ao fim do processo de *parse*, esta classe disponibiliza um objeto do tipo *MQueryTree*, que é uma árvore de objetos do tipo *MQueryNode*. Existem quatro tipos de *MQueryNode*: *FindNode* executa operações de busca de caminho mínimo, *MatchNode* e *GraphPatternNode* executam operações de correspondência de padrões de grafos, porém um retorna um grafo como resultado, e o outro uma lista de correspondências; e, por fim, *GraphOperationNode* executa operações entre grafos (união, diferença, etc).

A classe *MQueryParser* implementa uma fachada para todo o processo de compilação. Ela se encarrega de invocar o *parser* Mogwai com seu analisador léxico para produzir a árvore de sintaxe, e em seguida percorre a árvore com *MQueryPlanner* produzindo o plano de execução de consulta *MQueryTree*, conforme ilustra o diagrama de sequência 3.8.

A classe abstrata *MQueryEngine* define métodos para executar a consulta de fato utilizando uma *MQueryTree*. Cabe a cada implementação executar da melhor maneira possível o plano inicial de execução de consulta. A biblioteca Mogwai

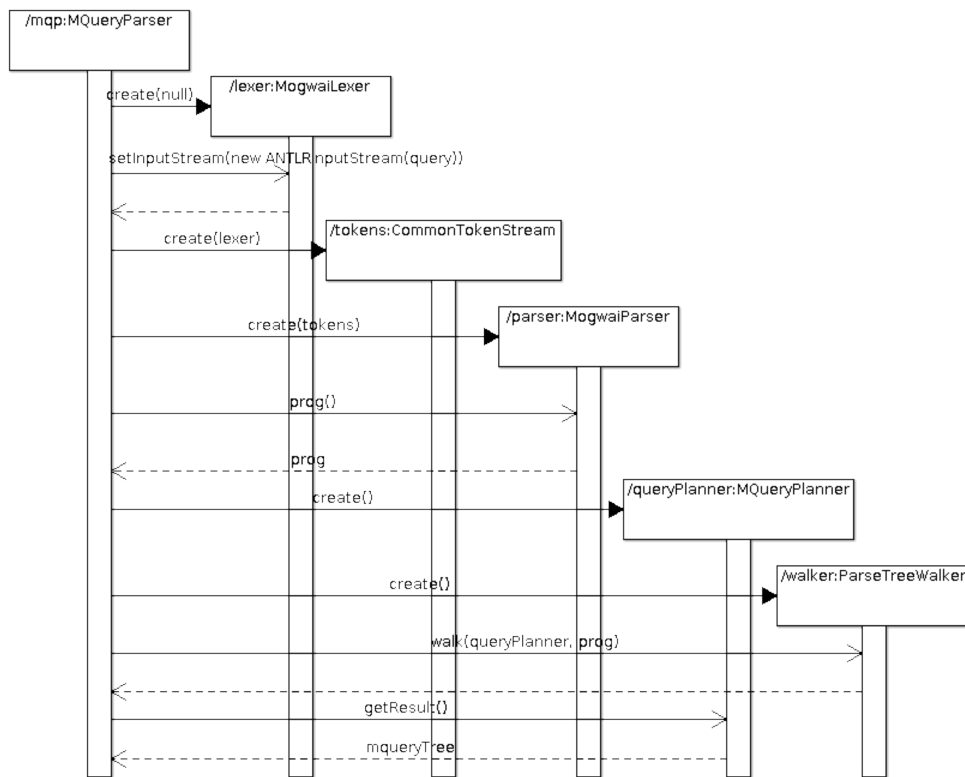


Figura 3.8: diagrama de sequência para obter plano de execução de consulta

oferece uma implementação que simplesmente executa o plano percorrendo-o de maneira *depth-first*.

3.7.4 JPA

Esta implementação utiliza a JPA para fazer a persistência dos dados. A utilização da JPA consiste em anotar as classes Java com metadados sobre os atributos, identificadores e relacionamentos a serem persistidos. Esse processo é denominado de mapeamento, e, quando a tecnologia envolvida é um SGBDR, os metadados incluem informações como nomes de tabelas, tipos e tamanhos de colunas, por exemplo. Neste caso, o processo é, então, denominado mapeamento objeto-relacional.

Uma vez feito o mapeamento, cabe à implementação do JPA traduzir os objetos Java em comandos para o SGBD utilizado. Ao aderir à especificação JPA torna-se possível utilizar uma ampla gama de SGBD compatíveis.

Esta implementação do Mogwai é acometida pelo problema de que as diferentes implementações da JPA não funcionam exatamente da mesma forma. Mais especificamente, determinados mapeamentos das entidades funcionam corretamente quando utilizados com um banco de dados orientado a objetos, mas não com um SGBDR.

Visando um funcionamento mais amplo, as entidades foram remodeladas de forma a atender pelo menos três mecanismos de persistência diferentes: H2 Database [38], MySQL [39], e ObjectDB [37]. A modelagem JPA quebrou a implementação da entidade *MProperty* em três classes: uma classe para propriedades de grafos, uma para propriedades de vértices, e uma para propriedades de arestas.

Nesta implementação, cada uma das entidades (grafo, vértice, aresta, propriedade de grafo, propriedade de vértice, e propriedade de aresta) é mapeada para uma tabela correspondente, no caso da JPA ser utilizada com um SGBDR.

Como erros de mapeamento são difíceis de serem detectados e corrigidos (pois são percebidos apenas em tempo de execução), optou-se por utilizar um modelo de domínio anêmico. Isto é, as classes das entidades não possuem lógica de negócio implementada, sendo meras estruturas de dados. A lógica de negócio necessária

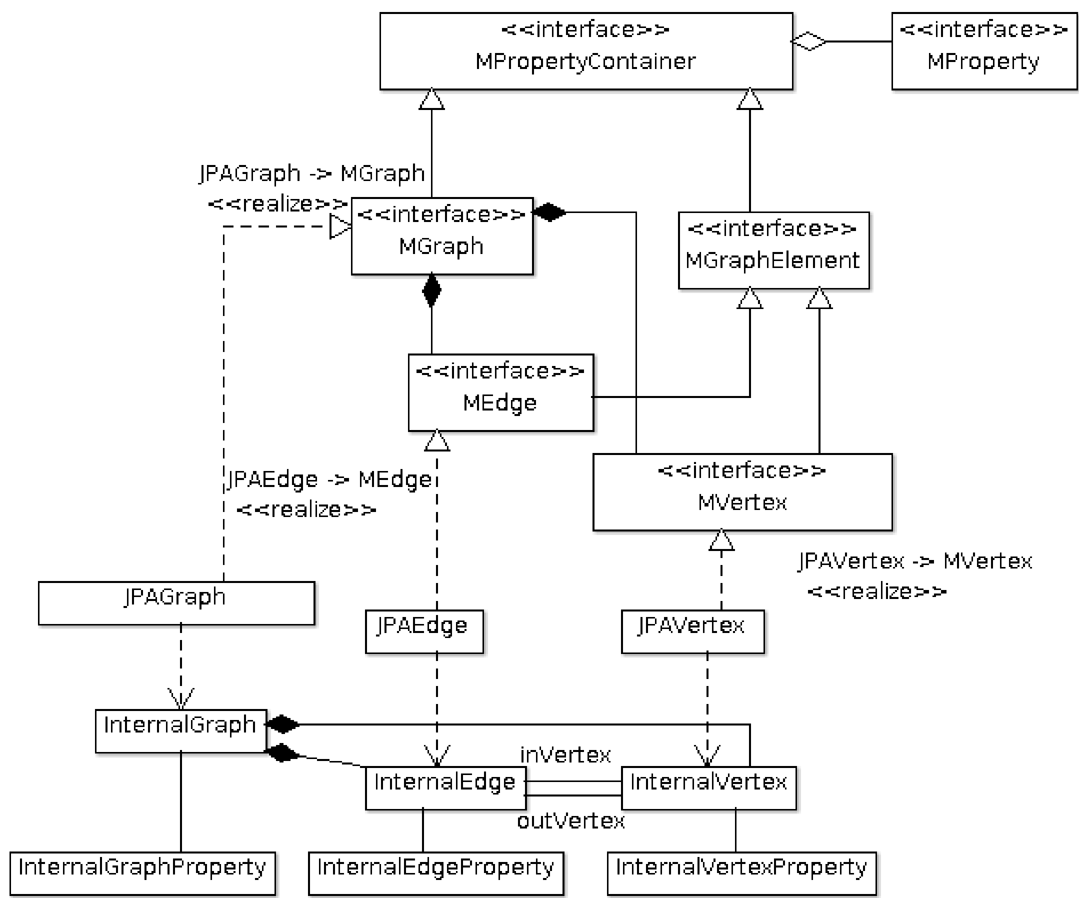


Figura 3.9: diagrama de classes JPA

para implementar a API ficou num conjunto de classes separado, conforme ilustra a figura 3.9.

Dado que a JPA não define uma interface para definição de índices, esta implementação faz uso da implementação Apache Lucene (Lucene) [40] provida pela biblioteca Mogwai como forma de garantir que qualquer implementação JPA se beneficiará do uso de indexação.

Para implementar os identificadores de tipos variáveis optou-se por armazená-los como *strings* no formato JavaScript Object Notation (JSON) , juntamente com um número inteiro que especifica o seu tipo. Este inteiro é necessário para realizar a deserialização do objeto JSON para o tipo Java apropriado. Visto que os identificadores necessitam ser únicos apenas dentro de um mesmo grafo, a chave primária de um elemento do grafo é na verdade uma chave composta pelo identificador do elemento e o identificador do grafo.

De forma análoga, as propriedades dos elementos também podem ser de qualquer tipo, e também são armazenadas no formato JSON com o auxílio de um inteiro.

O vértices JPA mantém referências para as arestas de entrada e de saída, não sendo especificado se o relacionamento será carregado imediatamente ou tardiamente (*lazy loading*), ficando o comportamento a cargo da implementação subjacente.

As arestas são objetos que mantém relacionamentos para os vértices de entrada e de saída.

O mapeamento dos relacionamentos entre os vértices e o grafo, e entre as arestas e o grafo é unidirecional, de tal forma que ao carregar o grafo não sejam carregados todos os seus elementos automaticamente.

Esta implementação faz cache de objetos com escopo de transação.

3.7.5 JDBC

Esta implementação utiliza o mecanismo mais básico para acesso a bancos de dados oferecido pelo Java, o *Java Database Connectivity (JDBC) API* [41], e foi nomeada de acordo. Ao aderir à JDBC e utilizar apenas SQL ANSI, esta implementação

torna-se passível de ser utilizada com qualquer SGBDR compatível.

Assim como na implementação JPA, cada uma das entidades (grafo, vértice, aresta, e suas propriedades) é mapeada para uma tabela correspondente. Não se faz necessário, no entanto, representar as diferentes propriedades com diferentes classes, conforme ilustram as figuras 3.11 e 3.10.

Inicialmente não se fazia uso do Lucene, e pressupunha-se que seriam definidos índices nas tabelas do banco de dados relacional subjacente. Entretanto, em alguns casos de testes notou-se que o desempenho na inserção de dados ficava a desejar. O Mogwai permite que sejam especificadas quais propriedades devem ser indexadas. Mas, ao utilizar índices nas tabelas de propriedades, a implementação JDBC é incapaz de fazer essa distinção, e indexa todas as propriedades. E isto, provavelmente, estava causando o desempenho inferior na inserção - o que não ocorre ao se utilizar o Apache Lucene. Este problema poderia ter sido resolvido utilizando-se duas tabelas para cada tabela de propriedades, uma indexada e a outra não. Entretanto, empregar esta estratégia complicaria as consultas de recuperação de propriedades com o uso de junções e uniões, o que teoricamente também degradaria o desempenho. Neste cenário, a opção escolhida foi de manter a simplicidade das tabelas, e ter ambas as opções de utilizar o Lucene ou a indexação na tabela, conforme necessário. A indexação da tabela de propriedades também melhorou ao se indexar apenas os 30 primeiros caracteres, ao invés de utilizar indexação *full-text*.

As características do modelo de grafo de propriedades do Mogwai exigem uma modelagem diferente da abordagem entidade-relacionamento tradicional. O uso de qualquer tipo de dado como identificador é especialmente problemático, pois exige que a coluna do banco de dados armazene um tipo genérico, como, por exemplo, um objeto serializado em bytes ou JSON. Assim como outros SGBDR, o MySQL[39] possui restrição de tamanho de chave primária, de modo que a solução adotada para este problema foi a utilização de uma chave interna primária fictícia (*surrogate key*), que consiste apenas num número inteiro sequencial. O identificador do usuário é então armazenado como uma coluna à parte da chave primária, que deve ser utilizado

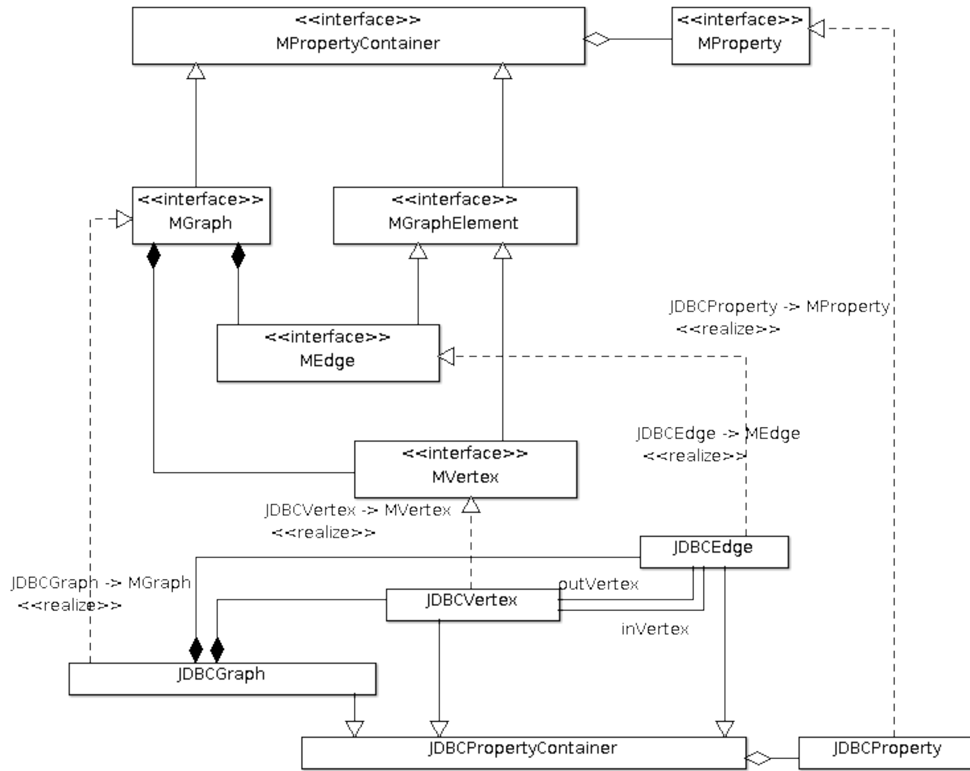


Figura 3.10: diagrama de classes JDBC

em conjunto com um índice de unicidade. Além do índice de unicidade, utiliza-se um *cache* em memória do mapeamento de identificadores do usuário e internos.

As figuras 3.9 e 3.11 ilustram a modelagem da implementação JDBC.

Apesar de as propriedades dos elementos estarem armazenadas em relações distintas, não há necessidade de refletir isso no modelo Java, de tal forma que qualquer tipo de propriedade utiliza a classe *JDBCProperty.* A classe *JDBCMDatabase* se encarrega de utilizar a relação apropriada para cada propriedade.

A tabela INDEXED_PROPERTY é utilizada para persistir o *schema* do Mogwai, quer dizer, mantém uma simples lista de quais propriedades devem ser indexadas.

