UFAM - Engenharia da Computação

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS ORIENTADOS A GRAFOS

Robson Rojas Andrade

Monografia de Graduação apresentada à Coordenação de Engenharia da Computação, UFAM, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Engenheiro de Computação.

Orientadores: Eduardo James Pereira
Thiago de Souza Rocha

Manaus

Agosto de 2015

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS ORIENTADOS A GRAFOS

Robson Rojas Andrade

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO CURSO DE ENGENHARIA DA COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DO AMAZONAS COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO.

Aprovada por:

Prof. Eduardo James Pereira , D.Sc.

Prof. Thiago de Souza Rocha , M.Sc.

Prof. Celso Barbosa Carvalho, D.Sc.

Prof. Eduardo Luzeiro Feitosa, D.Sc.

MANAUS, AM – BRASIL
AGOSTO DE 2015

Andrade, Robson Rojas

Análise de Desempenho de Bancos de Dados Orientados a Grafos/Robson Rojas Andrade. – Manaus: UFAM, 2015.

XIII, 57 p.: il.; 29, 7cm.

Orientadores: Eduardo James Pereira

Thiago de Souza Rocha

Monografia (graduação) – UFAM / Curso de Engenharia da Computação, 2015.

Referências Bibliográficas: p. 49 – 51.

1. Bancos de dados baseados em grafos. 2. performance. 3. hpc-sgab. 4. benchmark. I. , Eduardo James Pereira *et al.*. II. Universidade Federal do Amazonas, UFAM, Curso de Engenharia da Computação. III. Título.

À minha mãe Dominga.

Agradecimentos

- Agradeço a Deus em primeiro lugar, pois tem me acompanhado, me animado e me conduzido neste objetivo pessoal, sem o qual não estaria neste mundo e não teria conhecimento da sua graça.

- Agradeço à minha esposa Adriana David da Silva, que me dá apoio, motivação e consolo nos momentos de desânimo.

- Agradeço aos meus pais Raimundo dos Santos Andrade e Dominga Ercilete Bernardo Rojas pelos conselhos e pelas orações e preocupações mesmo distantes.

- Agradeço aos meus irmãos Dercio, Dercilan, Douglas e Rayane pelo incentivo e motivação.

- Agradeço a todos os professores, especialmente ao professor Eduardo Souto (Orientador) e ao Thiago Rocha (Co-orientador), que buscaram me apoiar, orientar e motivar para a finalização deste trabalho.

- Agradeço a todos os meus amigos da faculdade e de fora da faculdade, dentre os quais Bruno Auzier, Osmar Rubert, Cristóvão Braga, Ivanildo dos Santos, Pastor Wilson Dias, Pastora Franciane, Tayana, Elôa, William, Ana Simões e todos os irmãos em cristo que me apoiaram neste esforço.

- Agradeço à Universidade Federal do Amazonas pela oportunidade de realização deste curso.

Resumo da Monografia apresentada à UFAM como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS ORIENTADOS A GRAFOS

Robson Rojas Andrade

Agosto/2015

Orientadores: Eduardo James Pereira
Thiago de Souza Rocha

Programa: Engenharia da Computação

Sistemas de banco de dados orientados a grafos necessitam buscar, armazenar e processar dados de forma veloz. Existem muitos sistemas de armazenamento no mercado, portanto, é necessário realizar a comparação entre eles. Para isso podemos usar *benchmarks* que especificam vários casos de teste, que permitem analisar e comparar a maior gama possível de critérios de desempenho entre os sistemas e assim determinar o melhor, permitindo que todo o sistema tenha desempenho e custo otimizados. Com esse objetivo, neste trabalho foi desenvolvido um software, baseado na especificação do *benchmark hpc-sga (high performance computation for scalable graph analysis)*, para realizar testes sobre bancos de dados orientados a grafos. A partir dos resultados dos testes foi concluído que, que o sistema *Neo4j v.2.1.6* obteve melhor desempenho quando comparado ao sistema *Titan v. 0.4.4*. Também foram realizados estudos sobre os conceitos de grafos, bancos de dados orientados a grafos e *NoSQL*, no contexto de *Big Data*.

Abstract of Monograph presented to UFAM as a partial fulfillment of the requirements for the degree of Engineer

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS ORIENTADOS A GRAFOS

Robson Rojas Andrade

August/2015

Advisors: Eduardo James Pereira
Thiago de Souza Rocha

Department: Computer Engineering

Graph-oriented database systems need to search, store and process data fast way. As there are many storage systems on the market it is necessary to make a comparison between them. For this we can use benchmarks that specifies various test scenarios that allow users to analyze and compare the widest possible range of performance criteria between systems and thus determine the best, allowing the entire system to be with a optimized performance cost. A software was developed based on the specification of HPC-SGA benchmark (high performance computation for scalable graph analysis), to carry out tests on graph databases. From the test results it was concluded that the Neo4j system performed better when compared to the Titan system. Studies were also conducted on the concepts of graphs, graph databases and NoSQL in the Big Data context.

Sumário

Lista de Figuras

Lista de Tabelas

Abreviações

API - ***A**plication **P**rogramming **I**nterface*

CSV - ***C**omma **S**eparated **V**alues*

HPC-SGAB - ***H**igh **P**erformance **C**omputation for **S**calable **G**raph **A**nalysis **B**enchmark*

JSON - ***J**ava **S**cript **O**bject **N**otation*

NoSQL - ***N**ot **O**nly **S**ql*

RDBMS - ***R**elational **D**ata **B**ase **M**anagement **S**ystems*

ROI - ***R**eturn **O**f **I**nvestment*

R-MAT - ***R**ecursive **M**Matrix*

SQL - ***S**tructured **Q**uery **L**anguage*

TEPS - ***T**raversed **E**dges **P**er **S**econd*

XML - ***E**xtensible **M**arkup **L**anguage*

Capítulo 1

Introdução

Nas últimas duas décadas, os dispositivos de armazenamento de dados se desenvolveram de tal forma que o preço de 1 Megabyte de armazenamento passou de 18,51 centavos para 0,00633 centavos de dólar [1]. De igual modo, os processadores tornaram-se mais velozes[2]. A união desses fatores com a popularização da Internet levou empresas e pessoas a produzir e consumir cada vez mais dados.

Empresas como Google, Facebook e Amazon geram diariamente volumes de dados da ordem de terabytes por hora [3]. Enquanto isso, bilhões de usuários navegam na web, publicando e acessando vídeos, artigos, imagens e músicas em uma escala sem precedentes, abrindo caminho para sistemas de informação em escala web. Sistemas de comércio eletrônico, infraestrutura de TI, redes sociais, recomendação de produtos, processamento distribuído de dados, *streaming* de mídia, web 2.0, entre outros, têm sido responsáveis por uma demanda crescente por capacidade de armazenamento e processamento de informações [4].

Este aumento contínuo e acelerado no volume e variedade de dados desperta grande interesse nas empresas, gerando demanda por tecnologias que sejam capazes de extrair informações que auxiliem na tomada de decisões, melhorando produtos, serviços e as vendas.

Assim surgiu o *Big Data*, termo usado para se referir ao conjunto de tecnologias usadas para capturar, integrar, organizar e analizar grandes e variados volumes de dados [5]. Entre estas, destacam-se o *MapReduce* [6] e o *BigTable* [7], ambos da Google. O *MapReduce* é uma interface de alto nível que abstrai os detalhes de uma infraestrutura de processamento distribuído de dados, permitindo a implemen-

tação transparante de sistemas para processamento de grandes volumes de dados [8]. O *BigTable* é um sistema de armazenamento distribuído projetado para gerenciar dados estruturados, que possui a capacidade de acompanhar naturalmente o crescimento do volume de dados, suportando volumes da ordem de petabytes [7].

Estes exemplos mostram que, dentro do conjunto de tecnologias *BigData*, existem algumas que servem como ferramentas para processamento e análise de dados (*MapReduce*) e outras como armazenamento (*BigTable*). Assim, o *BigTable* faz parte de uma nova classe de banco de dados chamados *NoSQL*, uma abreviação para *Not Only SQL* [9]. O *NoSQL* consiste em modelos de armazenamento de dados que não seguem somente a filosofia *SQL* e são projetados para grandes e variados volumes de dados.

Apesar da existência destas tecnologias, ainda pode-se utilizar bancos de dados relacionais (*Relational Data Base Management Systems - RDBMS*) para lidar com estes grandes volumes de dados. Com o uso adequado de particionamento e replicação, o sistema poderia permitir o aumento da capacidade de armazenamento. Esta poderia ser uma boa opção, já que em uma migração para *NoSQL* existe um investimento em infraestrutura e capacitação que pode não ter o retorno esperado.

Porém, quando existe uma demanda constante por escalabilidade, o Modelo Relacional não é recomendado. As técnicas usadas para adequá-lo a esta demanda são custosas e complexas, pois o modelo usa estruturas formadas por tabelas e relacionamentos para modelar e armazenar dados (esquemas). Tais esquemas são difíceis de modificar e foram projetados para lidar somente com classes de dados estruturados, sendo difícil conciliar tal modelo com a demanda por escalabilidade [9]. Por outro lado, os bancos de dados *NoSQL* foram concebidos com suporte nativo à escalabilidade.

Quanto ao modelo de dados, os bancos de dados *NoSQL* são classificados em quatro modelos de armazenamento:

- Armazenamento por chave/valor: nesse modelo existe uma associação única entre uma coleção de chaves únicas e valores [9];
- Armazenemento orientado a colunas: os dados são armazenados em colunas de atributos, que podem formar famílias de colunas (repositórios de colunas) e super colunas [9];

- Armazenemento orientado a documentos: os documentos são a unidade básica de armazenamento, que podem ser coleções de pares de chaves-valor [9]; e
- Armazenamento orientado a grafos[10]: os dados são armazenados nas propriedades dos vértices e arestas de um grafo [9].

As características de cada modelo serão descritas com mais detalhes na seção 2.2.1. Cada modelo possui características que o torna mais adequado para determinados tipos de dados. Quando se trata de dados conectados ou relacionados, em que existe necessidade de alto desempenho em consultas que levariam a muitas junções no modelo relacional, o armazenamento orientado a grafos se apresenta como melhor alternativa [11].

No modelo relacional, para recuperar dados de um registro é necessário uma consulta *SQL* que realiza a junção de várias tabelas a fim de produzir um resultado com os dados requeridos. Enquanto isso, no modelo orientado a grafos não é necessário realizar junções, pois os dados relacionados estão armazenados em vértices, distantes entre si de apenas algumas arestas[11].

Outra vantagem do modelo em grafo é a sua flexibilidade na modelagem de dados, permitindo que novos tipos de relacionamentos, vértices e propriedades possam ser criados em tempo de execução [12]. Essa é uma característica interessante para aplicações que estejam em constante mudança na estrutura do modelo de dados, permitindo ao banco de dados acompanhar de perto a evolução da aplicação durante o desenvolvimento, sem grandes complicações na remodelagem.

Existem muitas opções entre os sistemas de bancos de dados orientados a grafos, cada um com suas características. Portanto, são necessários experimentos que permitam a análise e a comparação entre eles a fim de decidir qual a melhor alternativa.

Entretanto, atualmente existem poucos sistemas ou *benchmarks* disponíveis para a realização de experimentos sobre bancos de dados orientados a grafos. Dentre estes, o *HPC-SGAB (High Performance Computation for Scalable Graph Analysis Benchmark)* [13] foi selecionado por ser uma especificação desenvolvida a fim de realizar testes de desempenho, contemplando as operações mais representativas sobre grafos, como: carga de dados, busca, navegação e percorrimentos[13].

1.1 Justificativa

A escolha do sistema de armazenamento de dados é uma parte crucial no desenvolvimento de um sistema de software. Pois, no caso de uma decisão errada, o sistema pode apresentar desempenho indesejado, levando o desenvolvedor a substituir o banco de dados por outro, acarretando em mais custos financeiros e de tempo para todos os interessados. Em especial, isso pode prejudicar as relações entre cliente e desenvolvedor, pois o primeiro poderá decidir buscar outro parceiro de negócios. Portanto, é desejável que antes de utilizar um sistema de armazenamento, este esteja testado, analisado e comparado com outros a fim de determinar o mais adequado para lidar com os problemas do cliente.

1.2 Objetivos

O principal objetivo deste trabalho consiste na análise comparativa entre dois sistemas de bancos de dados orientados a grafos, Titan [14] e Neo4j [15], a fim de determinar o *GDB* com melhor desempenho.

Como objetivo secundário, definimos:

- Desenvolvimento de um software, a partir da especificação do *HPC-SGAB*, para a realização de experimentos de desempenho sobre os bancos de dados.

1.3 Metodologia

Para atingir os objetivos deste trabalho serão conduzidos estudos dos conceitos de grafos, *Big Data*, *NoSQL*, bancos de dados baseados em grafos e da especificação do *HPC-SGAB*.

Com o conhecimento adquirido, será desenvolvido o programa gerador de dados, assim como o software, com base nas especificações do *HPC-SGAB*, para a realização dos experimentos.

Com o sistema de experimentação desenvolvido, serão conduzidos experimentos sobre os bancos de dados *Neo4j* e *Titan*, com consequente geração de dados para análise. Com esses dados, serão realizadas análises e comparações entre os resultados para determinar o sistema com o melhor desempenho.

1.4 Organização da Monografia

Esta monografia está organizada da seguinte forma:

- O Capítulo 2 descreve os conceitos básicos relacionados a este trabalho. Serão detalhados os conceitos de *Big Data*, *NoSQL*, modelagem e armazenamento de dados usando grafos e Bancos de Dados orientados a Grafos;
- O Capítulo 3 apresenta um resumo dos trabalhos relacionados com análise de desempenho e *benchmarks* para bancos de dados orientados a grafos;
- O Capítulo 4 descreve o *HPC-SGAB* e o processo de desenvolvimento do software de experimentação;
- O Capítulo 5 apresenta os resultados do trabalho, descrevendo o ambiente e os experimentos realizados, as análises dos dados e comparações entre os bancos de dados;
- O Capítulo 6 apresenta as conclusões do trabalho com considerações e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, são apresentados os conceitos de *Big Data*, *NoSQL* , grafos, bancos de dados orientados a grafos e suas características.

2.1 Big Data

Big Data é definido como um grupo de tecnologias usadas para processar e armazenar conjuntos de dados com características especiais, como: alta velocidade no processamento, alto volume de armazenamento, alta variedade de fontes e estruturas de dados e, com menor ênfase, veracidade e valor gerado a partir dos dados [5]. Cada característica será explicada a seguir.

A alta velocidade no processamento é resultado da captura e disponibilização dos dados quase em tempo real [16]. A exploração desta característica pode dar vantagem competitiva no mercado a uma organização, pois, quanto mais rapidamente as informações são processadas e disponibilizadas, mais cedo serão tomadas as decisões estratégicas.

Para lidar com a velocidade, surgiram tecnologias como *stream processing* e *MapReduce*. O *stream processing* consiste em realizar processamento sobre uma corrente contínua e massiva de dados não estruturados [5]. Já o *MapReduce* consiste em uma interface de programação, em que os dados são divididos em unidades pequenas, distribuídos sobre uma rede de nós de processamento e então os resultados de cada nó são reunidos para compor o resultado final [8].

O alto volume está relacionado com o crescimento exponencial dos dados

disponíveis em forma digital, que são provenientes não só de sistemas convencionais, mas também de fontes como *web*, *RFID*, *smartphones*, sensores de diversos tipos, etc [17]. Assim, são muitas as fontes de dados, contendo quantidades imensas de informação, impondo desafios ao seu armazenamento e recuperação.

A alta variedade envolve as diversas fontes de dados que (no mesmo domínio) podem ser extremamente heterogêneas tanto no nível de esquema, sobre como são estruturados, quanto no nível da instância, sobre como elas descrevem a mesma entidade do mundo real, exibindo considerável variedade mesmo para entidades que são substancialmente similares [16]. Nestas fontes de informação, os dados podem ser estruturados, não estruturados ou semi-estruturados, como em arquivos *XML (Extensible Markup Language)* e *JSON (Java Script Object Notation)*, que são padrões de formatação de dados do tipo chave/valor e chave/documento, respectivamente.

