GRAPHENE: UM PROTÓTIPO DE SISTEMA DE GERÊNCIA DE BANCOS DE DADOS DISTRIBUÍDOS ORIENTADOS A GRAFOS

Fernando Vinicius Duarte Magalhães

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
Junho de 2014

GRAPHENE: UM PROTÓTIPO DE SISTEMA DE GERÊNCIA DE BANCOS DE DADOS DISTRIBUÍDOS ORIENTADOS A GRAFOS

Fernando Vinicius Duarte Magalhães

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.

_____
Prof. Marta Lima de Queirós Mattoso, D.Sc.

_____
Prof. Fábio André Machado Porto, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
JUNHO DE 2014

Magalhães, Fernando Vinicius Duarte

Graphene: Um Protótipo de Sistema de Gerência de Bancos de Dados Distribuídos Orientados a Grafos/ Fernando Vinicius Duarte Magalhães. – Rio de Janeiro: UFRJ/COPPE, 2014.

XVI, 138 p.: il.; 29,7 cm.

Orientador: Alexandre de Assis Bento Lima

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 104-110.

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

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

GRAPHENE: UM PROTÓTIPO DE SISTEMA DE GERÊNCIA DE BANCOS DE DADOS DISTRIBUÍDOS ORIENTADOS A GRAFOS

Fernando Vinicius Duarte Magalhães

Junho/2014

Orientador: Alexandre de Assis Bento Lima
Programa: Engenharia de Sistemas e Computação

Novas aplicações como análises de redes sociais e estudos moleculares auxiliados por computador têm motivado o desenvolvimento de Sistemas de Gerência de Banco de Dados de Grafos (SGBDG) e de outras técnicas relacionadas ao armazenamento e processamento destas estruturas. A maioria dos SGBDG hoje são centralizados ou, quando paralelos, usam bases de dados totalmente replicadas. No entanto, o aumento do volume de dados dos grafos, juntamente com sua complexidade, motiva a investigação de técnicas para a implementação de SGBDG Distribuídos. O uso de tais ferramentas possibilita a utilização de técnicas de processamento distribuído e paralelo para a realização de operações em grandes grafos com alto desempenho. Esta dissertação descreve o Graphene, um protótipo de camada intermediária de *software* desenvolvido para permitir a criação e utilização de bases de dados de grafos distribuídas, sem necessidade de replicação total e heterogêneas. Além disso, a fim de explorar o potencial desta tecnologia, foram implementados algoritmos distribuídos para operações típicas sobre grafos, como busca em largura e determinação do menor caminho entre vértices. A distribuição é transparente para aplicações cliente.

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

GRAPHENE: A DISTRIBUTED GRAPH-ORIENTED DATABASE MANAGEMENT SYSTEM PROTOTYPE

Fernando Vinicius Duarte Magalhães

June/2014

Advisor: Alexandre de Assis Bento Lima
Department: Systems and Computing Engineering

New applications like social network analysis and computer-aided experiments concerning molecular structures motivate the development of Graph-Oriented Database Management Systems (GDBMS) and other tools to improve the storage and processing of such data structures. Most of the existing GDBMS are centralized or, when parallel, use fully replicated databases. However, the increasing size and complexity of such graphs motivates the development of Distributed GDBMS. These tools make it feasible the development of distributed and parallel techniques for the execution of high performance operations on large graphs. This dissertation presents Graphene, a prototype *middleware* that allows for the creation and management of distributed, not fully replicated heterogeneous graph databases. Furthermore, in order to demonstrate the potentialities of the prototype, distributed algorithms were implemented for typical graph operations, *e.g.*, breadth first search and the determination of the shortest path between two vertices. Graphene makes the data distribution totally transparent for the client applications.

Dedicatória

À minha família, em especial meus pais e meus avós;
a meus amigos e a todas as pessoas que acreditaram e
me apoiaram durante esta jornada.

Agradecimentos

Primeiramente gostaria de agradecer a Deus, pois sem Ele nada seria possível.

Gostaria de agradecer ao meu orientador, Alexandre de Assis Bento Lima, pela paciência, motivação e orientação recebida durante todo o andamento da dissertação. Agradeço também a todos os professores do PESC pelo enorme conhecimento adquirido durante as disciplinas cursadas durante o Mestrado.

Agradeço à minha família, pelo incentivo e motivação fundamentais durante o curso de Mestrado, e aos meus amigos, pelo enorme apoio recebido.

Agradeço também a toda a equipe responsável pelo Grid5000 pela oportunidade de utilização deste ambiente e por todo o suporte prestado para a execução dos experimentos.

Agradeço ainda ao CNPq pelo apoio financeiro durante o Mestrado.

Sumário

Índice de Figuras

Índice de Tabelas

Lista de Siglas

BDG – Banco de Dados de Grafos

BSP – *Bulk Synchronous Parallel*

SGBDG – Sistema de Gerência de Bancos de Dados Orientados a Grafos

SGBDR – Sistema de Gerência de Bancos de Dados Relacionais

Capítulo 1 – Introdução

O recente crescimento de dados organizados como grafos impulsionou o surgimento de ferramentas capazes de armazenar, gerenciar e/ou processar, de forma eficiente, grandes quantidades de dados neste formato.

Sistemas de Gerência de Bancos de Dados Relacionais (SGBDR) representam a solução predominante no armazenamento de bases de dados atualmente. O relacionamento entre entidades armazenadas em um SGBDR é frequentemente representado através da utilização de chaves estrangeiras (RODRIGUEZ & NEUBAUER, 2010).

Quando os relacionamentos existentes entre as entidades do domínio são complexos e as operações de leitura envolvem uma frequente navegação entre estes relacionamentos, os SGBDR podem não ser a melhor opção a ser utilizada por não terem sido projetados para esta finalidade específica. Neste contexto, outras soluções como Sistemas de Gerência de Bancos de Dados Orientados a Grafos (SGBDG) podem ser mais apropriados, pelo fato de serem especializados tanto no armazenamento quanto na navegação de estruturas complexas no formato de grafos. Soluções de SGBDG podem ser encontrados em domínios como: redes sociais, sistemas de recomendação, logística, autorização e controle de acesso, dentre outros (ROBINSON *et al.*, 2013). Em bancos de dados de grafos as entidades são comumente representadas como vértices e os relacionamentos entre entidades são representados por arestas (DOMINGUEZ-SAL *et al.*, 2011).

Os SGBDG são, em sua maioria, voltados para aplicações OLTP (*On-Line Transaction Processing*), especializadas na gerência de consultas provenientes de múltiplos usuários simultâneos, que exploram a vizinhança de vértices percorrendo as arestas do grafo em busca de um objetivo específico predefinido. Consultas submetidas a um sistema OLTP são geralmente pequenas e não exigem analisar a estrutura do grafo como um todo. Diversos SGBDG estão disponíveis atualmente. Alguns, como o Neo4j (NEO4J, 2014) e o Sparksee (SPARKSEE, 2014), armazenam e gerenciam o acesso à base de dados de forma centralizada. Outros, como InfiniteGraph (INFINITEGRAPH,

2014), Titan (TITAN, 2014) e OrientDB (ORIENTDB, 2014) são especializados no armazenamento e processamento de grafos distribuídos.

Quando a estrutura global do grafo precisa ser analisada devido à execução de algoritmos como PageRank ou SSSP (*Single Source Shortest Path* – encontrar o menor caminho entre um vértice de origem e todos os outros vértices do grafo), por exemplo, um outro conjunto de soluções como os arcabouços de processamento paralelo costuma ser utilizado para permitir esta análise de forma eficiente. Uma ampla gama de soluções pode ser encontrada neste cenário. Os arcabouços de processamento paralelo geralmente são capazes de distribuir a carga do trabalho a ser executado entre múltiplos nós computacionais. O Pregel (MALEWICZ *et al.*, 2010) é um arcabouço que mantém toda a estrutura do grafo em memória principal, dividindo-a entre nós de um *cluster* de computadores. A comunicação entre os nós neste modelo é feita através de troca de mensagens. Outros arcabouços baseados no Pregel como o GraphLab (LOW *et al.*, 2010) e o PowerGraph (GONZALEZ *et al.*, 2012) utilizam um modelo de comunicação baseado em um espaço de memória distribuído compartilhado entre os nós. Arcabouços como o Grace (PRABHAKARAN *et al.*, 2012) são totalmente centralizados e utilizam múltiplos núcleos de processamento de máquinas de grande porte para a execução de algoritmos de grafos de forma eficiente. Trinity (SHAO *et al.*, 2013) propõe um mecanismo que utiliza a nuvem para armazenamento da estrutura do grafo. De um modo geral, todos os arcabouços de processamento paralelo precisam carregar previamente a estrutura do grafo na memória principal (*RAM*) para a execução dos algoritmos.

Os SGBDG podem ser vistos como soluções NoSQL (*Not Only SQL*). As soluções NoSQL são sistemas de gerenciamento de dados não relacionais especializadas no armazenamento de grandes quantidades de dados e na garantia de alta disponibilidade e tolerância a falhas. Sistemas NoSQL representam uma alternativa à utilização de SGBDR tradicionais e são projetados para facilitar a distribuição dos dados, permitindo o armazenamento e processamento de dados em grande escala através da utilização de hardware não especializado (MONIRUZZAMAN & HOSSAIN, 2013). Ao contrário da maioria das outras soluções NoSQL, como aquelas baseadas em bases de dados orientadas a pares chave/valor, famílias de colunas e documentos, a distribuição de uma base de dados de grafos entre múltiplos nós para proporcionar maior escalabilidade e

desempenho não é uma tarefa fácil (SADALAGE & FOWLER, 2012). Neste contexto, duas medidas são geralmente adotadas: a primeira estratégia é utilizar uma solução de escalabilidade vertical, aumentando a capacidade de processamento da máquina responsável pela gerência da base de dados de grafos. A segunda estratégia é realizar uma fragmentação da base de dados baseada no domínio da aplicação, distribuindo a base de dados original entre diferentes nós.

A distribuição de uma base de dados de grafos entre diferentes nós computacionais muitas vezes exige que cada aplicação cliente esteja ciente da distribuição dos dados e gerencie, manualmente, o acesso às bases de acordo com a necessidade. O mesmo ocorre em aplicações que precisem acessar diferentes bases de dados distribuídas geograficamente em múltiplos sítios.

Neste contexto propomos o Graphene, o protótipo de uma camada intermediária de *software* (*middleware*) desenvolvido para permitir a gerência de múltiplas bases de dados de grafos distribuídas geograficamente, fornecendo uma forma de acesso aos dados que torne transparente a sua distribuição, conforme esperado de um sistema de banco de dados distribuído (OZSU & VALDURIEZ, 2011). Desta forma é possível manipular a base de dados como se esta fosse centralizada, retirando das aplicações clientes a responsabilidade de acesso aos dados de forma distribuída.

O principal objetivo desta dissertação é criar uma camada de *software*, denominada Graphene, responsável por controlar a distribuição dos dados entre múltiplas bases de dados de grafos de forma a expor uma única base de dados logicamente centralizada, facilitando o acesso e tornando transparentes os aspectos de distribuição (e a complexidade que dela decorre) das aplicações clientes. A partir do momento em que bases de dados de grafos são gerenciadas pelo Graphene, relacionamentos entre vértices de diferentes bases de dados podem ser criados e percorridos durante travessias. Além disso, o Graphene é compatível com a especificação Blueprints (BLUEPRINTS, 2014), um padrão "de fato" responsável por proporcionar uma API (*Application Programming Interface* - Interface para Programação de Aplicações) uniforme consistente com as principais primitivas e operações realizáveis em bases de dados de grafos. As principais implementações de SGBDG adotam e são compatíveis com a API definida pelo Blueprints (CIGLAN *et al.*, 2012). Como consequência, o Graphene oferece suporte a

uma ampla gama de soluções de SGBDG e a outras soluções compatíveis com o Blueprints. Isso implica que os SGBDG utilizados em uma base de dados distribuída gerenciada pelo Graphene podem ser heterogêneos.

Todas as operações de grafos exigidas pelo contrato do Blueprints são implementadas pelo Graphene, que se comporta como um SGBDG distribuído. O *middleware* desenvolvido é capaz de realizar a inclusão e remoção de vértices e arestas, associar propriedades a eles bem como percorrer a estrutura do grafo e realizar consultas e travessias simples. A estratégia de distribuição dos dados é controlada por uma função que pode ser definida pelo usuário, apesar de o protótipo implementar uma estratégia padrão, que pode ser utilizada pelo usuário que não deseje criar uma nova estratégia.

Além das operações tradicionais sobre grafos exigidas pelo Blueprints, dois algoritmos distribuídos foram implementados no protótipo, a fim de permitir travessias sobre a estrutura do grafo de uma forma mais eficiente. Um algoritmo de travessia baseado em uma busca em largura distribuída e paralela foi desenvolvido e um algoritmo de busca de menor caminho entre dois vértices também foi implementado. Estas operações não são especificadas pelo padrão Blueprints, sendo específicas do Graphene. Os algoritmos paralelos e distribuídos desenvolvidos foram baseados no modelo de computação BSP - *Bulk Synchronous Parallel* (VALIANT, 1990).

O correto funcionamento do Graphene foi avaliado através da execução de consultas representativas das principais operações realizadas em bancos de dados de grafos. Os experimentos executados avaliaram a utilização do Graphene para distribuir bases de dados centralizadas de acordo com duas estratégias de fragmentação conhecidas: a estratégia de Espalhamento (*Hash*), que é alheia à estrutura do grafo e distribui os vértices entre fragmentos de acordo com o resultado de uma função de espalhamento que recebe como entrada o identificador do vértice; e a estratégia METIS (KARYPIS & KUMAR, 1998), que tenta obter fragmentos balanceados diminuindo as arestas de corte entre diferentes fragmentos do grafo.

Os resultados obtidos demonstraram que o Graphene pode ser utilizado para reunir bases gerenciadas por múltiplos SGBDG, formando um ambiente de banco de dados

distribuído. As principais operações executadas pelos SGBDG centralizados puderam também ser executadas no Graphene.

Os experimentos executados no decorrer desta dissertação permitiram analisar que a solução distribuída implementada possui um desempenho inferior do que aquele obtido por uma solução centralizada devido aos altos custos de comunicação envolvidos. Apesar disso, o Graphene torna possível a gerência de grafos que, devido ao seu tamanho, não poderiam ser gerenciados por uma base centralizada.

O modelo de computação BSP, utilizado para implementação dos algoritmos paralelos e distribuídos do Graphene, demonstrou um bom desempenho para a fragmentação baseada em Espalhamento, amplamente utilizada por arcabouços de processamento paralelo, mas demonstrou ser menos eficiente em estratégias de fragmentação mais direcionadas à estrutura do grafo, como o METIS.

Esta dissertação está organizada da seguinte forma: o Capítulo 2 descreve as principais soluções em SGBDG centralizadas e distribuídas bem como os arcabouços utilizados para o processamento de operações globais em grafos de grande escala. O Capítulo 3 apresenta o Graphene, solução proposta por esta dissertação para a gerência de bases de dados de grafos distribuídas. Detalhes de implementação e informações sobre como utilizar múltiplas bases de dados preexistentes junto ao Graphene são abordados também neste capítulo. O Capítulo 4 apresenta os resultados experimentais obtidos através da execução das principais consultas em banco de dados de grafos, justificando os valores observados na execução de cada cenário. O Capítulo 5 apresenta as conclusões da dissertação e propõe trabalhos futuros.