A tabela SEQUENCE é utilizada para gerar identificadores únicos de forma independente do SGBDR utilizado.

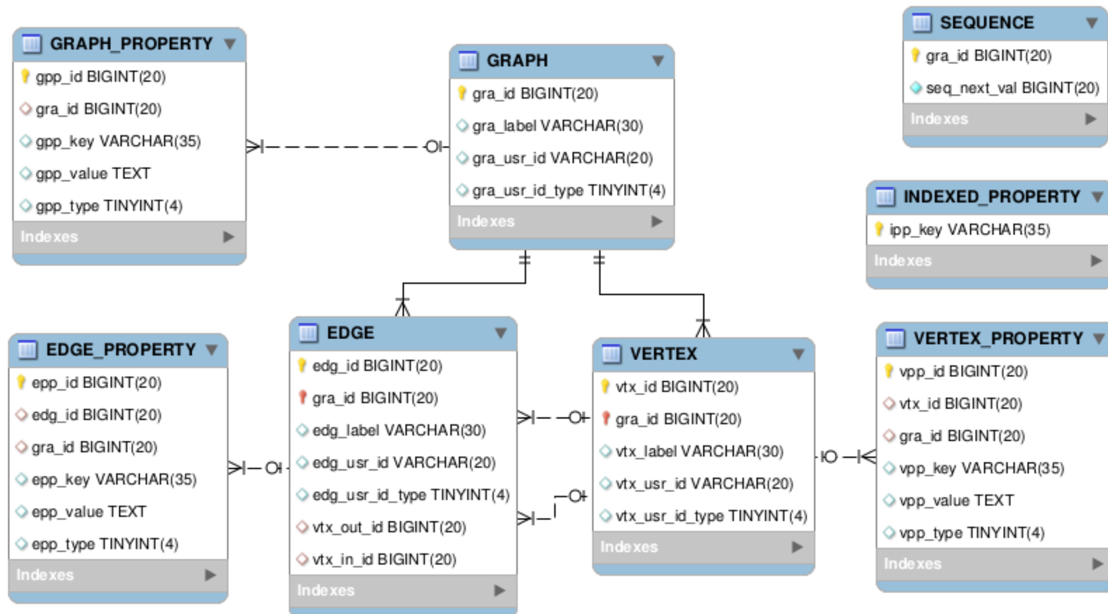


Figura 3.11: diagrama de entidades e relacionamentos

3.7.6 ObjectDB

A implementação ObjectDB[37] também é utilizada através da interface JPA , porém não foi necessário quebrar a implementação da *MProperty* em três, e nem utilizar o modelo de dados anêmico, ficando a lógica de negócio contida nas próprias entidades do modelo.

Como tanto a linguagem Java e o ObjectDB são orientados a objetos, não há impedância de correspondência. Desta forma, as entidades grafo, vértice, aresta e propriedade, que são representadas como classes em Java, são mapeadas para objetos correspondentes no banco de dados.

Apesar de ser possível definir índices na ferramenta, optou-se por utilizar o Lucene, pois o Lucene permite um controle mais fino da indexação que se mostrou necessário para ganhos de desempenho durante os testes.

Esta implementação não faz o próprio *cache* de objetos, ficando o mesmo a cargo exclusivo da implementação JPA do ObjectDB, sendo utilizada configuração padrão de tamanho de *cache*.

O ObjectDB oferece um recurso de aprimoramento de classes, que consiste em

uma ferramenta de pós compilação que melhora o desempenho através da modificação do *byte code* das classes. Este recurso é utilizado principalmente para as classes das entidades definidas pelo usuário que são persistidas. O aprimoramento melhora a eficiência de três maneiras.

Em primeiro lugar, o código aprimorado permite o rastreamento eficiente das modificações nos atributos persistentes, evitando a necessidade de comparações *snapshot*. Isto é feito pela adição de código nas classes aprimoradas que automaticamente notifica o ObjectDB toda vez que um campo persistente é modificado.

Em segundo lugar, o código aprimorado permite o carregamento tardio (*lazy loading*) dos objetos das entidades. Sem o aprimoramento, apenas os campos dos tipos coleções e mapas podem ser carregados tardiamente com o uso de objetos *proxy*. Os campos persistentes que referenciam objetos diretamente (relacionamentos um para um, como, por exemplo, entre a aresta e os vértices de entrada e de saída) têm que ser carregados imediatamente (*eagerly*).

Por fim, métodos otimizados especiais são adicionados às classes aprimoradas para serem utilizados pelo ObjectDB no lugar de reflexão. Segundo a documentação do ObjectDB, estes métodos executam muito mais rapidamente.

Este aprimoramento de pós compilação é realizado automaticamente ao compilar o projeto utilizando a ferramenta *Maven*.

A figura 3.12 ilustra a modelagem da implementação ObjectDB.

3.7.7 Neo4j

O Neo4j[17] utiliza o Lucene [40] internamente, e a implementação Neo4j do arcabouço Mogwai faz uso deste mecanismo através do envio de comandos de criação de índices.

O Neo4j é um SGBDG que utiliza o modelo de grafo de propriedades. Portanto, o mapeamento das entidades vértice, aresta, e propriedade é feito diretamente nas estruturas de dados do SGBDG. Apesar dessa relação direta, há necessidade de alterações devido ao fato que o Neo4j trabalha com apenas um grafo por base de

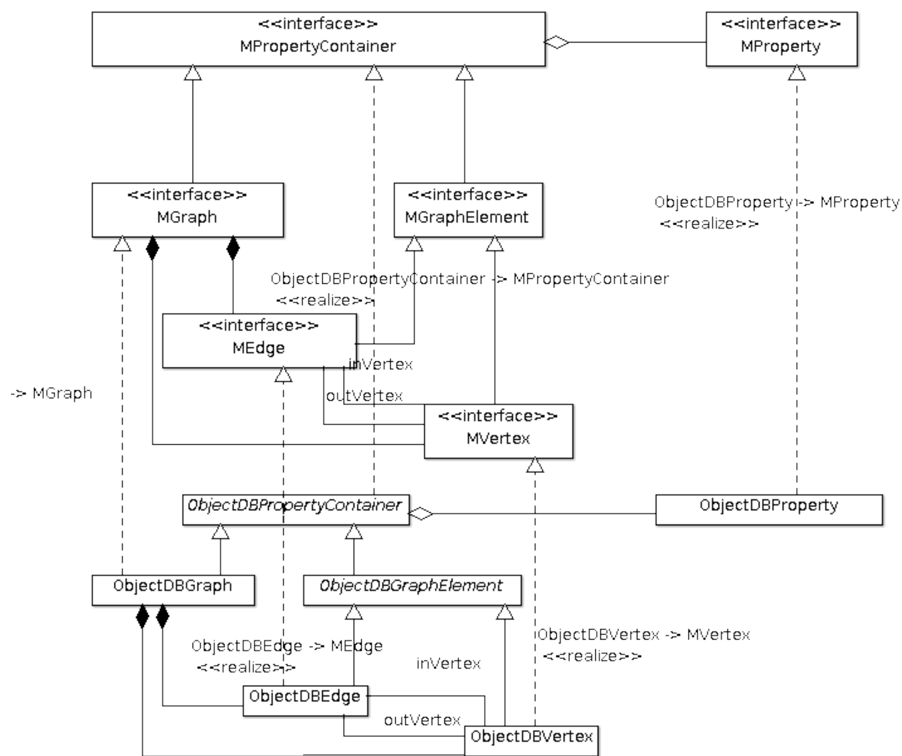


Figura 3.12: diagrama de classes ObjectDB

dados. A entidade grafo necessita de tratamento especial.

Problemas que envolvem múltiplos grafos necessitam trabalhar com uma modelagem alternativa à normal da ferramenta. Esta implementação do Mogwai visa prover isto de maneira transparente para o usuário, se encarregando de realizar esta modelagem.

O conceito de múltiplos grafos é implementado utilizando um recurso disponível a partir da versão 3.0 da ferramenta, que é a possibilidade de atribuir múltiplos rótulos para os vértices.

Uma entidade grafo é, então, representada por um vértice isolado com dois rótulos: o rótulo “MGRAPH” indica que o vértice representa um grafo, e o outro rótulo é o próprio identificador do grafo. Desta forma, os identificadores de grafos são limitados pelos valores válidos de rótulos do Neo4j. Uma entidade vértice, além do próprio rótulo, utiliza o rótulo identificador do grafo ao qual pertence. As arestas não possuem a possibilidade de possuírem múltiplos rótulos, mas, como não são permitidas arestas entre grafos distintos, a busca das arestas de um grafo é feita buscando-se as arestas que se originam em algum vértice deste grafo.

O Neo4j utiliza identificadores internos próprios, e não permite que o identificador de um elemento seja especificado pelo usuário. Para permitir tal funcionalidade, a solução é armazenar o identificadores dos elementos como propriedades dos mesmos. São utilizadas propriedades com nomes reservados para tal finalidade.

A execução de consultas pode ser realizada de duas maneiras. A primeira é utilizando o motor de execução de consultas do Mogwai, como fazem todas as outras implementações. E a segunda é através da tradução de uma consulta Mogwai para Cypher. A classe *MogwaiCypherCompiler* estende a *MogwaiBaseListener*, e compila uma consulta escrita em Mogwai para uma consulta Cypher. Desta forma é possível se beneficiar das otimizações internas do Neo4j.

A figura 3.13 ilustra a modelagem da implementação Neo4j.

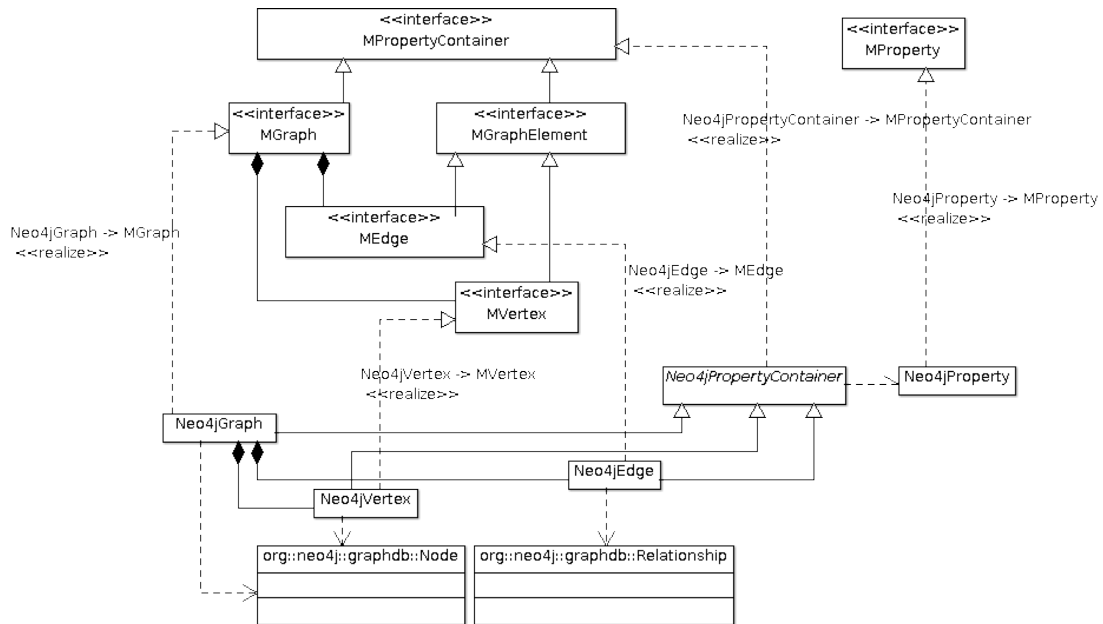


Figura 3.13: diagrama de classes Neo4j

3.7.8 Algoritmo de correspondência de padrões de grafos

A execução de uma consulta de correspondência de padrão de grafos (*graph pattern matching*) é uma operação extremamente custosa devido à natureza do problema cuja complexidade é NP. Para melhorar o desempenho dessa operação, o Mogwai se utiliza de mecanismos para diminuição do espaço de busca (poda, ou *prunning*).

O algoritmo básico para encontrar isomorfismos de subgrafos é descrito por Lee et al [42] da seguinte forma:

Algorithm 1 Procedimento genérico de consulta

1: **procedure** ProcGenericaConsulta(q, g)
2: $M \leftarrow \emptyset$
3: **for all** $u \in V(q)$ **do**
4: $C(u) \leftarrow FiltrarCandidatos(q, g, u, ...)$
5: **if** $C(u) = \emptyset$ **then**
6: **return**
7: **end if**
8: **end for**
9: BuscaSubgrafo(q, g, M, ...)
10: **end procedure**
11: **procedure** BuscaSubgrafo($q, g, M, ...$)
12: **if** $|M| = |V(q)|$ **then**
13: **report** M
14: **else**
15: $u \leftarrow$ ProxVerticeConsulta(...) $[[u \in V(q) \wedge \forall(u', v) \in M(u' \neq u)]]$
16: $C_R \leftarrow$ RefinarCandidatos(M, u, C(u), ...) $[[C_R \subseteq C(u)]]$
17: **for all** $v \in C_R$ tal que v ainda nao foi correspondido **do**
18: **if** PossivelJuntar(q, g, M, u, v, ...) **then**
19: $[[\forall(u', v') \in M(u, u') \in E(q) \implies (v, v') \in E(g) \subseteq L(u, u') = L(v, v'))]]$
20: AtualizarEstado(M, u, v, ...) $[[(u, v) \in M]]$
21: BuscaSubgrafo(q, g, M))
22: RestaurarEstado(M, u, v) $[[(u, v) \notin M]]$
23: **end if**
24: **end for**
25: **end if**
26: **end procedure**

O procedimento genérico de busca possui como parâmetros de entrada o grafo

q de consulta, e o g de dados. Este algoritmo se inicia fazendo uma busca pelos candidatos a cada um dos vértices do grafo de busca. Em outras palavras, para cada vértice u de q, o algoritmo busca um conjunto de vértices C(u) em g. O conjunto de vértices C(u) é formado pelos vértices que atendem as restrições do vértice u. Por exemplo, se o grafo de consulta especifica que u possui três arestas, C(u) será composto por vértices que possuem três ou mais arestas. Caso algum vértice não possua candidatos, o algoritmo pode responder que não há nenhum grafo isomórfico de q em g. Do contrário, o algoritmo inicia a busca pelos subgrafos.

O procedimento de busca por subgrafos é um procedimento recursivo. Ele utiliza a estrutura de dados M, que contém o mapeamento entre os vértices dos grafos q e g. A primeira instrução condicional deste procedimento verifica se encontrou-se uma solução. Chega-se a uma solução quando todos os vértices de q puderam ser mapeados com um vértice de g, neste caso, a solução é emitida.

O algoritmo genérico não entra nos detalhes técnicos de como realizar a ação de emitir uma solução, entre outros detalhes do algoritmo, o que pode ser notado, por exemplo, pela presença dos três pontos nos argumentos das funções.

A busca pelo padrão de grafo procede da seguinte forma. Escolhe-se um vértice u que ainda não tenha um mapeamento em M. Refina-se então a lista de candidatos para u. Para cada candidato v que ainda não foi escolhido, verifica-se se é possível juntar v ao mapeamento. Caso seja possível, v é adicionado ao mapeamento e inicia-se uma nova recursão da busca. No retorno desta recursão, v é removido do mapeamento, realizando o *backtracking*, para que um outro candidato v seja examinado.

A subrotina *PossivelJuntar* verifica se as arestas entre u e os outros vértices já mapeados possuem arestas correspondentes no grafo de dados. A subrotina *RefinarCandidatos* obtém um conjunto refinado de vértices que utiliza regras de poda específicos de um algoritmo não genérico.

O algoritmo desenvolvido para o arcabouço parte do algoritmo genérico básico e acrescenta algumas características próprias.