Para lidar com o alto volume e a alta variedade de fontes e tipos de dados, surgiu o *NoSQL* [9], termo que designa sistemas de armazenamento projetados para escalarem naturalmente e para serem pouco estruturados, a fim de aumentar a capacidade de armazenamento e suportar variados tipos de dados.

A veracidade está relacionada ao fato de que os dados não são "perfeitos", no sentido de que é preciso considerar o quão bons eles devem ser para que gerem informações úteis e também os custos para torná-los confiáveis [16]. Este aspecto envolve a garantia de que os dados fazem sentido, são autênticos, verdadeiros e precisos.

O valor refere-se àquilo que o *Big Data* oferece para seu negócio ou vida pessoal [18]. Para o investidor, isso se traduz no retorno do investimento (*Return Of Investment - ROI*). Para as pessoas, envolve produtos e serviços de melhor qualidade e preços mais acessíveis.

Para gerar valor no *Big Data*, é necessário investir no seu uso analítico e na criação de novos produtos a partir da análise dos dados processado [16]. O uso analítico pode levar a percepções de informações que estavam escondidadas pelo alto custo em processar tantas informações. Já o processamento pode revelar informações importantes sobre as demandas dos consumidores, permitindo oferecer produtos e serviços mais próximos das suas necessidades, sempre considerando a confiabilidade das informações geradas.

As características do *Big Data* costumam ser abordadas por mais de uma tecnologia, sendo difícil delimitar o espaço de atuação das diversas ferramentas, havendo casos onde uma tecnologia pode ser aplicada em diferentes características do *Big Data*.

2.1.1 Aplicações

Existem muitos exemplos de aplicações das tecnologias *Big Data* na literatura como: melhoria de tratamentos de saúde, *business inteligence*, redes sociais, gerenciamento de manufatura, gerenciamento de tráfego, detecção de ataques em redes de computadores, detecção de tendências no mercado e nos consumidores, entre outras. Vejamos alguns:

- Em um experimento conduzido pelo grupo de pesquisa do Prof. Alex Pentland, do *MIT Media Lab*, foram capturados dados relativos à localização de celulares de forma a inferir quantas pessoas colocaram seus carros nos estacionamentos de lojas do grupo americano *Macy's* no *Black Friday* de 2011 (data que marca o início da temporada de compras de Natal nos Estados Unidos). Isso permitiu estimar com precisão as vendas dessas lojas antes mesmo que elas ocorressem, gerando vantagens competitivas nas áreas comerciais, de marketing e de terceiros, como investidores em bolsas de valores [5].

- O *Facebook* armazena e usa *Big Data* na forma de perfis de usuários, fotos, mensagens e publicidades. Analisando esses dados, a companhia é capaz de entender seus usuários e descobrir quais conteúdos exibir para eles [18].

- O *Google* rastreia bilhões de páginas na web e tem uma vasta lista de outras fontes de *Big Data* como *Google Maps*, que contém um imenso volume de dados incluindo localizações físicas de ruas, imagens de satélites, fotos de rua, e mesmo visualizações internas de muitas construções. Todas essas informações o ajudam a direcionar publicidade para os consumidores mais interessados [18].

- O *LinkedIn* hospeda milhões de resumos online e contém o conhecimento sobre como pessoas estão conectadas umas com outras. A companhia usa todos esses dados para sugerir o subconjunto de pessoas com quem nós poderemos

nos conectar e para nos mostrar atualizações relevantes sobre nossos amigos e colegas [18] .

Em meio as várias tecnologias existentes no *Big Data*, se destacam os bancos de dados *NoSQL*, que como já explicado anteriormente, foram projetados para armazenar grandes volumes de dados e para serem flexíveis com relação às variedades de tipos e estruturações dos dados. Visto que o escopo deste trabalho envolve bancos de dados *NoSQL*, na seção a seguir será realizada uma apresentação mais aprofundada desta tecnologia.

2.2 NoSQL

O termo *NoSQL* foi primeiramente usado em 1998 para referir-se a uma base de dados relacional que não usaria o *SQL (Structured Query Language)* [19]. Entretanto, as formas de armazenamento não relacionais são anteriores ao surgimento do termo e também ao surgimento do modelo relacional.

O armazenamento não relacional foi primeiramente usado no armazenamento de dados pouco estruturados nos sistemas de autenticação do *Active Directory*, em que os dados dos usuários são armazenados em estruturas do tipo chave/valor [20]. Posteriormente, o modelo relacional foi proposto para lidar com o problema do armazenamento de dados vindos dos antigos sistemas de arquivos de dados digitais, que armazenavam dados na forma de fichas de dados organizados em pastas, vindo a se tornar a forma de armazenamento predominante [21].

Naquele contexto, não se pensava em lidar com dados de aplicações distribuídos geograficamente, processamento paralelo massivo, escalabilidade, grandes volumes de dados e dados não estruturados. Entretanto, com o advento das necessidades do *Big Data*, os bancos de dados *NoSQL* ressurgiram como proposta para atender a estas necessidades. A tabela 2.1 apresenta um resumo das diferenças entre os bancos de dados relacionais e *NoSQL* com relação às necessidades do *Big Data* [22].

Tabela 2.1: Modelo Relacional x *NoSQL* com relação ao *Big Data*

	Relacional	NoSQL
Escalonamento	Possível, mas complexo. Devido à natureza estruturada do modelo, a adição de forma dinâmica e transparente de novos nós num grid de armazenamento não é realizada de modo natural	Uma das principais vantagens desse modelo. Por não possuir nenhum tipo de esquema pré-definido, estes bancos de dados favorecem a inclusão transparente de outros nós de armazenamento.
Consistência	Ponto mais forte do modelo relacional. As regras de consistência presentes propiciam um maior grau de rigor quanto à consistência das informações.	Realizada de modo eventual: garante que, se nenhuma atualização for realizada sobre o item de dados, todos os acessos a esse item devolverão o último valor atualizado.
Disponibilidade	Dada a dificuldade de se conseguir trabalhar de forma eficiente com a distribuição dos dados, este modelo pode não suportar uma demanda muito grande de requisições de informações sobre o banco.	Outro fator fundamental do sucesso desse modelo. O alto grau de distribuição dos dados permite que um maior número de solicitações sejam atendidas aumentando a disponibilidade do sistema.
Desempenho	Cai muito quando o volume de dados a ser processado é grande ou as consultas realizam muitas junções.	Se mantém linear para alguns bancos de dados como os orientados a grafos.
Modelagem de dados	Esquemas rígidos, formados por tabelas e relacionamentos.	A modelagem não é rígida e os modelos de dados são livres de esquemas.
Garantias transacionais	Garantias *ACID(Atomicity, Consistency, Isolation, Durability)* para consistência e segurança dos dados.	*BASE (Basic Availability, Soft-state, Eventual consistency)*, relaxamento das garantias do *ACID*.
Linguagem de consulta	*SQL*	*Gremlin, SPARQL, CYPHER*. Um dos pontos fracos destes modelos, pois não possuem uma linguagem padrão de consulta.

No contexto *Big Data*, um dos apelos do *NoSQL* frente ao modelo relacional é o suporte à escalabilidade horizontal. Esta característica permite a adição de novos nós de armazenamento, formando um sistema distribuído, em que cada nó pode responder às requisições de dados das aplicações, o que aumenta a disponibilidade dos dados.

Contudo, o *NoSQL* não deve ser encarado como um substituto aos bancos de dados relacionais, pois a escolha pelo modelo relacional ou *NoSQL* depende dos requisitos do sistema. Quando a consistência dos dados é uma necessidade do sistema, bancos de dados relacionais são recomendados. Pois, neste aspecto, eles são maduros e bem estabelecidos [9]. Quando a disponibilidade dos dados e a escalabilidade são as prioridades, o *NoSQL* é mais recomendado. Além disso, sistemas que não operam com volumes de dados da ordem de *Terabytes* e sistemas que não necessitem de armazenamento distribuído podem usar normalmente bancos de dados relacionais.

2.2.1 Os tipos de NoSQL

Existem diversos critérios de classificação e de comparação dos bancos de dados *NoSQL*. Apresentaremos uma classificação baseada no modelo de dados, pois é mais comumente utilizada. De acordo com essa classificação, as formas de armazenamento dos bancos de dados *NoSQL* dividem-se em:

- Orientado a chave/valor: mantém um conjunto grande de pares de chaves/valor. A chave do par é um valor único no conjunto e permite que os dados sejam rapidamente acessados pela chave, principalmente em sistemas que possuem alta escalabilidade [11]. Costumam ser utilizados em sistemas de *caching* e permitem particionamento, replicação e versionamento dos dados. Exemplos: *Berkeley DB*, *Amazon's Dynamo* e *Project Voldemort* [19].

- Orientado a colunas: os dados são armazenados em uma estrutura orientada a colunas. Esta estrutura permite armazenamento efetivo dos dados diminuindo o consumo de espaço em disco, pois não há armazenamento de valores nulos. Uma característica interessante são as famílias de colunas, conceito que é usado para agrupar colunas que armazenam o mesmo tipo de dados [11]. O modelo permite particionamento e forte consistência dos dados. O maior exemplo é o

banco de dados desenvolvido pelo *Google*, chamado de *BigTable* [23]. Outros exemplos são: *Apache Cassandra* e *Apache Hbase*.

- Orientado a documentos: armazenam e recuperam documentos. Esses bancos de dados confiam na manutenção de índices, na maioria, para facilitar o acesso aos documentos através dos seus atributos. Os documentos tendem a formar listas e mapas, permitindo uma hierarquia natural que pode ser representada usando *JSON (JavaScript Object Notation - Notação de Objetos JavaScript)* ou *XML (eXtensible Markup Language - Linguagem de Marcação Extensível)* [11]. Este modelo de armazenamento não depende de um esquema rígido sendo a modelagem de dados altamente flexível. Exemplos: *Apache CouchDB*, *MongoDB*.

- Orientado a grafos: são sistemas de banco de dados com operações sobre os dados que expõem um modelo de dados em grafo [12]. Apresenta ganho de performance quando as consultas a serem realizadas são complexas, custosas e realizadas sobre elementos interconectados ou em rede. Exemplos: *Neo4j*, *Titan*, *Sparksee*, *AllegroGraph* e *Virtuoso*.

Além da classificação por modelo de dados, os bancos de dados *NoSQL* também podem ser classificados pela distribuição de dados. Nesta classificação, o particionamento e a replicação dos dados podem ser realizados no sistema ou no cliente; e pelo modo de armazenamento dos dados, em que os bancos mantêm suas informações em memória e/ou em disco [9].

A despeito das diversas classificações e modelos de dados, não é possível defender a superioridade de um tipo de armazenamento sobre outro em todos os cenários. Porém, existem cenários mais adequados para cada categoria, por exemplo: bancos orientados a colunas são mais adequados quando há necessidade de alta disponibilidade e consistência; bancos orientados a documentos e a chave-valor podem ser adequados para manipulação de dados estatísticos e para sistemas de *caching*; e bancos orientados a grafos são recomendados quando as consultas são complexas e necessitam ter alto desempenho sobre dados interconectados.

Visto que o foco principal deste trabalho está na execução de testes sobre bancos de dados orientados a grafos é importante que o entendimento sobre esta

forma de armazenamento seja aprofundado, o que será realizado na próxima seção.

2.3 Modelagem e armazenamento de dados em grafos

Para facilitar o entendimento do armazenamento orientado a grafos são apresentados nesta seção os conceitos de grafos, os tipos e modelos de grafos usados como base para a construção desses bancos de dados.

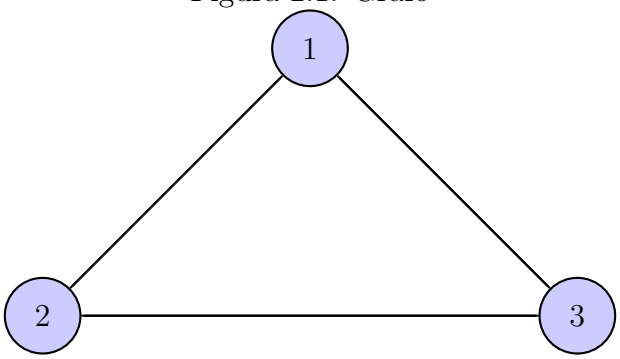
2.3.1 Grafo

Um grafo pode ser definido como um par $(V(G), E(G))$, onde $V(G)$ é um conjunto finito, não vazio, de vértices e $E(G)$ é um conjunto finito, não ordenado, de arestas distintas de $V(G)$ [24]. São estruturas de propósito geral que nos permitem modelar todos os tipos de cenários, como: construção de foguetes espaciais, sistemas de estradas, cadeias de suprimentos, históricos médicos para populações, problemas de logísticas para transporte de cargas, cálculo de rotas entre dois pontos ou qualquer outro problema onde existam objetos interconectados [12].

Os grafos podem ser classificados de várias maneiras, entre elas, pela direção das arestas e pela topologia. Quanto á direção da aresta entre dois vértices, um grafo pode ser:

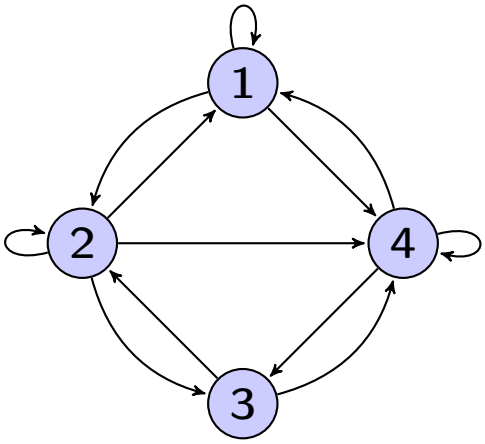
- Não direcionado: suas arestas não tem uma direção definida. Veja o exemplo na Figura 2.1.

Figura 2.1: Grafo



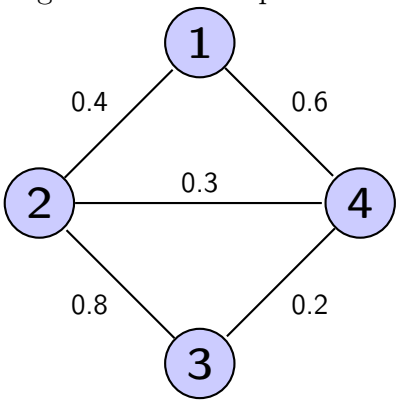
- Direcionado ou dígrafo: possui arestas (A) que apontam para uma direção a partir de um vértice (V). Conforme o exemplo na Figura 2.2.

Figura 2.2: Grafo direcionado



Em ambos os casos, as arestas podem conter pesos (grafo ponderado) que podem ser usados para indicar o custo de uma ligação entre dois vértices, de acordo com a Figura 2.3.

Figura 2.3: Grafo ponderado



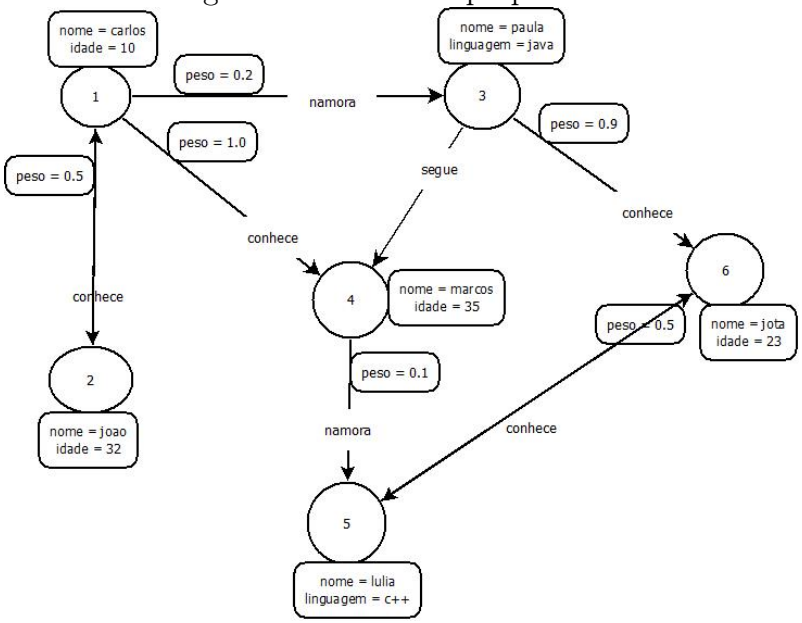
Quanto á topologia um grafo pode ser:

- Rotulado: Todas as arestas tem um rótulo que indica o tipo de aresta. Uma extensão deste tipo permite que até mesmo os vértices sejam rotulados [25].
- Multi-grafo: É um grafo onde podem existir múltiplas arestas entre dois vértices [26].
- Atribuído: Os elementos do grafo podem ter atributos anexados a elas a fim de conter dados adicionais. Os atributos são expressados usualmente como uma lista de chaves/valor [25].