Capítulo 2 – Revisão da Literatura

Este capítulo descreve os principais sistemas de gerência de banco de dados de grafos, arcabouços de processamento paralelo em grafos e algumas das principais técnicas de fragmentação de grafos propostas na literatura. As seções 2.1 e 2.2 dão ênfase às questões de escalabilidade e às estratégias adotadas por cada uma delas para lidar com grandes volumes de dados. Os principais arcabouços de processamento paralelo em grafos são apresentados e comparados na seção 2.3. Questões de distribuição e fragmentação em grafos são discutidas na seção 2.4, realçando as vantagens e desvantagens da utilização de cada solução.

2.1. Bancos de Dados de Grafos

Um grafo $G = (V, E)$ é uma estrutura de dados constituída por um conjunto de vértices V e um conjunto de arestas E tal que $E \subseteq (V \times V)$ para grafos direcionados e $E \subseteq \{V \times V\}$ para grafos não direcionados (RODRIGUEZ & NEUBAUER, 2010).

Sistemas de Gerência de Banco de Dados de Grafos (SGBDG) são especializados em armazenar e processar estruturas de grafos de forma eficiente. O principal modelo de dados utilizado por estes sistemas é o de grafos de propriedades (ROBINSON *et al.*, 2013). Um grafo de propriedades é um grafo $G = (V, E, \lambda, \mu)$ tal que as arestas são direcionadas ($E \subseteq (V \times V)$), as arestas possuem um tipo associado ($\lambda: E \rightarrow \Sigma$) e um conjunto de propriedades no formato chave e valor pode ser associado aos vértices e arestas do grafo ($\mu: (V \cup E) \times R \rightarrow S$) (RODRIGUEZ & NEUBAUER, 2010). Vértices e arestas algumas vezes são denominados, de uma forma mais geral, simplesmente como elementos.

O modelo de dados relacional (CODD, 1970) é amplamente utilizado atualmente como o principal modelo de representação de dados. Sistemas de Gerência de Banco de Dados Relacionais (SGBDR) têm sido utilizados desde a década de 80. Dados de um domínio são frequentemente divididos em múltiplas tabelas e colunas de uma tabela são responsáveis por referenciar registros presentes em outra tabela. A recuperação da

informação neste modelo é fortemente baseada na junção das tabelas e na utilização das estruturas de índices. Os SGBDG, por outro lado, armazenam todos os dados em uma estrutura que é geralmente otimizada utilizando ponteiros para indicar diretamente as adjacências dos elementos, sem a necessidade de consultar índices. Esta característica torna possível a recuperação das adjacências de um elemento em tempo constante, independentemente do tamanho do grafo como um todo.

Arestas em um banco de dados de grafos são representadas como "cidadãs de primeira classe" (*first-class citizens*) (ROBINSON *et al.*, 2013), ao contrário dos bancos de dados Relacionais, onde os relacionamentos são geralmente inferidos por meio de chaves estrangeiras. Desta forma as arestas, assim como os vértices, são tratadas como entidades do domínio. As consultas e operações em um banco de dados de grafos não implicam na necessidade de escrever regras complexas para navegar entre os relacionamentos (GÜTING, 1994).

Percorrer os relacionamentos existentes entre os vértices de forma eficiente é uma tarefa crucial em um SGBDG. A utilização de bases de dados em formato de grafos é particularmente útil quando o domínio exige analisar de que forma os vértices estão relacionados entre si. Ou seja, os relacionamentos entre os vértices passam a ser uma informação tão importante quanto os próprios vértices que estão sendo armazenados. Alguns importantes SGBDG encontrados atualmente são: Sparksee (SPARKSEE, 2014), Neo4j (NEO4J, 2014), OrientDB (ORIENTDB, 2014), InfiniteGraph (INFINITEGRAPH, 2014), Titan (TITAN, 2014), dentre outros. Estes sistemas são soluções OLTP capazes de gerenciar um grande número de transações simultâneas.

Outra característica importante em bases de dados de grafos é a existência de um esquema de dados mais flexível, ideal para a representação de dados semiestruturados. Novos tipos de relacionamentos podem ser identificados e representados em tempo de execução, sem a necessidade de modificar esquemas de bases de dados.

Esta dissertação tem como foco principal os grafos de propriedades. A seguir, são descritas as principais soluções de banco de dados para esta classe de grafos.

2.1.1. Neo4j

O Neo4j (NEO4J, 2014) é um SGBDG centralizado, mantido pela empresa Neo Technology. Trabalha com o modelo de dados de grafos de propriedades e suporta nativamente o armazenamento e processamento de grafos. Possui suporte a transações, garantindo as propriedades ACID (atomicidade, consistência, isolamento e durabilidade). Pode ser utilizado na arquitetura cliente e servidor ou de forma embarcada junto da aplicação do usuário.

Algoritmos de grafos como cálculo de menor caminho, todos os caminhos entre dois vértices, Dijkstra e A* são suportados pelo Neo4j. Outros recursos como consultas baseadas em travessias a partir de determinados vértices de origem também podem ser utilizados. Tanto as operações de algoritmos de grafos quanto as travessias suportam uma série de restrições com relação aos caminhos que serão percorridos.

As operações de travessias podem ser personalizadas com uma série de parâmetros definidos pelo usuário. A direção e tipo das arestas a serem percorridas, restrições de caminhos que serão percorridos, restrições de unicidade e a ordem da travessia são exemplos de parâmetros que podem ser estabelecidos pelo usuário ao especificar uma consulta de travessia neste banco de dados de grafos.

O Neo4j possui uma linguagem de consulta própria chamada Cypher. O Cypher é uma linguagem declarativa que permite expressar consultas de maneira simples e eficiente. Sua estrutura foi influenciada por uma série de linguagens: cláusulas WHERE e ORDER BY foram inspirados pela linguagem SQL, navegação e casamento de padrões teve influência do SPARQL, e a semântica das estruturas de dados de conjuntos e coleções veio de linguagens como Haskell e Python. É otimizada para consultas que envolvam apenas leitura da base de dados, apesar de também permitir operações de alteração de dados.

Escalabilidade e alta disponibilidade podem ser obtidos através da solução de alta disponibilidade Neo4j HA, que está presente na distribuição comercial *Enterprise Edition*. A base de dados é replicada em uma série de nós computacionais (que formam um *cluster*) e um balanceador de carga (que não faz parte da solução de alta

disponibilidade e deve ser provido pelo usuário) distribui as consultas entre os nós existentes. Um nó é eleito mestre e fica responsável por centralizar as operações de escrita. Tanto o nó mestre quanto os nós escravos podem ser responsáveis pelas operações de leitura. A tolerância a falhas é alcançada por esta solução uma vez que os dados permanecem acessíveis enquanto existir pelo menos um nó funcional no *cluster*, mesmo que outros nós apresentem falhas. A principal desvantagem desta solução é a replicação total da base de dados entre os nós do *cluster*. Com isso, torna-se necessário garantir que toda a base de dados possa ser inteiramente alocada em cada nó participante. Além disso, o processamento de cada consulta está condicionado aos limites de processamento impostos por uma única instância do Neo4j.

Apesar de uma única instância do Neo4j ser capaz de lidar com bases de dados de milhões de vértices e arestas, caso a base de dados seja suficientemente grande, esta solução poderá enfrentar problemas ao gerenciá-la. Isso ocorre porque o bom desempenho no Neo4j está fortemente associado ao uso de *caches* para manter a maior parte possível do grafo em memória principal. Eventualmente, os recursos de uma única máquina podem ser esgotados, como os limites da memória RAM por exemplo. A partir deste ponto, uma grande queda de desempenho poderá ser verificada, devido à intensa permuta de informações no cache. Para solucionar este problema, o Neo4j utiliza uma estratégia baseada em fragmentação de cache nas soluções de alta disponibilidade. Cada nó computacional de um *cluster* especializa o seu *cache* local de acordo com as consultas de um grupo de usuários. Esta ideia parte do princípio que as consultas de diferentes usuários envolverão diferentes regiões de vizinhança do grafo. Desta forma, esta medida diminui a sobreposição de elementos nos diferentes caches de cada nó do *cluster*.

2.1.2. Sparksee

O Sparksee (SPARKSEE, 2014) é um SGBDG proprietário, centralizado, que é mantido pela empresa Sparsity Technologies. Este sistema era conhecido anteriormente como DEX e teve seu nome alterado no lançamento da versão 5.0 em fevereiro de 2014. Seu núcleo foi desenvolvido em C++ e possui API que permitem a sua utilização com: C++, Java, .NET e Python.

Este SGBDG armazena grafos de propriedades e suporta transações, garantindo as propriedades ACID. Oferece as operações básicas necessárias para recuperar os elementos armazenados, bem como explorar travessias em torno de sua vizinhança. Suporta alguns algoritmos de grafos como cálculo de menor caminho e conectividade, por exemplo.

As travessias são utilizadas para visitar os vértices em um grafo, partindo de um determinado vértice, em uma determinada ordem e de acordo com restrições que podem ser previamente definidas pelo usuário. No Sparksee, a ordem de visitação dos vértices pode seguir uma busca em largura ou em profundidade. O usuário pode restringir os tipos de vértices e arestas que serão percorridos, a direção da travessia e a profundidade máxima a ser alcançada a partir do vértice de origem.

Os algoritmos de cálculo de menor caminho têm como objetivo encontrar um caminho entre dois vértices de tal forma que a soma dos pesos das arestas que o constitui seja mínima. Duas implementações são suportadas: a primeira é baseada na busca em largura e é útil para resolver o problema do menor caminho nos casos onde o grafo é não ponderado. A segunda implementação utiliza o algoritmo de Dijkstra e pode ser utilizada com grafos ponderados, desde que não existam pesos negativos. Os algoritmos de menor caminho também suportam as mesmas restrições que podem ser definidas pelo usuário em travessias.

Algoritmos de conectividade são fornecidos para a descoberta de componentes conexos do grafo. Um componente conexo é um subgrafo tal que existe um caminho entre quaisquer dois vértices deste subgrafo e, ao mesmo tempo, não existe conexão com qualquer outro vértice do supergrafo.

O Sparksee fornece uma solução de alta disponibilidade e escalabilidade horizontal chamada Sparksee HA. Esta solução é bastante semelhante à solução de alta disponibilidade oferecida pelo Neo4j. Neste caso, um balanceador de carga distribui as requisições dentre diversas instâncias Sparksee. Uma instância é eleita mestre e as outras são denominadas escravas. Operações de escrita são sempre realizadas pela instância mestre e são assincronamente propagadas para as instâncias escravas. Operações de leitura podem ser realizadas pelas instâncias escravas diretamente. A coordenação entre

as instâncias Sparksee é feita por meio do Apache ZooKeeper (HUNT *et al.*, 2010), que é um serviço de coordenação de aplicações distribuídas.

A forma de representação das entidades no Sparksee é baseada em estruturas de dados como *bitmaps* e mapas. A estrutura do grafo é representada por meio de uma matriz de adjacências, que é dividida em vários pequenos índices com o objetivo de melhorar a gestão das cargas de trabalho, aprimorando as operações de E/S e políticas de *cache*. A lista de adjacências de cada vértice é representada no formato bitmap que possui um *bit* representativo indicando cada adjacência daquele elemento. Os bitmaps passam por um processo de compactação para reduzir o consumo de memória e facilitar o seu gerenciamento. Os atributos armazenados nos vértices e arestas podem ser indexados. Duas estruturas de mapas principais são mantidas pelo SGBDG, onde um deles relaciona o identificador do elemento (chave) com os valores de propriedades que aquele elemento possui (valor) e o outro relaciona os valores das propriedades (chave) com o identificador dos elementos onde ela aparece (valor). Estas e outras informações sobre a arquitetura da solução e decisões de implementação são abordados em (MARTÍNEZ-BAZAN *et al.*, 2007).

Em (BARGUÑÓ *et al.*, 2011) foi conduzido um estudo utilizando o DEX com o intuito de desenvolver uma solução de banco de dados de grafo distribuída e paralela de forma a alcançar um bom desempenho em grafos de grande escala (milhões de vértices e arestas). Este trabalho focou na especialização dos caches das bases de dados e demonstrou ser possível construir SGBDG distribuídos e escaláveis fragmentando e distribuindo os dados entre os nós computacionais no nível físico, sem considerar o formato de grafo. Técnicas de fragmentação mais complicadas puderam ser evitadas utilizando o método de especialização de cache proposto pelo trabalho. Resultados de aceleração super linear foram apresentados para base de dados de milhões de objetos.

2.2. Bancos de Dados de Grafos Distribuídos

Esta seção descreve os principais SGBDG que suportam nativamente a distribuição dos dados entre múltiplos nós computacionais. Diferente das soluções

tradicionais como o Sparksee e o Neo4j, eles não exigem que a base de dados seja totalmente replicada em todos os nós. Neste caso o subsistema é preparado para distribuir e localizar os elementos do grafo entre os nós existentes. Réplicas também podem ser criadas para lidar com as questões de alta disponibilidade e escalabilidade geralmente oferecidas por estas soluções.

2.2.1 InfiniteGraph

O InfiniteGraph (INFINITEGRAPH, 2014) é um SGBDG que suporta distribuição de dados e a execução de travessias com processamento paralelo. Permite a configuração de um esquema de dados mais rígido, que impõe restrições de integridade quanto aos tipos de dados e relacionamentos armazenados. Controle de transação e concorrência, suporte a índices e presença de cache para manter porções mais acessadas do grafo em memória são algumas das características presentes. Possui uma arquitetura modular que permite a utilização de *plug-ins* de forma a expandir as suas funcionalidades originais.

As operações de travessias efetuadas pelo InfiniteGraph seguem uma forma de execução bastante parecida com a do Neo4j e do Sparksee. A descrição da travessia inclui definir o vértice de origem, a estratégia de travessia, um conjunto de qualificadores e um processador de resultados.

A estratégia de travessia inclui as operações de busca em largura ou busca em profundidade. Ao contrário das outras soluções, o InfiniteGraph também suporta a implementação de uma estratégia de travessia personalizada onde o próprio usuário pode definir como ocorrerá a expansão dos elementos da busca bastando, para isso, escrever sua própria estratégia (que deverá implementar a interface *Guide* do InfiniteGraph).

Ao realizar uma operação de travessia, o usuário geralmente está interessado em obter como resposta um conjunto de caminhos que seguem um determinado critério preestabelecido. O conjunto de qualificadores é utilizado para este fim e tem como objetivo determinar o quão próximo o estado atual da busca se encontra do resultado esperado pelo usuário. Utilizando qualificadores, os usuários podem escrever funções que

analisam o estado atual da busca e podam caminhos que não sejam mais relevantes para a consulta pretendida. Eles também são utilizados para definir, dentre todos os caminhos percorridos, aqueles elegíveis como resultados finais da busca.

O processador de resultados é uma funcionalidade que está associada ao qualificador de resultados e executa uma lógica definida pelo usuário para cada caminho da travessia que é considerado elegível de ser retornado como resultado pelo qualificador. Pode ser utilizado para propósitos simples como contar o número de resultados retornados por uma travessia ou para operações mais complexas que envolvam avaliar os elementos retornados por cada resultado da busca. Seu propósito geral é permitir que operações definidas pelo usuário possam ser executadas para cada resultado da busca durante o andamento da mesma.