Em primeiro lugar, na operação inicial de filtrar os candidatos, o Mogwai, ao invés de obter as referências para todos os candidatos, ele apenas consulta a quantidade de vértices candidatos, e a partir desta quantidade estabelece uma ordem de busca para os elementos do grafo de consulta. Os vértices candidatos são determinados a partir das restrições existentes no padrão de consulta, como por exemplo, possuir em determinado rótulo, ou uma propriedade com um determinado valor.

A ordem de busca é determinada verificando as restrições impostas na consulta. Primeiramente, se o vértice possui uma restrição de identificador, esta restrição é utilizada. Do contrário, são verificadas as restrições de propriedades do vértice. Caso não haja tais restrições, verifica-se a existência de uma restrição de identificador de grafo. Não possuindo esta restrição, verifica-se o rótulo do mesmo. Por fim, verifica-se se alguma aresta adjacente possui alguma restrição. Apenas uma destas verificações é realizada para que este processo não seja demorado. É uma limitação da implementação atual que todas as restrições não possam ser verificadas. Os vértices são então ordenados pela ordem crescente da cardinalidade de seus candidatos. Espera-se desta forma, reduzir o espaço de busca.

Depois de estabelecida a ordem de busca, a busca no grafo inicia-se a partir de um vértice inicial e um conjunto de candidatos inicial; e prossegue razoavelmente da mesma forma que o algoritmo genérico, com algumas diferenças importantes. A primeira diferença notável é que as arestas podem ter rótulos e restrições de propriedade. Logo, não é possível realizar a busca apenas nos vértices, e induzir um subgrafo isomórfico a partir dos mesmos, mas sim é necessário verificar todos os elementos do grafo de consulta. Outra diferença notável é que no Mogwai é possível especificar caminhos, então as operações, como, por exemplo, *PossivelJuntar*, que determina se um elemento pode ser adicionado ao mapeamento M, precisam verificar tanto vértices, como arestas ou caminhos. Além destes fatores, o algoritmo deve lidar com as restrições comparativas entre os elementos do subgrafo. Estas comparações são realizadas assim que ambos elementos de uma restrição binária estiverem mapeados.

Considerando que o SGBDG não será capaz de manter todos os grafos em memória o tempo inteiro, e que, se tratando de grafos de propriedades, pode ser o caso que as propriedades dos elementos ocupem muito mais memória do que as informações sobre a estrutura do grafo, uma estratégia que pode se mostrar interessante é a de manter em memória apenas a estrutura dos grafos e deixar para recuperar as propriedades com carregamento tardio (*lazy loading*). A exploração do padrão de grafo poderia se dar então por passos de expansão baseados na estrutura do grafo, seguidos por passos de filtragem, baseados nas restrições de propriedades dos elementos. Os passos de filtragem se beneficiariam, então, da rápida recuperação dos valores das propriedades a partir dos identificadores dos elementos. Esta estratégia, no entanto, não foi realizada neste trabalho, ficando como um possível trabalho futuro.

3.8 Utilitários

3.8.1 Importação de bases de dados relacionais

Os SGBDR ainda são a solução padrão *de facto* para armazenamento persistente de bancos de dados. Porém, Martin Fowler [43] é da opinião de que há uma tendência de que qualquer empresa de tamanho razoável passará a ter uma variedade de tecnologias de armazenamento para diferentes tipos de dados. A utilização de diversas tecnologias de armazenamento de dados é denominada persistência poliglota. Para Martin Fowler, cada vez mais as pessoas vão se perguntar em como os dados serão manipulados antes de determinar qual é a tecnologia mais adequada para a tarefa. Na persistência poliglota, diferentes tecnologias são empregadas para resolver diferentes partes de uma aplicação complexa. Por exemplo, numa aplicação web hipotética de vendas, as sessões do usuário e o carrinho de compras utilizariam um banco de dados do tipo chave-valor, os dados financeiros e relatórios ficariam armazenados em um SGBDR, o catálogo de produto estaria num banco de dados orientado a documentos, e o sistema de recomendações utilizaria um SGBDG.

Nos cenários de utilização de múltiplos mecanismos de persistência de dados, eventualmente há a necessidade de migrar os dados de uma base para outra, de tal forma que uma ferramenta de importação de dados se faz útil. A biblioteca Mogwai provê uma ferramenta que realiza a importação de base de dados relacionais utilizando um mapeamento simples: cada entidade relacional é mapeada para um nó de um determinado rótulo, e os relacionamentos entre as entidades são mapeados para arestas de um determinado rótulo.

3.8.2 Importação de arquivos

A *Graph eXchange Language* (GXL) [44] é um formato de troca de dados baseado em XML para compartilhamento de dados entre ferramentas. Resumidamente, o modelo de dados é composto por grafos direcionados, onde os elementos do grafo podem possuir atributos tipados. Uma vez que o modelo de dados da GXL é mais restritivo que o utilizado pelo arcabouço, a importação dos dados é direta, sendo realizada de maneira simples. O procedimento de importação consiste em ler os arquivos XML para estruturas de dados em memória que refletem a estrutura definida no *Document Type Definition* (DTD) e em seguida executar os métodos de persistência definidos na API para cada uma das entidades.

A GXL não prevê a utilização de rótulos nos elementos, de tal forma que essa informação deve ser codificada de outra forma, por exemplo, como um atributo; e, nos grafos que não possuem rótulos, um rótulo padrão deve ser aplicado na sua importação.

3.8.3 Interface web para consultas

Finalizando os detalhes de implementação, apresenta-se a interface web para consultas. Apesar de ser apenas um protótipo bastante limitado, a interface web para consultas permite que o usuário faça consultas ao banco de dados sem a necessidade de escrever um programa.

A interface web para consultas provê um mecanismo para visualizar graficamente

Figura 3.14: Interface web para consultas

o resultado da execução das consultas. Trata-se de uma aplicação web composta por uma única página, conforme ilustra a figura 3.14. No topo da tela encontram-se a caixa de texto para o usuário informar a consulta na linguagem Mogwai e o botão para executá-la. Ao centro situa-se o grafo resultado, e na parte de baixo fica um painel para visualizar os detalhes de um elemento do grafo. Os detalhes de um elemento são exibidos quando o mesmo é clicado pelo usuário. Também é possível navegar pelo resultado arrastando o *canvas* para a direção desejada, e é possível aumentar ou diminuir o *zoom* utilizando o botão de rolagem do *mouse*.

A interface web propriamente dita é um arquivo html estático, que faz requisições *Ajax* para um *web service* no servidor web. A visualização do grafo é realizada utilizando a biblioteca *JavaScript* VisJS. Esta biblioteca trabalha com o conceito de um grafo único, então, antes de enviar o resultado em *JSON* é necessário trocar os identificadores dos elementos para serem uma composição do identificador do grafo com o identificador do elemento.

O SGBDG Neo4j oferece um console de administração acessado via navegador de Internet. Para acessá-lo é necessário iniciar o SGBDG no modo console, e abrir o navegador no endereço que o SGBDG informa. Ao ser iniciado no modo console, o Neo4j inicia automaticamente um servidor web, de forma que o usuário não necessita configurar um manualmente. O arcabouço Mogwai funciona de forma análoga. A interface web para consultas fica disponível ao executar a classe principal Java *br.ufrj.coppe.mogwai.Application*. Esta classe faz parte de um projeto *Maven* configurado para utilizar o *SpringBoot* [45]. O *SpringBoot* se encarrega de incluir e iniciar o servidor web automaticamente na aplicação.

Capítulo 4

Resultados e Discussões

Este capítulo discute os resultados da solução proposta nesta dissertação. Na primeira seção é realizada uma análise qualitativa do arcabouço Mogwai. A seção seguinte apresenta e discute em termos de desempenho os resultados experimentais obtidos com as implementações da API.

A análise de desempenho é dividida em duas baterias de testes. A primeira visa medir o desempenho numa base de múltiplos grafos onde os elementos dos grafos possuem atributos diversos, e a segunda visa medir o desempenho numa base de dados mais volumosa, composta por um único grafo que não possui atributos nos seus elementos.

É importante salientar que a análise de desempenho tem o objetivo primordial de verificar a aplicabilidade do arcabouço relativamente a um SGBDG nativo. Especificamente, o Neo4j, um SGBDG que implementa armazenamento e processamento de grafos nativo.

Nas implementações feitas, a MogwaiQL não é traduzida diretamente para uma única consulta no SGBD, mas sim para um conjunto de operações, excetuando-se quando se traduz a MogwaiQL para Cypher. O objetivo dos testes, então, é observar se a tradução da MogwaiQL em várias operações obtém desempenho compatível com a execução (quase direta) da Cypher no Neo4j. A influência do arcabouço é mínima neste último caso, e, portanto, o desempenho das consultas Cypher é utilizado como base de comparação.

Portanto, a comparação entre os SGBD é um aspecto secundário. Até porque as tecnologias não competem exatamente em pé de igualdade, por exemplo, enquanto algumas tecnologias executam no mesmo processo na Java Virtual Machine, outras se comunicam pela interface de rede. Ademais, o desempenho das implementações ainda depende de como cada driver foi implementado.

4.1 Comparação com outras API

Esta seção objetiva comparar o arcabouço Mogwai com outras duas API de grafos: a API do SGBDG Neo4j e a API do Apache Tinker Pop. Estas duas API foram escolhidas pois são os padrões *de facto* atualmente. A comparação é feita apenas sobre as interfaces que o usuário da API interage. É possível estabelecer um paralelo entre os elementos básicos, o que facilita a comparação.

Na API do Neo4, a abstração de mais alto nível é a interface *GraphDatabase-Service*. Ela é responsável por inicializar o SGBDG, e é o ponto de partida para as interações do usuário. A partir dela é possível criar nós (vértices), encontrar nós e relacionamentos (arestas), e realizar as consultas em Cypher. Também há métodos para gerenciar os rótulos, o *schema* e os índices. A interface é composta por 30 métodos.

No Apache Tinker Pop, o usuário interage com o SGBDG através da interface *Graph*. A API trabalha com um grafo único. *Graph* é bastante compacta, possui apenas 14 métodos. Dos quais dois são para adicionar vértices, e os outros métodos mais relevantes são para acessar as coleções de vértices e arestas (possivelmente por identificador) e realizar travessias (a principal operação), entre outros.

A abstração de mais alto nível no Mogwai é a interface *MDatabase*. Esta interface é composta por 52 métodos, mais complexa do que as demais. A primeira diferença primordial é que o Mogwai possui métodos para trabalhar com múltiplos grafos, enquanto *GraphDatabaseService* e *Graph* não possuem.

Além disso, o Mogwai necessita expor um pouco do seu funcionamento interno para que seja possível que a linguagem MogwaiQL opere sobre a API de forma que

a execução das consultas possa ser otimizada em diferentes situações. Em outras palavras, o Mogwai disponibiliza métodos para consultar os índices diretamente.

Podemos dizer que as API do Neo4j e do Apache Tinker Pop fazem implementações mais coesas. Por exemplo, enquanto na interface *MDatabase* há métodos para o *CRUD* das propriedades de um elemento, não há métodos desse tipo nem na *GraphDatabaseService* e nem na *Graph*. As operações de propriedades são realizadas no próprio elemento ao qual elas pertencem.

Na API do Neo4j e do Mogwai é possível executar consultas declarativas com as linguagens Cypher e MogwaiQL, respectivamente. As consultas em Cypher retornam apenas no formato de tuplas, enquanto que em MogwaiQL as consultas podem retornar como tuplas ou como grafos. Na Apache Tinker Pop a priori não é possível fazer consultas declarativas exatamente da mesma forma. Os dados são obtidos principalmente através de travessias, de forma imperativa. Mas a partir da versão 3 da Apache Tinker Pop, há um passo de travessia denominado *match* onde é possível especificar padrões de grafos. O Neo4j também oferece métodos para realizar travessias.

Os vértices são representados pela interface *Node* no Neo4j. Há métodos para incluir, alterar e remover rótulos e propriedades, e métodos para recuperar os relacionamentos de diversas formas, num total de 32 métodos. Já no Mogwai, os vértices são representados pela interface *MVertex*, que contém 19 métodos para realizar as mesmas tarefas. No Apache Tinker Pop, a interface *Vertex* possui 14 métodos, sendo a mais compacta de todas. Apesar das funcionalidades semelhantes entre todos, o Neo4j é o único que permite que um vértice tenha múltiplos rótulos.

A interface *Relationship* representa os relacionamentos no Neo4j. Ela possui 20 métodos para o *CRUD* de propriedades e acesso aos nós de início e fim. No Mogwai, a *MEdge* realiza estas operações com 11 métodos, e a *Edge*, do Apache Tinker Pop, com 14 métodos.

As interfaces de vértices e arestas se assemelham bastante, diferindo em pequenos aspectos, como, por exemplo, no Neo4j há diversos métodos para navegar pelos

relacionamentos. Provavelmente esta disponibilidade de métodos permite ao Neo4j otimizar travessias em diferentes consultas ou situações. O Neo4j também possui métodos para apenas verificar se um elemento possui uma determinada propriedade.

Um pequeno detalhe é que no Neo4j as propriedades são obtidas "diretamente" através de um objeto do tipo *hash*, enquanto que no Mogwai há ainda uma abstração antes de obter o dado propriamente dito. Ao solicitar uma propriedade, obtém-se uma instância de *MProperty*, para então solicitar o valor com o método *value*. O mesmo ocorre no Apache Tinker Pop, com as abstrações *VertexProperty* e *Property*.

Resumidamente, as três API são muito semelhantes no que diz respeito a "expor o modelo de grafo de propriedades". Em relação ao acesso aos dados, tanto o Neo4j quanto o Apache Tinker Pop trabalham tanto com padrões de grafos quanto travessias, enquanto que o Mogwai trabalha apenas com padrões de grafos. Os dois primeiros diferem na maneira como fazem suas travessias e consultas por padrões de grafos, cabendo ao usuário determinar qual é a mais apropriada para seu problema em questão. Além disto, ambas API disponibilizam recursos avançados que o Mogwai não possui, como, por exemplo, a Apache Tinker Pop tem uma interface *GraphComputer* projetada para permitir a computação de "programas orientados a vértice" de forma distribuída, e o Neo4j permite a criação de procedimentos armazenados. Por outro lado, a Mogwai é a única API que trabalha com múltiplos grafos, e, nos problemas em que este recurso é necessário, o arcabouço Mogwai se sobressai, pois não requer o uso de artifícios para simular múltiplos grafos em um grafo só.

4.2 Análise de Desempenho

Esta análise de desempenho objetiva verificar a aplicabilidade do arcabouço Mogwai. Para tanto, o desempenho do Neo4j é utilizado como *baseline*. Ele foi escolhido por ser o SGBDG de uso mais amplo atualmente.

O Neo4j é utilizado através da interface Mogwai apenas para conveniência na construção dos testes. As consultas nas implementações Neo4jCypherNoIndex,

Neo4jCypherGraphIndex e Neo4jCypherGlobalIndex são traduzidas para Cypher, de forma que a influência do Mogwai no desempenho da consulta é praticamente irrelevante. O desempenho destas implementações forma a base de comparação com as outras implementações.

Para avaliar o desempenho das implementações do Mogwai foram projetadas duas baterias de testes. A primeira é voltada para a correspondência de padrões de grafos em uma base de múltiplos grafos. Ela utiliza a base de dados de proteínas proposta por Riesen et al.[46], e consiste na execução de nove diferentes consultas que visam a identificar similaridades entre as diversas proteínas do banco de dados. As consultas foram projetadas de forma a exercitar as diferentes estratégias de otimização de consulta, como um teste *white-box*.

A segunda bateria de testes visa testar operações comuns de grafos numa base de dados mais volumosa, a base de dados web-Stanford[47]. Esta base de dados é composta pelo grafo web de Stanford.edu, contendo 281.904 vértices e 2.312.497 arestas.