- Grafo de propriedades: é definido como um multi-grafo direcionado atribuído. É o modelo implementado por vários bancos de dados orientados a grafos por permitir mais flexibilidade e representatividade aos dados armazenados [12].

As três topologias são geralmente utilizadas nos sistemas de armazenamento orientados a grafos. Especialmente o grafo de propriedades e o multi-grafo são utilizados no *Titan* e *Neo4j*, respectivamente. Na Figura 2.4 temos um exemplo de grafo de propriedades. Este exemplo representa uma pequena rede social, onde todas as arestas e vértices tem rótulos, alguns vértices tem múltiplas arestas entre si e tanto vértices como arestas tem atributos, ou seja, é um grafo de propriedades.

Figura 2.4: Grafo de propriedades



A seção sequinte fará uma apresentação mais aprofundada sobre os sistemas que utilizam grafos para armazenar dados.

2.3.2 Bancos de dados orientados a grafos

Como foi dito anteriormente, bancos de dados orientados a grafos expõem para as aplicações um modelo de dados em grafo, permitindo grande flexibilidade na modelagem e permitindo que técnicas de programação e algoritmos voltados para grafos sejam executados no banco de dados.

Entretanto, nem todo banco de dados orientado a grafos armazena os dados

em uma estrutura de grafo real. Portanto, devemos conhecer os elementos importantes da estrutura interna desses banco de dados, que são:

- O armazenamento: muitos *GDBs* (*graph databases*) usam armazenamento nativo em grafo. Porém, alguns apenas serializam os dados do grafo em bancos de dados relacionais, orientados a objetos ou algum outro tipo de armazenamento *NoSQL* [12].

- O processamento: alguns sistemas implementam o conceito de "adjacência livre de índice" (do inglês *index-free adjacency*). Isso significa que vértices conectados "apontam" fisicamente uns para os outros no banco de dados. Isso acrescenta significante ganho de performance para o banco de dados em operações que percorrem caminhos através dos relacionamentos entre vértices [12].

É evidente o ganho de desempenho quando o banco de dados orientado a grafos implementa armazenamento nativo e a adjacência livre de índices. Também está claro o poder das técnicas de modelagem que usam grafos. Porém, isso não é justificativa suficiente para substituir uma plataforma de dados bem estabelecida e bem entendida; deve existir um benefício imediato e muito significativo [12]. Portanto, é necessário analisar os pontos positivos e negativos do modelo orientado a grafos e validar contra as necessidades dos clientes e suas aplicações a fim de ter bons resultados. Tomando como referência os bancos de dados relacionais, podemos citar como pontos positivos os seguintes itens:

- Desempenho com dados interconectados: Comparado aos bancos de dados relacionais, os bancos de dados em grafos apresentam um grande ganho de performance quando usados em operações sobre elementos relacionados. Isso ocorre por que as consultas são localizadas a uma porção do grafo, não necessitando usar junções de tabelas, em parte devido à adjacência livre de índices [12].

- Flexibilidade de modelagem: O modelo de dados em grafo permite representar o domínio do problema de forma fiel sem a necessidade de acrescentar metadados, como as chaves primárias do modelo relacional, e nem de ignorar informações existentes no domínio quando o banco de dados não suporta

certas informações, por exemplo, dados não estruturados. Assim, a estrutura dos dados que for modelada para o armazenamento emerge facilmente com o crescimento do entendimento do escopo do problema [12].

- Agilidade na remodelagem: A baixa esquematização e a flexibilidade do modelo de dados em grafo facilita grandemente a modificação da estrutura de armazenamento e permite a adição de novos tipos de vértice, relacionamentos e até subgrafos à estrutura pré-existente sem que ela seja prejudicada ou tenha que ser modificada [12]. Dessa forma, o armazenamento em grafo é capaz de acompanhar a evolução das aplicações conforme seus requisitos mudam.

Como pontos negativos, temos:

- Consumo de memória: Como já dito anteriormente, alguns bancos de dados orientados a grafos implementam o conceito de adjacência livre de índices, onde o relacionamento entre dois vértices aponta diretamente para o outro nó em memória. Isso aumenta consideravelmente o consumo de memória, pois os dados carregados irão permanecer em memória. Em casos onde a quantidade de memória é limitada, é preferível utilizar um banco de dados relacional ou orientado a colunas, que armazenam os dados em disco.

- Voltado para dados conectados: Os bancos de dados orientados a grafos são mais adequados para armazenar dados que possam ser organizados de forma interconectada ou em rede. Também são mais adequados para operações que busquem encontrar padrões de relacionamentos entre indivíduos no grafo. Se, por exemplo, os dados tem origem em sistemas de cadastro de formulários, é mais recomendado o uso de bancos de dados relacionais.

2.4 Sistemas de banco de dados orientados a grafos

Nesta seção são descritos os bancos de dados orientados a grafos testados e analisados neste trabalho. Foram selecionados para avaliação o *Titan* e o *Neo4j*. O primeiro por ser um banco de dados recente e *opensource*, e o segundo por ser um sistema já maduro e ter desempenho comprovado em outros trabalhos.

2.4.1 Titan

O *Titan* é uma base de dados distribuída otimizada para armazenamento de grafos contendo bilhões de vértices. Os dados do grafo podem ser distribuídos sobre um *cluster* de máquinas que pode escalar facilmente para suportar o crescimento do volume de dados e clientes. Pode suportar centenas de clientes concorrentes executando percorrimentos complexos em grafo [14]. Implementa a pilha *Tinkerpop*, suportando realização de consultas através da linguagem *Gremlin*, acesso ao sistema através de *web services* com o servidor de grafo *Rexster* e da *API Blueprints*. Resultando em uma arquitetura de armazenamento plugável [14].

Entre as muitas funcionalidades e características destacam-se as seguintes [14]:

- Escalabilidade elástica e linear para crescimento de dados e clientes;
- Distribuição e replicação de dados para desempenho e tolerância a falhas;
- Alta disponibilidade através de múltiplos datacenters;
- Suporte a transações *ACID* (Atomicidade, Consistência, Isolamento, Durabilidade) e (*BASE (Basically Available, Soft state, Eventually consistency - Basicamente disponível, Estado Leve, Eventualmente consistente)*;
- Suporte a *backends* de armazenamento;
- Código aberto;
- Vários níveis de configuração para melhoria de desempenho;
- Índices nos vértices do grafo, que permitem consultas em nível de vértices para aliviar problemas como o *super node problem* (vértices com número desproporcionalmente grande de arestas incidentes);

Entre os vários *backends* de armazenamento se destacam os seguintes:

- *Apache HBase*: garante consistência e particionabilidade, enquanto não garante alta disponibilidade;

- *Apache Cassandra*: garante alta disponibilidade e particionabilidade, mas não garante consistência;

- *Oracle BerkleyDB*: garante consistência e alta disponibilidad, mas não garante particionabilidade.

Funcionamento

O sistema de armazenamento expõe para as aplicações uma estrutura de grafos de propriedades (*graph-property-store*), enquanto que o processamento expõe operações de percorrimentos e navegação no grafo (ex. usando a linguagem *Gremlin traversal*). A união das duas tecnologias é chamada de computação baseada em grafo (do inglês *graph-based-computing*), que é a tecnologia com a qual funciona o *Titan*.

Entretando, o armazenamento dos dados não é realizado em uma estrutura de grafo nativa, sendo em vez disso usados sistemas de terceiros para armazenar os dados. O *Titan* pode suportar várias *engines*, sendo elas classificadas de acordo com o modo de armazenamento, em:

- Memória: os dados são mantidos em memória e o tamanho do grafo é limitado pelo tamanho da *RAM* da máquina. Normalmente funcionam sobre uma única máquina. Ex: *jung* e *networkx*.

- Disco: os dados são mantidos em disco e são carregados na memória conforme a demanda ou de acordo com a política da *engine*. O tamanho do grafo é limitado pelo tamanho do disco local. Normalmente são *engines* otimizadas para algoritmos de grafo local. Exemplos: *OrientdB*, *Neo4j Community* e *Sparksee*.

- *Cluster*: os dados podem ser armazenados tanto em disco como em memória. O limite de armazenamento depende do tamanho total do espaço em disco do *cluster*. A *engine* é otimizada para a execução de algoritmos de grafo global. Exemplos: *Goldenorb*, *Harma*, *Apache HBase*, *Apache Cassandra*, *Neo4j Enterprize* e *Sparksee*.

Além disso, as operações de percorrimento expostas também são abstrações das reais operações realizadas sobre os bancos de dados usados como *backend*. Por-

tanto, a forma real como as operações são realizadas depende do *backend* utilizado, seja para armazenar dados ou para percorrer os vértices do grafo.

As principais formas de realizar operações sobre o grafo no *Titan* são:

- *Gremlin*: uma linguagem funcional, voltada para operações transversais, ou seja, que permitem realizar percorrimentos sobre as arestas e vértices do grafo;
- *Blueprints API*: também implementadada nativamente pelo *Titan*, permitindo que uma aplicação utilize qualquer uma das ferramentas da pilha do *TinkerPop*.

2.4.2 Neo4j

O *Neo4j* é um banco de dados orientado a grafos *open-source* projetado e mantido pela *Neo Tecnology*. Os dados são armazenados em uma estrutura de vértices diretamente conectados, em que os relacionamentos possuem propriedades, também conhecido como grafo de propriedades [12]. Pode ser visto como um "quadro branco amigável", onde um projeto criado com caixas e linhas em um quadro branco pode ser facilmente convertido numa estrutura de armazenamento.

O foco do *Neo4j* é maior nas relações entre valores do que sobre os pontos em comum entre conjuntos de valores (tais como coleções de documentos ou tabelas de linhas). Deste modo, ele pode armazenar dados altamente variáveis de uma forma fácil e natural [27].

De forma geral, o *Neo4J* possui características semelhantes ao *Titan* e é caracterizado por ser um sistema [12]:

- Intuitivo: usando um modelo em grafo para representação de dados.
- Seguro: com suporte a transações *ACID* completas.
- Durável e rápido: usando um armazenamento nativo baseado em disco.
- Massivamente escalável: para bilhões de vértices/relacionamentos/propriedades.
- Altamente disponível: quando distribuído através múltiplas máquinas (Na versão *Enterprize*).

- Expressivo: com uma linguagem de consulta em grafo poderosa (*CYPHER*) e humanamente legível.

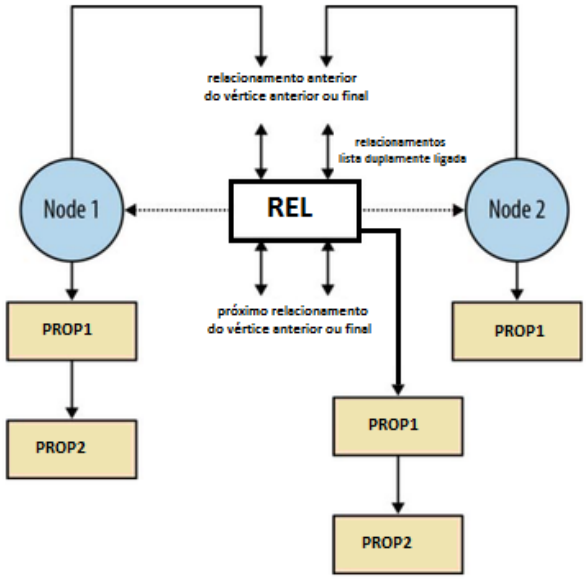
Funcionamento

O sistema de armazenamento expõe para as aplicações uma estrutura de grafos de propriedades e processamento por percorrimentos no grafo (ex. usando a linguagem *CYPHER*), suportando a computação e armazenamento nativo em grafo e adjacência livre de índices.

Os três principais componentes do grafo (vértices, arestas e propriedades dos elementos) são armazenados em três arquivos separados. Vértices são armazenados com um ponteiro para a sua primeira aresta e primeira propriedade. Propriedades são armazenadas como um grupo de listas ligadas, onde há uma lista ligada para cada vértice. Finalmente, arestas são armazenadas em duas listas duplamente ligadas, juntamente com o tipo da aresta e um ponteiro para a primeira propriedade [25].

O exemplo da Figura 2.5 permite visualizar esta estrutura, onde os vértices (NODE 1 e 2) apontam para uma lista de propriedades e apontam para a lista duplamente encadeada onde estão armazenados os relacionamentos, em que cada relacionamento (REL) aponta para uma lista de propriedades (PROP1 e PROP2).

Figura 2.5: Exemplo da organização interna entre vértices, relacionamentos e propriedades do Neo4j



As operações sobre o grafo podem ser realizadas seguindo os relacionamentos ou usando as funcionalidades de caminhamento declarativo para recuperar os dados

desejados. O sistema de armazenamento suporta o conceito de transações *ACID*, mas esse suporte é deteriorado quando o banco de dados é distribuído sobre um cluster de servidores, pois a propriedade consistência do *ACID* é substituída pela consistência eventual do *BASE*.

Para permitir estas operações, o *Neo4j* implementa nativamente a linguagem *CYPHER*, que é uma linguagem declarativa. Existem outras linguagens, entretanto o *Neo4j* recomenda o uso desta. Além desta *API* existem ainda:

- *API Kernel*: permite ao código do usuário ser notificado sobre as transações que ocorrem através do *kernel* do banco de dados, e assim reagir de acordo com o conteúdo e estágio das transações;

- *API Core*: *API* imperativa em Java que expõe as primitivas dos vértices, relacionamentos e propriedades do grafo para o usuário. Permite que os desenvolvedores façam ajustes finos sobre suas consultas de forma a ter alta afinidade com o grafo subjacente, significando que uma consulta usando esta *API* pode ser mais rápida que outras abordagens;

- *API Traversal*: *API* declarativa Java que permite ao usuário especificar um conjunto de regras que limitam os elementos do grafo que o percorrimento pode visitar (tipos de relacionamento, direção, etc).

- *API Blueprints*: também implementadada nativamente pelo *Neo4j*.

As três primeiras *APIs* tornam o código mais dependente da estrutura interna do banco de dados. Por esse motivo, o *Neo4j* recomenda que sejam usadas somente em casos em que o *CYPHER* não puder atender às necessidades da aplicação, pois em caso de mudanças na estrutura interna do banco de dados o sistema desenvolvido com o *CYPHER* não é afetado.

Capítulo 3

Trabalhos relacionados

Neste capítulo, são descritos cinco trabalhos que abordam a comparação do desempenho de bancos de dados orientados a grafos. Embora exista uma lista extensa de estudos nessa área, somente foram selecionados aqueles que apresentam objetivos semelhantes com este. Cada trabalho é descrito de acordo com suas características mais relevantes e ao final é realizada uma comparação das características mais importantes.

Em [28], é proposto um *benchmark* que compara a performance de bancos de dados orientados a grafos através de operações transversais, ou seja, que caminham sobre o grafo visitando seus nós e arestas. O objetivo do *benchmark* é medir o desempenho dos bancos de dados em realizar operações transversais. Foram implementados dois grupos de operações:

- Operações que testam a habilidade de um banco de dados orientado a grafos de executar operações transversais a partir de um único vértice no grafo, através de operações de: carregamento dos dados no grafo; cálculo do coeficiente de clusterização local; e percorrimento via *breadth-first* transversal, com três saltos em 10 mil vértices escolhidos randomicamente a partir de um vértice dado;

- Operações que armazenam informações em memória. O objetivo é testar o percorrimento do grafo inteiro usando a detecção de componentes conectados seguindo as arestas de saída e de entrada de cada nó. Durante a execução desta operação foram armazenadas informações intermediárias de duas formas: na

própria estrutura de dados do grafo na forma de propriedades e em memória externa ao banco de dados.