Por possuir uma arquitetura modular, o InfiniteGraph permite que detalhes da camada de armazenamento dos dados possam ser amplamente personalizados pelo usuário. Todos os elementos são armazenados em um ou mais arquivos do banco de dados. Cada arquivo pode possuir um ou mais recipientes, que são estruturas capazes de agrupar os elementos fisicamente em memória ou no disco e são a menor unidade de bloqueio (*lock*) possível. Cada recipiente pode conter duas ou mais páginas. As páginas são unidades mínimas de crescimento do recipiente e de transferência para o disco. Cada página pode conter qualquer número de elementos do grafo. Regras podem ser definidas para gerenciar a localização desses elementos e definir estratégias de criação e povoamento dos recipientes, dos arquivos de banco de dados e da localização onde estes arquivos serão finalmente armazenados.

As estratégias de distribuição do InfiniteGraph permitem que os arquivos de banco de dados possam ser distribuídos em diferentes localizações. Zonas podem ser definidas para englobar um conjunto de localizações geralmente próximas geograficamente. Estas preferências são configuráveis em arquivos XML que funcionam como descritores utilizados pelo InfiniteGraph. Tanto nas operações de escrita quando nas operações de leitura, o InfiniteGraph tenta seguir a ordem de prioridade de distribuição definida pelo usuário. Caso a localização preferencial não possua mais espaço para armazenamento ou esteja inacessível, o InfiniteGraph tentará utilizar a próxima localização definida.

Para leitura e execução de travessias eficientes sobre os dados distribuídos, o InfiniteGraph propõe duas estratégias: a primeira é permitir que o usuário escolha uma estratégia de fragmentação baseada no domínio da aplicação de forma a tentar balancear os fragmentos e diminuir o número de arestas de corte. A segunda estratégia é a execução de travessias baseadas em troca de mensagens entre os nós, executando o máximo da travessia em memória antes de comunicar-se com outros nós uma vez que esta operação é muito custosa. Uma solução de cache distribuído também foi implementada para execução da travessia de forma eficiente. Uma porção do grafo local de um nó é mantido em memória e os elementos externos que forem muito referenciados também são armazenados em um cache local.

2.2.2 Titan

O Titan (TITAN, 2014) é um SGBDG escalável capaz de armazenar e consultar grafos de propriedades em um ambiente de múltiplos nós computacionais. Possui uma arquitetura de armazenamento modular, sendo desta forma compatível com várias soluções de armazenamento. Soluções como Oracle Berkeley DB[1], Apache Cassandra[2] e Apache HBase[3] são atualmente suportadas como formas de armazenamento. Apesar de ser um banco de dados de grafos, o Titan utiliza formas de armazenamento de dados que não são especializados no armazenamento de estruturas de grafos.

O BerkeleyDB é um SGBD do tipo chave/valor. O modelo de dados do Cassandra, assim como o do HBase, é baseado no armazenamento de dados estruturados em família de colunas. O sistema de armazenamento de dados do Cassandra é baseado no DynamoDB (DECANDIA *et al.*, 2007) enquanto o do HBase é baseado no BigTable (CHANG *et al.*, 2008). A escolha de uma solução de armazenamento é dependente de

[1] Oracle Berkley DB - http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[2] Apache Cassandra - http://cassandra.apache.org/

[3] Apache HBase - http://hbase.apache.org/

uma análise da relação custo/benefício entre tolerância a falhas na rede, disponibilidade e consistência uma vez que o teorema CAP (GILBERT & LYNCH, 2002) estabelece que em um sistema distribuído não pode apresentar as três características ao mesmo tempo.

O Titan é SGBDG voltado para aplicações OLTP, que suporta gerenciar uma grande quantidade de transações simultaneamente. Esta solução dá suporte à criação de um esquema de dados que é capaz de estabelecer restrições de integridade às propriedades atribuídas aos elementos do grafo. Esta característica da solução permite que o usuário possa definir os tipos dos dados que serão armazenados em determinadas arestas ou propriedades do grafo, e é capaz de garantir uma série de restrições como: unicidade, direcionamento de arestas, cardinalidade nos relacionamentos criados, dentre outras. O suporte a um esquema de dados também permite que esta solução consiga alcançar uma maior eficiência no armazenamento dos dados.

Uma importante característica do Titan é o suporte a índices centrados em vértices. Esta característica torna possível recuperar rapidamente os vizinhos de um vértice, obtendo rapidamente um conjunto de arestas que seja relevante para uma determinada travessia. Esta característica é particularmente útil em bases de dados que possuam super vértices, isto é, vértices que possuem alto grau de conectividade. As operações de travessia são aceleradas pois as arestas relevantes podem ser retornadas diretamente pelos índices sem a necessidade de percorrer toda a lista de arestas adjacentes ao vértice.

Apesar de ser uma solução de banco de dados distribuído, o Titan atualmente não aproveita os aspectos de distribuição para execução de consultas ou travessias em paralelo.

2.2.3 OrientDB

O OrientDB (ORIENTDB, 2014) é um SGBDG distribuído que utiliza um modelo de dados híbrido combinado com o armazenamento de documentos. É uma solução aderente às propriedades ACID, compatível com um subconjunto da linguagem SQL e pode ser utilizado como uma solução embarcada.

A distribuição e replicação dos dados é gerenciada por meio de uma biblioteca externa ao projeto chamada Hazelcast[4]. Esta biblioteca é capaz de simular um ambiente de memória compartilhada entre os nós participantes do *cluster* e garante alta disponibilidade em caso de falhas. O usuário também é capaz de ajustar as configurações de distribuição definindo qual nó do cluster é responsável por armazenar algum tipo de dado específico, por exemplo. Nas versões mais recentes também foi adicionado o suporte a transações distribuídas. Esta solução não é capaz, no momento, de tirar proveito da distribuição de dados para execução de consultas em paralelo entre as bases existentes.

2.3. Arcabouços de Processamento Paralelo em Grafos

Os arcabouços de processamento em grafos são soluções especializadas em realizar análises em dados armazenados em formato de grafos. São utilizados para executar uma computação específica sobre uma base de dados de grafos. Algumas computações frequentes incluem o cálculo de métricas como: medidas de centralidade, conectividade (detecção de componentes fracamente e fortemente conectados), cálculo de diâmetro da rede, densidade do grafo, grau médio dos vértices, PageRank. Além dos exemplos citados, outros algoritmos mais gerais definidos pelos usuários também podem ser executados por alguns desses arcabouços cujo objetivo é proporcionar um modelo de computação paralela e distribuída, abstraindo detalhes de implementação para facilitar o processamento de grafos em grande escala. Estas soluções são mais utilizadas quando o domínio exige analisar ou realizar algum tipo de computação na estrutura de um grafo como um todo. Os arcabouços não são, portanto, SGBDG. São apenas ferramentas para a execução de operações em grandes grafos com alto desempenho. Eles não suportam transações *on-line*, controle de acesso e outras operações características tradicionais de SGBD.

O modelo de computação destas soluções, ao contrário dos SGBDG, é baseado em processamento em lote. Outra diferença é que geralmente lidam com uma carga de

[4] Hazelcast - http://hazelcast.com/

trabalho fixa e bem definida do início ao fim do processamento. Aplicações destas ferramentas são encontradas em diversas áreas como análise de redes sociais, mineração de dados e gestão do conhecimento (inferência de conhecimento a partir de relações identificadas em grandes bases de dados de grafos, por exemplo). Alguns exemplos de arcabouços de processamento paralelo em grafos são: SNAP (BADER & MADDURI, 2008), Parallel Boost Graph Library (GREGOR & LUMSDAINE, 2005), Multithreaded Graph Library (BARRETT *et al.*, 2009), Knowledge Discovery Toolbox (LUGOWSKI *et al.*, 2012), GraphCT (EDIGER *et al.*, 2013) e STINGER (BADER *et al.*, 2009).

O SNAP é um arcabouço desenvolvido para realizar análise de redes sociais e foi projetado para a arquitetura de memória compartilhada. Os grafos originados a partir de redes sociais pertencem ao conjunto de grafos naturais, que são aqueles obtidos a partir de interações do mundo real. Características dos grafos naturais como: pequeno diâmetro de rede (característica das redes de pequeno mundo), conectividade esparsa, distribuição irregular de graus entre os vértices foram levadas em consideração para desenvolver uma solução capaz de obter maior grau de paralelismo e escalabilidade nestes cenários.

O Parallel Boost Graph Library (PBGL) é uma biblioteca de processamento paralelo e distribuído em grafos. Foi desenvolvido para a arquitetura de memória distribuída e utiliza o MPI (GEIST *et al.*, 1996) para troca de mensagens entre os nós. Foi desenvolvida em C++ e oferece suporte à execução de uma série de algoritmos em grafos como: busca em largura, busca em profundidade, busca de caminhos mínimos (Dijkstra), detecção de árvore geradora mínima, componentes conectados, PageRank, medidas de centralidade, coloração de grafos, dentre outros.

O Multithreaded Graph Library (MTGL) é uma biblioteca em C++ cujo desenvolvimento foi inspirado na PBGL. Ao contrário da PBGL, a MTGL foi desenvolvida para a arquitetura de memória compartilhada, mais especificamente para ser compatível com a arquitetura de alto desempenho presente em supercomputadores como no Cray, que é chamada de MTA. Devido a dificuldades no processamento de grafos em grande escala como baixa localidade, comunicação não estruturada e balanceamento de carga ineficiente, muitas vezes seu processamento tem sido realizado por arquiteturas específicas, com alto grau de paralelismo e de alto desempenho como o Cray XMT. Aplicações desenvolvidas para a arquitetura MTA só podem ser executadas

nesta mesma arquitetura. O MTGL surge como uma coleção de algoritmos e estruturas de dados preparados para serem utilizados nas arquiteturas paralelas como a encontrada no Cray e, com a ajuda da biblioteca Qthreads, também nas arquiteturas de multiprocessamento simétrico (SMP) e hardware *commodity* como estações de trabalho com múltiplos núcleos de processamento.

O Knowledge Discovery Toolbox (KDT) oferece um conjunto de operações de alto nível em Python para o processamento de grafos. O objetivo é permitir que especialistas de um determinado domínio possam realizar diversas computações de alto nível em grafos sem a necessidade de programar em paralelo ou ser um especialista na área de grafos. Escalabilidade e bom desempenho são alcançados armazenando a estrutura de grafos em matrizes esparsas e utilizando o Combinatorial BLAS (BULUC & GILBERT, 2011) como infraestrutura subjacente para computação das primitivas de álgebra linear que envolvem as operações de grafos.

O GraphCT e o STINGER são arcabouços que permitem a análise de grafos de grande escala em um ambiente de memória compartilhada. Desenvolvidos para arquiteturas altamente *multithread* como o Cray XMT. O GraphCT permite análises comuns como detecção de comunidades, componentes conectados e medidas de centralidade. O STINGER é especializado em *streaming* de grafos e armazena dados temporais sobre as alterações em topologia, permitindo que o usuário monitore como determinadas características do grafo (como a formação de comunidades, por exemplo) se comportam e sofrem alterações no decorrer do tempo.

2.3.1. Map/Reduce

O Map/Reduce (DEAN & GHEMAWAT, 2008) foi utilizado por bastante tempo como o arcabouço predominante para computação em grafos de grande escala. Este arcabouço facilita a distribuição e processamento dos dados em paralelo escondendo a complexidade destas operações da aplicação do usuário. É responsável por permitir o processamento de grandes volumes de dados em paralelo, dividindo o trabalho em um conjunto de tarefas independentes. Proporciona recursos e facilidades adicionais às

aplicações do usuário, tais como: distribuição e paralelismo transparente ao usuário, tolerância a falhas, escalonador de E/S que distribui as tarefas dinamicamente entre os nós disponíveis e ferramentas para monitorar o andamento dos trabalhos agendados.

O modelo de programação do Map/Reduce inclui um conjunto de dados de entrada e saída no formato chave e valor, bem como duas funções que são definidas pelo usuário: a função de mapeamento (*map*) e a função de redução (*reduce*). A função de mapeamento processa cada entrada no formato chave e valor e produz uma tupla de saída com uma chave de saída e um conjunto de valores intermediários. Cada instância da função de redução agrupa todas as tuplas de mesma chave geradas pelas funções de mapeamento e gera um resultado que corresponde à saída da computação para a chave correspondente, realizando alguma operação a partir dos dados de entrada. A vantagem deste modelo de programação é que o usuário deve transformar o seu problema em operações de mapeamento e redução e, uma vez definidas estas funções, as demais questões de distribuição, paralelismo e tolerância a falhas são gerenciadas automaticamente pelo próprio ambiente de processamento.

A arquitetura inclui uma série de nós computacionais formando um *cluster* onde um nó é eleito mestre e os demais nós são chamados de trabalhadores. O arquivo de entrada é armazenado de forma distribuída entre os nós utilizando o Google File System – GFS (GHEMAWAT *et al.*, 2003). O nó mestre aloca os nós trabalhadores ociosos para realizar operações de mapeamento ou redução. O arquivo de entrada é dividido em fragmentos que são consumidos pelos nós trabalhadores responsáveis por processar as funções de mapeamento, responsáveis por processar a entrada e armazenar os resultados intermediários obtidos pela execução desta função. Os trabalhadores responsáveis por executar as funções de redução agrupam e ordenam os resultados intermediários que foram gerados, passando estes dados à função de redução definida pelo usuário. Os resultados calculados pelos trabalhadores que executam a função de redução são escritos em arquivos de saída, também armazenados no GFS. A computação termina quando as operações de mapeamento e redução de todos os nós foram completadas. Este modelo é representado na Fig. 1. A implementação original da computação Map/Reduce proposta por (DEAN & GHEMAWAT, 2008) pertence à empresa Google. A distribuição

equivalente mais utilizada é o Apache Hadoop[5], que é disponibilizado livremente e possui código fonte aberto.

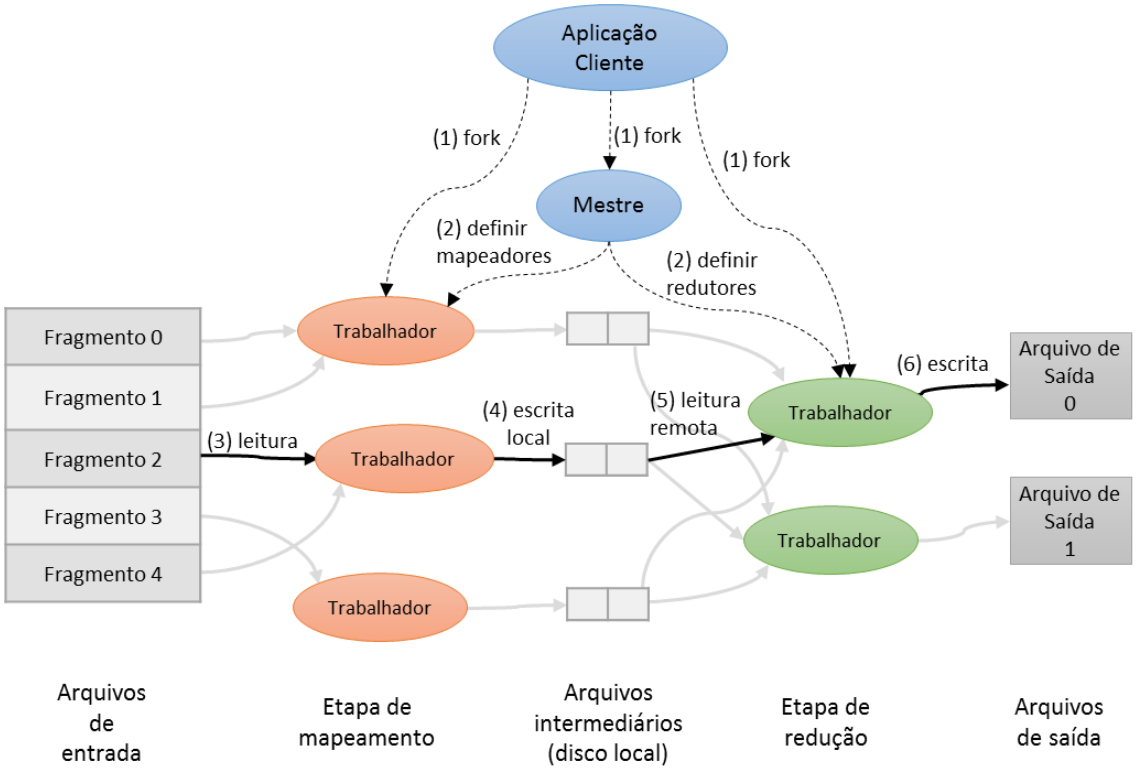


Fig. 1- Arquitetura do arcabouço Map/Reduce. Adaptado de (DEAN & GHEMAWAT, 2008).