4.2.1 Bateria de testes 1

Este primeiro conjunto de testes trata de consultas de correspondência por padrões (isomorfismo) sobre múltiplos grafos. As consultas foram construídas a partir de proteínas existentes na base de dados proposta por Riesen et al[46]. Esta base de dados é constituída por 600 grafos, com um total de 19.580 vértices e 37.282 arestas. Os padrões de grafos das consultas foram criados subtraindo alguns elementos (vértices e arestas) dos grafos das proteínas da base. Desta forma, garante-se todas as consultas possuem no mínimo um resultado. As consultas são nomeadas de acordo com a proteína que lhe deu origem. É interessante notar que as consultas não especificam um determinado grafo sobre o qual serão executadas. Cada consulta executa sobre os 600 grafos constantes da base de dados. Mas, se fosse o caso de restringir a busca em apenas um grafo, isto também seria possível de fazer na MogwaiQL.

Um pequeno detalhe é que na base de dados original as arestas não possuem rótulo. Os rótulos são inseridos durante a carga de dados. Por brevidade, o rótulo “NEIGHBOR” será substituído por “N” nas consultas abaixo.

Nesta bateria de testes, a implementação MogwaiNeo4J foi testada de três formas diferentes: a primeira sem indexação, a segunda com indexação local nos grafos, e a terceira com indexação global. Mais especificamente, os índices do SGBDG Neo4j são criados especificando-se uma propriedade e um rótulo. Porém, os elementos da massa de dados não possuem rótulos.

A implementação MogwaiNeo4J simula vários grafos em um só fazendo com que cada vértice possua como rótulo o identificador do grafo a qual pertence. A chamada indexação local foi realizada criando índices com estes rótulos. Portanto, foram criados diversos índices.

A indexação global necessitou a criação de um teste específico. Durante a carga de dados, todos os elementos do grafo receberam um rótulo especial. E este rótulo foi utilizado para indexar as propriedades.

O teste com o MogwaiJDBC utilizou índices compostos (nome e valor da propriedade) nas tabelas de propriedades. As demais implementações utilizam o Apache Lucene.

Esta bateria de testes foi executada numa máquina com processador Intel Core i7-4770, com 16 GB de memória RAM, porém apenas 6 GB foram destinados à maquina virtual Java que executou os testes. A máquina utiliza disco rígido rotacional de 7200 RPM.

Os testes foram executados sem nenhuma rotina para “aquecimento de cache”, ou pré-carregamento de dados em memória, que não fosse feita pelo próprio SGBD utilizado. Cada teste foi executado três vezes, foi realizada uma média dos valores de execução de cada consulta, e obtido um desvio padrão.

Consulta 1

```
match (v1)-[e0:N]->(v0) where e0.frequency = 1
AND v0.sequence = 'KEDVDAAVKQLLSLKAEYKEKTG';
```

Esta consulta foi construída a partir da proteína pdb1fyj, e objetiva testar a busca por um padrão simples. Após a sua execução percebe-se que ela tem uma característica muito peculiar: o atributo "sequence" com valor "KEDVDAAVKQLLSLKAEYKEKTG" ocorre apenas uma vez na base de dados. Esta característica é capturada pela poda do espaço de busca, que, de acordo com o log de execução, ocorre em cerca de 80 ms nas implementações que utilizam o Apache Lucene, e faz com que a consulta execute rapidamente. As implementações tiveram desempenho superior ao *benchmark*. Os tempos de execução desta consulta constam na tabela 4.1 e são ilustrados na figura 4.1.

Tabela 4.1: tempos de execução da consulta 1

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	2.461,33 ms	1.103,34 ms
MogwaiCypherNoIndex	1.427,33 ms	26,28 ms
MogwaiObjectDB	423,33 ms	0,47 ms
MogwaiJpaObjectDB	543,67 ms	10,62 ms
MogwaiJpaMySQL	364,33 ms	7.54 ms
MogwaiJpaH2	364,00 ms	9,27 ms
MogwaiJdbcMySQL	473,33 ms	7,54 ms
MogwaiCypherGraphIndex	1.965,67 ms	96,14 ms
MogwaiCypherGlobalIndex	1.402,33 ms	11,44 ms

Consulta 2

```
match (v2)-[e0:N]->(v0), (v1)-[e1:N]->(v0),
(v2)-[e2:N]->(v1) where e0.frequency = 1
AND e1.frequency = 1 AND e2.frequency = 1
AND v0.sequence = 'SQ'
```

Apesar de possuir uma triangulação, a segunda consulta, construída a partir da proteína pdb1i8g, é similar à primeira em termos de restrições nos atributos dos elementos. São realizadas restrições nos atributos "frequency" e "sequence".

Após a execução, observa-se nas implementações Mogwai que a poda do espaço

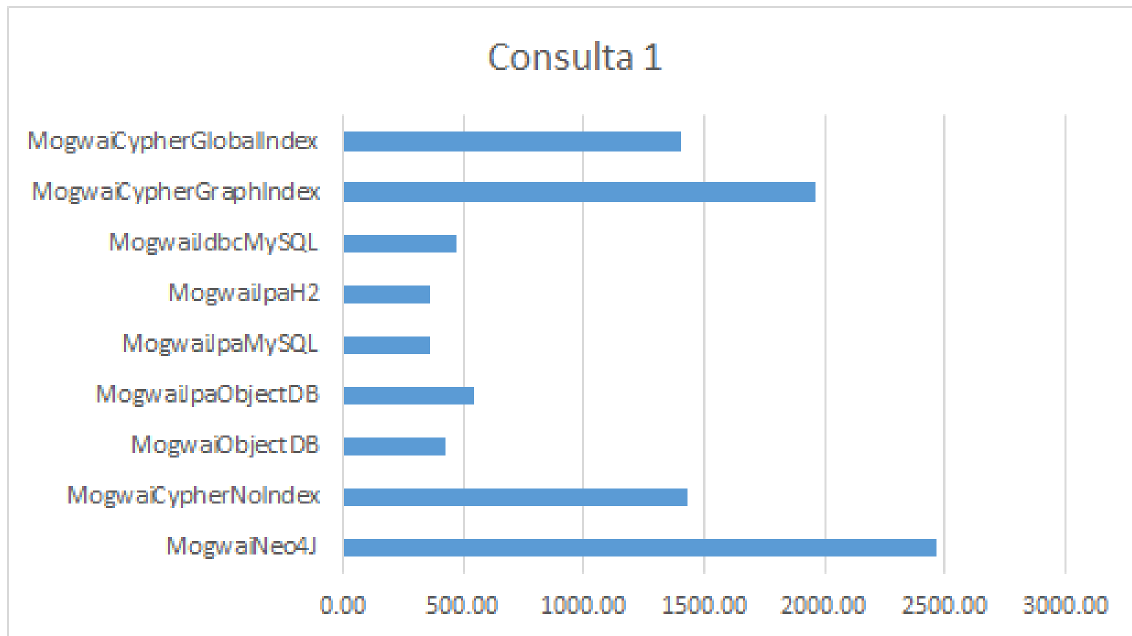


Figura 4.1: desempenho da consulta 1 em milisegundos

de busca foi realizada através do atributo de vértice “sequence”, que ocorre apenas 7 vezes com o valor “SQ” no banco de dados. Este pequeno espaço de busca não se traduz em tempos de desempenho similares entre as implementações, como ocorre na consulta 1.

Os tempos de cada implementação são listados na tabela 4.2, e ilustrados na figura 4.2. Um ponto interessante é que nesta consulta a implementação Cypher que não utiliza índices foi a mais rápida de todas, mais do que as implementações que utilizam índice. Este comportamento é um tanto inesperado, pois espera-se que a poda pela restrição de atributo seja o fator determinante para o tempo de execução. Além disto, implementação MogwaiJpaObjectDB possui um desempenho que se destaca negativamente das outras implementações, até mesmo da outra implementação com ObjectDB.

Tabela 4.2: tempos de execução da consulta 2

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	270,00 ms	15,93 ms
MogwaiCypherNoIndex	172 ms	10,67 ms
MogwaiObjectDB	428 ms	6,98 ms
MogwaiJpaObjectDB	5.158,33 ms	81,08 ms
MogwaiJpaMySQL	441,67 ms	14,61 ms
MogwaiJpaH2	542,33 ms	50,76 ms
MogwaiJdbcMySQL	1.550,00 ms	7,07 ms
MogwaiCypherGraphIndex	1.752,33 ms	178,74 ms
MogwaiCypherGlobalIndex	1.497,00 ms	16,97 ms

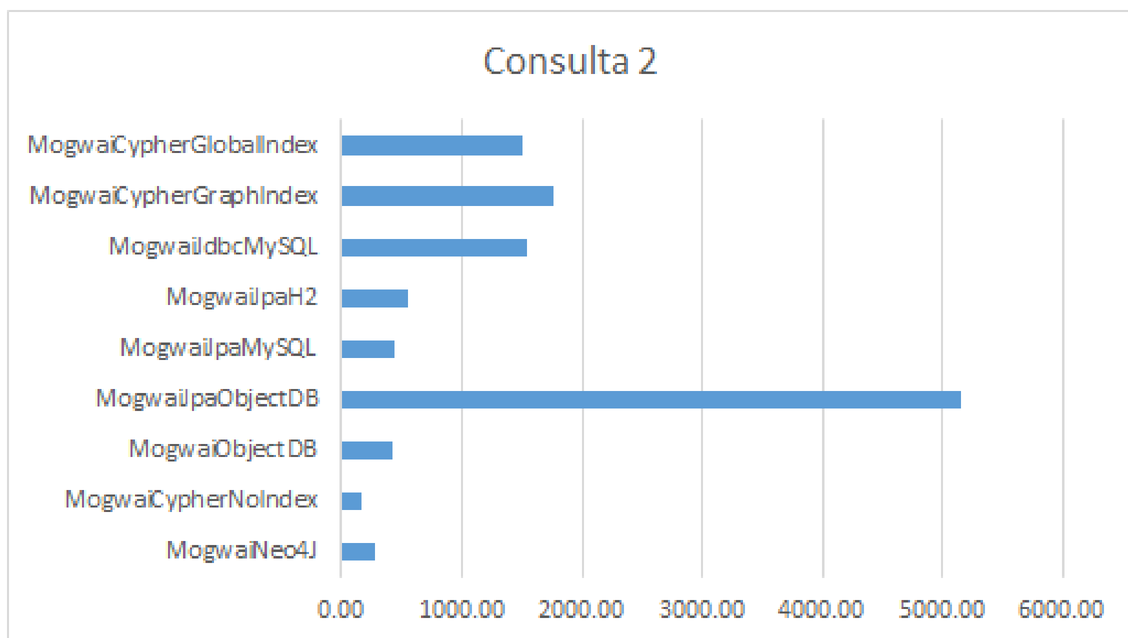


Figura 4.2: desempenho da consulta 2 em milisegundos

Consulta 3

```
match (v1)-[e0:N]->(v0), (v2)-[e1:N]->(v1),
(v3)-[e2:N]->(v0), (v0)-[e3:N]->(v2),
(v3)-[e4:N]->(v1), (v3)-[e5:N]->(v2)
where e0.frequency = 2 AND e1.frequency = 2
AND e2.frequency = 1 ;
```

A consulta 3, projetada a partir da proteína pdb1mea, é a consulta que demanda mais tempo de processamento entre todas. Esta consulta foi projetada para não utilizar restrições nos vértices. Portanto, a estratégia para redução do espaço de busca, diferentemente das anteriores, utiliza os predicados das arestas. Após a

execução nota-se que a restrição do atributo "frequency" é pouco restritiva, quer dizer, ocorre muitas vezes no banco de dados. O algoritmo necessita buscar 6334 arestas na base de dados para começar a busca pelo padrão.

O desempenho do Neo4j em todas as implementações que o utilizam foi muito superior aos outros, conforme a tabela 4.3 e a figura 4.3. A julgar pelo log de execução da consulta, o gargalo neste processamento está não apenas em trazer as arestas do SGBD para a memória, mas também na exploração de cada vértice do espaço de busca inicial, sugerindo que o Neo4j é muito superior em termos de travessia. Dentre as demais implementações, a MogwaiJpaMySQL obtém o terceiro melhor desempenho, cerca de 20 vezes mais lento que o melhor desempenho do Neo4j.

Tabela 4.3: tempos de execução da consulta 3

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	1.018,67 ms	32,49 ms
MogwaiCypherNoIndex	695,33 ms	10,04 ms
MogwaiObjectDB	227.505,30 ms	3.876,93 ms
MogwaiJpaObjectDB	737.534,00 ms	5.926,14 ms
MogwaiJpaMySQL	13.728,33 ms	99,68 ms
MogwaiJpaH2	15.867,00 ms	527,07 ms
MogwaiJdbcMySQL	843.114,30 ms	20.768,02 ms
MogwaiCypherGraphIndex	2.407,33 ms	128,26 ms
MogwaiCypherGlobalIndex	2.317,67 ms	27,39 ms

Consulta 4

```
match (v3)-[e0:N]-$>$(v2), (v0)-[e1:N]-$>$(v3),
(v4)-[e2:N]-$>$(v1), (v0)-[e3:N]-$>$(v2),
(v3)-[e4:N]-$>$(v1), (v4)-[e5:N]-$>$(v2),
(v0)-[e6:N]-$>$(v1), (v1)-[e7:N]-$>$(v2),
(v4)-[e8:N]-$>$(v3) where e3.frequency = 1
AND e4.frequency = 1 AND e5.frequency = 1
AND e6.frequency = 1 AND e7.frequency = 1
AND e8.frequency = 1;
```

Esta consulta foi montada com base na proteína pdb1a6x. Ela possui mais

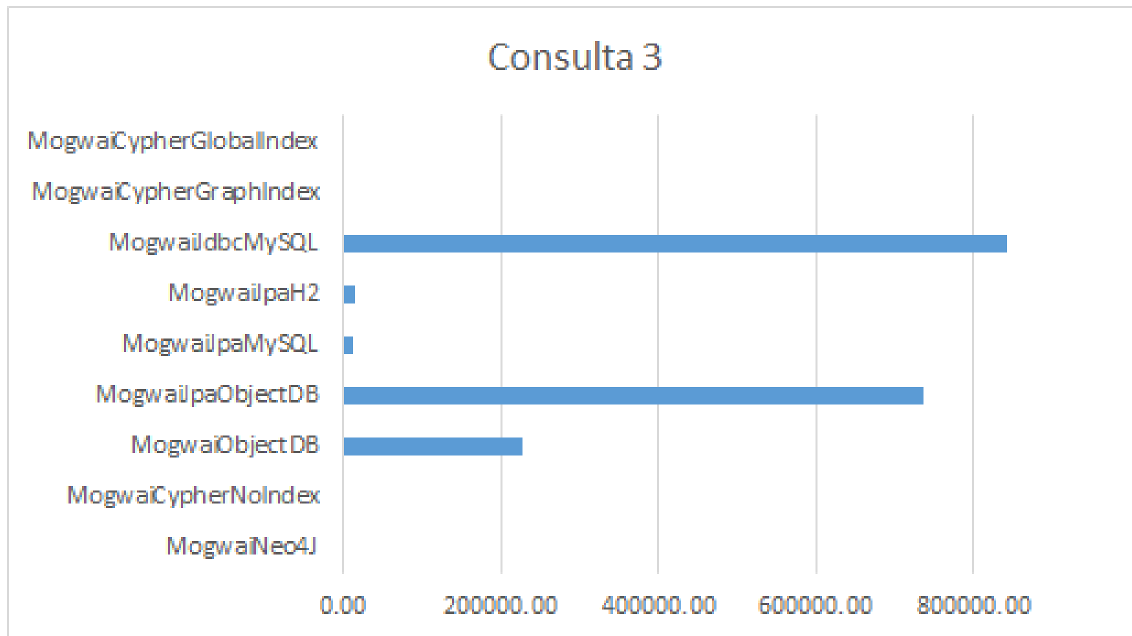


Figura 4.3: desempenho da consulta 3 em milisegundos

elementos e mais restrições que a consulta 3, e objetiva verificar como isto impacta os tempos de execução - constantes na tabela 4.4 e figura 4.4. Observa-se durante a sua execução que seu comportamento é similar à consulta 3, pois o algoritmo necessita utilizar o atributo “frequency” para reduzir o espaço inicial de busca. Este é constituído por 30.948 arestas, quase cinco vezes mais do que a consulta 3. Os tempos de execução, no entanto, não variam da mesma forma.