Os autores não deram um nome para o *benchmark* proposto, por isso tomamos a liberdade de chamá-lo de *TraversalBench*. O *TraversalBench* foi desenvolvido na linguagem Java, utilizando a *API Blueprints*. Os experimentos foram realizados sobre uma máquina local com processador *Intel* de dois núcleos com 2.4GHz e 2 gigabytes de RAM.

Foram testados cinco bancos de dados: *Neo4j*, *DEX*, *OrientDB*, *NativeSail* e *SGDB* (um protótipo de pesquisa). Os resultados mostram que o *SGDB* teve o melhor desempenho, com o *Neo4j* em segundo e o *NativeSail* em terceiro.

Em [29], é proposto o *XGDBench*, um *framework* de *benchmarking* de bancos de dados orientados a grafos que explora as capacidades de sistemas *multi-core* para melhorar o tempo de geração de grafos, bem como para realizar grandes cargas de trabalho sobre estes. No *benchmark* foram implementadas operações de carga de trabalho transversais, foram caracterizadas as performances de servidores de bancos de dados orientados a grafos e foram realizadas operações sobre bancos de dados distribuídos.

O *benchmark* foi desenvolvido usando a linguagem de programação *open-source X10*, que se propõe a prover um modelo de programação robusto para lidar com os desafios impostos nos sistemas *multi-core*, na aceleração de hardware, nos *clusters* e supercomputadores. Para gerar os dados do grafo, foi utilizado o modelo de dados em grafo chamado *Multiplicative Attribute Graph (MAG)*, uma abordagem para modelar estruturas de redes com atributos nos nós.

Foram testados os seguintes sistemas: *OrientDB*, *AllegroGraph*, *Neo4j*, *Fuseki* e *Titan*. Os experimentos foram realizados no *Tsuname*, um ambiente de computação em nuvem. Os resultados mostraram que o *OrientDB* teve o melhor desempenho, seguido do *Titan* e *AllegroGraph* em segundo, e com o *Neo4j* e *Fuseki* em último. Os autores explicaram que o *OrientDB* obteve este resultado por que foi utilizada a *API* Java nativa do próprio banco, enquanto que nos outros foi utilizado um serviço web *HTTP/REST* que gerou um gargalo na comunicação.

Em [30], é apresentado o *GDB*, um *framework* de comparação de bancos de dados orientados a grafos distribuídos.

Neste *framework*, é disponibilizada uma interface para que o usuário defina seu próprio *benchmark*. Este *benchmark* contém uma lista de bancos de dados para comparar e uma série de operações (*workloads*) para serem realizadas sobre os bancos selecionados. O *benchmark* é executado pelo módulo *Operacional*, que inicializa os bancos de dados e mede o tempo requerido para realizar as operações especificadas. Por fim, é gerado um sumário dos resultados.

O *benchmark* foi desenvolvido na linguagem Java, através da pilha *Blueprints TinkerPop*, para permitir uma comparação sobre a mesma base evitando qualquer favorecimento entre os sistemas. O *GDB* pode ser distribuído em multiplos computadores para simular qualquer número de clientes concorrentes enviando requisições para um banco de dados.

O servidor e os bancos de dados foram executados em uma mesma máquina, enquanto que os clientes foram executados em outra, com ambas as máquinas fazendo parte da mesma rede local. Todos os bancos de dados foram executados de dentro da *Java Virtual Machine* e não foram realizados ajustes para melhorar a performance dos bancos de dados.

Foram testados os seguintes sistemas: *Neo4j 1.9*, *DEX 4.7*, *Titan 0.3* e *OrientDB 1.3*. Os resultados mostraram que o *Neo4j* teve o melhor resultado com *workloads* transversais. Para *workloads* de apenas leitura, *Neo4j*, *DEX*, *Titan-Berkeley* e *OrientDB* tireram resultados similares. Para *workloads* de leitura-escrita, *DEX* e *Titan-Cassandra* tiveram melhor resultado.

Em [25], é proposto o *BlueBench*, um *benchmark* para realizar comparação e análise de bancos de dados orientados a grafos.

O *BlueBench* é dividido em três *benchmarks* padrões que executam um conjunto de operações individuais sobre os bancos de dados em testes. As operações são selecionadas de um grupo de operações pré-estabelecidas, que buscam refletir a maioria das operações que devem ser suportadas nos sistemas de bancos de dados orientados a grafos e que incorporam tanto operações simples quanto complexas.

O *benchmark* foi desenvolvido na linguagem Java usando a interface *Blueprints* da pilha *Tinkerpop* para acessar aos bancos de dados. Todos os bancos de dados testados foram executados dentro da *Java Virtual Machine*, a fim de garantir que haja maior precisão e acurácia dos resultados.

Foram testados os seguintes sistemas: *DEX*, *InfiniteGraph*, *Neo4j*, *OrientDB* e *Titan*. Dentre estes, a análise dos resultados concluiu que o *DEX* e o *Neo4j* tiveram os melhores resultados.

Em [13], foi utlizada a especificação do *benchmark HPC-SGA (High Performance Computation for scalable graph analysis)* para realizar comparação e análise de bancos de dados orientados a grafos.

O *benchmark* é dividido em quatro núcleos (*kernels*) que realizam operações sobre o banco de dado testado. Todas as operações são temporizadas e este tempo é usado como critério de comparação entre os bancos de dados. No quarto *kernel*, também é calculada a velocidade de visitação de arestas sobre o grafo chamada de *TEPS - Traversed Edges Per Second*.

O *benchmark* foi desenvolvido na linguagem Java, usando as interfaces nativas de cada sistema para acessar os grafos e realizar as operações. Todos os bancos de dados testados foram executados de forma embarcada a fim de evitar problemas relacionados com tráfego de rede.

Foram testados os seguintes sistemas: *DEX 3.0*, *HypergraphDB 1.0*, *Neo4j 1.0* e *Jena 2.6.2*. A análise dos resultados concluiu que o *DEX* e o *Neo4j* são os melhores bancos de dados orientados a grafos.

A Tabela 3.1 resume algumas características dos trabalhos apresentados. Estas características não envolvem a abordagem ou a forma como foram avaliados os bancos de dados.

O estudo bibliográfico realizado e as informações apresentadas na tabela 3.1 mostram que, embora existam muitos trabalhos que procuram avaliar o desempenho dos bancos de dados orientados a grafos, infelizmente estes trabahos nem sempre convergem para uma metodologia padrão, havendo diferenças na quantidade e na forma como os sistemas são avaliados. Os resultados também não são unânimes, embora *Neo4j* e *DEX* tenham tido o melhor resultado mais vezes, há casos em que um sistema obteve péssimos resultados em um trabalho e em outro obteve o melhor resultado. Estas diferenças tornam necessária a realização de estudos adicionais, pois os resultados encontrados nestes trabalhos não são suficientes para determinar o melhor sistema. Nesse sentido, foram observados alguns elementos importantes para a realização de experimentos, que são:

Tabela 3.1: Resumo de características

Nome	**Linguagem**	**API**	**Sistemas avaliados**	**Melhor sistema**
TraversalBench [28]	*Java*	*Blueprints*	Neo4j, DEX, OrientDB, NativeSail e SGDB	*SGDB*
XGDBench [29]	*X10*	*Nativa e HTTP/REST*	*OrientDB, AllegroGraph, Neo4j, Fuseki e Titan*	*OrientDB*
GDB [29]	*Java*	*Blueprints Tinkerpop*	*Neo4j 1.9, DEX 4.7, Titan 0.3 e OrientDB 1.3*	*Neo4j*
BlueBench [25]	*Java*	*Blueprints Tinkerpop*	*DEX, InfiniteGraph, Neo4j, OrientDB e Titan*	*DEX e Neo4j*
HPC-SGAB [13]	*Java*	Nativa	*DEX 3.0, HypergraphDB 1.0, Neo4j 1.0 e Jena 2.6.2*	*DEX*

- Todo experimento deve executar operações transversais (percorrimentos, navegações ou caminhamentos) sobre o grafo, por ser este um requisito a ser atendido nestes sistemas;
- Deve-se usar uma *API* comum para acessar o banco de dados, a fim de evitar favorecimento a um dos sistemas em teste através do uso de sua *API* nativa. Neste caso, destaca-se a pilha *Blueprints Tinkerpop*;
- Deve-se usar as configurações de fábrica de cada sistema em teste, evitando ajustes ou otimizações;
- Os grafos devem ser construídos com dados gerados a partir de modelos que melhor representem os dados do mundo real.

Neste trabalho procuramos aplicar estas informações, pois são muito importantes e ajudam a melhorar a confiabilidade e a precisão dos resultados.

Capítulo 4

O *Benchmark HPC-SGA*

Nesta capítulo são descritos a arquitetura e o desenvolvimento do software, baseado na especificação do *Benchmark HPC-SGA (High Performance Computation for Scalable Graph Analysis)*, para realização do testes sobre os bancos de dados. Também são descritas as ferramentas utilizadas para implementar o software e os experimentos realizados.

4.1 Introdução

Este *benchmark* foi escolhido por ter sido desenvolvido por pesquisadores acadêmicos, com a participação de membros de várias companhias industriais, a fim de realizar as operações mais representativas sobre bancos de dados orientados a grafos, com o objetivo de determinar o seu desempenho [13].

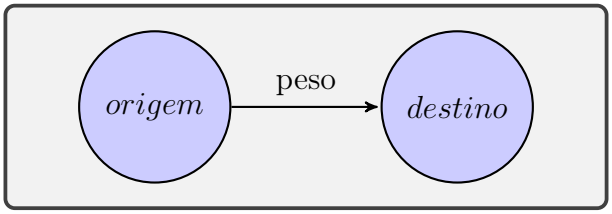
O *benchmark* conta com múltiplos cenários de teste (*kernels*), que acessam uma estrutura de dados simples representando um grafo direcionado e ponderado (com pesos). São quatro *kernels* mais um gerador escalável de dados. A seguir, serão descritos os componentes do *benchmark*.

4.1.1 Modelo de dados

O *benchmark HPC-SGA* utiliza o modelo de dados exibido na figura 4.1, sobre o qual todas as operações são realizadas. Neste modelo, um nó do grafo pode ser uma *origem* ou *destino*. Os nós *origem* têm uma ou mais arestas de saída e os nós de *destino* tem uma ou mais arestas de entrada. Cada nó tem uma propriedade

chamada *id* que o identifica e cada aresta tem uma propriedade *peso* que indica a força do relacionamento entre dois nós.

Figura 4.1: Modelo de dados do *HPC-SGAB*



4.1.2 Gerador de dados

O gerador produz uma lista de arestas em que cada aresta contém o vértice inicial, o vértice final e o peso da aresta, de acordo com o modelo de dados descrito anteriormente. Ele implementa o algoritmo *Recursive Matrix (R-MAT) scale-free graph generation algorithm* [31] e usa uma matriz de adjacências que é subdividida recursivamente em quatro partições de tamanho igual. Nestas partições os pesos das arestas são distribuídos usando probabilidades *a, b, c e d* diferentes e pré-definidas. A recursão termina quando não é mais possível subdividir em novas partições.

Durante a geração de dados, o algoritmo pode gerar ciclos no grafo e os arcos podem conter alto grau de localidade, por isso o gerador deve realizar permutações randômicas desses arcos e os *kernels* do *benchmark* deverão ignorar esses ciclos. As arestas geradas serão posteriormente inseridas no grafo. Para calcular o número de vértices (N), o número de arestas(M) e o máximo peso inteiro de uma aresta(C), são usadas as expressões a seguir:

$$N = 2^{SCALE}$$

$$M = 8 * N$$

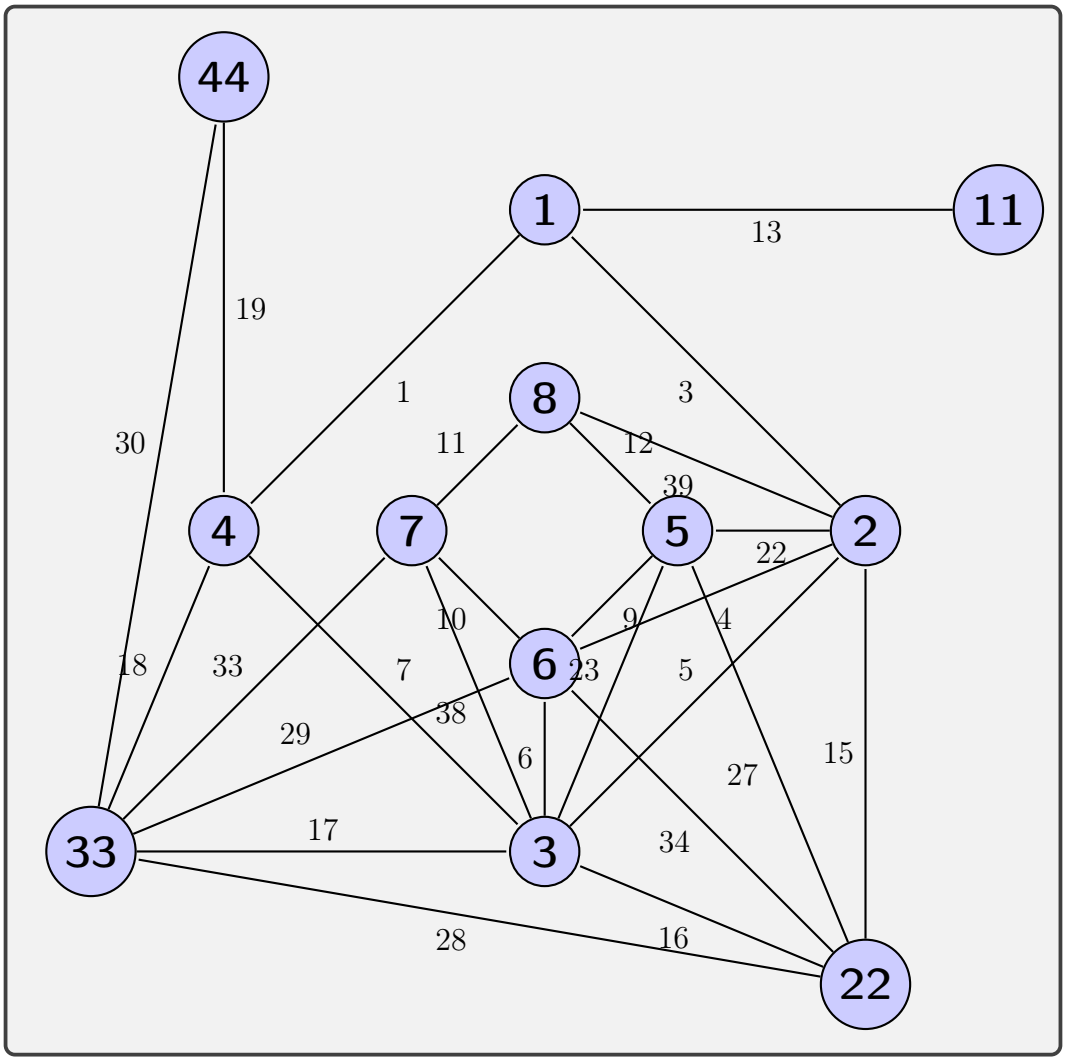
$$C = 2^{SCALE}$$

4.1.3 Kernel 1

Este *kernel* tem a função de medir o tempo necessário para popular o banco de dados com as arestas produzidas no gerador, formando um grafo esparso. Este

grafo pode ser representado de qualquer forma, mas não pode ser modificado nos próximos *kernels*. Esta operação pode ser utilizada quando é necessário inserir dados no sistema a partir de outras fontes, como arquivos *XML*, *CSV (Comma Separated Values)* ou outros bancos de dados.