Uma importante aplicação do arcabouço do Map/Reduce sendo utilizado para processamento de grafos é encontrado em (YANG *et al.*, 2009). Outra aplicação relevante é encontrada na solução conhecida por Faunus (FAUNUS, 2014), que propõe transformar consultas escritas pelo usuário em uma linguagem de alto nível em uma série de funções Map/Reduce, agilizando o processamento de grafos em grande escala. O Faunus é uma solução comumente utilizada junto com o banco de dados Titan para execução de operações de análises de dados (OLAP – *On-Line Analytical Processing*) em banco de dados de grafos.

[5] Apache Hadoop - http://hadoop.apache.org/

Um dos problemas deste modelo é que ele apresenta problemas de escalabilidade e desempenho quando há muita dependência entre os dados que estão sendo processados. Pelo fato dos grafos apresentarem estas características, o Map/Reduce nem sempre é a melhor opção para processá-los. Muitas vezes, uma determinada computação de grafo precisa ser transformada não em uma, mas em uma sequência encadeada de execuções Map/Reduce de tal forma que o final de uma execução sirva de entrada para a execução seguinte. Sua natureza não permite armazenar os estados no decorrer desta computação, tornando necessária a transferência do estado do grafo entre cada execução, gerando sobrecarga e grande latência durante a computação. A grande dependência entre os dados acaba se tornando um ponto de contenção para este modelo de computação.

Por conta desta limitação, algumas iniciativas surgiram para tentar acelerar o desempenho da utilização do Map/Reduce para o processamento de grafos, como podemos verificar em (LIN & SCHATZ, 2010). Outros trabalhos relacionados envolvem a criação de novas soluções ou modificações no modelo de computação e na arquitetura do Map/Reduce para atender a este domínio específico. Dentre eles podemos citar: (LIN & DYER, 2010), (BU *et al.*, 2012), (KANG *et al.*, 2009), (ZHANG *et al.*, 2012), (CHEN *et al.*, 2010) e (ELNIKETY *et al.*, 2011).

2.3.2. Pregel

O Pregel (MALEWICZ *et al.*, 2010) surgiu como uma alternativa ao modelo de computação Map/Reduce. Assim como o Map/Reduce, o Pregel proporciona um modelo de computação que funciona como uma abstração para facilitar o processamento de dados em grande escala. Uma diferença importante é que o Pregel foi desenvolvido especialmente para lidar com grafos.

O modo de execução da computação no Pregel é inspirado no modelo de computação *Bulk Synchronous Parallel* (VALIANT, 1990). O modelo de computação do BSP possui basicamente três etapas. Na primeira etapa, cada nó computacional realiza o processamento local. Na segunda etapa, ocorre a troca de mensagens entre os nós. A terceira etapa é uma barreira de sincronização que garante que todos os nós terão

alcançado o mesmo ponto antes de prosseguir com a computação. Esta etapa é importante para garantir que todas as mensagens foram enviadas e recebidas conforme necessário e todos os nós possuem os valores atualizados a serem utilizados na próxima etapa da computação. Uma computação é composta por um conjunto de superpassos (*supersteps*) onde cada um engloba as três etapas descritas acima conforme representado na Fig. 2.

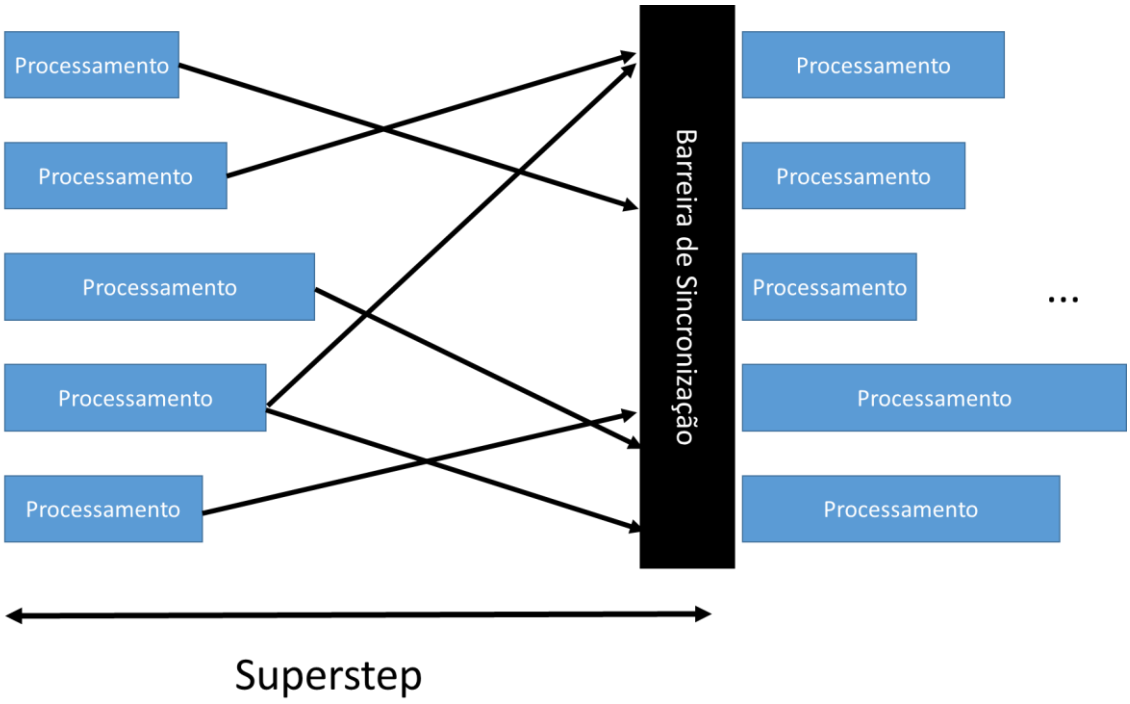


Fig. 2 - Modelo de processamento BSP. Etapa de processamento local, troca de mensagens e barreira de sincronização. O conjunto que compreende as três etapas forma um superpasso.

A entrada de uma computação no Pregel é um grafo direcionado. Cada vértice deve possuir um identificador único e um valor qualquer definido pelo usuário que poderá ser modificado no decorrer da computação. O modelo de computação sugere uma abordagem orientada ao ponto de vista de um vértice. É conhecida como computação centrada em vértice. Durante a execução de um superpasso, o arcabouço invoca uma função definida pelo usuário para cada vértice do grafo. Estas execuções ocorrem potencialmente em paralelo.

A função definida pelo usuário utiliza as informações do próprio vértice e das mensagens que foram recebidas por ele (enviadas no superpasso anterior) para definir o seu comportamento naquela etapa da computação. Ao final do processamento do superpasso atual segue a etapa de troca de mensagens entre o vértice e todos os seus vizinhos que estão conectados a ele por uma aresta de saída. Também é possível enviar mensagens diretamente a outros vértices mesmo que eles não sejam vizinhos bastando, para isso, saber o identificador do vértice de destino. As mensagens que forem enviadas nesta etapa serão recebidas pelos destinatários no superpasso seguinte. A presença de pontos de sincronização globais ao final do superpasso garante o sincronismo entre todos os nós envolvidos.

Um vértice é representado no modelo de computação do Pregel por dois estados possíveis: ativo ou inativo. No início da computação todos os vértices estão no estado ativo. Durante a execução da função definida pelo usuário um vértice pode mudar de estado ou trocar o estado de qualquer vértice vizinho na direção de saída. Um vértice que em um determinado superpasso se encontre no estado inativo não executa nenhum processamento. Um vértice inativo torna-se ativo novamente quando recebe uma mensagem enviada por outro vértice. As arestas não são "cidadãs de primeira classe" no modelo de computação do Pregel e não possuem nenhum tipo de computação associada a elas. A computação do Pregel termina quando todos os vértices se encontram no estado inativo e não existem mais mensagens sendo trocadas. O resultado da computação são os valores finais associados a cada um dos vértices.

Uma importante característica do Pregel é que o grafo pode sofrer alterações topológicas durante a execução da computação. Vértices e arestas podem ser adicionados ou removidos conforme necessário dependendo da lógica presente na função executada pelo usuário.

Durante o início do processamento, vértices e arestas são distribuídos entre os nós do *cluster* e comunicam-se uns com os outros por meio de troca de mensagens. Um vértice pode enviar qualquer número de mensagens durante a execução de um superpasso. Esta forma de comunicação é utilizada como uma maneira de simular um ambiente de memória compartilhada em uma arquitetura de memória distribuída. Em comparação ao modelo proposto pelo Map/Reduce, a troca de mensagens representa uma menor

sobrecarga computacional. Enquanto no Map/Reduce os nós redutores efetuam leituras remotas dos dados gerados pelos nós mapeadores (o que representa um alto custo de processamento devido à leitura de dados em disco e à comunicação entre os nós), no Pregel a latência gerada pelas trocas de mensagens pode ser amortizada pela entrega assíncrona e em lote.

Uma outra forma utilizada pelo arcabouço para diminuir a sobrecarga causada pelo envio de mensagens é o uso de combinadores. Os combinadores são funções que acumulam uma sequência de mensagens direcionadas a um determinado vértice e agrupam estas informações, enviando uma única mensagem ao vértice de destino diminuindo, assim, a quantidade de mensagens em tráfego na rede. Um exemplo de uso de combinador seria uma função definida pelo usuário que conta a quantidade de mensagens recebidas por um determinado vértice. Supondo que esta informação seja relevante para uma determinada finalidade, a vantagem de utilizar combinadores neste cenário é que somente uma mensagem final com a quantidade de mensagens recebidas seria enviada a cada vértice de destino. Sem a sua utilização, todos os vértices receberiam inúmeras mensagens e seriam responsáveis por contabilizar a quantidade recebida, o que geraria um tráfego muito maior na rede. As operações realizadas por uma função combinadora especificada pelo usuário devem ser comutativas e associativas.

Os agregadores são mecanismos que permitem uma comunicação global e são úteis para monitoramento e obtenção de dados estatísticos. Cada vértice pode fornecer valores a um agregador global, que deve combinar os valores recebidos pelos diferentes vértices e gerar um valor final baseado em uma função predefinida ou especificada pelo usuário. Algumas funções predefinidas como: menor valor, maior valor e soma já são providos pelo arcabouço, por exemplo. O usuário também pode escrever uma função personalizada para um agregador, conforme necessário. As operações realizadas por estas funções devem ser comutativas e associativas.

O grafo de entrada pode ser fornecido em uma série de diferentes formatos como arquivos de texto, tabelas de um banco de dados relacional, dentre outras. Durante o desenvolvimento do arcabouço, optou-se por não definir um padrão com relação ao formato de entrada e saída da computação. Desta forma, diversos leitores e escritores

foram implementados e fornecidos junto da solução para lidar com as diferentes representações possíveis para o grafo.

O Pregel foi desenvolvido para execução em um cluster formado por milhares de PC de uso geral interligados por uma rede de alta largura de banda. No início da computação, o grafo original é dividido em fragmentos. Cada fragmento possui um conjunto de vértices e todas as arestas de saída desses vértices. Por padrão, um vértice é alocado em um determinado fragmento de acordo com o seu identificador e a função de fragmentação utilizada pelo Pregel é: $hash(ID)\ mod\ N$, onde N representa o número de fragmentos. O usuário também pode modificar a forma de fragmentação padrão conforme desejado.

No início da execução da computação cada nó do cluster recebe uma cópia do programa definido pelo usuário. Um nó é eleito mestre e os outros são os nós trabalhadores. Inicialmente, o nó mestre define em quantos fragmentos o grafo original será dividido e atribui cada fragmento a um nó trabalhador. Cada nó trabalhador pode receber um ou mais fragmentos. O número de fragmentos pode ser definido pelo próprio arcabouço ou gerenciado pelo usuário. O nó mestre fica responsável por coordenar as atividades dos nós trabalhadores, gerenciando o início, a troca de mensagens e o final de cada superpasso com a barreira de sincronização. Ao final de cada superpasso, cada nó trabalhador informa ao mestre quais são os vértices que estão no estado ativo e que, portanto, devem ser processados no próximo superpasso. O mestre também é responsável por instruir os trabalhadores a salvar o estado atual da computação em um meio persistente no início de cada superpasso. Desta forma, em caso de falha de um nó trabalhador, o mestre poderá redistribuir os fragmentos entre os nós trabalhadores presentes e reiniciar a computação a partir do último estado salvo.

O Pregel é um arcabouço proprietário que inspirou uma série de trabalhos que seguem a mesma abordagem de realizar uma computação centrada em vértices. Dentre os trabalhos semelhantes podemos citar: Giraph (CHING & KUNZ, 2011), GPS (SALIHOGLU & WIDOM, 2013), Trinity (SHAO *et al.*, 2013), GRE (YAN *et al.*, 2013) e Mizan (KHAYYAT *et al.*, 2013).

Ao contrário do Map/Reduce onde o modelo de computação é focado na leitura e escrita dos dados em um meio persistente, o Pregel exige que todo o grafo seja carregado em memória. Esta característica pode algumas vezes limitar o tamanho dos grafos processados por este arcabouço.

2.3.3. GraphLab

O GraphLab (LOW *et al.*, 2010) é um arcabouço voltado para execução de algoritmos de aprendizado de máquina utilizando grandes bases de dados de grafos em um ambiente de computação em nuvem. Este arcabouço propõe um modelo de computação com alto nível de abstração para lidar com análise de grafos em grande escala.

Este arcabouço implementa um modelo de computação diferente do BSP utilizado pelo Pregel. O modelo BSP se mostrou ineficiente para execução de alguns algoritmos de aprendizado de máquina como: *belief propagation,* filtragem colaborativa e PageRank, por exemplo. O desempenho do modelo BSP acaba sendo limitado pelo tempo de execução do nó mais lento em cada superpasso. Se o tempo de execução deste nó for muito superior aos demais, uma grande parcela de tempo da computação será perdida com nós ociosos aguardando o término do processamento de seus vizinhos. Para alcançar um bom desempenho nestas aplicações, seria fundamental utilizar um novo modelo de computação diferente do BSP.

Assim como o Pregel, este arcabouço permite que o usuário escreva um algoritmo que resolva um determinado problema sob o ponto de vista de um vértice e o próprio arcabouço fica responsável por gerar execuções paralelas de alto desempenho. Ao contrário do Pregel, o GraphLab utiliza um modelo de computação assíncrono. Outra diferença é que as arestas podem possuir valores associados que fazem parte da computação realizada. No Pregel, as arestas têm um propósito mais simples de apenas indicar a direção de comunicação entre os vértices.