Tabela 4.4: tempos de execução da consulta 4

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	3.320,00 ms	142,97 ms
MogwaiCypherNoIndex	1.555,33 ms	54,95 ms
MogwaiObjectDB	255.594,30 ms	4.676,08 ms
MogwaiJpaObjectDB	771.163,30 ms	7.151,98 ms
MogwaiJpaMySQL	31.335,33 ms	325,83 ms
MogwaiJpaH2	35.001,67 ms	517,63 ms
MogwaiJdbcMySQL	16.937.279,00 ms	4.511,80 ms
MogwaiCypherGraphIndex	3.713,00 ms	96,20 ms
MogwaiCypherGlobalIndex	3.665,33 ms	53,01 ms

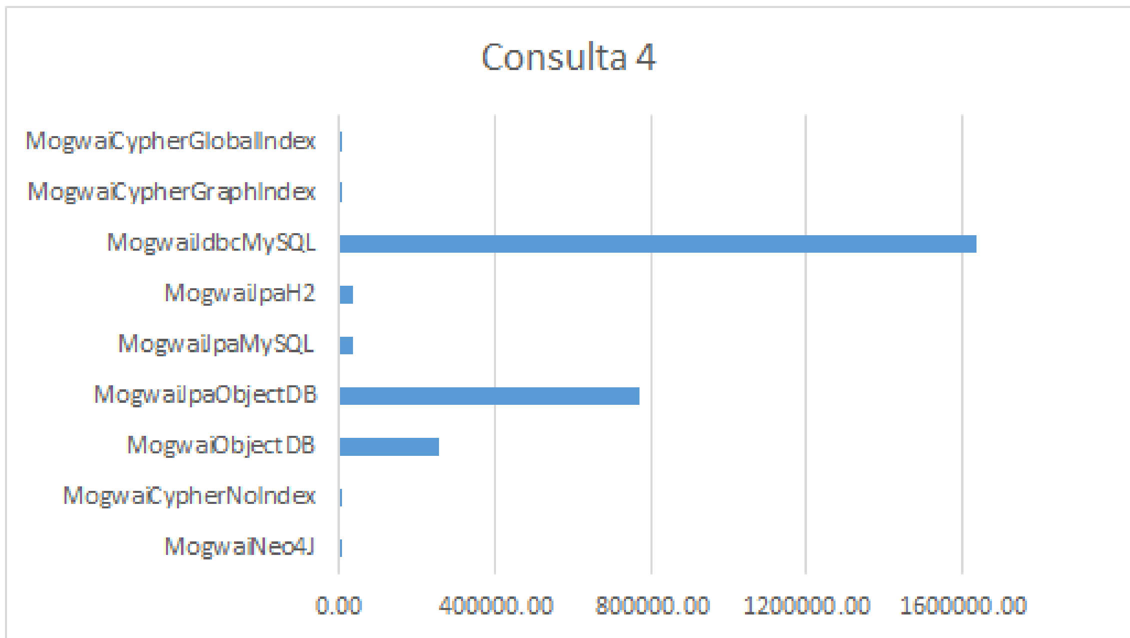


Figura 4.4: desempenho da consulta 4 em milisegundos

Consulta 5

```
match (v1)-[e0:N]->(v0), (v3)-[e1:N]->(v1),
(v1)-[e2:N]->(v5), (v2)-[e3:N]->(v0),
(v3)-[e4:N]->(v5), (v4)-[e5:N]->(v0),
(v3)-[e6:N]->(v0), (v4)-[e7:N]->(v5),
(v5)-[e8:N]->(v0), (v4)-[e9:N]->(v2),
(v5)-[e10:N]->(v2) where e0.frequency = 2
AND e1.frequency = 2 AND v5.sequence = 'QCVEV' ;
```

O objetivo desta consulta, derivada da proteína pdb1blu, é continuar a explorar a relação entre a complexidade do padrão de grafo, o tamanho do espaço de busca e os tempos de execução, que constam na tabela 4.5 e figura 4.5.

Mesmo o padrão de grafo sendo constituído por mais elementos que a consulta 1, o espaço inicial de busca é o mesmo, pois o predicado "sequence" com valor "QC-VEV" possui apenas uma ocorrência no banco de dados, de forma que o resultado é similar ao da consulta 1.

Nesta consulta, assim como na consulta 1, as implementações que utilizam a

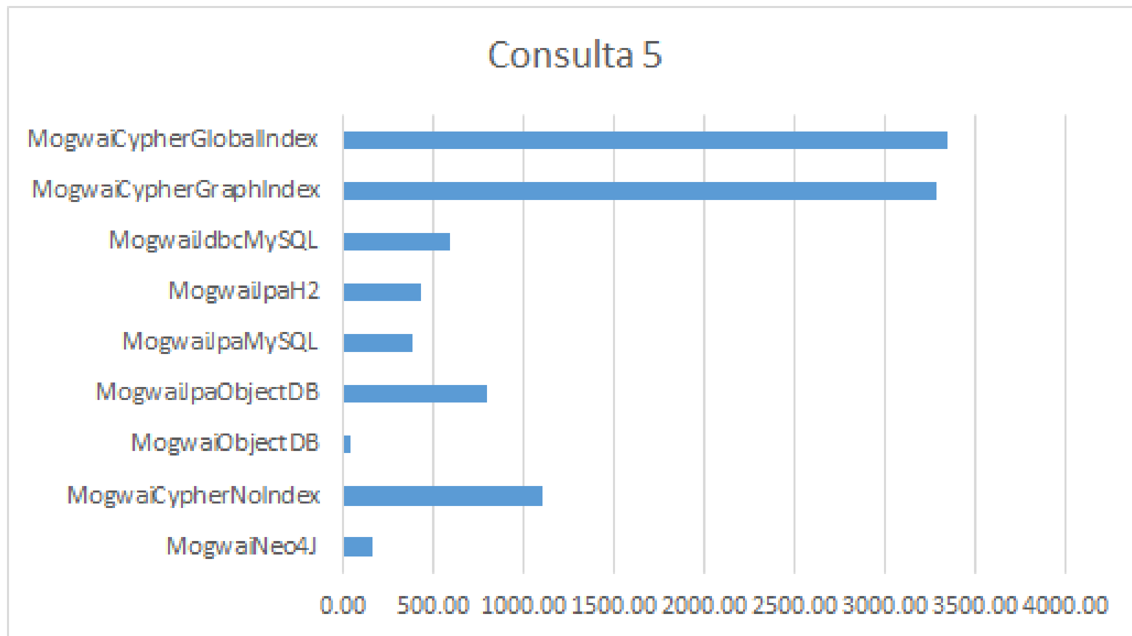


Figura 4.5: desempenho da consulta 5 em milisegundos

MogwaiQL se sobressaíram sobre as que utilizam Cypher.

Tabela 4.5: tempos de execução da consulta 5

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	162,67 ms	7,32 ms
MogwaiCypherNoIndex	1.106,00 ms	20,99 ms
MogwaiObjectDB	41 ms	0,82 ms
MogwaiJpaObjectDB	796,00 ms	13,06 ms
MogwaiJpaMySQL	389,67 ms	12,66 ms
MogwaiJpaH2	432,67 ms	6,65 ms
MogwaiJdbcMySQL	595,00 ms	4,97 ms
MogwaiCypherGraphIndex	3.281,33 ms	44,25 ms
MogwaiCypherGlobalIndex	3.350,00 ms	25,57 ms

Consulta 6

```
match (v5)-[e0:N]->(v2), (v1)-[e1:N]->(v0),
(v4)-[e2:N]->(v1), (v2)-[e3:N]->(v1),
(v3)-[e4:N]->(v1), (v5)-[e5:N]->(v1),
(v1)-[e6:N]->(v6), (v5)-[e7:N]->(v0),
(v6)-[e8:N]->(v3), (v2)-[e9:N]->(v0),
(v4)-[e10:N]->(v3), (v6)-[e11:N]->(v4)
where e0.frequency = 2 AND e1.frequency = 2
AND e2.frequency = 2 AND e3.frequency = 1
AND e4.frequency = 1 AND e5.frequency = 1
AND e6.frequency = 1 AND e7.frequency = 1
AND e8.frequency = 1 AND e9.frequency = 1
AND e10.frequency = 1 AND e11.frequency = 1 ;
```

Esta consulta tem por objetivo identificar o comportamento quando há várias restrições de arestas. Construída a partir da proteína pdb1iyf, sua execução é similar à 3, pois o espaço de busca é restrito pela arestas com atributo “frequency” com valor “2”.

Observa-se pela tabela de resultados 4.6 que o padrão de grafo mais complexo não se traduziu necessariamente em tempos de execução maiores na implementações Mogwai. Nas implementações que utilizam o Neo4j esta relação é mais clara.

Tabela 4.6: tempos de execução da consulta 6

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	1151,67 ms	84,39 ms
MogwaiCypherNoIndex	2.310,33 ms	57,51 ms
MogwaiObjectDB	4.181,33 ms	52,26 ms
MogwaiJpaObjectDB	757.721,70 ms	5.664,11 ms
MogwaiJpaMySQL	12.074,33 ms	315,12 ms
MogwaiJpaH2	13.855,33 ms	558,67 ms
MogwaiJdbcMySQL	605.737,30 ms	4.199,03 ms
MogwaiCypherGraphIndex	4.519,33 ms	14,61 ms
MogwaiCypherGlobalIndex	4.667,00 ms	30,73 ms

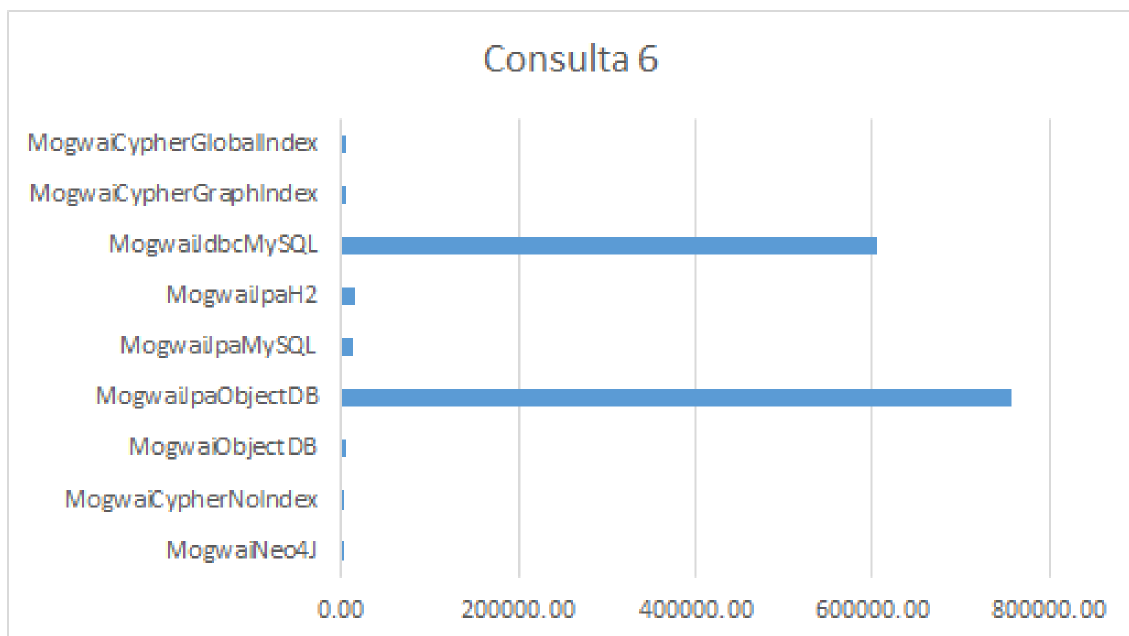


Figura 4.6: desempenho da consulta 6 em milisegundos

Consulta 7

```
match (v5)-[e0:N]->(v1), (v2)-[e1:N]->(v3),
(v6)-[e2:N]->(v2), (v0)-[e3:N]->(v4),
(v1)-[e4:N]->(v4), (v4)-[e5:N]->(v2),
(v6)-[e6:N]->(v3), (v7)-[e7:N]->(v3),
(v0)-[e8:N]->(v5), (v5)-[e9:N]->(v4),
(v7)-[e10:N]->(v2), (v0)-[e11:N]->(v1),
(v4)-[e12:N]->(v3), (v7)-[e13:N]->(v6)
where e11.frequency = 1 AND e12.frequency = 1
AND e13.frequency = 1 AND v0.aaLength = 9
AND v0.type = 1 ;
```

Esta consulta, derivada da proteína pdb1krs, continua com o objetivo de testar padrões com mais elementos, porém com número maior de diferentes atributos nas restrições. Sua execução reduz o espaço de busca procurando pelos vértices com a propriedade “aaLength” com valor “9”, o que resulta em 943 vértices. De acordo com a tabela 4.7 de resultados, as implementações do Mogwai obtiveram desempenho

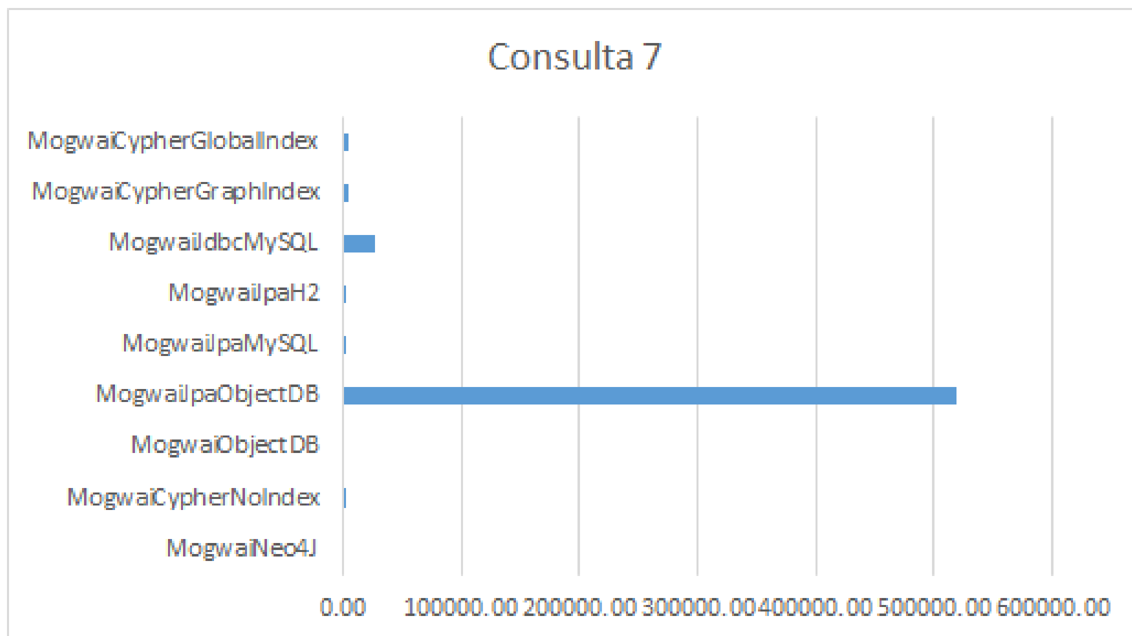


Figura 4.7: desempenho da consulta 7 em milisegundos

compatível com o *baseline*, excetuando-se a implementação MogwaiJpaObjectDB, que se destaca negativamente outra vez, fato que é claramente ilustrado na figura 4.7.