Figura 4.2: Exemplo de grafo gerado construido pelo kernel 1 com SCALE=4

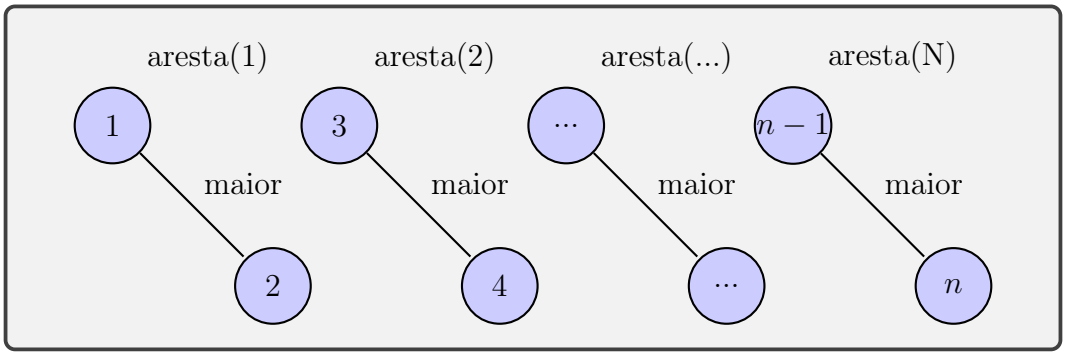


4.1.4 Kernel 2

Com o grafo criado pelo *kernel 1*, este *kernel* mede o tempo necessário para examinar todos os pesos das arestas do grafo para extrair uma lista de arestas, cujos pesos são os maiores pesos de todas as arestas do grafo. Veja um exemplo na figura 4.3. Esta lista poderá, excepcionalmente, conter apenas uma aresta com o maior valor, mas normalmente haverá mais de uma aresta. Este processo pode ser útil quando se deseja consultar os pares de vértices com maior afinidade dentro do grafo.

Por exemplo, em uma rede social existem as pessoas com maior número de interações entre si, esse grupo de pessoas pode ser útil para a propagação de publicidade.

Figura 4.3: Exemplo de lista de arestas extraídas pelo kernel 2



4.1.5 Kernel 3

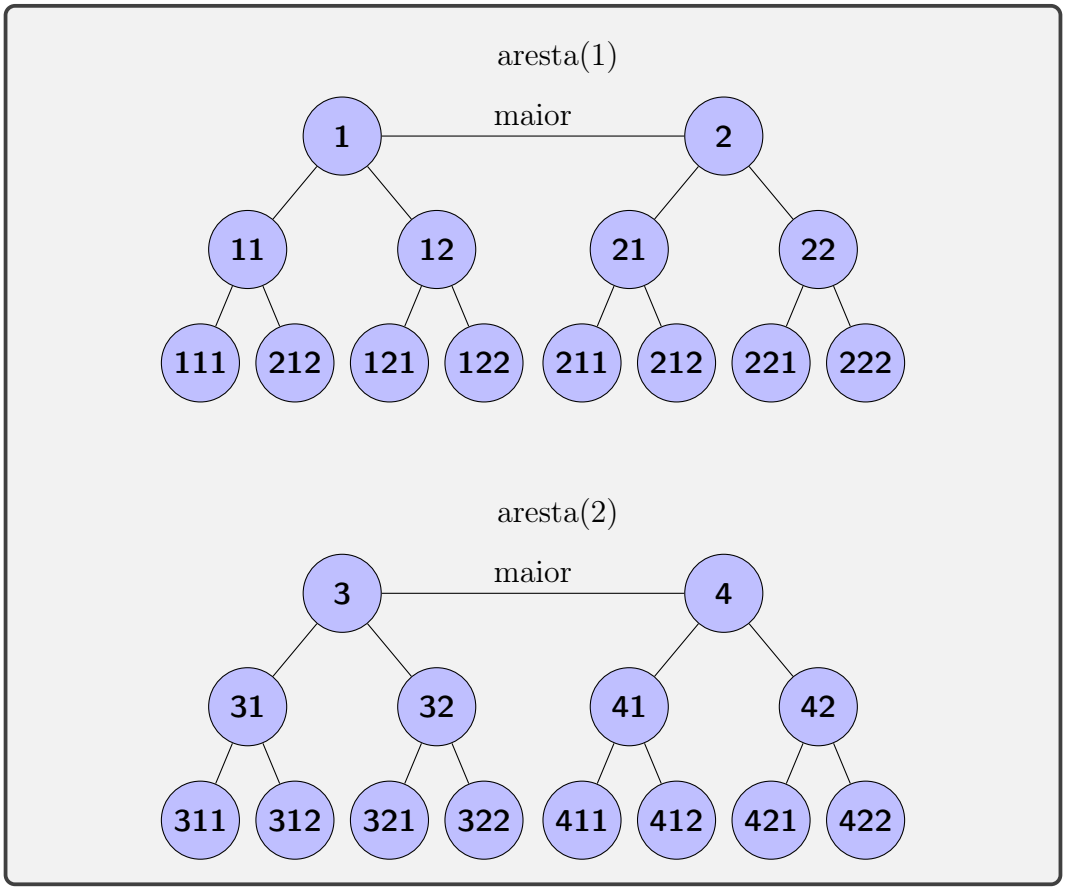
Este *kernel*, mede o tempo necessário para produzir um subgrafo de comprimento fixo, para cada vértice da lista de arestas geradas no *kernel 2*. Veja um exemplo na figura 4.4. Tomando como base os indivíduos com maior afinidade do *kernel 2*, esta operação buscará as pessoas que mantém comunicação mais próxima com aqueles indivíduos, que serão alcançadas pelas publicidades.

4.1.6 Kernel 4

A partir do grafo criado no *kernel 1*, este *kernel* mede o tempo necessário para identificar o conjunto de vértices no grafo com o maior valor de centralidade de intermediação (*Betweness Centrality*). O tempo medido é utilizado para estimar o valor de *TEPS* (Número de Arestas Percorridas por Segundo - *Traversed Edges Per Second*). Na figura 4.5 vemos um exemplo de um grafo onde os vértices estão destacados conforme a centralidade.

A centralidade de intermediação mede o controle de um vértice sobre a comunicação na rede e pode ser usada para identificar os principais intervenientes na rede [32]. Por exemplo, no cenário econômico, um ator (vértice) conecta indiretamente diversos grupos de atores na economia que não são diretamente ligados entre si; para um ator acessar outro em setor distante da rede, terá de estabelecer um caminho que muito provavelmente passará por indivíduos com elevada centralidade [33]. A

Figura 4.4: Exemplo de lista de subgrafos extraídos pelo kernel 3 com os vértices 1, 2, 3 e 4 da lista extraída pelo kernel 2



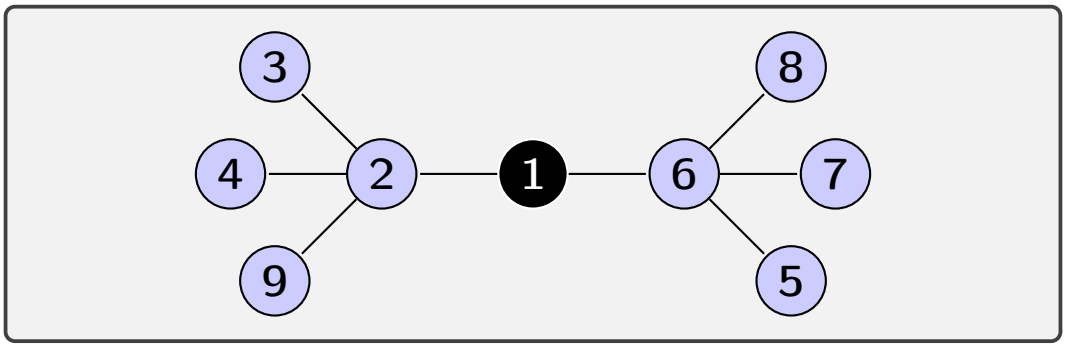
intermediação de um vértice pode ser calculada usando a expressão:

$$C_B(v) = \sum_{s!=v!=t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Onde σ é o número total de caminhos curtos do nó s ao nó t e $\sigma_{st}(v)$ é o número desses caminhos que passam por v. A intermediação envolve o cálculo dos caminhos mais curtos entre todos os pares de vértices de um grafo, que requer um tempo $O(V^3)$ com o algoritmo de *Floyd-Warshall*. No entanto, em grafos dispersos, o algoritmo de *Johnson* pode ser mais eficiente, tendo tempo $O(V^2 * \log(V) + V * E)$. Para grafos não ponderados, os cálculos podem ser feitos com o algoritmo de *Brandes* [34], o que leva o tempo $O(V * E)$. Estes algoritmos assumem que os grafos estão sem direção, conectados sem ciclos e sem arestas múltiplas.

Para diminuir o custo computacional do *kernel* 4, é recomendado o uso de um filtro de arestas e o uso de amostras de nós do grafo. O filtro exclui da computação os arcos cujos pesos não são divisíveis por 8 e a amostragem usa o parâmetro *K4approx*

Figura 4.5: Exemplo de grafo, em que o vértice 1 tem maior centralidade que os outros



para determinar o tamanho da amostra de acordo com a expressão $V_s = 2^{K4approx}$. O *kernel 4* é executado em cada amostra e o resultado ($time_{k4}(N, V_s)$) será usado para calcular o valor de *TEPS* ($TEPS(N, V_s)$), a fim de determinar com que tamanho de amostra seu valor se estabiliza. A *TEPS* é estimada usando a expressão [35]:

$$TEPS(N, V_s) = \frac{7 * N * V_s}{time_{k4}(N, V_s)}$$

4.2 Framework Blueprints

O *Blueprints* é uma coleção de interfaces, implementações e suites de teste para um modelo de grafo de propriedades. É análogo ao *JDBC*, mas para bancos de dados orientados a grafo. Como tal, ele provê um conjunto comum de interfaces para permitir a desenvolvedores usar seus *backends* de bancos de dados orientados a grafos de forma *plug-and-play*. Além disso, os softwares desenvolvidos sobre o *Blueprints* funcionam sobre todos os bancos de dados em grafo que suportam o *framework*. Dentro da pilha de software *Tinkerpop* [36], um *framework* de computação em grafo, o *Blueprints* serve como a tecnologia de base para várias ferramentas criadas para auxiliar em operações sobre bases de dados orientadas a grafos, desde servidores de bancos de dados de grafos até linguagens de consulta [36].

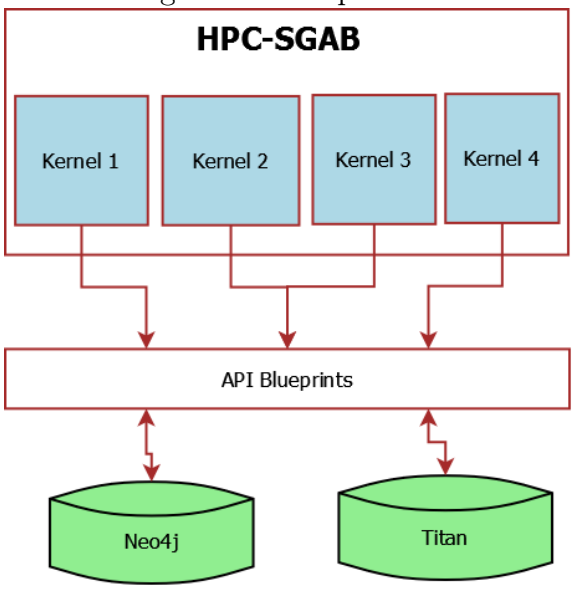
Tanto o *Neo4j* quanto o *Titan* suportam o *Blueprints* e seu *Modelo de dados para grafos*. Por isso, o *Blueprints* foi escolhido para ser usado no desenvolvimento do software de testes com base no *HPC-SGAB*. Isso permitirá que as operações sejam implementadas e executadas da mesma forma independentemente do banco de dados testado, evitando assim que haja qualquer vantagem referente à utilização

da *API* nativa do banco de dados.

4.3 Implementação do *HPC-SGAB* para *Titan* e *Neo4j* em *Java* com a *API Blueprints*

A Figura 4.6 apresenta a arquitetura do sistema desenvolvido para realizar os testes sobre os bancos de dados *Neo4j* e *Titan*. Os quatro *kernels* do *benchmark* acessam e realizam operações sobre os dois bancos de dados, através da *API Blueprints*.

Figura 4.6: Arquitetura



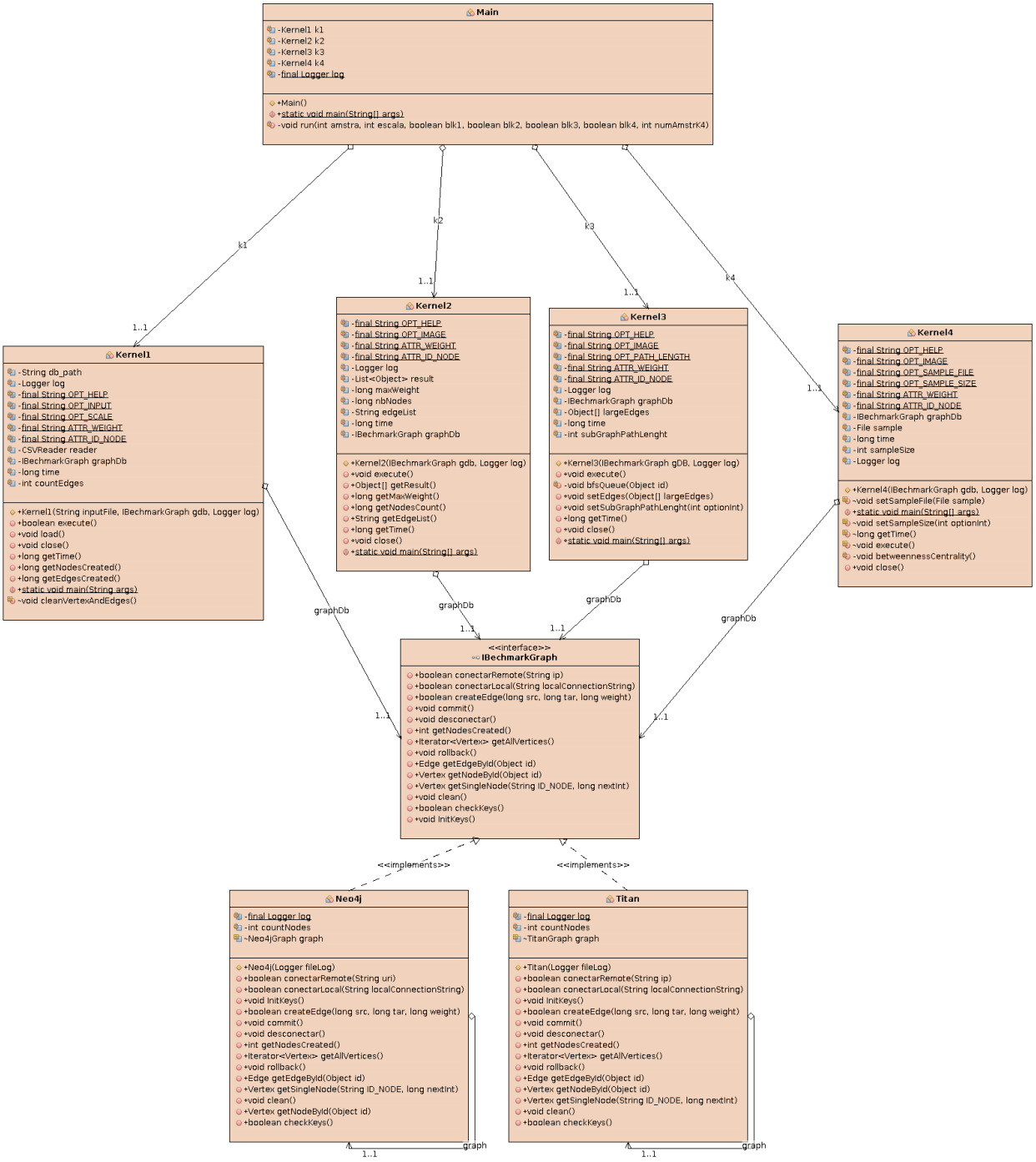
A Figura 4.7 apresenta o diagrama de classes do sistema desenvolvido.

As classes *Kernel1*, *Kernel2*, *Kernel3* e *Kernel4* implementam as operações da especificação do *HPC-SGAB*. Foi utilizada como base a implementação disponível em [37]. Esta foi desenvolvida em Java utilizando as *APIs* nativas de cada banco de dados testado, sendo portada para o nosso sistema que utiliza a *API Blueprints*.

Cada classe de *kernel* acessa o banco de dados através da interface *IBenchmarkGraph*. As classes *Neo4j* e *Titan* implementam esta interface e acessam ao banco de dados para realizar as operações.