Da mesma forma que o Pregel, cada vértice executa um trecho de código definido pelo usuário em paralelo. Durante esta execução, o vértice possui acesso as suas arestas

e aos seus vértices vizinhos. Por utilizar uma abordagem assíncrona, a execução de uma computação neste arcabouço pode esbarrar em alguns problemas tradicionais da computação paralela como condições de corrida, por exemplo. As condições de corrida ocorrem quando existe uma sobreposição entre a vizinhança de dois vértices executados em paralelo. Um vértice pode estar escrevendo e modificando dados de sua vizinhança enquanto o outro vértice executa uma operação de leitura sobre esses mesmos dados. As condições de corrida podem levar a situações completamente distintas a cada execução do algoritmo. Alguns algoritmos toleram bem a falta de consistência e determinismo causado pelas condições de corrida enquanto outros algoritmos não geram resultados satisfatórios sob essas condições. Por esse motivo este arcabouço permite que o usuário configure o nível de consistência desejado conforme necessário, realizando uma troca entre o nível de consistência desejado e o grau de paralelismo alcançado pela computação. Quanto menor a exigência de consistência, maior o grau de paralelismo obtido e vice-versa.

2.3.4. PowerGraph

O PowerGraph (GONZALEZ *et al.*, 2012) é um arcabouço que surgiu como uma proposta de evolução do GraphLab. Durante a execução de experimentos, verificou-se que o GraphLab não apresentava um bom desempenho quando lidava com grafos naturais. Os grafos naturais são uma classe de grafos gerados a partir de interações do mundo real. Grafos naturais são encontrados por exemplo nas relações entre pessoas em uma rede social ou em interações entre usuários dentro de um sistema de recomendação. Os grafos naturais apresentam características particulares que dificultam o seu processamento na maioria dos arcabouços existentes. O PowerGraph apresenta um modelo de computação específico para lidar com grafos naturais de forma eficiente.

Os graus dos vértices presentes em grafos naturais tendem a seguir uma distribuição de lei de potência, isto é, uma pequena parcela dos vértices está interligada à grande maioria de todos os outros vértices do grafo. Esta característica dá origem a super-vértices. Os super-vértices são vértices que possuem alto grau de conectividade e são muito difíceis de fragmentar de forma eficiente nos arcabouços como o GraphLab e o

Pregel pois, independente da forma de fragmentação escolhida, o surgimento de um grande número de arestas de corte implica em um alto custo de comunicação entre os nós.

Em (LESKOVEC *et al.*, 2009) e (LANG, 2004) é discutido o fato de que os grafos naturais não possuem cortes balanceados de baixo custo. O custo computacional com a troca de mensagens é linear em relação ao número de arestas de corte.

Os arcabouços como GraphLab, Pregel e trabalhos semelhantes assumem que o grafo a ser processado possui uma distribuição de vértices e arestas aproximadamente uniforme. Estes arcabouços realizam uma fragmentação de Espalhamento, onde os vértices são alocados entre os nós disponíveis de acordo com o resultado de uma função de espalhamento predefinida. As características dos grafos naturais implicam em um alto custo de comunicação neste modelo, tornando difícil a sua utilização prática para este problema. De um modo mais geral, utilizando a fragmentação de Espalhamento, a porcentagem de arestas de corte presente no grafo será dada por: $1 - \frac{1}{p}$, onde p é o número de nós disponíveis. Um grande percentual de arestas de corte implica em pouco ganho de desempenho na utilização destes arcabouços comparado à utilização do Map/Reduce.

Existem algumas semelhanças entre este arcabouço e o Pregel. O processamento centrado em vértices por exemplo é semelhante. Além disso, cada vértice pode estar no estado ativo ou inativo e possui um valor associado a ele. Ao contrário dos outros arcabouços, no PowerGraph as arestas também possuem valores associados a elas.

Para lidar com grafos naturais de uma forma eficiente, o PowerGraph propõe dividir a computação em três fases principais: reunião *(gather)*, aplicação *(apply)* e dispersão *(scatter)*. Este modelo foi chamado de decomposição GAS. Além disso, ao contrário da fragmentação proposta pelos arcabouços anteriores, onde os vértices são divididos entre os nós existentes, este arcabouço propõe uma fragmentação do grafo original de tal forma que cada aresta e seus dois vértices sejam alocados unicamente em um determinado nó.

Na fase de reunião, as informações sobre os vizinhos são coletadas em paralelo pelo arcabouço para cada vértice ativo na computação. Do ponto de vista de um vértice específico, esta etapa da computação recupera todas as suas arestas vizinhas retornando

um conjunto de triplas contendo: dados da aresta, do vértice de origem e do vértice de destino. Cada tripla possui um valor final específico definido pela função de reunião implementada pelo usuário. Após o cálculo do valor de cada tripla, é invocada uma função de soma associativa e comutativa que também é definida pelo usuário e deve gerar um valor final baseado nos valores das triplas recebidas como entrada. O valor final gerado pela função de soma indica o próximo valor que aquele vértice em questão irá assumir. A fase de aplicação utiliza as informações do próprio vértice combinando-as com o valor obtido pela função de soma e gera um novo valor para o vértice que está sendo processado de acordo com uma função definida pelo usuário. A fase de dispersão também é uma função definida pelo usuário, que tem como objetivo atualizar os valores das arestas vizinhas ao vértice que teve seu valor modificado. Esta função recebe como parâmetro o valor atualizado do vértice calculado na fase anterior junto com informações da própria aresta e de seu vizinho e, com base nesses parâmetros, atualiza o valor de cada aresta. Esta etapa também é executada em paralelo pelo arcabouço.

O PowerGraph oferece suporte a três tipos de fragmentação possíveis: aleatória, gulosa coordenada e gulosa indiferente. Na estratégia de fragmentação aleatória cada aresta é unicamente alocada em um fragmento. As estratégias gulosas dão preferência a nós que já possuam o vértice de origem ou o vértice de destino no momento de alocação de uma aresta. Em caso de empate, o nó que possui a menor quantidade de fragmentos alocados fica responsável por armazenar a aresta de forma a manter o balanceamento dos fragmentos. A diferença entre as duas estratégias gulosas é que a coordenada mantém uma estrutura distribuída em memória para coordenar a alocação de cada aresta enquanto a estratégia indiferente realiza um processamento local, sem comunicação com os outros nós. A estratégia coordenada é mais lenta, mas oferece fragmentos de melhor qualidade enquanto a estratégia indiferente é mais ágil, mas possui menor qualidade quanto os fragmentos gerados.

As formas de execução suportadas pelo arcabouço são: execução síncrona, assíncrona e assíncrona serializável. A execução síncrona segue um modelo semelhante ao BSP, onde cada vértice executa as fases do modelo GAS de forma síncrona e coordenada. A forma de execução assíncrona permite intercalar as fases e o arcabouço

tenta evitar condições de corrida. A forma de execução assíncrona serializável garante que vértices vizinhos não são executados de forma concorrente.

Os arcabouços como o Pregel e o PowerGraph são amplamente conhecidos como sistemas de paralelismo em grafos enquanto arcabouços baseados em Map/Reduce, de uma forma mais geral, são conhecidos como sistemas de processamento de dados em grande escala. Um problema que geralmente não é abordado por arcabouços de paralelismo em grafos é a carga, construção e transformação da base de dados de grafos na qual o algoritmo será executado. Por se tratar de grafos em grande escala, estas operações podem acabar se tornando tão problemáticas quanto a própria execução do algoritmo. Os arcabouços baseados em Map/Reduce, por outro lado, conseguem gerenciar as operações de extração, transformação e carga (ETL) de forma eficiente. Por este motivo, em (XIN *et al.*, 2013) é possível encontrar uma proposta de combinar sistemas de processamento em grafos com sistemas de processamento de dados em grande escala de forma a conseguir aproveitar as vantagens oferecidas por cada solução. Este trabalho utiliza o Spark (ZAHARIA *et al.*, 2012), que é uma solução baseada no arcabouço Map/Reduce para realizar, de forma eficiente, a extração, transformação e carga da base de dados de grafos. Um novo modelo de abstração para computação em grafos chamado RDG (*Resident Distributed Graphs*) é proposto, criando primitivas para manipulação de grafos no Spark. Implementações dos modelos do Pregel e PowerGraph foram construídos sobre o RDG utilizando as operações primitivas. O desempenho desta solução, que ficou conhecida por GraphX (XIN *et al.*, 2013), foi comparado ao Apache Mahout[6] (baseado no Hadoop) e ao PowerGraph. Os tempos de execução demonstraram larga vantagem do GraphX sobre o Mahout apesar de ter tido um desempenho inferior ao do PowerGraph.

[6] Apache Mahout - https://mahout.apache.org/

2.3.5. Mizan

O Mizan (KHAYYAT *et al.*, 2013) é um arcabouço baseado no Pregel que propõe uma melhoria de desempenho no tempo de execução e um melhor balanceamento de carga entre os nós analisando e modificando dinamicamente a estratégia de alocação dos elementos do grafo. De acordo com este trabalho, a origem da falta de balanceamento na execução do algoritmo pode estar em dois pontos principais: na própria estrutura do grafo ou no algoritmo executado pelo usuário.

O desequilíbrio ocorre na estrutura do grafo quando um nó se torna responsável por um grande número de vértices densamente conectados, o que gera uma crescente necessidade de processamento local e de comunicação com seus vizinhos. Quanto aos algoritmos executados, verificou-se que podem ser divididos em duas categorias principais: algoritmos estacionários e não estacionários. Os algoritmos estacionários apresentam um comportamento constante desde o início até o fim da computação. Um algoritmo é considerado estacionário quando seus vértices ativos enviam e recebem a mesma distribuição de mensagens no decorrer dos superpassos. No final da computação, todos os vértices passam para o estado inativo no mesmo superpasso. Os algoritmos não estacionários, por sua vez, são aqueles onde o destino ou a quantidade de mensagens trocadas entre os vértices varia no decorrer dos superpassos. Algoritmos não estacionários contribuem para o problema da falta de balanceamento entre os nós e esta questão serve de motivação para a proposta de gerenciar o balanceamento de forma dinâmica, em tempo de execução, e não de maneira estática como feito por outros arcabouços.

Para realizar o balanceamento dinâmico, o Mizan monitora métricas de cada vértice com a finalidade de detectar os possíveis pontos de contenção. As métricas obtidas são: número de mensagens de saída para outros nós trabalhadores, número de mensagens de entrada e tempo de execução do superpasso atual. Valores altos para qualquer uma dessas métricas indicam um possível elemento candidato à realocação. O plano de migração é executado por cada nó trabalhador em paralelo, sem a necessidade de um coordenador central. O arcabouço utiliza uma segunda barreira de sincronização, entre a barreira BSP e o início do próximo processamento local no superpasso seguinte, chamada de barreira de migração. Tarefas referentes à identificação dos candidatos à migração e à efetiva migração de elementos ocorrem nesta etapa, entre a barreira BSP e a barreira de

migração. As decisões que são tomadas quanto à migração dos elementos não utilizam nenhuma informação prévia sobre o grafo ou sobre o algoritmo especificado pelo usuário. Toda decisão de migração é baseada nas métricas obtidas em tempo de execução.

2.4. Fragmentação em Grafos

O problema de fragmentação em grafos consiste em dividir um grafo $G = (V, E)$ em componentes menores de forma que cada componente ou fragmento seja um subgrafo de G. Cada fragmento contém um subconjunto V' e E' do conjunto original de vértices V e arestas E. Algumas arestas ligam vértices pertencentes a fragmentos diferentes, garantindo a conectividade presente no grafo original. Tais arestas são denominadas "arestas de corte". Algumas técnicas evitam arestas de corte duplicando vértices ao alocá-los em mais de um fragmento. O problema de fragmentação em grafos tem aplicações em áreas de computação científica, programação paralela e distribuída, processamento de imagens, detecção de comunidades em redes sociais, biológicas, dentre outras (BULUC *et al.*, 2013).

As estratégias de fragmentação de grafos, de um modo geral, podem ser classificadas em duas categorias: estratégias que levam em consideração a estrutura do grafo e estratégias que são alheias a ela. As estratégias que levam em consideração a estrutura do grafo geralmente tentam obter fragmentos com aproximadamente a mesma quantidade de elementos enquanto o número de arestas de corte entre fragmentos é minimizado. As estratégias alheias à estrutura do grafo, por sua vez, adotam formas mais simples de dividir o conjunto de vértices e arestas entre os fragmentos, utilizando uma estratégia de alocação aleatória por exemplo.

A determinação de fragmentos nas estratégias de fragmentação que levam em consideração a estrutura do grafo é um problema computacional que pertence à classe NP-difícil. Por este motivo, heurísticas e algoritmos aproximativos são amplamente utilizados neste sentido. As primeiras iniciativas que surgiram em (KERNIGHAN & LIN, 1970) e (FIDUCCIA & MATTHEYSES, 1982) utilizavam estratégias de fragmentação local e fragmentavam o grafo gerando uma saída com dois fragmentos distintos. A

utilização destas técnicas de fragmentação local implica em uma fragmentação inicial arbitrária do conjunto de vértices, o que pode levar à baixa qualidade nos fragmentos obtidos.

Uma estratégia de fragmentação mais recente sugere a utilização de algoritmos de fragmentação global, isto é, algoritmos que levam em conta a estrutura do grafo como um todo para a obtenção dos fragmentos. A estratégia global mais comum é a fragmentação espectral. A técnica de fragmentação espectral proposta em (HAGEN & KAHNG, 1992) utiliza a matriz de adjacências A_{ij} (onde o valor 1 representa uma aresta entre o vértice i e o vértice j) e a matriz diagonal de graus dos vértices D_{ii} que representa o grau do vértice i, calculando, a partir delas, a matriz laplaciana $L = D - A$. De posse da matriz laplaciana L, os autovalores (λ) desta matriz são calculados. O segundo menor autovalor encontrado (o menor autovalor é considerado uma solução trivial para o problema) é utilizado para identificar o ponto de corte onde serão gerados os dois fragmentos. Uma fragmentação em um número maior de fragmentos pode ser alcançada repetindo o processo para cada fragmento encontrado.

Uma importante técnica de fragmentação utilizada atualmente é a fragmentação multi nível (KARYPIS & KUMAR, 1995). Esta técnica de fragmentação é geralmente utilizada em grandes bases de dados de grafos e é dividida em três fases: contração (*coarsening*), fragmentação e expansão (*uncoarsening*). Na fase de contração, a cada iteração o grafo original é transformado em grafos menores, sempre com uma menor quantidade de elementos. Vértices e arestas são agrupados em elementos menores de tal forma que os pesos se mantenham constantes. É necessário que o grafo original possa ser recalculado a partir do grafo reduzido obtido nesta fase. Após a fase de contração do grafo original, quando os elementos obtidos já são suficientemente pequenos, a fase de fragmentação se inicia e encontra o ponto que melhor divide o grafo em dois fragmentos. A técnica utilizada para encontrar este ponto pode ser a fragmentação espectral, geométrica ou métodos combinatórios. Qualquer que seja o método escolhido, neste ponto ele é executado muito mais rapidamente do que no grafo original já que o grafo passou pela etapa de contração de seus elementos anteriormente. Finalmente, após a etapa de fragmentação, ocorre a fase de expansão, que se comporta de forma contrária à fase de contração. A cada passo, o grafo reduzido expande e volta a dar origem ao grafo original,

já fragmentado. Durante esta etapa, a cada passo os vértices passam a adquirir um maior grau de liberdade. Por conta disso, um refinamento sobre a fragmentação obtida é executado para ajustar a fragmentação obtida na fase anterior reduzindo, desta forma, o número de arestas de corte. As fases de contração, fragmentação e expansão são representadas na Fig. 3. A fragmentação multinível é uma técnica que exige a análise global do grafo e uma grande quantidade de memória para a obtenção dos fragmentos. As soluções de fragmentação multinível mais conhecidas são: METIS (KARYPIS & KUMAR, 1998), hMETIS (KARYPIS & KUMAR, 1999) para fragmentação de hipergrafos e ParMETIS (KARYPIS & KUMAR, 1996) que estende as funcionalidades do METIS adicionando aspectos de paralelismo aos algoritmos de fragmentação.