Tabela 4.7: tempos de execução da consulta 7

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	370,33 ms	21,70 ms
MogwaiCypherNoIndex	2.303,67 ms	38,42 ms
MogwaiObjectDB	726,00 ms	26,55 ms
MogwaiJpaObjectDB	518.701,70 ms	10.922,36
MogwaiJpaMySQL	2.293,00 ms	38,38 ms
MogwaiJpaH2	3.090 ms	212,99 ms
MogwaiJdbcMySQL	27.322,33 ms	171,90 ms
MogwaiCypherGraphIndex	4.401,67 ms	72,62 ms
MogwaiCypherGlobalIndex	4.401,67 ms	72,62 ms

Consulta 8

```
match (v2)-[e0:N]->(v1), (v5)-[e1:N]->(v4),
(v0)-[e2:N]->(v1), (v3)-[e3:N]->(v4),
(v6)-[e4:N]->(v7), (v8)-[e5:N]->(v7),
(v3)-[e6:N]->(v0), (v6)-[e7:N]->(v0)
where v0.sequence = 'TMMHGQDHLMTTILLKDVIHHLIELY'
AND v1.sequence = 'AKADSLVVEAGSCIAEAHSSQTGMLAREA';
```

A consulta 8, gerada a partir da proteína pdb1e2a, utiliza apenas restrições nos vértices. Apesar de possuir o padrão de grafo com diversos elementos, ela se comporta de maneira similar às demais consultas cujos espaços de busca são reduzidos, conforme demonstram os resultados da tabela 4.8.

Tabela 4.8: tempos de execução da consulta 8

Implementação	**média**	**desvio padrão**
MogwaiNeo4j	61,66 ms	4,50 ms
MogwaiCypherNoIndex	354,33	7,31 ms
MogwaiObjectDB	47 ms	0,82 ms
MogwaiJpaObjectDB	863,33 ms	14,82 ms
MogwaiJpaMySQL	358,67 ms	12,66 ms
MogwaiJpaH2	364,67 ms	3,30 ms
MogwaiJdbcMySQL	416,67 ms	3,30 ms
MogwaiCypherGraphIndex	2.303,67 ms	7,54 ms
MogwaiCypherGlobalIndex	2.185,67 ms	12,55 ms

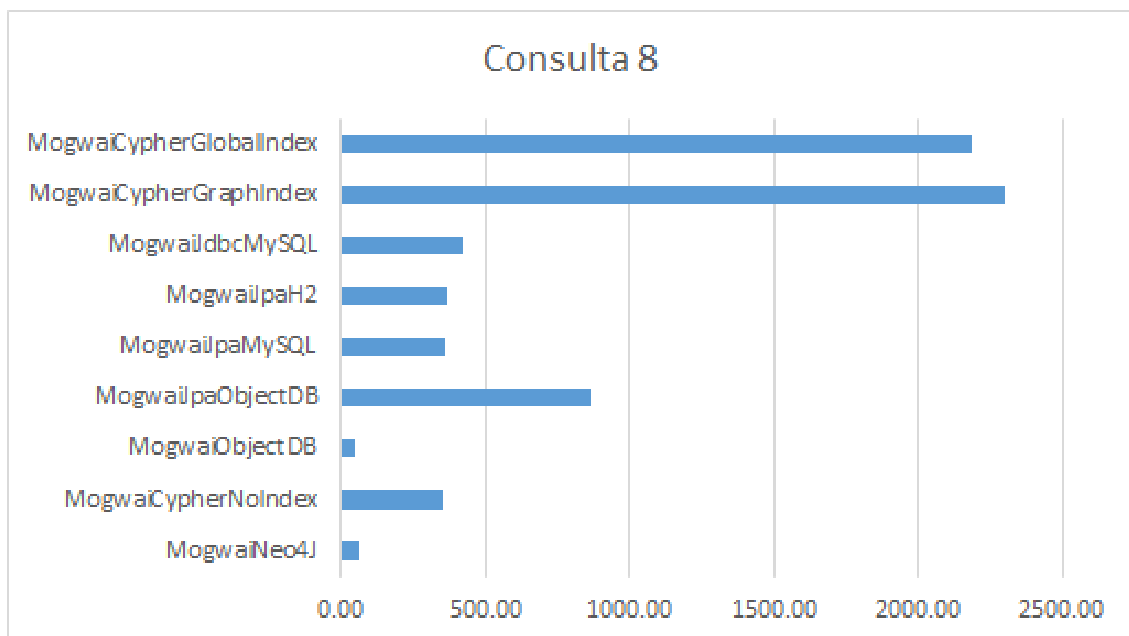


Figura 4.8: desempenho da consulta 8 em milisegundos

Consulta 9

```
match (v2)-[e0:N]->(v1), (v7)-[e1:N]->(v2),
(v5)-[e2:N]->(v4), (v0)-[e3:N]->(v1),
(v3)-[e4:N]->(v1), (v6)-[e5:N]->(v0),
(v8)-[e6:N]->(v0), (v9)-[e7:N]->(v2),
(v0)-[e8:N]->(v4), (v1)-[e9:N]->(v5),
(v2)-[e10:N]->(v4), (v3)-[e11:N]->(v4),
(v7)-[e12:N]->(v5), (v8)-[e13:N]->(v3),
(v9)-[e14:N]->(v5), (v3)-[e15:N]->(v0),
(v8)-[e16:N]->(v6)
where v0.sequence = 'anDerWaals' AND v0.aaLength = 9 ;
```

A consulta 9 foi desenvolvida a partir da proteína pdb1a1e, e é a consulta que possui mais elementos no padrão de grafo. O espaço inicial de busca é novamente reduzido pela propriedade “aaLength” de valor “9” para 943 vértices. Os tempos de execução são listados na tabela 4.9, e, mais uma vez, são compatíveis com o *baseline*, excetuando-se a implementação MogwaiJpaObjectDB.

Tabela 4.9: tempos de execução da consulta 9

Implementação	média	desvio padrão
MogwaiNeo4j	165,67 ms	10,84 ms
MogwaiCypherNoIndex	3051,33 ms	50,39 ms
MogwaiObjectDB	63,00 ms	0,82 ms
MogwaiJpaObjectDB	510.938,30 ms	9.308,29 ms
MogwaiJpaMySQL	1.498,00 ms	33,79 ms
MogwaiJpaH2	2.126,00 ms	195,33 ms
MogwaiJdbcMySQL	1.003,67 ms	8,58 ms
MogwaiCypherGraphIndex	4.882,67 ms	76,34 ms
MogwaiCypherGlobalIndex	5.611,33 ms	38,38 ms

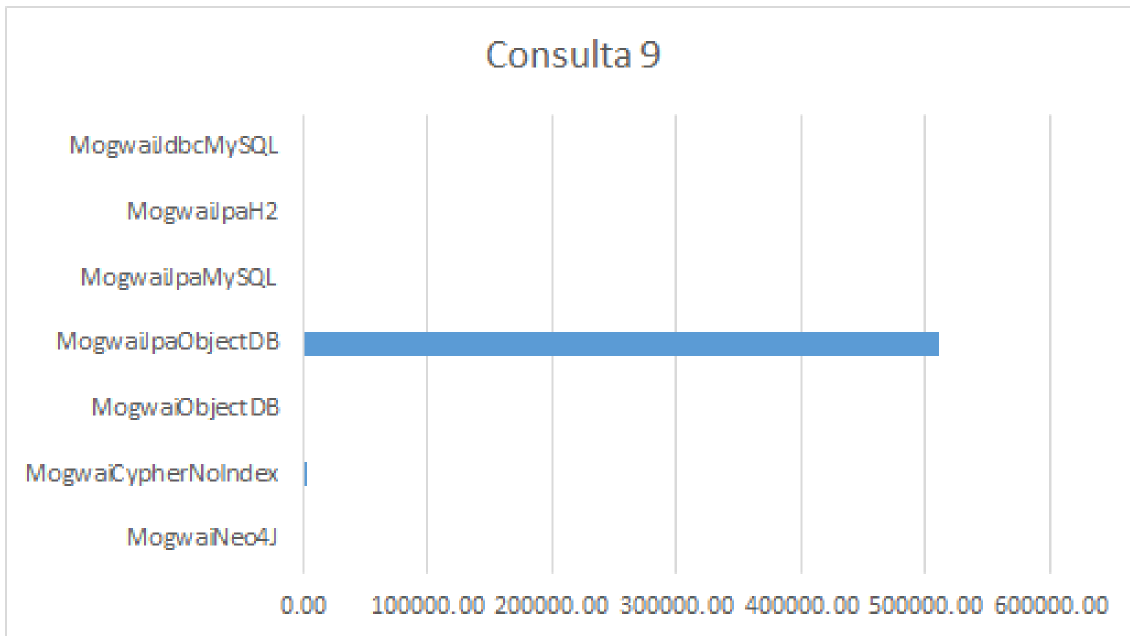


Figura 4.9: desempenho da consulta 9

4.2.2 Bateria de testes 2

Nesta bateria de testes foi utilizada a base de dados do grafo web da universidade de Stanford [48]. Esta base de dados é composta por um único grafo com 281.903 vértices e 2.312.497 arestas. Esta base de dados não possui atributos nem nos vértices e nem nas arestas além do identificador do elemento. O objetivo desta bateria de testes é verificar o comportamento quando o Mogwai não consegue reduzir o espaço de busca.

Esta bateria de testes foi executada em servidores na nuvem da Amazon [49], com a configuração i3.large: *2 vCPUs, High Frequency Intel Xeon E5-2686 v4 (Broadwell) Processors with base frequency of 2.3 GHz, 15.25 GiB memory, 1 x 0.475 NVMe*

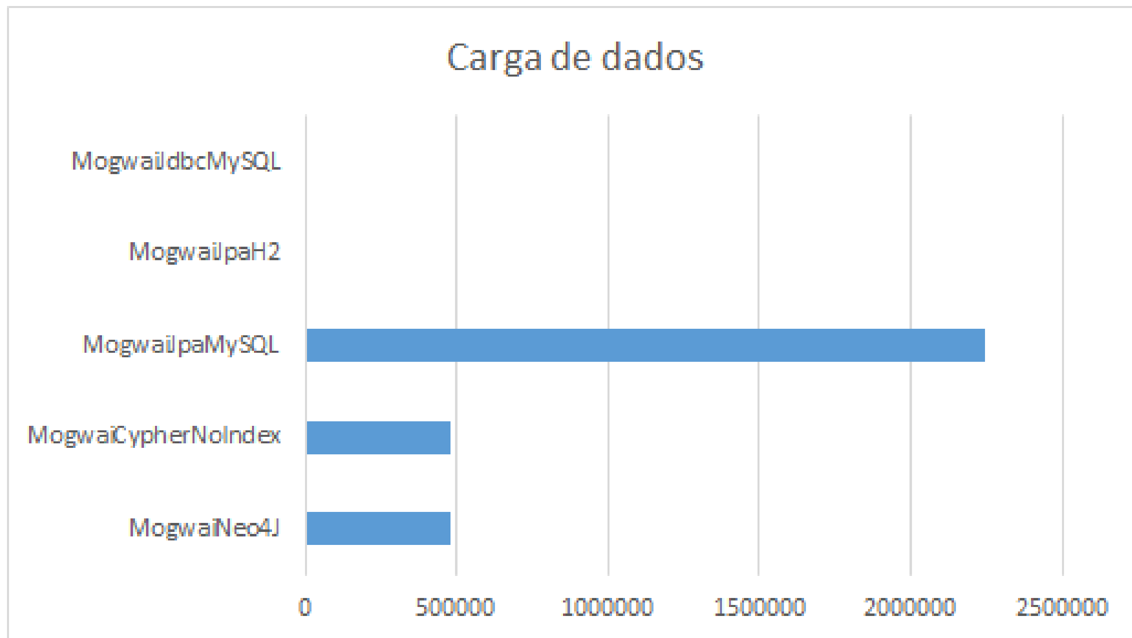


Figura 4.10: tempo de carregamento em segundos

SSD Storage.

Carga de dados

A carga de dados para esta bateria de testes é significativamente mais complicada. Não apenas pelo maior volume de dados, mas também pela característica da entrada dos dados. A base de dados do grafo web é composta por dois arquivos, um de vértices, e outro de arestas. O procedimento mais imediato para carregar estes dados é carregar primeiro os vértices, e depois as arestas, pois as arestas fazem referências aos vértices. Entretanto, este procedimento demonstrou lentidão com algumas tecnologias. Acredita-se que a lentidão é causada pela necessidade de reorganizar o arquivo em disco. Mas para afirmar categoricamente seria necessário utilizar alguma ferramenta apropriada para tal.

Tabela 4.10: tempos de carregamento da base de dados

MogwaiNeo4j	478.007.054 ms
MogwaiJpaMySQL	2.246.400.000 ms
MogwaiJpaH2	2.126,00 ms
MogwaiJdbcMySQL	699.537 ms

Observa-se que o carregamento da base de dados relacional

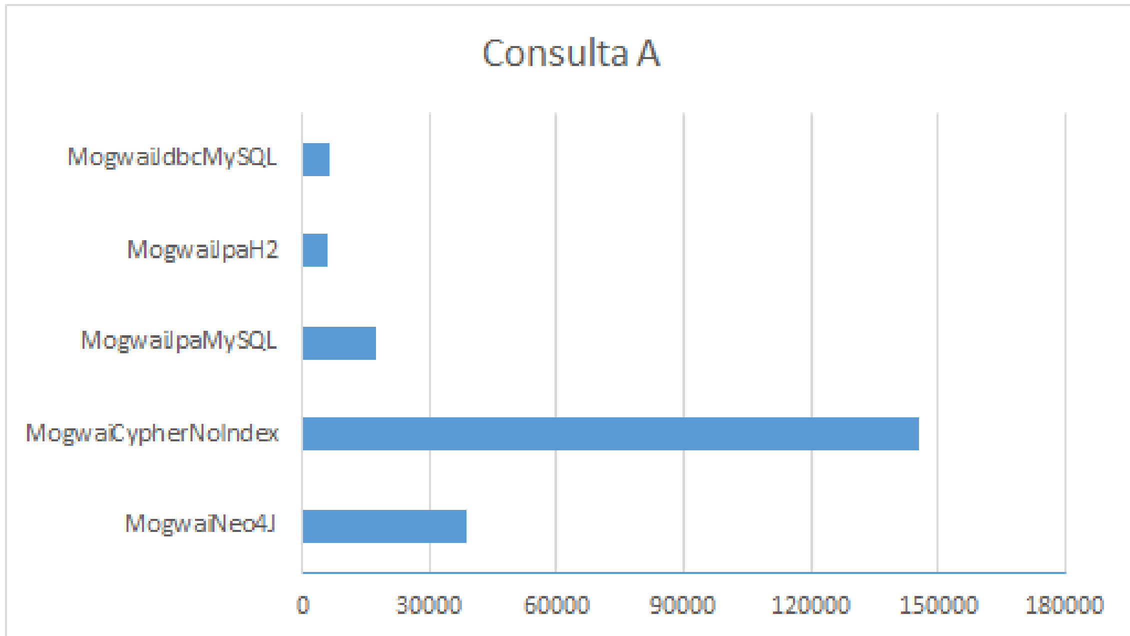


Figura 4.11: desempenho da seleção em milisegundos

(MogwaiJdbcMySQL) teve desempenho bastante superior aos demais.

Consulta A

A primeira consulta efetua a seleção de 1000 vértices escolhidos aleatoriamente.

```
match (v #'webStanford' $identificadorVertice);
```

O desempenho em milisegundos no teste de seleção consta na tabela 4.11.

Esperava-se de forma geral que o SGBD H2 desempenhasse melhor. A consulta A é a única onde a MogwaiJpaH2 tem o melhor desempenho. O segundo e o terceiro melhores resultados são das implementações que utilizam o SGBDR MySQL, superando o *baseline*.