A classe *Main* cria uma instância das classes *Neo4j* e *Titan* e as usa para instanciar cada um dos *kernels*. Por fim, realiza a execução de cada kernel de acordo com os parâmetros passados via linha de comando.

Figura 4.7: Diagrama de classes



Para gerar os dados a serem inseridos no grafo foi utilizado o algoritmo gerador de dados do programa da especificação executável, descrito na seção 4.1.2 e disponível em [38]. O programa foi modificado para, além de gerar os dados, também escrevê-los em um arquivo (extensão .csv) lido pelo *kernel* 1 e para gerar as amostras de vértices usados no *kernel* 4, em que para cada valor do parâmetro *SCALE* são extraídos dezesseis amostras em quantidades que variam de 1 a 16.

Capítulo 5

Resultados

Neste capítulo são apresentados o ambiente de execução dos experimentos, os resultados obtidos e a análise dos resultados.

5.1 Ambiente de teste

Os experimentos do *benchmark* foram realizados nos bancos de dados executando na mesma máquina local, embarcados no próprio *benchmark* utilizando a interface *Blueprints*. Assim, não há risco significativo de perda de conectividade e de inconsistência dos dados. No *Titan*, foi escolhido o *backend* Cassandra, pois é um banco de dados que garante consistência dos dados. O Cassandra acompanha a distribuição do *Titan*, sendo de fácil execução e não necessita de nenhuma configuração extra. O *Neo4j* não utiliza *backend*, por isso não é necessário se preocupar com esta questão.

A máquina utilizada para executar os experimentos é uma máquina virtual do *google cloud* com um servidor Linux/Ubuntu com 13 Gigabytes de memória, uma CPU com quatro núcleos e 10 Gigabytes de espaço em disco.

O *benchmark* tem como dependência o *JDK Java* e o *Maven*, sendo utilizado o *JDK-8 da Oracle* e o *Maven 3*. Os dois foram instalados seguindo os passos descritos no apêndice A e B. As versões dos bancos de dados são *Neo4j 2.1.6* e *Titan 0.4.4*, instalados seguindo os passos descritos no apêndice C e D .

5.2 Procedimento experimental

O experimentos foram realizados da seguinte forma:

Os dados foram gerados de acordo com a descrição na seção 4.3 para valores do parâmetro *SCALE* entre 10 e 20. Onde a quantidade de vértices gerados para *SCALE=10* e *SCALE=20* foram *N=1048576* e *N=1048576*, respectivamente. No ambiente de teste configurado, o máximo valor de escala dos grafos suportados pelos sistemas testados foi de *SCALE=20*.

Para cada valor de escala, foram realizados 5 execuções completas do software de teste, em que uma execução consiste nos quatro *kernels* do *benchmark*.

Os resultados das 5 execuções foram usados para calcular uma média dos tempos para os *kernels 1, 2 e 3* e o valor da *TEPS* no *kernel 4* para cada uma das escalas.

5.3 Resultados dos testes

Os resultados dos experimentos do *benchmark* foram compilados para escalas de 10 a 17, que foram realizadas corretamente sobre os dois sistemas. Nas figuras 5.1 a 5.8 , são apresentados os resultados dos testes para os *kernels 1, 2 e 3*.

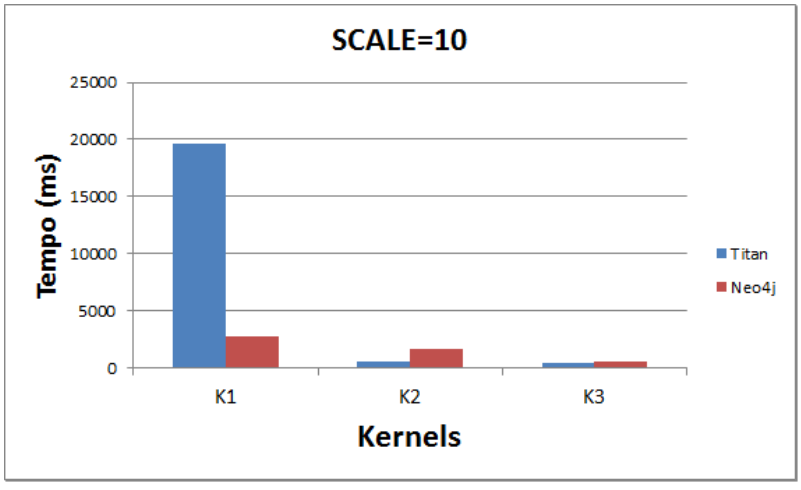


Figura 5.1: *kernels 1, 2 e 3 SCALE=10*

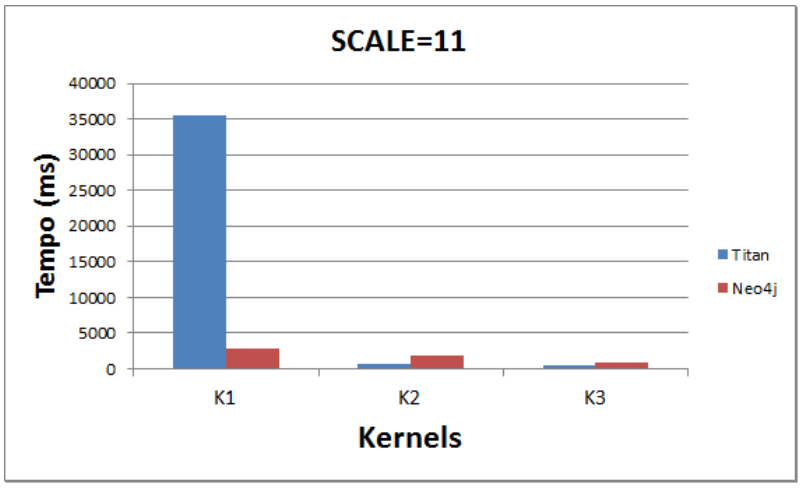


Figura 5.2: *kernels 1, 2 e 3 SCALE=11*

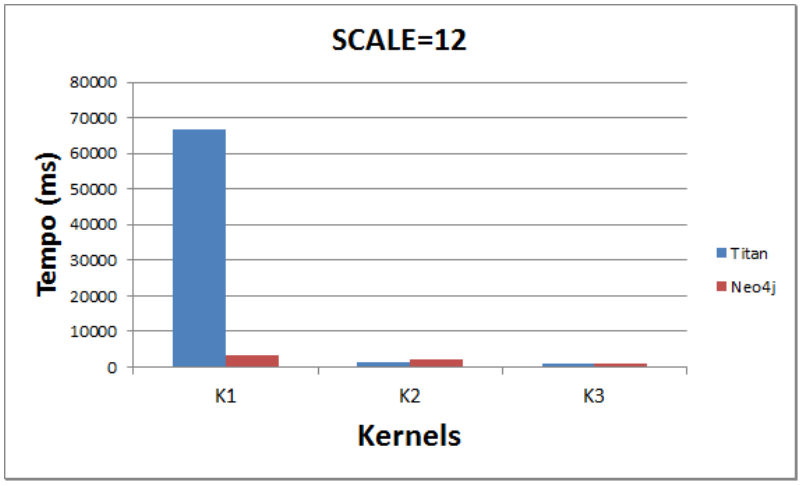


Figura 5.3: *kernels 1, 2 e 3 SCALE=12*

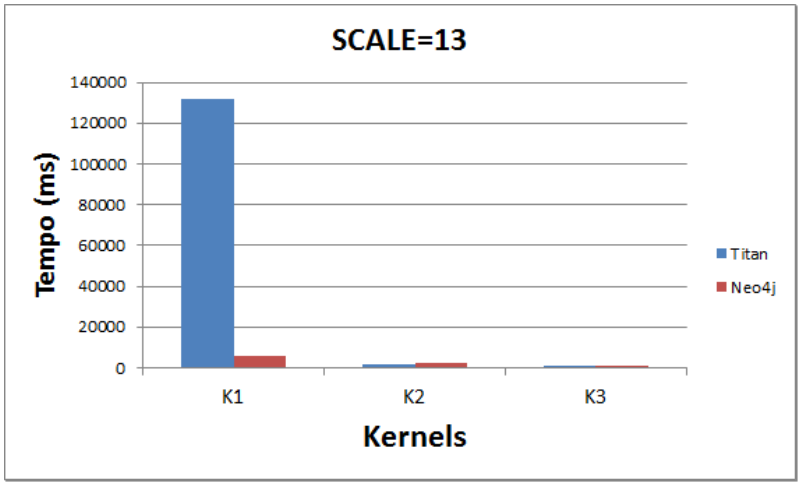


Figura 5.4: *kernels 1, 2 e 3 SCALE=13*

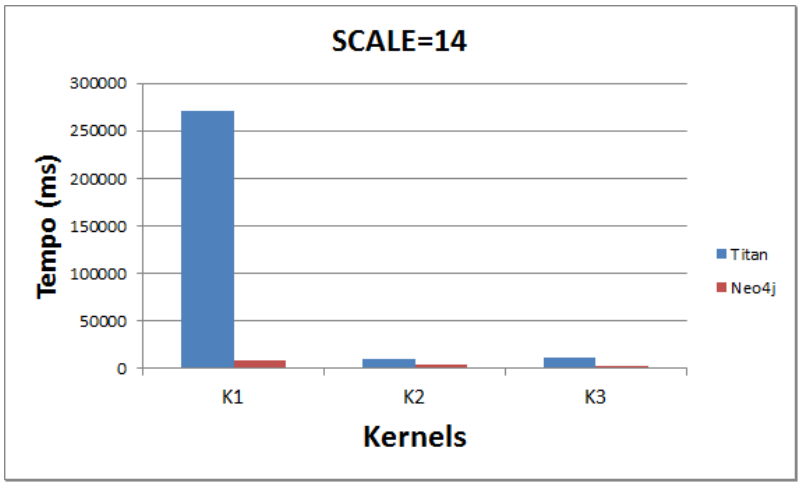


Figura 5.5: *kernels 1, 2 e 3 SCALE=14*

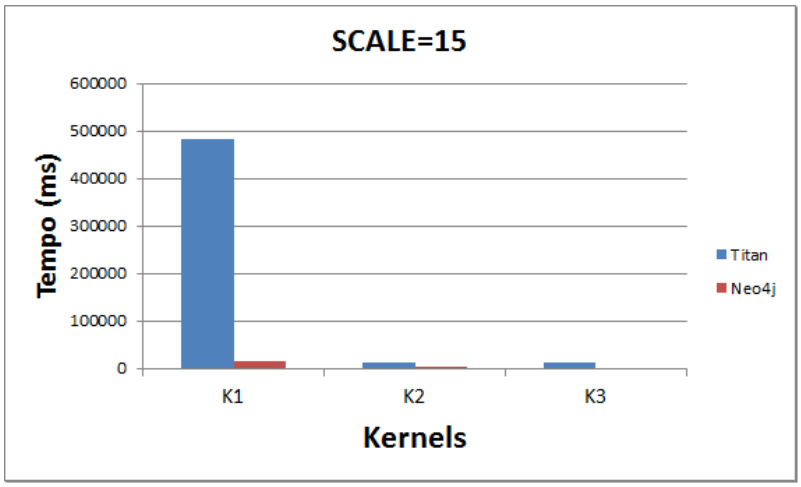


Figura 5.6: *kernels 1, 2 e 3 SCALE=15*

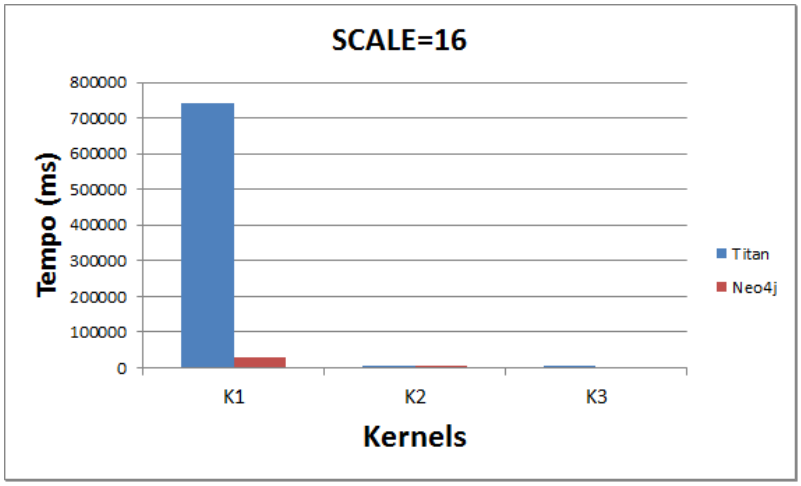


Figura 5.7: *kernels 1, 2 e 3 SCALE=16*

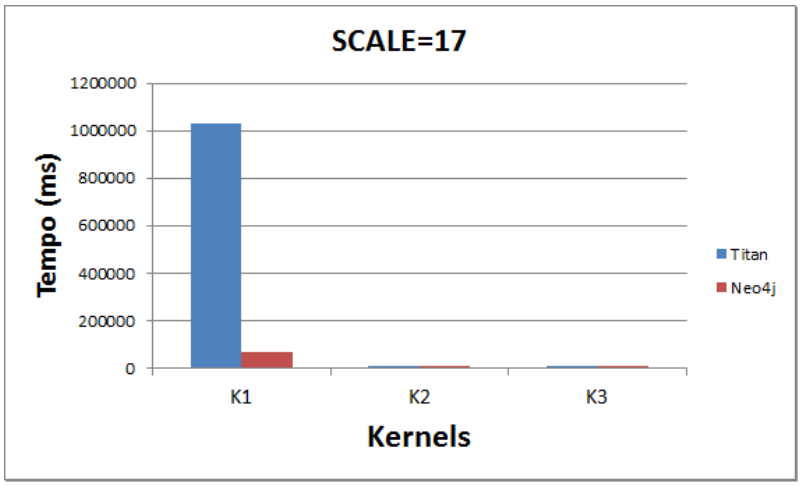


Figura 5.8: *kernels 1, 2 e 3 SCALE=17*

Nas figuras 5.9 a 5.16, são apresentados os resultados dos testes para o *kernel 4*, para cada número de vértices nas amostras.