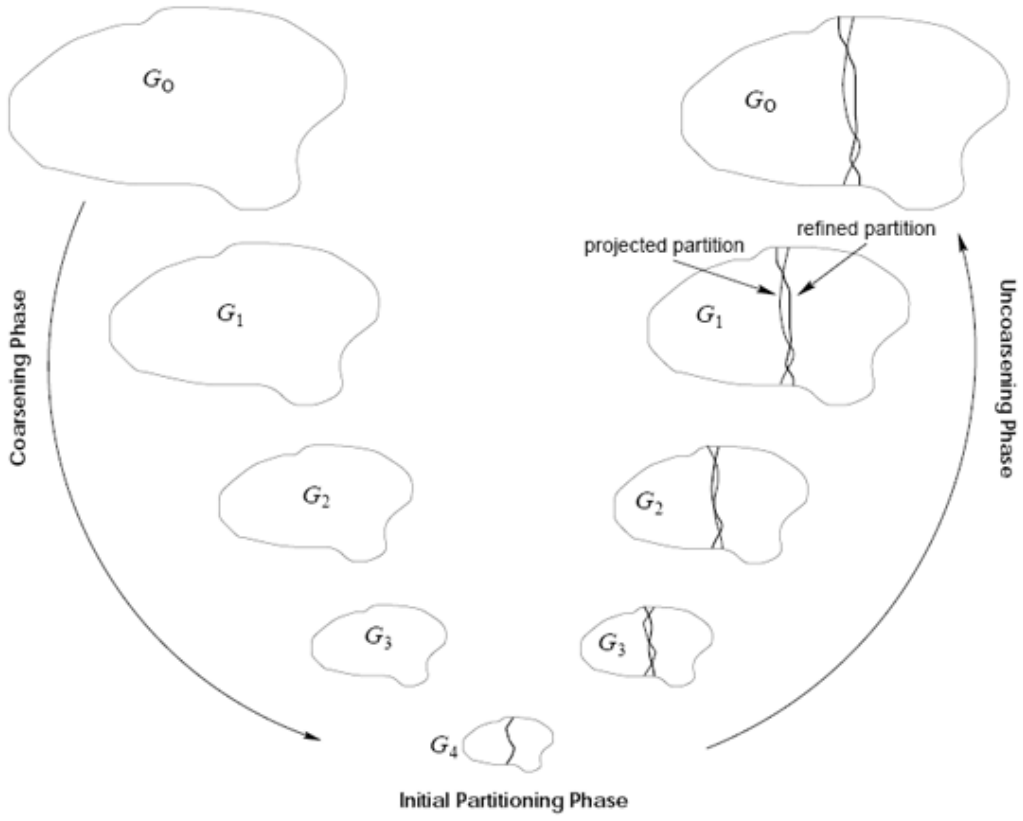


Fig. 3 – Fragmentação multinível representada nas etapas de contração, fragmentação e expansão.

Estas e outras inúmeras técnicas de fragmentação em grafos são suportadas por soluções populares de fragmentação como: Chaco (HENDRICKSON & LELAND, 1995), Jostle (WALSHAW & CROSS, 2000), Party (DIEKMANN *et al.*, 2000) e Scotch (CHEVALIER & PELLEGRINI, 2009).

As soluções de distribuição e paralelismo em grafos de modo geral fazem pouco uso de técnicas de fragmentações mais sofisticadas, que levam em consideração a estrutura do grafo. Isto ocorre pelo fato de que o cálculo dos fragmentos pode ser custoso e exigir uma grande quantidade de memória, o que poderia limitar o tamanho dos problemas abordados por estas soluções. Além disso, a maioria das soluções de distribuição e paralelismo em grafos (como os arcabouços descritos anteriormente) possui um propósito geral de forma a obter um desempenho razoável independente do formato do grafo utilizado como entrada. Ferramentas de fragmentação populares têm demonstrado baixo desempenho e baixa qualidade de fragmentos em grafos naturais ou com a presença de vértices densamente conectados (ABOU-RJEILI & KARYPIS, 2006). Técnicas de fragmentação mais simples alheias à estrutura do grafo como fragmentação aleatória, gulosa, 1D e 2D têm sido utilizadas para permitir a distribuição dos dados de forma eficiente entre os nós do *cluster*.

Arcabouços de processamento em grafos como o Pregel e suas implementações utilizam, por padrão, uma fragmentação de Espalhamento. Vértices são distribuídos entre os nós trabalhadores que ficam responsáveis também por todas as arestas de saída relacionadas a eles. Este tipo de fragmentação é chamado de *edge-cut*. O PowerGraph, ao contrário do Pregel, realiza uma fragmentação aleatória ou gulosa nas arestas, alocando-as entre os nós disponíveis. Como consequência, diversas cópias do mesmo vértice aparecem em fragmentos distintos. Este método é chamado de *vertex-cut* e sua principal vantagem é o menor custo de comunicação envolvido para realização da computação. A Fig. 4 representa uma comparação entre os métodos de fragmentação *edge-cut* e *vertex-cut*.

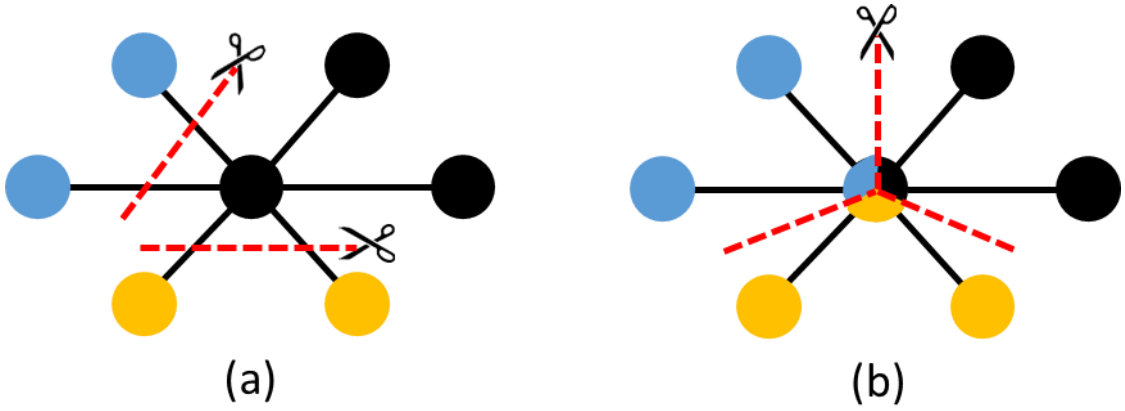


Fig. 4 - Comparação entre os métodos de fragmentação edge-cut (a) e vertex-cut (b) propostos por arcabouços de processamento paralelo em grafos. Divisão em três fragmentos distintos. Adaptado de (XIN *et al.*, 2013).

Além das principais estratégias utilizadas pelos arcabouços de processamento paralelo em grafos, outra estratégia de fragmentação utilizada é a fragmentação 1D e 2D. Esta estratégia de fragmentação é útil particularmente quando a estrutura do grafo é representada por uma matriz de adjacências.

Em (YOO *et al.*, 2005), (MUNTÉS-MULERO *et al.*, 2010) e (BULUÇ & MADDURI, 2011), por exemplo, são abordadas técnicas eficientes para execução de algoritmos paralelos e distribuídos de busca em largura em ambientes de clusters de memória distribuída. Estes trabalhos optaram por utilizar uma estrutura de dados de matriz de adjacências para armazenar e representar o grafo original. As operações de busca foram realizadas sobre esta estrutura de dados específica. Por este motivo, o problema de distribuição dos elementos do grafo passou a ser equivalente ao problema de fragmentar a matriz de adjacências entre os nós participantes. Neste contexto surgiram duas estratégias principais de fragmentação: 1D e 2D.

A fragmentação 1D consiste em dividir a matriz de adjacências igualmente entre os nós participantes. Os vértices e todas as arestas relacionadas a ele são atribuídos a um determinado nó. Já na fragmentação 2D, a matriz de adjacências é dividida em $R \times C$ blocos de linhas e C blocos de colunas conforme representado na Fig. 5. Os valores são escolhidos de tal forma que P, o número de processadores disponíveis, seja igual a $R \times C$.

A notação $A^{(*)}_{(i,j)}$ representa um bloco que pertence ao nó computacional (i,j). Cada nó fica responsável por armazenar C blocos (ver Fig. 6).

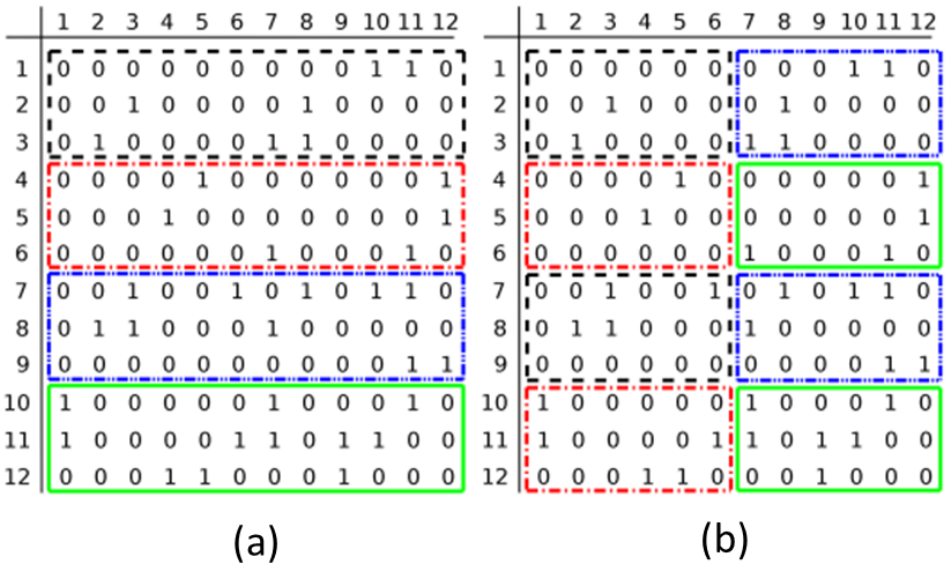


Fig. 5- Comparação entre as estratégias de fragmentação 1D (a) e 2D (b) distribuídos entre quatro nós computacionais. Adaptado de (MUNTÉS-MULERO *et al.*, 2010).

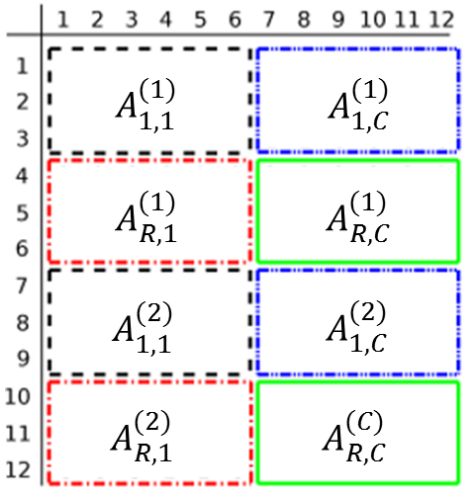


Fig. 6 – Representação da divisão em blocos de linhas e blocos de colunas proposto pela fragmentação 2D para o exemplo da Fig. 5. R = 2 e C = 2 formando 4 blocos de linhas e 2 blocos de colunas.

A fragmentação 2D distribui as arestas dentre os nós disponíveis mas cada vértice fica alocado unicamente em um único nó computacional. Por definição, o nó (i, j) fica responsável por armazenar os vértices do bloco de linha: $(j-1) \times R + i$. Assim sendo, para o exemplo acima, temos:

$$A_{(i,j)} = (j-1)R + i$$

$$\begin{cases} A_{(1,1)} = 1 \rightarrow nó\ (1,1)\ fica\ responsável\ por\ armazenar\ os\ vértices\ 1,2,3 \\ A_{(1,2)} = 3 \rightarrow nó\ (1,2)\ fica\ responsável\ por\ armazenar\ os\ vértices\ 7,8,9 \\ A_{(2,1)} = 2 \rightarrow nó\ (2,1)\ fica\ responsável\ por\ armazenar\ os\ vértices\ 4,5,6 \\ A_{(2,2)} = 4 \rightarrow nó\ (2,2)\ fica\ responsável\ por\ armazenar\ os\ vértices\ 10,11,12 \end{cases}$$

Na fragmentação, 1D cada vértice possui acesso direto à lista completa de adjacências, ou seja, nenhuma comunicação é necessária para descobrir quais são os vizinhos de um vértice uma vez que todas as arestas são armazenadas junto com o vértice neste tipo de fragmentação. A operação de expandir um nó exige, porém, uma comunicação com todos os nós restantes no pior caso. Na fragmentação 2D, por outro lado, a comunicação com nós vizinhos se faz necessária para descobrir todas as adjacências de um vértice uma vez que cada nó armazena somente um subconjunto das adjacências. Dessa forma, explorar as adjacências de um vértice na fragmentação 2D envolve duas operações: *expand* e *fold*. Na operação de *expand*, o dono do vértice a ser explorado envia mensagens para outros nós com o objetivo de recuperar a lista completa de suas adjacências. Na operação de *fold* as listas de adjacências parciais retornadas pelos vizinhos são fundidas para gerar a lista final completa de adjacências. A vantagem da fragmentação 2D sobre a fragmentação 1D é que a operação de expandir um vértice exige menor troca de mensagens entre os nós uma vez que cada nó comunica-se com, no máximo, $R + C - 2$ nós (MUNTÉS-MULERO *et al.*, 2010) ao invés de envolver todos os vizinhos como no caso da fragmentação 1D.

No contexto de processamento de grafos em grande escala, a fragmentação é uma técnica comumente utilizada quando o armazenamento ou processamento do grafo original ultrapassa os limites de um único nó computacional. No capítulo seguinte descrevemos o Graphene, que é o protótipo proposto por esta dissertação capaz de

armazenar a estrutura de um grafo distribuindo-o entre múltiplos nós computacionais. A estratégia de fragmentação pode ser definida pelo usuário.

Capítulo 3 – Graphene: Um Protótipo de SGBD Distribuído Orientado a Grafos

Esta dissertação propõe a utilização de um *middleware* distribuído que foi implementado para distribuição e gerenciamento de bases de dados. O *middleware* proposto e desenvolvido permite a fusão de múltiplas bases de dados autônomas, possivelmente heterogêneas e geograficamente descentralizadas em um único banco de dados federado. O usuário final interage com o *middleware* por meio de operações definidas em uma API, sem se preocupar com a forma através da qual os dados estão armazenados e distribuídos. Toda interação entre o usuário e o *middleware* é realizada como se ele estivesse trabalhando com um banco de dados único, homogêneo e centralizado. Este *middleware* recebeu o nome de Graphene.