Tabela 4.11: tempos de execução teste de seleção

MogwaiJpaH2	5.745 ms
MogwaiJdbcMySQL	6.555 ms
MogwaiJpaMySQL	17.510 ms
MogwaiNeo4j	38.656 ms
MogwaiCypherNoIndex	145.416 ms

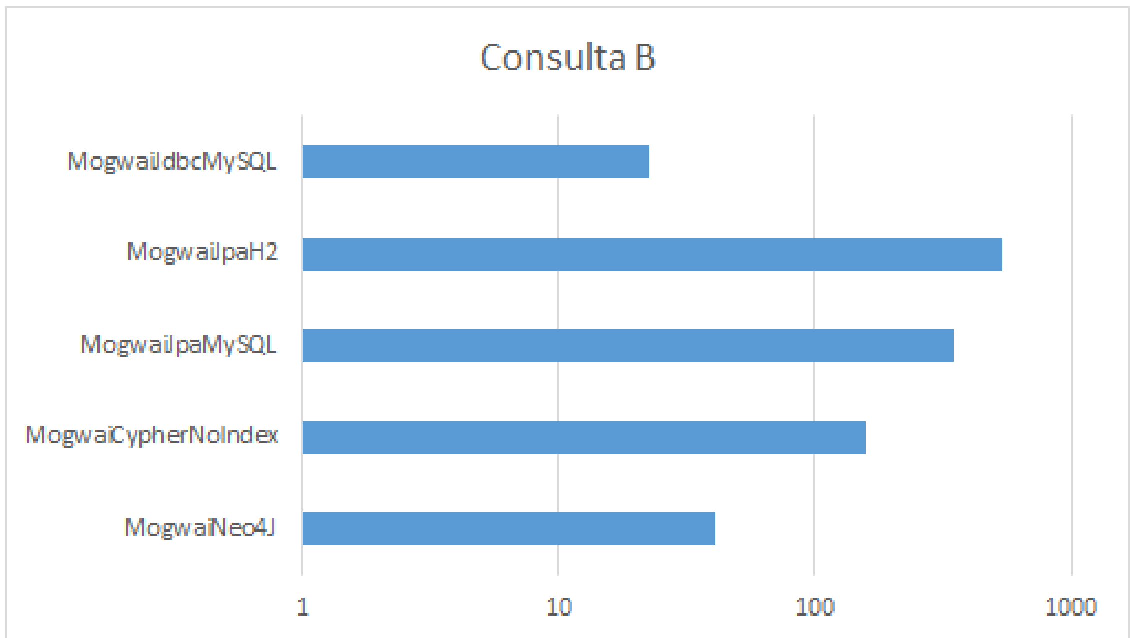


Figura 4.12: desempenho da vizinhança em milisegundos

Consulta B

A segunda consulta é uma consulta de adjacência. Ela busca os vizinhos e os vizinhos dos vizinhos dos mesmos 1000 vértices escolhidos aleatoriamente. O resultado é ilustrado pela tabela 4.12.

```
match (v #'webStanford' $identificadorVertice)-[1..2]->(n);
```

O melhor resultado é obtido pela MogwaiJdbcMySQL, seguido pelas implementações Neo4jse pelas implementações JPA. A MogwaiJpaH2 tem o pior resultado.

Tabela 4.12: tempos de execução teste de vizinhança

MogwaiJdbcMySQL	22.467 ms
MogwaiNeo4j	41.078 ms
MogwaiCypherNoIndex	159.232 ms
MogwaiJpaMySQL	351.794 ms
MogwaiJpaH2	534.649 ms

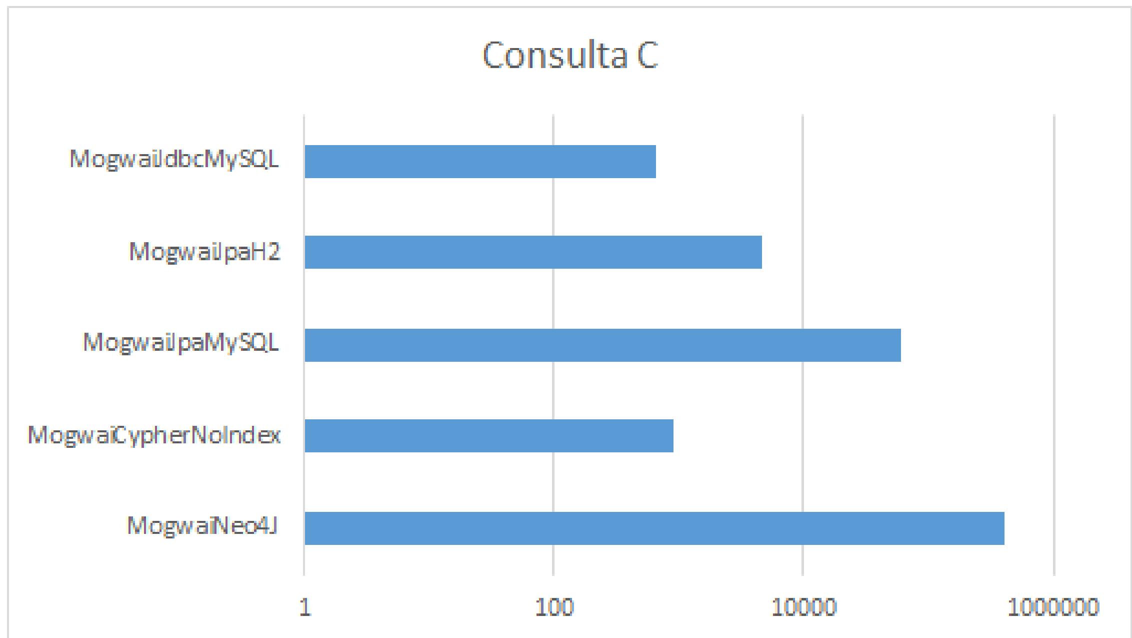


Figura 4.13: desempenho do caminho mais curto em milisegundos

Consulta C

A terceira consulta é uma consulta de alcance. Ela busca o caminho mais curto entre 50 pares de vértices.

```
find path between identificadorVerticeOrigem
AND identificadorVerticeDestino on graph 'webStanford';
```

A MogwaiJdbcMySQL obtém o melhor desempenho, seguida pela MogwaiCypherNoIndex e MogwaiJpaH2. A MogwaiJpaMySQL teve desempenho cerca de 90 vezes mais lento que a MogwaiJdbcMySQL. Este resultado evidencia como que uma mesma tecnologia de SGBD pode ter resultados tão distintos para realizar uma mesma tarefa, dependendo da forma como é utilizada. Neste caso específico, o *overhead* da orientação a objetos parece pesar na resolução do problema.

Tabela 4.13: tempos de execução teste caminho mais curto

MogwaiNeo4j	411.205 ms
MogwaiCypherNoIndex	906 ms
MogwaiJpaMySQL	59.314 ms
MogwaiJpaH2	4.674 ms
MogwaiJdbcMySQL	654 ms

A partir dos testes realizados é possível chegar a algumas conclusões.

Na bateria de testes 1, quase todas as implementações que não utilizam o Neo4j são mais rápidas do que as que o utilizam nas consultas 1, 2, 5, 7, 8, 9. A principal característica destas consultas é que o espaço inicial de busca é pequeno, de no máximo 943 vértices. Conforme este espaço cresce, o desempenho do Neo4j passa a ser superior. Este comportamento sugere que o Mogwai realiza a poda do espaço de busca de forma mais eficiente, enquanto que o Neo4j é mais eficiente em consultas de travessias.

Duas classificações foram montadas sobre os resultados da bateria de testes 1. A primeira, denominada classificação por posição, é montada somando-se a posição de cada implementação em cada consulta (qual foi a 1ª, 2ª, 3ª mais rápida, e assim por diante), e consta na tabela 4.14. A segunda classificação, denominada por tempo total, é montada através da soma dos tempos médios de cada implementação em cada consulta, e é exibida na tabela 4.15.

A MogwaiNeo4j foi a implementação mais rápida de todas em ambas as classificações, com pontuação 23, tendo ficado nas posições 9, 2, 2, 2, 2, 1, 1, 2 e 2, e tempo total de 8982 ms. Podemos argumentar que ela extrai o melhor das duas tecnologias: ela se utiliza da poda realizada pelo Mogwai, e pela rapidez das travessias do Neo4j.

O MogwaiObjectDB aparece em segundo lugar por posição, mas é preciso observar pela tabela de tempos totais, que quando o Neo4j é mais rápido, ele é muito mais rápido, de forma que a MogwaiObjectDB ocupa a 7ª classificação por tempo.

A MogwaiCypherNoIndex é a segunda mais rápida por tempo total. Este bom resultado ressalta a ideia de que realizar as travessias com rapidez é tão ou mais importante do que ter mecanismos de indexação.

As implementações MogwaiCypherGlobalIndex e MogwaiCypherGraphIndex ocupam a terceira e quarta classificações por tempo, respectivamente.

A MogwaiJpaMySQL aparece em quinto lugar, com tempo total acima do dobro da MogwaiCypherGraphIndex. Apesar disso, pode-se argumentar que o tempo é

bem razoável (não é ordens de magnitude mais lento), deixando acesa a ideia de que SGBDR podem ser uma boa solução para a computação de grafos.

Pelo fato de que o H2 é um SGBD em memória, esperava-se que o MogwaiJpaH2 obtivesse melhor colocação do que a sexta. Este resultado provavelmente é devido pelo fato que os procedimentos de carga e consulta são separados, necessitando persistir os dados em disco. Como não há procedimento de aquecimento nas consultas, esta implementação pode ter sido "prejudicada".

Devido à facilidade que o modelo de dados de objetos consegue representar o modelo de grafos, esperava-se bons resultados de desempenho da MogwaiObjectDB. De fato ela está bem classificada em termos de posição, mas seu tempo total não é bom. Seu desempenho é muito bom quando o espaço de busca é pequeno, mas muito ruim quando grande. Este comportamento enfatiza a necessidade de serem realizados testes com bases de dados com diferentes tamanhos e características.

A MogwaiJdbcMySQL e a MogwaiJpaObjectDB possuem os piores desempenhos, com seus números chegando aos 7 dígitos. Isto não quer dizer, entretanto, que os SGBD utilizados são os menos adequados a trabalhar com grafos. Conforme visto, as implementações MogwaiJpaMySQL e MogwaiObjectDB possuem desempenhos melhores. As diferenças de desempenho ilustram, na verdade, o impacto dos diferentes usos dos SGBD.

Tabela 4.14: Classificação por posição na bateria de testes 1

Classificação	**Implementação**	**Somatório das posições**
1	MogwaiNeo4J	23
2	MogwaiObjectDB	28
3	MogwaiCypherNoIndex	32
4	MogwaiJpaMySQL	36
5	MogwaiJpaH2	44
6	MogwaiCypherGlobalIndex	54
7	MogwaiJdbcMySQL	59
8	MogwaiCypherGraphIndex	59
9	MogwaiJpaObjectDB	70

A segunda bateria de testes tem um resultado bastante diferente. Enquanto na primeira as implementações que utilizam SGBDR desempenharam mal, na segunda, a implementação com o melhor desempenho foi a que utiliza o SGBDR MySQL

Tabela 4.15: Classificação por tempo total bateria de testes 1

Classificação	**Implementação**	**Somatório das médias dos tempos**
1	MogwaiNeo4J	8.982,00 ms
2	MogwaiCypherNoIndex	12.975,67 ms
3	MogwaiCypherGlobalIndex	29.098,00 ms
4	MogwaiCypherGraphIndex	29.369,67 ms
5	MogwaiJpaMySQL	62.483,33 ms
6	MogwaiJpaH2	71.643,67 ms
7	MogwaiObjectDB	489.009,33 ms
8	MogwaiJdbcMySQL	3.117.492,00 ms
9	MogwaiJpaObjectDB	3.303.420,33 ms

através da JDBC. Ela se sobressaiu ao *baseline* nas três consultas. Isto ilustra como as consultas de grafos diferem significativamente em termos da presença ou não de atributos nos elementos do grafo. São dois casos de uso muito distintos.

Durante a realização dos testes, notou-se que a utilização de *cache* é de suma importância. O desempenho na sua ausência é demasiado prejudicado. Os caches utilizados são apenas locais à *thread* que executa a consulta, ficando qualquer cache global a cargo do SGBD utilizado.

Todas as implementações do Mogwai utilizaram uma abordagem orientada a objetos. Podemos argumentar que as implementações de pior desempenho constituem verdadeiras formas de não implementar um SGBDG, enquanto que as de melhor desempenho sugerem possíveis caminhos a serem adotados, necessitando apenas de melhorias em determinados casos de uso. As deficiências atuais podem eventualmente ser superadas por melhorias diversas, como otimizar pontualmente métodos de consulta, criar planos alternativos de execução, ou até mesmo alterar o algoritmo de correspondência de grafos.

Uma outra possibilidade é a de utilizar múltiplos SGBD armazenando o grafo de forma redundante, e executar a consulta no SGBD mais adequado conforme a necessidade da consulta.

Em suma, os testes demonstraram a aplicabilidade do arcabouço Mogwai. Suas implementações tiveram por vezes desempenho superior, por vezes desempenho compatível, e em alguns casos desempenho bastante inferior ao *baseline*. Ao mesmo tempo, igualmente importante às comparações entre as implementações (ou entre

os SGBD) é a demonstração de que os diferentes casos de uso de grafos demandam diferentes estratégias para obtenção de resultados ótimos. Este fato é evidenciado pelo fato de que nenhuma implementação obtém o melhor resultado em todas as consultas. Ainda neste sentido, é de grande valia comparar diferentes usos de um mesmo SGBD.

O algoritmo de correspondência de padrões de grafos utilizado demonstrou ser efetivo em determinados casos de uso.

Capítulo 5

Conclusões e Trabalhos Futuros

Esta dissertação descreveu o Mogwai, um arcabouço para banco de dados de múltiplos grafos, a MogwaiQL, linguagem descritiva de consulta, e a álgebra a partir da qual a MogwaiQL foi desenvolvida. Foram descritas as características da arquitetura do arcabouço, assim como os detalhes da álgebra e da linguagem de consulta. Para verificar a validade e a viabilidade do arcabouço, diferentes implementações com diferentes tecnologias foram realizadas. Procurou-se utilizar ao máximo os recursos oferecidos pelas tecnologias utilizadas nas diferentes implementações. As consultas dos principais tipos de consultas de grafos foram submetidas ao Mogwai, e também consultas particulares de banco de dados de múltiplos grafos, e os resultados foram exibidos e avaliados.

Conforme discussão na análise dos resultados, a utilização do Mogwai apresenta algumas vantagens quando comparada com outras soluções de banco de dados para grafos. Notoriamente, são poucas as ferramentas que, ao mesmo tempo, trabalham com o modelo de grafo de propriedades, com múltiplos grafos, e que possuem linguagem de consulta, de forma que o Mogwai apresenta-se como alternativa mais adequada para as aplicações que exigem tais recursos do que as que não os possuem todos.

A álgebra proposta possui operações que compreendem os principais tipos de consultas (de adjacência, alcance, correspondência de padrões de grafos, e sumarização) e operações entre grafos. Os resultados das operações são tuplas ou grafos.

Esta última característica é desejável (e não é tão comum), pois quando os resultados são informados apenas na forma de tuplas, o usuário é obrigado a remontar o grafo (ou subgrafo) manualmente (se este for seu interesse, obviamente).

Além disso, a MogwaiQL, por ser uma linguagem de consulta declarativa, alcança os usuários que não estão aptos a escrever programas utilizando as API de outras ferramentas, ao mesmo tempo em que favorece a adoção e difusão das tecnologias de SGBDG devido à sua facilidade de uso. Outros fatores que estimulam esta adoção e difusão são, apesar de não serem capacidades exclusivas, os fatos do arcabouço poder ser executado sobre um SGBDR, de contar com uma interface web para execução de consultas, e de possuir uma ferramenta para importação de dados.