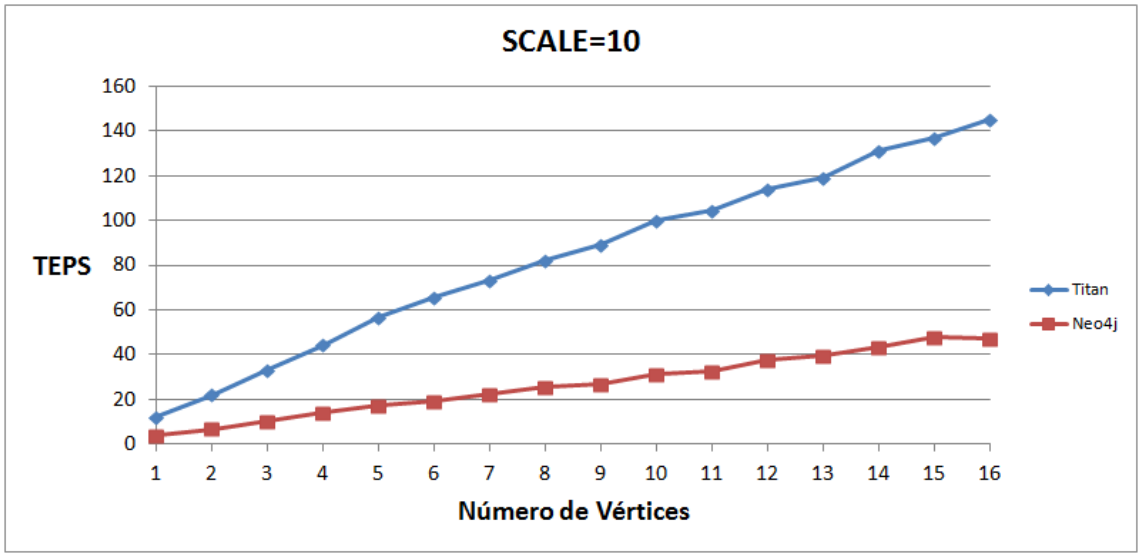


Figura 5.9: *kernel 4 SCALE=10*

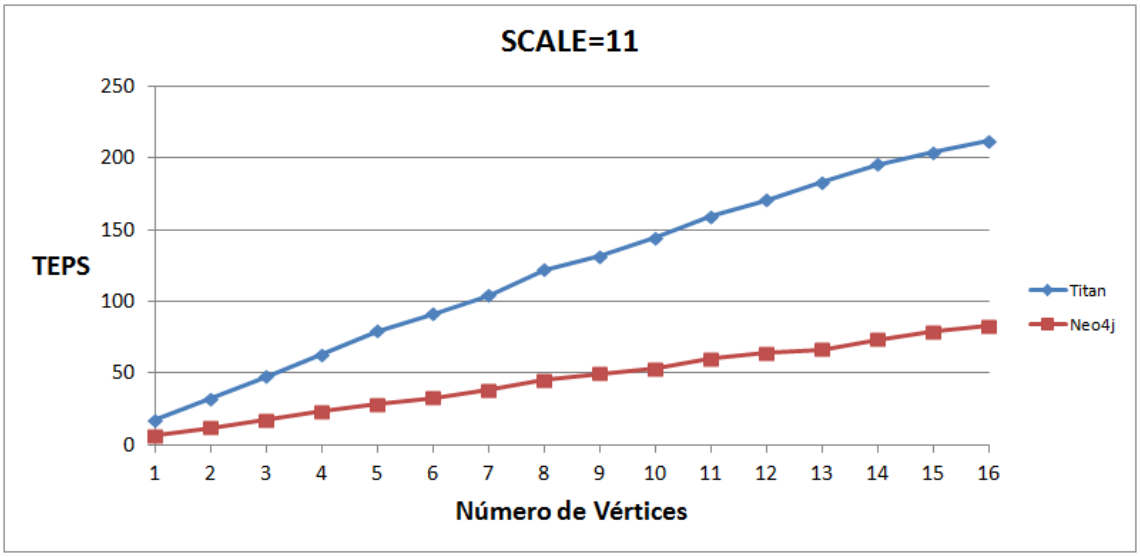


Figura 5.10: *kernel 4 SCALE=11*

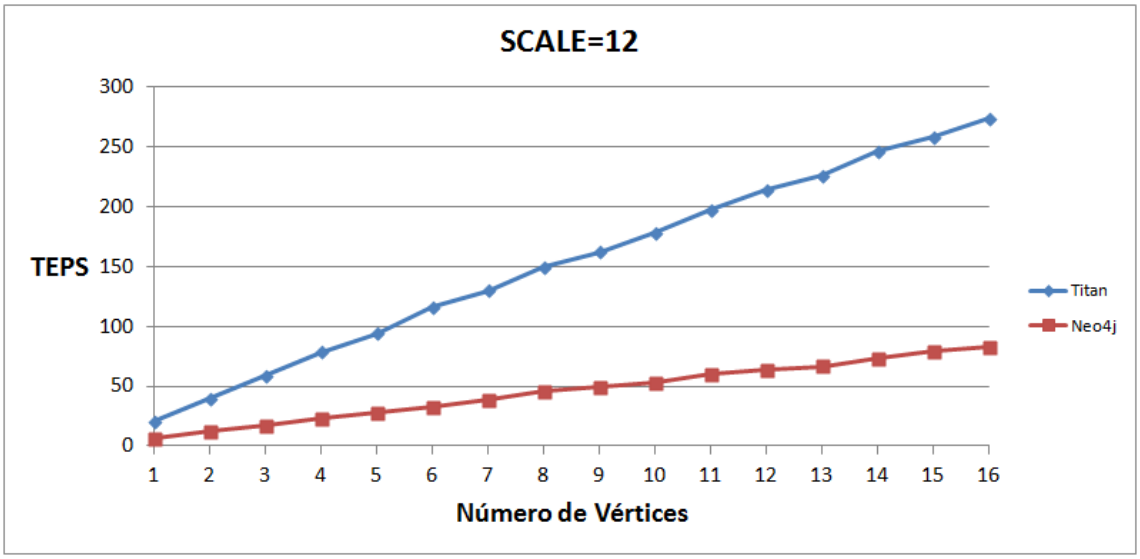


Figura 5.11: *kernel 4 SCALE=12*

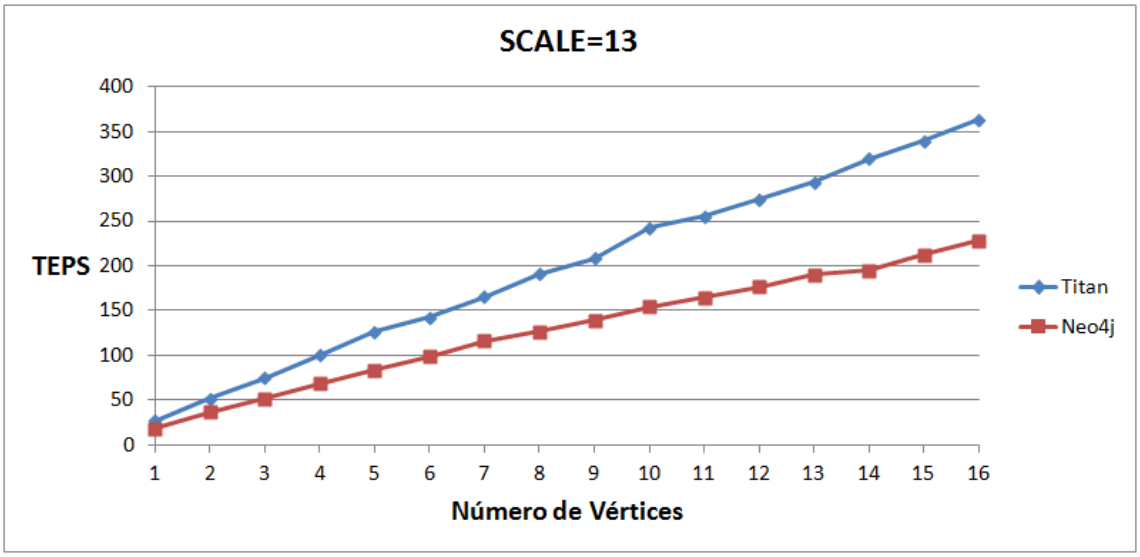


Figura 5.12: *kernel 4 SCALE=13*

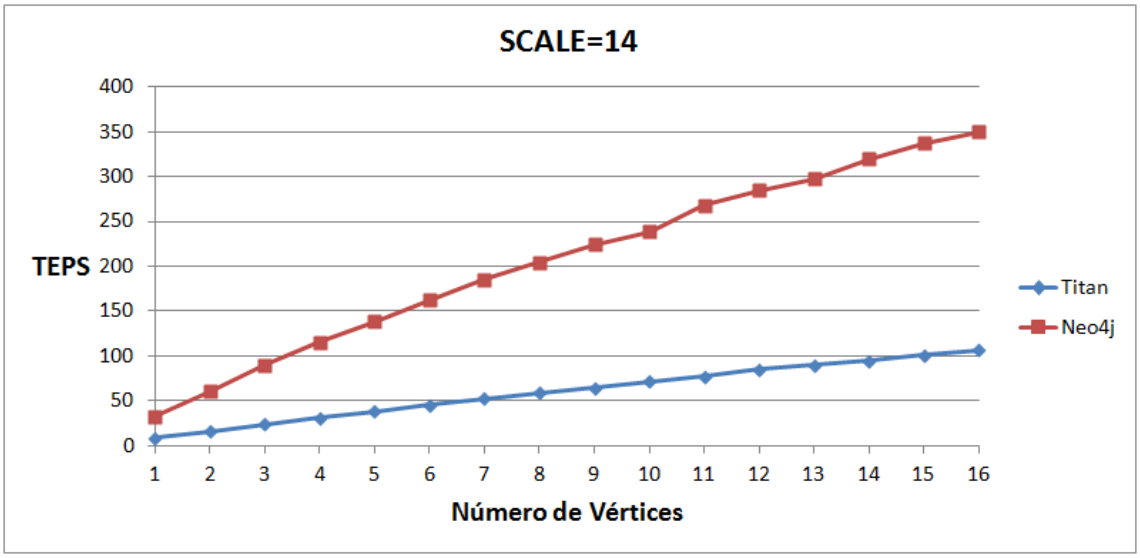


Figura 5.13: *kernel 4 SCALE=14*

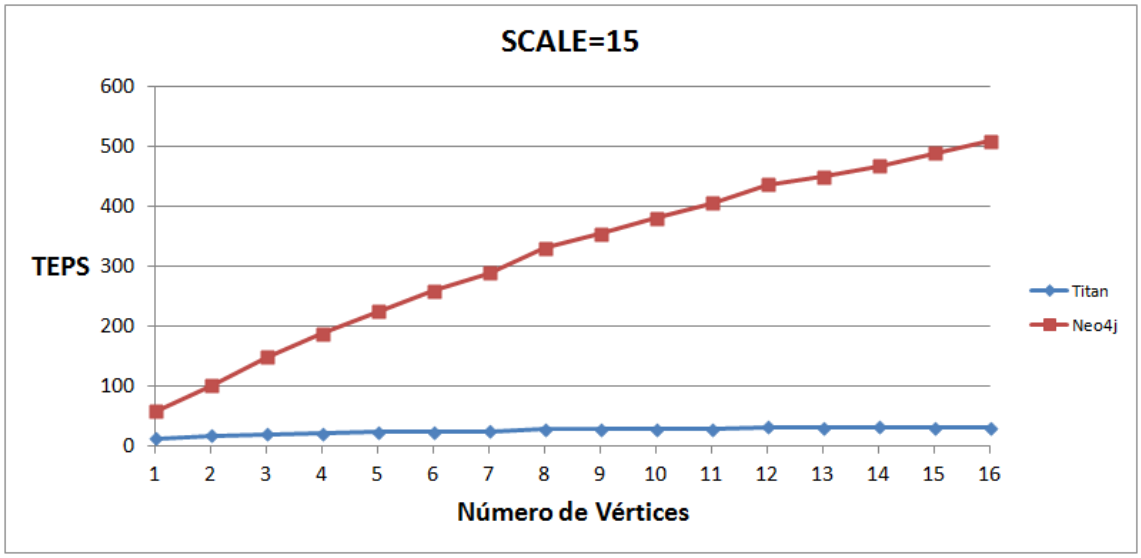


Figura 5.14: *kernel 4 SCALE=15*

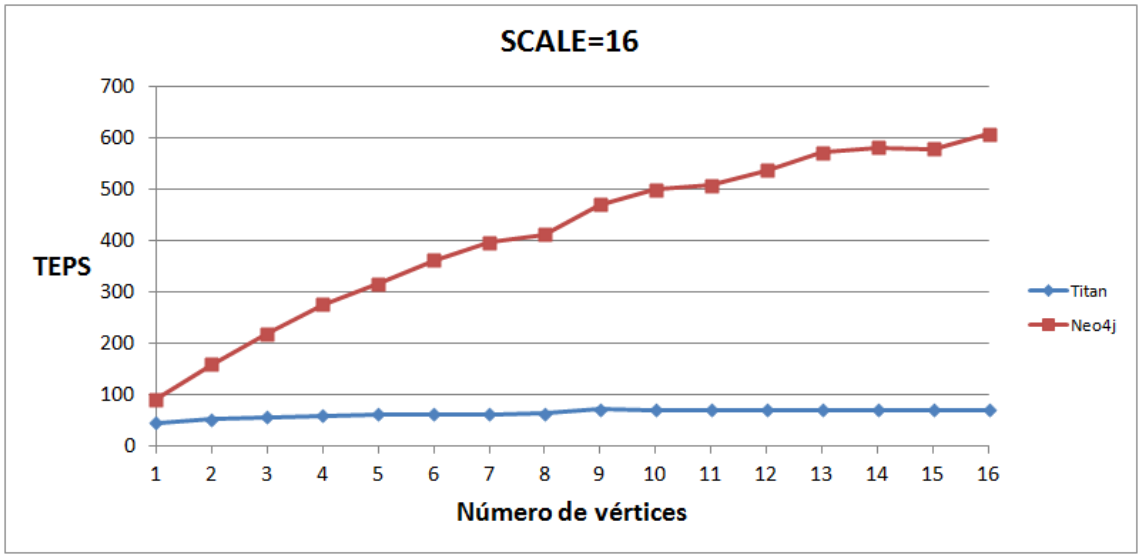


Figura 5.15: *kernel 4 SCALE=16*

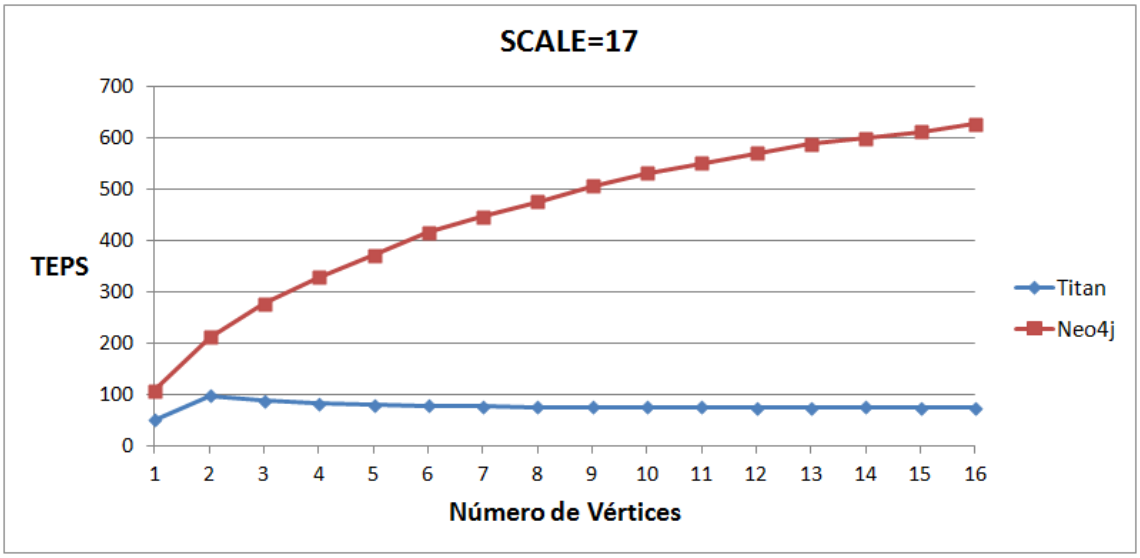


Figura 5.16: *kernel 4 SCALE=17*

5.4 Análise dos Resultados

A partir dos resultados apresentados nos gráficos, foram identificadas as seguintes informações:

Com relação ao *kernel 1*, o *Neo4j* teve o melhor resultado em todas as escalas. Este *kernel* realiza a construção do grafo no banco de dados a partir de uma lista de arestas previamente geradas.

A grande diferença entre os resultados dos dois sistemas é explicada pela forma como os dados são mantidos em cada sistema e pela forma como este *kernel* insere os dados no banco de dados. O *kernel 1* sempre verifica se o vértice a ser inserido já existe no grafo, caso exista ele o utiliza. O *Neo4j* procura manter os elementos já acessados ou criados em memória por tempo indeterminado [39]. Mas, o *Titan* limita a permanência dos dados em memória por apenas alguns milisegundos [40]. Como consequência, o *kernel 1* faz a busca pelo vértice, o *Neo4j* consulta a memória e o *Titan* na maioria das vezes consulta o disco.

Para os *kernels 2 e 3*, inicialmente nas escalas entre 10 e 12, o *Titan* teve o melhor resultado. Contudo, a partir da escala 14, o *Neo4j* passou a ter o melhor resultado, mostrando que em escalas maiores de dados o *Neo4j* tem melhor desempenho. O *kernel 2* realiza uma consulta por arestas do grafo, cujos pesos têm o maior valor. O *kernel 3* usa os vértices que resultaram desta consulta como

ponto de partida para realizar consultas sobre o grafo a fim de construir árvores com profundidade pré-determinada.