O Graphene é compatível com o Blueprints, um padrão de fato utilizado atualmente pela maioria dos SGBDG. O Blueprints é uma coleção de interfaces definidas a partir das principais operações realizáveis em banco de dados de grafos de propriedades. Funciona como uma API genérica para grafos de propriedades e tem como objetivo facilitar a portabilidade entre diferentes SGBDG. O Blueprints faz parte de uma pilha de soluções que compõem o ecossistema Tinkerpop[7] e tem sido amplamente utilizado pelas soluções de banco de dados de grafos atuais de forma a conseguir aproximar-se da definição de um padrão, bem como conseguir tirar proveito de outras soluções presentes no ecossistema.

3.1. Visão Geral

O Graphene é um *middleware* distribuído que permite o armazenamento e a manipulação de bases de dados de grafos fragmentadas e distribuídas entre múltiplos nós

[7] Tinkerpop - https://github.com/tinkerpop/blueprints/wiki

de uma rede de computadores. A estratégia de fragmentação dos dados pode ser controlada pelo usuário.

A distribuição dos dados de acordo com uma lógica do domínio da aplicação pode trazer benefícios com relação à questão de gerência dos dados envolvidos, uma vez que os dados podem ser distribuídos entre as bases de acordo com questões de segurança, desempenho ou qualquer outra lógica definida pelo usuário através da função de fragmentação. Esta característica pode contribuir facilitando a gerência dos dados envolvidos de acordo com as necessidades de cada cenário.

Outra funcionalidade do *middleware* desenvolvido é permitir o agrupamento de bases de dados já consolidadas, preexistentes e centralizadas. Bases de dados centralizadas que já são utilizadas em outros ambientes podem ser agrupadas, formando uma base centralizada do ponto de vista do usuário enquanto armazenada de forma distribuída. A partir do momento que estas bases distintas são agrupadas ao *middleware*, novos tipos de relacionamento podem ser identificados e criados entre elementos de diferentes bases. Mesmo que o objetivo não seja a identificação e criação de elementos de diferentes bases, o agrupamento transparente de bases distintas ainda pode ser vantajoso para fornecer ao usuário final um repositório aparentemente centralizado, responsável pelas questões referentes à distribuição dos dados. Isso diminui a complexidade das consultas e a responsabilidade por acessar múltiplas bases distintas por parte do cliente. Maiores detalhes sobre a utilização do Graphene para o agrupamento de bases de dados são abordados na seção 3.3 deste capítulo.

A maioria das soluções de banco de dados de grafos ainda segue uma abordagem centralizada. A distribuição dos dados visando melhoria no desempenho não costuma ser uma estratégia adotada por estas soluções devido a uma série de dificuldades e desafios encontrados no processamento paralelo e distribuído de grafos em grande escala. Dentre as principais dificuldades podemos citar a baixa localidade e grande dependência entre os dados, limitando a quantidade de trabalho independente que pode ser executada por um determinado nó computacional. Além disso a maioria dos grafos apresentam uma estrutura bastante irregular e este fato torna difícil a identificação de uma estratégia de fragmentação que consiga priorizar execuções paralelas. Discussões sobre as principais dificuldades encontradas no processamento paralelo em grafos são abordados por

(LUMSDAINE *et al.*, 2007). A implementação do Graphene tira proveito da distribuição dos dados para execução de operações em paralelo quando possível, apesar de não estar focada em garantir execuções paralelas de alto desempenho.

O *middleware* proposto representa uma inciativa em proporcionar uma forma distribuída para o armazenamento de base de dados de grafos bem como permitir a execução de travessias e consultas em paralelo nestas bases. Em comparação com os demais trabalhos relacionados, apenas o InfiniteGraph (INFINITEGRAPH, 2014) e Titan (TITAN, 2014) são capazes de realizar distribuição e travessias em paralelo em um banco de dados de grafos. As demais soluções são completamente centralizadas ou utilizam a distribuição dos dados para fins de replicação e garantia de alta disponibilidade. Além disso, a distribuição e execução de travessias em paralelo implementadas no Graphene foi desenvolvida para ser utilizada em *hardware* não especializado, de baixo custo, em um ambiente de memória distribuída, não necessitando de arquiteturas paralelas específicas de alto custo. A Fig. 7 representa um exemplo de duas bases de dados de grafos participando do Graphene. Arestas entre elementos de diferentes bases são representados por arestas tracejadas.

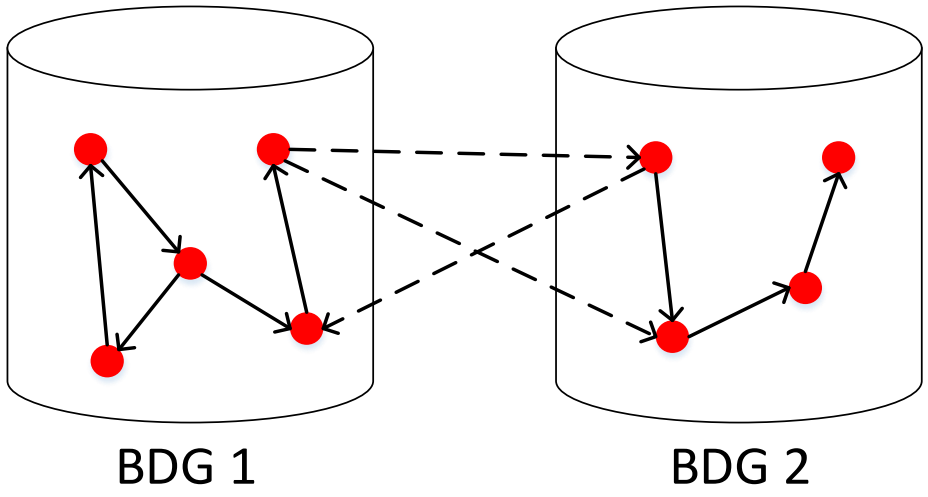


Fig. 7 - Exemplo conceitual de bancos de dados de grafos (BDG) participando do *middleware* distribuído. Arestas tracejadas representam conexões entre vértices presentes em banco de dados de grafos distintos. Identificadores, etiquetas nas arestas e propriedades dos elementos foram omitidos por motivo de simplificação.

Ao contrário de soluções de banco de dados distribuídas como o Titan (TITAN, 2014) e o OrientDB (ORIENTDB, 2014), o Graphene utiliza preferencialmente soluções de banco de dados de grafos como forma de armazenamento dos dados. Não é uma restrição, porém, que as bases de dados utilizadas sejam implementações nativas de banco de dados de grafos. Por ser uma solução compatível com a especificação Blueprints, uma ampla gama de soluções de armazenamento é suportada, dentre as quais podemos citar tanto soluções de SGBDG nativas quanto soluções especializadas em outras formas de armazenamento.

3.1.1. Estratégia de Distribuição

Para realizar a distribuição dos dados, atualmente o Graphene suporta a fragmentação 1D. Cada vértice é alocado em um único nó, juntamente com todas as suas arestas de saída. Para recriar o grafo original a partir dos dados distribuídos, o Graphene utiliza um conjunto de propriedades que são associadas aos vértices e arestas para representar os elementos e adjacências virtuais. Arestas virtuais são relacionamentos entre vértices alocados em nós distintos.

Existem três propriedades principais criadas e gerenciadas internamente pelo Graphene para representação de identificadores e arestas virtuais. Estas propriedades possuem chaves denominadas: __id, _partition e _destVertexId. Por serem utilizados para gerência interna dos dados, o *middleware* não permite que um usuário utilize esses nomes reservados em propriedades próprias.

A propriedade de chave identificadora __id é utilizada para armazenar o identificador de um elemento de uma forma única. É necessário garantir que cada elemento do grafo possua um identificador único. Cada solução de banco de dados de grafos possui uma estratégia própria quanto à atribuição de identificadores aos seus elementos. Algumas soluções como o Neo4j (NEO4J, 2014) geram identificadores

sequenciais de forma automática enquanto outras como o TinkerGraph[8] (banco de dados de grafo leve, em memória), por exemplo, permitem que o identificador do elemento possa ser escolhido pelo usuário. Algumas soluções podem ainda reutilizar o valor de identificadores atribuídos anteriormente a elementos do grafo que já foram removidos. Por esse motivo, é uma boa prática que os usuários de banco de dados de grafos criem identificadores próprios aos elementos do domínio através de propriedades associados aos vértices e arestas, gerenciando manualmente a atribuição destes valores ao invés de utilizar os valores gerados pelo próprio SGBDG.

Cada SGBDG participante na solução distribuída funciona de maneira independente e não conhece a existência dos outros SGBDG, pois o Graphene é a camada superior responsável por conhecer e orquestrar as operações entre os SGBDG existentes. Por este motivo, elementos diferentes com identificadores idênticos podem surgir entre os BDG participantes. Durante a criação de um novo elemento o usuário informa o identificador que aquele elemento irá receber. Este identificador é gravado na propriedade __id associada ao elemento. O SGBDG responsável pela criação do novo elemento pode escolher qualquer identificador a seu critério para este elemento mas este valor não será utilizado pelo Graphene, uma vez que qualquer referência futura que o usuário venha fazer a este elemento será feita utilizando o identificador definido pelo usuário durante a criação do elemento, que foi o valor armazenado na propriedade __id.

As propriedades _partition e _destVertexId são utilizadas pelo *middleware* para representar arestas virtuais. As arestas virtuais são arestas do Graphene que conectam dois vértices que estão em fragmentos distintos. Quando uma aresta é criada entre dois vértices que estão no mesmo fragmento, isto é, pertencem ao mesmo BDG, as arestas são criadas diretamente entre os elementos da mesma forma que seriam criadas no banco de dados original. Quando os vértices de origem e destino estão em BDG diferentes, uma aresta virtual é criada pelo Graphene para representar a ligação entre os elementos. Para representar uma aresta virtual, optou-se por criar uma aresta no vértice de origem que aponta para o próprio vértice e que possui uma propriedade com chave "_partition" e

[8] Tinkergraph - https://github.com/tinkerpop/blueprints/wiki/TinkerGraph

valor “virtual”. As demais arestas são marcadas com a propriedade “_partition” com valor “original”. No caso das arestas virtuais, uma outra propriedade com chave _destVertexId é adicionada ao elemento e seu valor é o identificador do vértice de destino (__id).

O Graphene suporta três políticas de fragmentação predefinidas: fragmentação baseada em Espalhamento, circular (*round-robin*) ou baseada em intervalo (*range*). Outras políticas de fragmentação personalizadas podem ser definidas pelo usuário. Cada nó computacional participante possui um identificador próprio, único conhecido pelo *middleware*. A política de fragmentação é uma função que recebe como entrada o identificador único de um vértice e retorna, como saída, o identificador do nó responsável por armazenar aquele vértice. O usuário pode utilizar uma das funções de fragmentação predefinidas ou implementar sua própria lógica de fragmentação dos dados.

A fragmentação baseada em Espalhamento utiliza o identificador do vértice como entrada em uma função de Espalhamento. A saída desta função é um número inteiro. O valor absoluto do número inteiro é recuperado e dividido pelo número de nós existentes na solução distribuída. O resto da divisão é utilizado para indicar qual nó é responsável pelo vértice cujo identificador foi utilizado como entrada da função de Espalhamento. O nó de destino para um vértice que possui um identificador igual a $idVerticeEntrada$ é calculado pela Equação 1.

$$f(idVerticeEntrada) = (|hash(idVerticeEntrada)| \ mod \ NUM_{NÓS}) + 1$$

Equação 1 – Função de distribuição de Espalhamento predefinida disponível no Graphene.

A ideia da fragmentação baseada em intervalo (*range*) é garantir que vértices com identificadores consecutivos sejam alocados no mesmo nó. É útil particularmente quando a estrutura do grafo é conhecida previamente. É importante que o número total de vértices que serão armazenados seja conhecido previamente. A fragmentação baseada em intervalo recebe o identificador de um vértice como entrada e utiliza dois parâmetros para calcular o destino daquele vértice: o número total de vértices que serão armazenados no grafo $NUM_{TOTAL\_VERTICES}$ e o número de nós participantes na solução $NUM_{NÓS}$. O nó de destino é dado pela Equação 2.

$$f(idVerticeEntrada) = \left\lfloor \frac{idVerticeEntrada * NUM_{NÓs}}{NUM_{TOTAL\_VERTICES}} \right\rfloor + 1$$

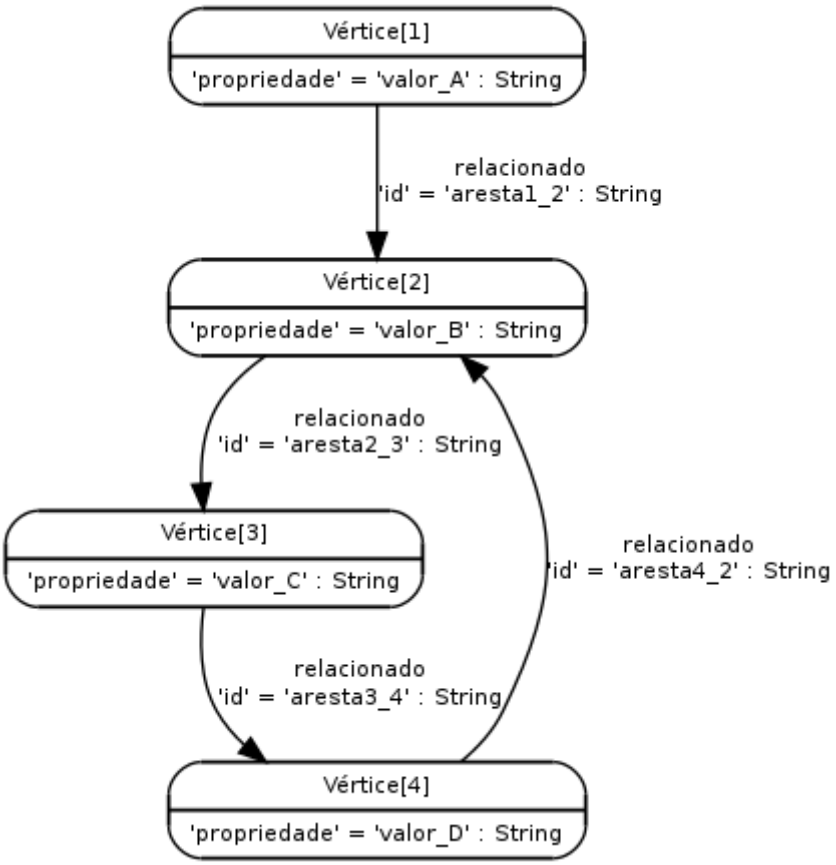
Equação 2 – Função de distribuição baseada em intervalo predefinida disponível no Graphene.

A terceira função de distribuição presente no Graphene é uma função circular. Esta função de distribuição garante que os vértices com identificadores consecutivos sejam alocados em diferentes bases de dados. Cada vértice consecutivo é distribuído em um nó diferente seguindo uma estratégia circular, em sequência. O nó de destino é dado pela Equação 3.