Junto às implementações do Mogwai foi criado um conjunto de testes que testa a quase totalidade das interfaces do arcabouço, o que torna a tarefa de desenvolver uma nova implementação relativamente fácil.

Sobre as implementações do Mogwai, por elas utilizarem a ferramenta Maven, elas são ferramentas fáceis de serem adicionadas a qualquer projeto de sistema em Java (ou linguagens de programação que executam na JVM).

A definição formal da linguagem de consulta a partir de uma álgebra, e a sua implementação sobre o arcabouço com utilização da ferramenta ANTLR, contribuem para o desenvolvimento das linguagens de consulta dos SGBDG e para a difusão deste tipo de tecnologia, pois o seu estudo, o seu uso e a sua extensão são facilitados.

Os experimentos executados nesta dissertação foram realizados com diferentes modelagens e tecnologias de banco de dados. De forma geral, as implementações todas obtiveram desempenho compatível com o SGBDG Neo4J, o que não só comprova a capacidade do arcabouço de ser utilizado como um SGBD de propósito geral, mas também dá força a ideia de que os SGBDG nativos não necessariamente obtém resultados de desempenho melhores do que os não-nativos. Apesar do modelo de dados orientado a objetos ser mais próximo do orientado a grafos, o desempenho da implementação Mogwai ObjectDB não se destacou positivamente das demais, obtendo desempenho similar. Pelo contrário, ao utilizar a modelagem da implementação

JPA, o ObjectDB obteve alguns resultados expressivamente piores. A questão em si não são os resultados desta ferramenta: o importante é destacar que pequenas alterações na modelagem impactam profundamente o desempenho do sistema, de forma que testar outras modelagens constitui um possível trabalho futuro.

Era de se esperar que a implementação que utiliza os dados todos em memória obtivesse o melhor resultado. Mas não foi exatamente o caso. Além disto, ela está limitada à quantidade de memória disponível para a Java Virtual Machine. Um trabalho futuro interessante consiste em manter uma distinção clara entre a estrutura do grafo e suas propriedades, uma vez que a estrutura do grafo pode ser representada utilizando menos memória do que os valores das propriedades, e adaptar os algoritmos para se aproveitar desta divisão. O desempenho da MogwaiNeo4jNoIndex foi superior em diversas consultas, e reforça a ideia de focar a execução das consultas na estrutura do grafo ao invés de nos atributos dos elementos.

A natureza do problema de busca de subgrafos isomórficos, um problema NP, faz com que o processo de redução do espaço de busca seja importante. O arcabouço realiza a poda utilizando os predicados nos atributos dos vértices e arestas da consulta. Entretanto, consultas sem estes predicados ficam sem recursos de otimização. Um outro trabalho futuro consiste em utilizar outros mecanismos de indexação que utilizem outras características do grafo.

Por fim, outros trabalhos futuros incluem executar experimentos com bases de dados maiores (com atributos nos elementos dos grafos), executar o arcabouço de forma distribuída, utilizar algoritmos paralelos, executar as consultas com iteradores, gerar planos de execução de consulta alternativos, implementar o armazenamento nativo de grafos, e tornar o arcabouço compatível com o arcabouço Apache Tinkerpop, que atualmente é a API mais difundida entre os SGBDG comerciais.

Referências Bibliográficas

[1] LESER, U. "A query language for biological networks", *Bioinformatics*, v. 21, n. suppl 2, pp. ii33–ii39, 2005.

[2] ROBINSON, I., WEBBER, J., EIFREM, E. *Graph Databases: New Opportunities for Connected Data*. "O'Reilly Media, Inc.", 2015.

[3] STEFANI, S., TORRIERO, A. "Networks, topology and dynamics", *Quality & Quantity*, v. 48, n. 4, pp. 1817–1819, 2014.

[4] RIAZ, F., ALI, K. M. "Applications of graph theory in computer science". In: *Computational Intelligence, Communication Systems and Networks (CICSyN), 2011 Third International Conference on*, pp. 142–145. IEEE, 2011.

[5] BALABAN, A. T. "Applications of graph theory in chemistry", *Journal of chemical information and computer sciences*, v. 25, n. 3, pp. 334–343, 1985.

[6] OLKEN, F. "Graph data management for molecular biology", *OMICS A Journal of Integrative Biology*, v. 7, n. 1, pp. 75–78, 2003.

[7] "RDF Current Status - W3C". `https://www.w3.org/standards/techs/rdf`, . Accessed: 2017-03-13.

[8] PRUD, E., SEABORNE, A., OTHERS. "SPARQL query language for RDF", 2006.

[9] "RDF - Semantic Web Standards". `https://www.w3.org/RDF/`, . Accessed: 2017-09-09.

[10] EL-JAICK, D., MATTOSO, M., LIMA, A. A. "SGProv: Mecanismo de Sumarização para Múltiplos Grafos de Proveniência." In: *SBBD (Short Papers)*, pp. 17–1, 2013.

[11] ANGLES, R. “A comparison of current graph database models”. In: *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pp. 171–177. IEEE, 2012.

[12] BUERLI, M., OBISPO, C. “The current state of graph databases”, *Department of Computer Science, Cal Poly San Luis Obispo, mbuerli@ calpoly. edu*, v. 32, n. 3, pp. 67–83, 2012.

[13] VERKROOST, Y. “Evaluation of Graph Management Systems for Monitoring and Analyzing Social Media Content with OBI4wan”, 2015.

[14] “The Gremlin Graph Traversal Machine and Language”. `http://tinkerpop.apache.org/gremlin.html`, . Accessed: 2017-01-25.

[15] “What is openCypher?” `http://www.opencypher.org/`. Accessed: 2017-01-25.

[16] “The Gremlin Graph Traversal Machine and Language”. `http://tinkerpop.apache.org/`, . Accessed: 2017-09-09.

[17] “Neo4j: The World's Leading Graph Database”. `https://neo4j.com/`. Accessed: 2017-01-25.

[18] HOLZSCHUHER, F., PEINL, R. “Querying a graph database–language selection and performance considerations”, *Journal of Computer and System Sciences*, v. 82, n. 1, pp. 45–68, 2016.

[19] ELMASRI, R., NAVATHE, S. B. “Sistemas de banco de dados”, 2005.

[20] “Apache Hadoop”. `http://hadoop.apache.org`. Accessed: 2017-01-26.

[21] STONEBRAKER, M. “SQL databases v. NoSQL databases”, *Communications of the ACM*, v. 53, n. 4, pp. 10–11, 2010.

[22] “Neo4j's Graph Query Language: An Introduction to Cypher”. `https://neo4j.com/developer/cypher-query-language/`. Accessed: 2017-03-13.

[23] MAGALHÃES, F. V. D. *Graphene: um Protótipo de Sistema de Gerência de Bancos de Dados Distribídos Orientados a Grafos*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2014.

[24] AGNARSSON, G., GREENLAW, R. *Graph Theory: Modeling, applications, and algorithms*. Prentice-Hall, Inc., 2006.

[25] RODRIGUEZ, M. A., NEUBAUER, P. "A path algebra for multi-relational graphs", *arXiv preprint arXiv:1011.0390*, 2010.

[26] HE, H., SINGH, A. K. "Graphs-at-a-time: query language and access methods for graph databases". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 405–418. ACM, 2008.

[27] PARR, T., FISHER, K. "LL (*): the foundation of the ANTLR parser generator", *ACM SIGPLAN Notices*, v. 46, n. 6, pp. 425–436, 2011.

[28] SHENG, L., OZSOYOGLU, Z. M., OZSOYOGLU, G. "A graph query language and its query processing". In: *Data Engineering, 1999. Proceedings., 15th International Conference on*, pp. 572–581. IEEE, 1999.

[29] CONSENS, M. P., MENDELZON, A. O. "GraphLog: a visual formalism for real life recursion". In: *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 404–416. ACM, 1990.

[30] GIUGNO, R., SHASHA, D. "Graphgrep: A fast and universal method for querying graphs". In: *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, v. 2, pp. 112–115. IEEE, 2002.

[31] "Introduction - ArangoDB v3.0.12 AQL Documentation". `https://docs.arangodb.com/3.0/AQL/`. Accessed: 2017-03-13.

[32] STEINHAUS, R., OLTEANU, D., FURCHE, T. "G-Store: a storage manager for graph data", *Master's thesis, University of Oxford*, 2011.

[33] KERAMOPOULOS, E., POUYIOUTAS, P., SADLER, C. "GOQL, a graphical query language for object-oriented database systems". In: *Information Technology, 1997. BIWIT'97., Proceedings of the Third Basque International Workshop on*, pp. 35–45. IEEE, 1997.

[34] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., etal. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146. ACM, 2010.

[35] "Giraph - Welcome to Apache Giraph!" `http://giraph.apache.org`. Accessed: 2017-01-22.

[36] "GraphX — Apache Spark". `https://spark.apache.org/graphx/`. Accessed: 2017-01-26.

[37] "ObjectDB - Fast Object Database for Java with JPA/JDO support". http://www.objectdb.com/. Accessed: 2017-01-25.

[38] "H2 Database Engine". http://www.h2database.com/html/main.html. Accessed: 2017-01-25.

[39] "MySQL The world's most popular open source database". https://www.mysql.com/. Accessed: 2017-01-25.

[40] "Apache Lucene - Welcome to Apache Lucene". https://lucene.apache.org/. Accessed: 2017-01-25.

[41] "Java JDBC API". https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. Accessed: 2017-03-13.

[42] LEE, J., HAN, W.-S., KASPEROVICS, R., etal. "An in-depth comparison of subgraph isomorphism algorithms in graph databases". In: *Proceedings of the VLDB Endowment*, v. 6, pp. 133–144. VLDB Endowment, 2012.

[43] "What is openCypher?" http://https://martinfowler.com/bliki/PolyglotPersistence.html/. Accessed: 2017-05-30.

[44] WINTER, A., KULLBACH, B., RIEDIGER, V. "An overview of the GXL graph exchange language". In: *Software Visualization*, Springer, pp. 324–336, 2002.

[45] "Spring Boot". https://projects.spring.io/spring-boot/. Accessed: 2017-07-26.

[46] RIESEN, K., BUNKE, H. "IAM graph database repository for graph based pattern recognition and machine learning". In: *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pp. 287–297. Springer, 2008.

[47] "Stanford Large Network Dataset Collection". https://snap.stanford.edu/data/. Accessed: 2017-09-13.

[48] "Stanford Large Network Dataset Collection". https://snap.stanford.edu/data/. Accessed: 2017-09-10.

[49] "Amazon EC2 Instance Types - Amazon Web Services (AWS)". https://aws.amazon.com/ec2/instance-types/. Accessed: 2017-09-13.

Apêndice A

Gramática

```
grammar Mogwai;

@header{
package br.ufrj.coppe.mogwai.parser;
}
prog: command ;
command : (select | match | find | create | insert | connect | update) ';' ;
select: 'select' selectStatements;
selectStatements: graphPattern                                  #SingleSelect
  selectStatements ('UNION' | 'INTERSECT' | 'SUBTRACT')
  selectStatements                                              #SelectOperation
  |  '(' selectStatements ('UNION' | 'INTERSECT' | 'SUBTRACT')
  selectStatements ')'                                          #GroupedSelectOperation
  | selectStatements 'UNIFY' '(' selectStatements ')'
  'ON' unifyExpressions+                                        #SelectUnify
                  ;
find: 'find' ('shortest')? 'path' 'between' value 'AND' value 'on graph' value ;
match: 'match' graphPattern returnStatement?;
returnStatement: 'return' property (',' property)*     #simpleReturn
  | 'group by' groupByProperties 'return'
  returnProperty (',' returnProperty)*                 #groupedReturn;
groupByProperties: groupByProperty (',' groupByProperty)*;
returnProperty: property                               #simpleReturnProperty
  |  AGGREGATEFUNCTION '(' property ') as' ID          #aggregateReturnProperty;
graphPattern: matchStatements whereClause?;
matchStatements: matchStatement (',' matchStatement)*;
matchStatement: leftVertex                             #MatchVertexOnly
  |leftVertex ((inEdge|outEdge|joinEdge) rightVertex)* #MatchVertexAndEdge
```

```
  | '#' ID                                                  #MatchWholeGraph;
whereClause: 'where' whereExpressions+;
whereExpressions: whereExpression
                  | '(' whereExpressions ')'
                  | 'NOT' '(' whereExpressions ')'
                  | whereExpressions booleanOperator whereExpressions;


whereExpression: property OPERATOR property
               | property OPERATOR value
               | property BOOLEANOPERATOR BOOLEAN;

unifyExpressions: unifyExpression
                  | '(' unifyExpressions ')'
                  | 'NOT' '(' unifyExpressions ')'
                  | unifyExpressions booleanOperator unifyExpressions;

unifyExpression: unifyProperty OPERATOR unifyProperty
               | unifyProperty OPERATOR value;

booleanOperator: 'AND' | 'OR';


OPERATOR: '='
        | '<'
        | '<='
        | '>'
        | '>='
        | '!=';

AGGREGATEFUNCTION: 'SUM'
        | 'MAX'
        | 'MIN'
        | 'COUNT'
        | 'AVG';

BOOLEANOPERATOR: ' IS '
        | ' IS NOT ';
```

```
property:   ID '.' ID;
groupByProperty:   ID '.' ID;
unifyProperty:   ('V1' | 'V2') '.' ID;
value:  INT | SHORT | BYTE | LONG | FLOAT | DOUBLE | BOOLEAN | STRINGLITERAL;

leftVertex : '(' ID? (':' ID)?  ('#' value)? ('$' value)? ')';
rightVertex : '(' ID? (':' ID)? ('#' value)? ('$' value)? ')';

outEdge    : '-->'
           | '-[' ID? (':' ID)? pathPattern? ']->'
           ;
inEdge     : '<--'
           | '<-[' ID? (':' ID)? pathPattern? ']-'
           ;

pathPattern: INT '..' (INT | '*');

joinEdge  : '==' ID '==';
retrieve: 'return';
expr: expr ('*'|'/') expr
    | expr ('+'|'-') expr
    | INT
    | '(' expr ')'
    ;


create: 'create (#' value (':' ID)? ')' ('{'
    definingProperty (',' definingProperty)* '}')?;
insert: 'insert (#' value '$' value (':' ID)? ')' ('{'
    definingProperty (',' definingProperty)* '}')?;
connect: 'connect (#' value '$' value ')' '-[' '$' value (':' ID)? ']->'
    '(' '$' value ')' ('{' definingProperty (',' definingProperty)* '}')?;
update: 'update (#' value ('$' value)? (':' ID)? ')' ('{'
    definingProperty (',' definingProperty)* '}')?;
definingProperty: ID ':' value;

NEWLINE : [\r\n]+ ;
INT     : '-'? [0-9]+ ;
SHORT   : '-'? [0-9]+ 'S';
```

```
BYTE    : '-'? [0-9]+ 'B';
LONG    : '-'? [0-9]+ 'L';
FLOAT   : '-'? [0-9]+ '.' [0-9]+ 'F' ;
DOUBLE  : '-'? [0-9]+ '.' [0-9]+ 'D' ;
BOOLEAN : 'TRUE' | 'FALSE';
STRINGLITERAL
        : '\''
        (  EscapeSequence
        | ~('\\' | '\'' | '\r' | '\n')
        ) *
        '\'';

fragment
EscapeSequence
        : '\\' ('b' | 't' | 'n' | 'f' | 'r' | '\'' | '\"' | ('0'..'3')
        ('0'..'7') | ('0'..'7')('0'..'7')|('0'..'7'));
ID      : ('a'..'z'|'A'..'Z'|'_'|[0-9])+;
WS      : (' ' | '\t' | '\n') {skip();};
```