Nestes *kernels*, o cache de dados em memória não favorece a nenhum dos dois sistemas. Pois, cada busca no *kernel 1* por arestas não faz uso de elementos já consultados em buscas anteriores. Assim, os resultados são consequência somente da forma como os elementos são recuperados em cada sistema. Os resultados superiores do *Titan* ocorrem pelo pequeno tamanho das colunas do *Cassandra*, seu *backend*, em que as junções de colunas são mais velozes que a navegação pelas arestas do *Neo4j*, em escalas até 13. Quando o tamanho das colunas cresce, o *Neo4j* passa a superar o *Titan*. Pois, o processo de junção de colunas passa a ser mais lento que as consultas no *Neo4j*, que se beneficiam da propriedade de *Adjacência Livre de Índices*.

Com relação ao *kernel 4*, entre as escalas 10 e 13, o *Titan* executa com um valor de *TEPS* (Número de Arestas Percorridas por Segundo - *Traversed Edges Per Second*) maior que o *Neo4j*. Para escalas acima de 13, os valores do *Titan* se tornam inferiores aos do *Neo4j*. Isso é uma repetição do que ocorre nos *kernels 2 e 3*. E demonstra que para altas escalas de dados o *Neo4j* é muito melhor que o *Titan*. Este *kernel* realiza uma operação sobre todo o grafo para determinar os vértices com maior valor de centralidade de intermediação, esta operação tem complexidade $O(N^3)$ sobre todo o grafo. Com relação à evolução da curva da *TEPS* com a quantidade de vértices no *kernel 4*, observou-se que em escalas acima de 14 as curvas do *Neo4j* adquirem uma tendência à estabilização, conforme a quantidade de vértices aumenta. Enquanto isso, a curva do *Titan* apresenta uma tendência de queda. Essa tendência fica mais evidente na escala 17.

O resultado do *kernel 4* é uma combinação do processo de cache de elementos do grafo com a forma como os dados são consultados em cada sistema. Quando o *kernel 4* realiza o cálculo da centralidade para um vértice, ele necessita percorrer todo o grafo. No *Neo4j*, isso fará com que os elementos do grafo sejam carregados em memória. No *Titan*, eles serão carregados, mas permanecerão por pouco tempo. Isso resulta em um ganho de desempenho maior para o *Neo4j*, quando precisar calcular a centralidade para um novo vértice, pois não precisará consultar em disco por elementos já carregados em memória. Por outro lado, o *Titan* não mantém esses dados por muito tempo, isso significa que a cada cálculo de um novo vértice,

muitos dos elementos consultados em um vértice anterior precisarão ser consultados em disco. Esse efeito é melhor percebido quando o tamanho do grafo é grande, o que ocorre a partir de valores de escala acima de 13.

Contudo, a partir da escala 18, não foi possível a execução de experimentos sobre o *Neo4j*, pois ocorriam falhas por falta de memória. O *Titan* por outro lado suportou escalas de até 19. Ou seja, embora o *Neo4j* tenha maior velocidade, esta se dá por um maior consumo de memória. Já o *Titan* consome menos memória, usando mais disco, pois o seu *backend* é o Cassandra, permitindo-o escalar melhor que o *Neo4j*.

Capítulo 6

Conclusões

Neste trabalho foi realizada uma análise comparativa entre dois sistemas de bancos de dados orientados a grafos. Os dois sistemas analisados foram *Neo4j* e *Titan*.

Para realizar a comparação foram realizados experimentos com um sistema de teste implementado com base na especificação do *Benchmark HPC-SGA (High Performance Computation for Scalable Graph Analysis)*, que executa sobre os sistemas testados quatro operações: carga (*kernel 1*), busca de arestas com filtragem de peso(*kernel 2*), criação de subgrafos a partir de uma lista de nós(*kernel 3*) e computação de centralidade de intermediação, com operação transversal sobre todo o grafo (*kernel 4*). Todas estas operações foram temporizadas e seus tempos foram utilizados como métrica de comparação entre os dois sistemas.

Todos os experimentos foram realizados usando a mesma máquina, com os sistemas testados executando de forma embarcada no software de teste.

A partir dos resultados, concluiu-se que para altas escalas de dados o sistema com melhor desempenho, em termos de velocidade na realização das operações, é o *Neo4j*. Contudo, embora o *Titan* tenha tido desempenho inferior foi observado que ele tem um desempenho melhor em termos de escalabilidade, pois pôde realizar operações sobre escalas de dados acima das alcançadas pelo *Neo4j*. Também observou-se que, a métrica de desempenho *TEPS* pode ser determinada no *kernel 4* calculando a centralidade de intermediação para um número de vértices próximo de 16. Reduzindo o tempo de execução deste *kernel*.

6.1 Trabalhos Futuros

Embora o *benchmark* utilizado para realizar os experimentos englobe um conjunto interessante de operações, infelizmente alguns aspectos não foram contemplados neste trabalho. Em vista disso, como trabalhos futuros pretende-se:

- Verificar do impacto da atualização dos atributos dos vértices e arestas sobre o tempo de execução;
- Verificar o impacto do tráfego dos pacotes de dados, via rede, sobre o tempo de execução;
- Verificar o impacto do acesso concorrente, de múltiplos clientes, sobre o tempo de execução;
- Incluir outros bancos de dados, para também verificar seu desempenho;
- Realizar experimentos no *Titan* usando outros *backends*, para verificar seu desempenho.

Referências Bibliográficas

[1] GLEANER, N. S. E. *Cost of Hard Drive Storage Space*. Disponível em: <http://ns1758.ca/winch/winchest.html>.

[2] DANOWITZ, A. et al. Cpu db: recording microprocessor history. *Communications of the ACM*, ACM, v. 55, n. 4, p. 55–63, 2012.

[3] ZIKOPOULOS, P.; EATON, C. et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. [S.l.]: McGraw-Hill Osborne Media, 2011.

[4] SEGAL, B. et al. Grid computing: The european data grid project. In: *IEEE Nuclear Science Symposium and Medical Imaging Conference*. [S.l.: s.n.], 2000. v. 1, p. 2.

[5] NUGENT, A. et al. *Big data for dummies*. [S.l.]: John Wiley & Sons, 2013.

[6] DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, ACM, v. 51, n. 1, p. 107–113, 2008.

[7] CHANG, F. et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 26, n. 2, p. 4, 2008.

[8] FILHO, C. L. A. D. S. Processamento de dados em larga escala na computação distribuída. 2011.

[9] BRITO, R. W. Bancos de dados nosql x sgbds relacionais: análise comparativa. *Faculdade Farias Brito e Universidade de Fortaleza*, 2010.

[10] WEST, D. B. et al. *Introduction to graph theory*. [S.l.]: Prentice hall Upper Saddle River, 2001. v. 2.

[11] LÓSCIO, B. F.; OLIVEIRA, H. R. d.; PONTES, J. C. d. S. Nosql no desenvolvimento de aplicações web colaborativas. *VIII SIMPÓSIO BRASILEIRO DE SISTEMAS COLABORATIVOS, Paraty, RJ: SBC*, 2011.

[12] ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph databases*. [S.l.]: O'Reilly Media, Inc., 2013.

[13] DOMINGUEZ-SAL, D. et al. Survey of graph database performance on the hpc scalable graph analysis benchmark. In: *Web-Age Information Management*. [S.l.]: Springer, 2010. p. 37–48.

[14] INDREES, R. *Beginning Titan - The Distributed Graph Database*. [S.l.]: Oplicate, 2013.

[15] BRUGGEN, R. V. *Learning Neo4j*. [S.l.]: "Packt Publishing Ltd", 2014.

[16] DONG, X. L.; SRIVASTAVA, D. Big data integration. In: IEEE. *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. [S.l.], 2013. p. 1245–1248.

[17] BRETERNITZ, V. J.; SILVA, L. A. Big data: Um novo conceito gerando oportunidades e desafios. *Revista Eletrônica de Tecnologia e Cultura*, v. 2, n. 13, 2013.

[18] MANNING, P. *Big Data Bootcamp: What Managers Need to Know to Profit from the Big Data Revolution*. [S.l.]: Springer Science+Business Media, 2014.

[19] STRAUCH, C. Nosql databases (2011). *URL http://www. christof-strauch. de/-nosqldbs. pdf. http://www. christof-strauch. de/nosqldbs. pdf*, 2012.

[20] ALLEN, R.; HUNTER, L. *Active directory cookbook*. [S.l.]: "O'Reilly Media, Inc.", 2006.

[21] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, ACM, v. 13, n. 6, p. 377–387, 1970.

[22] MCCREARY, D.; KELLY, A. Making sense of nosql. *Greenwich, Conn.: Manning Publications*, 2013.

[23] TIWARI, S. *Professional NoSQL*. [S.l.]: Wrox, 2011.

[24] WILSON, R. J. *An introduction to graph theory*. [S.l.]: Pearson Education India, 1970.

[25] KOLOMICENKO, V.

Analysis and Experiemental Comparison of Graph Databases — Charles University in Prague, 2013.

[26] OLIVEIRA, H. M.; FONSECA, N. L. da. Protection in elastic optical networks against up to two failures based fipp p-cycle. In: IEEE. *Computer Networks and Distributed Systems (SBRC), 2014 Brazilian Symposium on*. [S.l.], 2014. p. 369–375.

[27] REDMOND, E.; WILSON, J. R. Seven databases in seven weeks. *The Pragmatic Bookshelf*, 2012.

[28] CIGLAN, M.; AVERBUCH, A.; HLUCHY, L. Benchmarking traversal operations over graph databases. In: IEEE. *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*. [S.l.], 2012. p. 186–189.

[29] DAYARATHNA, M.; SUZUMURA, T. Graph database benchmarking on cloud environments with xgdbench. *Automated Software Engineering*, Springer, v. 21, n. 4, p. 509–533, 2014.

[30] JOUILI, S.; VANSTEENBERGHE, V. An empirical comparison of graph databases. In: IEEE. *Social Computing (SocialCom), 2013 International Conference on*. [S.l.], 2013. p. 708–715.

[31] CHAKRABARTI, D.; ZHAN, Y.; FALOUTSOS, C. R-mat: A recursive model for graph mining. In: SIAM. *SDM*. [S.l.], 2004. v. 4, p. 442–446.

[32] ISIDO, D. et al. Betweeness-centrality of grid networks. In: *2009 International Conference on Computer Technology and Development*. [S.l.: s.n.], 2009. v. 1, p. 407–410.

[33] LAZZARINI, S. *Capitalismo de Laços*. Elsevier Science, 2012. ISBN 9788535252613. Disponível em: <https://books.google.com.br/books?id=z00aBQAAQBAJ>.

[34] BRANDES, U. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, Taylor & Francis, v. 25, n. 2, p. 163–177, 2001.

[35] BADER, D. A. et al. Hpc scalable graph analysis benchmark. *Citeseer. Citeseer*, p. 1–10, 2009.

[36] BLUEPRINTS Tinkerpop Wiki. 2013. Visitado em 27/07/2015. Disponível em: <https://github.com/tinkerpop/blueprints/wiki>.

[37] HPC-SGAB Source. 2010. Disponível em: <http://www.dama.upc.edu/publications/files/dama-upc-iwgd2010-hpca-sgab-source.zip/view>.

[38] HPC Graph Analysis. 2007. Visitado em 27/07/2015. Disponível em: <http://www.graphanalysis.org/>.

[39] HPC-SGAB Source. 2015. Disponível em: <http://neo4j.com/docs/stable/>.

[40] TITAN Data Caching. 2014. Disponível em: <https://github.com/thinkaurelius/titan/wiki/Data-Caching>.

Apêndice A

Instalação do JDK 8

Para instalar o *JDK* 8 nas distribuições *Debian* siga os seguintes passos:

Devemos remover o *OpenJDK*. No terminal (root), digite:

 # apt-get remove –purge openjdk-*

Digite **s** para confirmar a remoção do *OpenJDK*.

Aguarde a remoção ser concluída.

Agora, vamos instalar o *Java Oracle (JDK 8)*. No terminal (root), copie e cole os comandos abaixo:

 # echo "deb http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main tee /etc/apt/sources.list.d/webupd8team-java.list # echo "deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main tee -a /etc/apt/sources.list.d/webupd8team-java.list # apt-key adv –keyserver keyserver.ubuntu.com –recv-keys EEA14886

Após adicionar os repositórios e a key, atualize a lista de pacotes:

 # apt-get update

Instale o *JDK* 8, digite:

 # apt-get install oracle-java8-installer

Clique em OK e em Aceitar.

Aguarde a instalação ser concluída.

Após o término da instalação, digite no terminal:

$ java -version

Irá aparecer algo assim:

java version "1.8.0_05"

Java(TM) SE Runtime Environment (build 1.8.0_05-b13)

Java HotSpot(TM) 64-Bit Server VM (build 25.5-b02, mixed mode)

O Java 8 está instalado e pronto para usar.

Apêndice B

Instalação do Maven 3

Antes de instalar o Maven 3, é necessário remover o Maven 2. Faça isso com o seguinte comando:

sudo apt-get remove maven2

Para instalar o *Maven* 3 nas distribuições *Debian* siga os seguintes passos:

Devemos adicionar o repositório do Maven 3 no arquivo de sources. Siga os seguintes passos:

sudo -H gedit /etc/apt/sources.list

Adicione a linha seguinte no arquivo sources.list:

deb http://ppa.launchpad.net/natecarlson/maven3/ubuntu precise main

deb-src http://ppa.launchpad.net/natecarlson/maven3/ubuntu precise main

Execute os comandos:

sudo apt-get update

sudo apt-get install maven3

sudo ln -s /usr/share/maven3/bin/mvn /usr/bin/mvn

Apêndice C

Titan - Instalação, configuração e teste

Neste trabalho foi utilizado como *backend* o *Apache Cassandra*, banco de dados orientado a colunas.

Contudo, antes de instalar o *Titan* é necessário que a versão mais recente do Java 8 esteja instalada, sendo preferível o *JDK Oracle/Sun*, *OpenJdk* ou *IBM jvm*. Para instalar o *JDK* 8 nas distribuições *Debian* siga os passos descritos A. Também é necessário que esteja instalado o *Maven 3* de acordo com os passos descritos no B

Será usada a versão 0.4.4 do *Titan*. Baixe e descompacte com os comandos seguintes:

wget <http://s3.thinkaurelius.com/downloads/titan/titan-all-0.4.4.zip>

unzip titan-all.0.0.4.zip

O pacote do *Titan* vem acompanhado como um executável do *Apache Cassandra*. Todos os binários estão localizados na pasta:

~/ROOT_TITAN_FOLDER/bin.

O *Cassandra* também pode ser baixado no link <http://www.apache.org/dyn/closer.cgi?path=/cassandra/2.0.9/apache-cassandra-2.0.9-bin.tar.gz> ou <http://ftp.unicamp.br/pub/apache/cassandra/2.0.9/apache-cassandra2.0.9-bin.tar.gz>. Para executar use o seguinte comando:

~/ROOT_TITAN_FOLDER/bin/cassandra -f

O *Titan* também vem acompanhado do *Gremlin*, um programa que dá su-

porte á linguagem *Gremlin* para realizar operações sobre o banco de dados. A conexão do *Gremlin* com o *backend* pode ser local ou de distribuída. Para executá-lo use o comando:

~/ROOT_TITAN_FOLDER/bin/gremlin

O *Gremlin* irá exibir um prompt de comandos para a linguagem *Gremlin*. Para configurar e conectar com o *Cassandra* localmente use os comandos:

conf = new BaseConfiguration();

conf.setProperty("storage.backend", "embededcassandra");

conf.setProperty("storage.hostname", "~/ROOT_TITAN_FOLDER/data/cassandra.yaml");

g = TitanFactory.open(conf);

Com os ultimos comandos o *Titan* está pronto par uso. Agora passemos á apresentação do *Neo4j*.

Apêndice D

Neo4j - Instalação, configuração e teste

Será usada uma instância local da *Community Edition* para Linux do *Neo4j*. Faça download da *release* 2.1.6 no site <http://www.neo4j.org/download>, e descompacte. Esta já vem configurada para acesso local via navegador.

Para iniciar o sistema entre na pasta raiz do *Neo4j* descompactado e execute o comando:

~/ROOT_NEO4J_FOLDER/bin/neo4j start

Após estes passos o *Neo4j* estrá disponível para teste no link <http://localhost:7474/browser> e para conexão local na forma embarcada via Java *Blueprints* através da pasta /tmp/hsperfdata_$(USER).