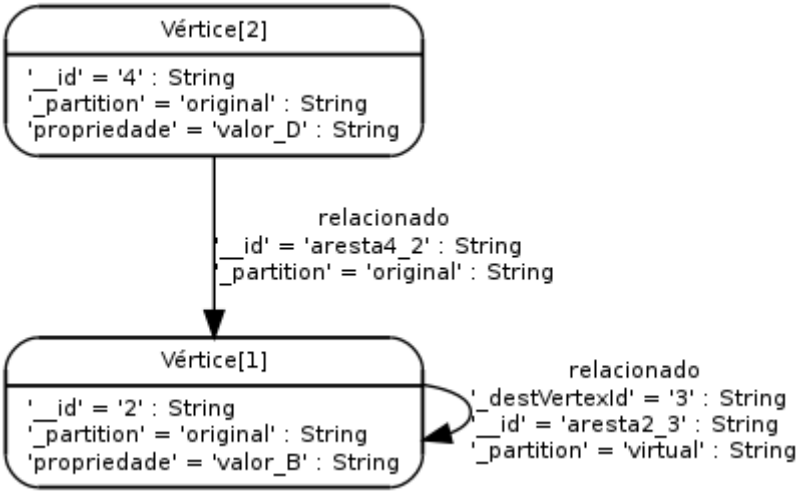
$$f(idVerticeEntrada) = (idVerticeEntrada \; mod \; NUM_{NÓS}) + 1$$

Equação 3 – Função de distribuição circular predefinida disponível no Graphene.

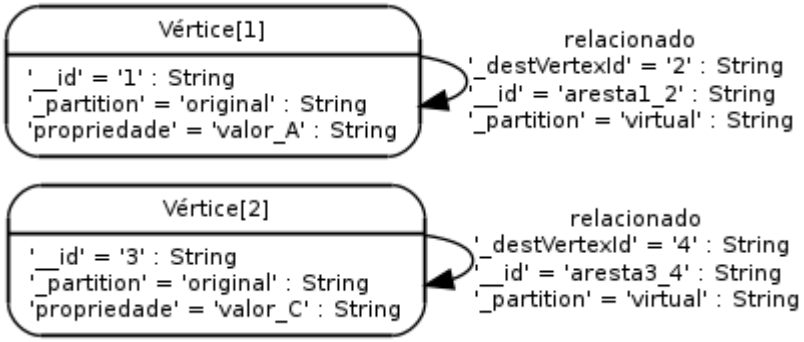
Para facilitar o entendimento sobre como os dados são armazenados de forma distribuída no *middleware*, um pequeno grafo é utilizado como exemplo na Fig. 8. O grafo original possui quatro vértices e quatro arestas. Algumas propriedades foram associadas aos vértices que se conectam aos outros vértices através de arestas com etiqueta do tipo "relacionado". Os vértices são identificados por valores de 1 a 4. Este grafo foi carregado no Graphene e distribuído em duas bases de dados distintas, usando uma política de fragmentação de Espalhamento para distribuir os vértices entre dois nós. O SGBDG 1 ficou responsável por armazenar os vértices de identificador 2 e 4 enquanto o SGBDG 2 armazenou os vértices 1 e 3. A aresta que liga os vértices 4 e 2 é representada diretamente no BDG 1 já que ambos os vértices são armazenados pelo mesmo nó. As demais arestas ligam vértices armazenados por nós distintos e, por esse motivo, estas adjacências são representadas por arestas virtuais que possuem como propriedade o identificador do vértice de destino.



(a) – Grafo Original



(b) – BDG 1



(c) – BDG 2

Fig. 8 - Grafo original (a) distribuído no Graphene entre dois BDG de acordo com a fragmentação baseada em Espalhamento. Em (b) é possível verificar a porção do grafo armazenada no BDG 1 enquanto (c) representa a porção armazenada no BDG 2.

3.1.2. Descrição da Arquitetura

A arquitetura do Graphene está dividida em três tipos de módulos principais: cliente, coordenador central e coordenadores locais. A Fig. 9 representa uma visão geral da arquitetura.

O módulo cliente é disponibilizado para ser utilizado pela aplicação do usuário final. Este módulo expõe as operações comuns em banco de dados de grafos como: adição e recuperação de vértices e arestas, associação de propriedades aos elementos do grafo, indexação de elementos e operações de consulta e travessia. A junção entre diferentes banco de dados ou a distribuição de um banco de dados centralizado em diferentes bases distribuídas é alcançada definindo uma função de fragmentação que será utilizada pelo Graphene. O módulo cliente informa ao servidor qual é a função de fragmentação que este deve utilizar para localizar os elementos nas bases distribuídas.

O coordenador central é um módulo leve que tem como objetivo avaliar e repassar uma operação solicitada pelo cliente a um módulo coordenador local, que será responsável pela sua execução. Ao final da operação, o coordenador central repassa a resposta recebida ao módulo cliente, que por sua vez repassará ao método invocado pela aplicação do usuário. Algumas operações como a inclusão de vértices por exemplo possuem um nó específico de destino, que será responsável pelo armazenamento deste elemento do grafo. Outras operações como, por exemplo, a recuperação de todo o

conjunto de vértices do grafo envolverá a comunicação entre todos os nós participantes. Em ambos os casos, o coordenador central elege um coordenador local para ser responsável pela operação e a repassa a este coordenador. O coordenador central distribui as consultas recebidas entre os coordenadores locais seguindo uma política de rodízio circular (*round-robin*). A ideia neste caso é tentar obter um melhor desempenho e maior grau de paralelismo, distribuindo as consultas entre os coordenadores locais existentes e evitando sobrecarregar um coordenador específico quando possível.

O coordenador local é um módulo responsável por um determinado fragmento do grafo. Este módulo só pode acessar as informações do próprio fragmento que coordena mas pode comunicar-se com outros coordenadores locais via troca de mensagens para solicitar algum dado necessário para concluir uma operação que esteja realizando. O coordenador local contém toda a lógica de negócio com relação à forma como o banco de dados encontra-se distribuído e quais são as etapas necessárias para responder às operações recebidas a partir da aplicação do usuário. Uma operação invocada pelo usuário poderia ser direcionada diretamente a um coordenador local qualquer sem passar pelo coordenador central, mesmo que o coordenador local em questão não fosse responsável pelos BDG participantes daquela operação. Neste caso, o coordenador local verifica que não é responsável pela consulta recebida e a repassa ao coordenador vizinho correspondente. A decisão de manter uma camada com o coordenador central na arquitetura do *middleware* pode ser justificada por um melhor balanceamento na distribuição das consultas entre os nós participantes.

Se a operação envolver a participação de múltiplos SGBDG (como no caso de recuperar todo o conjunto de vértices, por exemplo) então o coordenador local eleito é responsável por invocar estas operações em paralelo em todos os SGBDG participantes sempre que possível. Por exemplo, uma operação para recuperar o conjunto de todos os vértices é repassada do coordenador central a qualquer coordenador local, que se tornará responsável por aquela operação. O coordenador local responsável se comunica com os outros coordenadores, iniciando uma linha de execução paralela em cada nó. Cada linha de execução paralela executada por cada nó recupera o conjunto de vértices pelo qual ele é responsável. Os conjuntos recuperados são retornados ao coordenador local responsável

por aquela operação, que combina os resultados e os retorna ao coordenador central que, por sua vez, os retorna à aplicação cliente.

O último componente que faz parte da arquitetura da solução adotada é um catálogo centralizado, gerenciado por qualquer nó e que guarda informações sobre a localização de cada nó do cluster. Todos os componentes do Graphene registram-se e localizam os outros módulos por meio deste catálogo durante a etapa de inicialização.

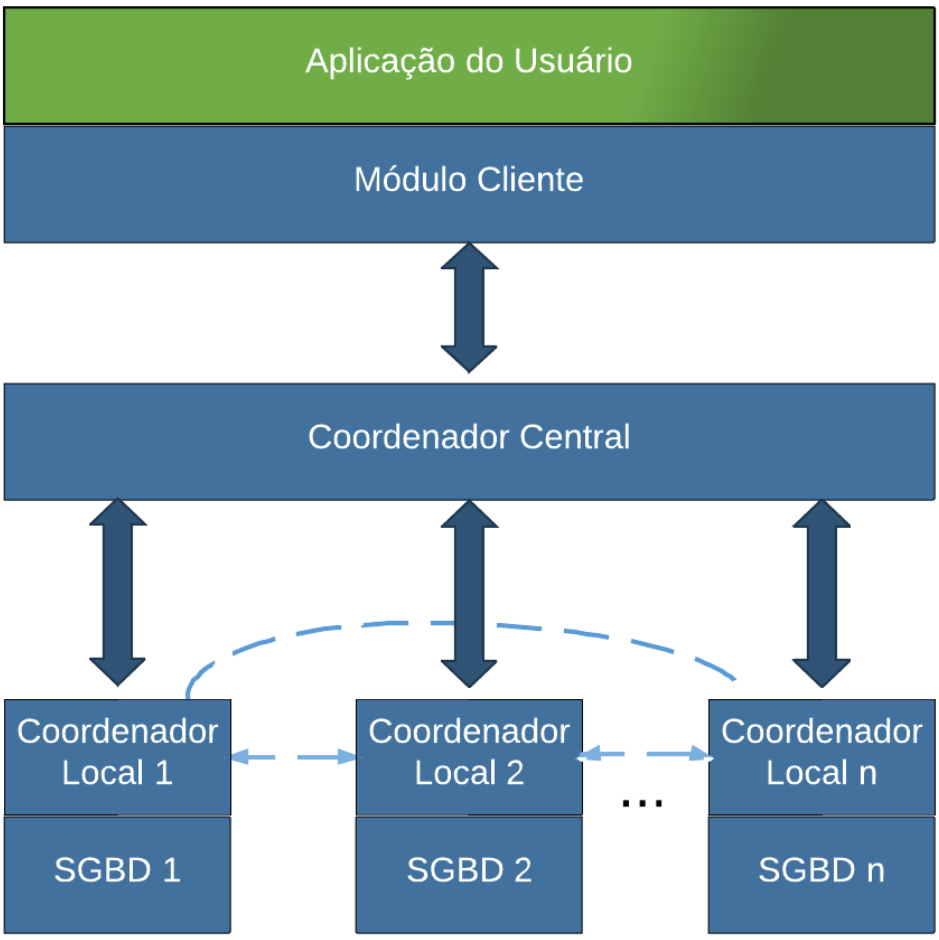


Fig. 9 - Arquitetura do *middleware* distribuído. As setas indicam a comunicação bidirecional entre os módulos cliente, coordenador central e coordenadores locais participantes na solução.

Para lidar com as questões de escalabilidade, as principais soluções de banco de dados de grafos como o Sparksee e o Neo4j utilizam políticas de *cache* de elementos. O

cache de elementos é uma funcionalidade importante para garantir um bom desempenho de um SGBDG. O Graphene também realiza *cache* dos elementos acessados.

O Graphene utiliza cache de elementos em dois níveis: no módulo coordenador central e no módulo coordenador local. Quando uma operação é recebida pelo *middleware*, antes de ser repassada a um coordenador local, o coordenador central verifica se ela pode ser respondida a partir de alguma informação presente no seu cache. Em caso positivo, a operação é respondida imediatamente antes de ser enviada a um coordenador local. Em caso negativo, a requisição é enviada para um coordenador local. Quando a requisição é enviada para um coordenador local, primeiro é verificado em seu cache local se a informação está presente. Em caso negativo o coordenador local envia uma mensagem aos outros coordenadores locais perguntando se alguém possui aquela informação em seu cache. Em caso positivo, a informação é retornada ao coordenador local atual que insere a informação recuperada em seu próprio cache. Em caso negativo, o coordenador local executa a consulta normalmente e insere o resultado em seu cache.

Neste momento o *middleware* utiliza o cache para armazenar as seguintes informações: os objetos que representam os vértices ou as arestas dado seu identificador, vértices ou arestas que possuam uma propriedade com uma determinada chave e valor, conjunto de arestas vizinhas a um determinado vértice que possuam um tipo e direção escolhidos, vértices vizinhos a um determinado vértice navegando por arestas com uma determinada etiqueta e direção e o conjunto de arestas virtuais dado o identificador de um determinado vértice. O *cache* possui um tamanho máximo predefinido e quando este limite está próximo de ser atingido o Graphene começa a eliminar os elementos que estiveram mais tempo no cache sem terem sido referenciados.

3.1.3. Operações sobre Grafos

As principais operações sobre grafos suportadas pelo Graphene são: adicionar vértices e arestas, recuperar uma aresta ou vértice dado um identificador, recuperar a lista de todos os vértices ou arestas do grafo, recuperar um vértice ou uma aresta por meio de uma propriedade com determinada chave e valor, recuperar os vértices de origem ou

destino de uma aresta, recuperar vértices vizinhos a um vértice em uma determinada direção e com uma determinada etiqueta associada, recuperar arestas incidentes a um vértice dado uma determinada direção e etiqueta associada, remover um vértice ou aresta do grafo, criar e remover índices em propriedades de vértices e arestas, iniciar e terminar transações, e executar operações de caminhos mínimos e travessias.

Para tirar proveito da distribuição dos dados, algumas operações realizadas sobre a base distribuída podem ser executadas em paralelo pelos nós participantes. Dentre as operações executadas em paralelo, podemos citar: retornar todos os vértices ou arestas do grafo, retornar uma aresta específica dado o seu identificador, remover uma aresta dado o seu identificador e retornar um conjunto de vértices ou arestas de acordo com propriedades que possuam uma determinada chave e valor.

O Graphene suporta três formas principais de operações de consultas em bases de dados de grafos distribuídas: através da interface *Query* do Blueprints, da linguagem de travessia Gremlin[9] ou através da execução de uma travessia em paralelo que foi desenvolvida especialmente para permitir operações de busca mais complexas no Graphene.

A primeira forma de consulta é através da interface *Query*. Esta interface oferece filtros e modificadores que podem ser utilizados para selecionar arestas a partir de um vértice. Dentre as operações suportadas, podemos citar: filtrar arestas por direção, por etiqueta e por propriedade. O filtro por propriedade permite que o usuário utilize comparações para retornar, por exemplo, arestas que possuam uma propriedade maior do que um valor predefinido. Também é possível verificar se o valor de uma propriedade está dentro de um intervalo definido pelo usuário. Operações para limitar o número de arestas retornadas também estão presentes e são particularmente úteis ao percorrer grafos com vértices de alto grau de conectividade. O resultado de uma consulta realizada por meio da interface *Query* é um conjunto de arestas, o conjunto de vértices resultantes que

[9] Gremlin - https://github.com/tinkerpop/gremlin/wiki

são conectados por essas arestas ou então somente o número de elementos que atendem aos filtros especificados pelo usuário.

A segunda forma de consulta é através da linguagem de travessia Gremlin. O Gremlin é uma linguagem específica de domínio para travessia de grafos de propriedades. Esta linguagem tem aplicação nas áreas de consulta, análise de manipulação de dados em formato de grafos. O Gremlin é uma linguagem de código aberto que faz parte do ecossistema Tinkerpop. Esta linguagem pode ser utilizada por soluções de banco de dados de grafos ou outras ferramentas que implementam o modelo de dados especificado pelo Blueprints. Atualmente possui implementações para as linguagens Java e Groovy.

Existem vários benefícios na utilização da linguagem Gremlin para travessia de grafos. É possível especificar consultas complexas de forma sucinta utilizando esta linguagem. Além disso, sua utilização permite que o usuário não esteja preso a nenhuma implementação de banco de dados de grafo específica.

O Gremlin é uma linguagem funcional na qual os operadores de travessia são encadeados para formar expressões de caminho. Consultas complexas nesta linguagem são formadas pelo encadeamento de operações mais simples. A linguagem executa as operações sequencialmente, da esquerda para a direita onde o resultado de cada operação é utilizado como entrada da operação seguinte. O resultado da última operação da cadeia é retornado como o resultado da consulta. As consultas expressas em Gremlin são executadas no Graphene de forma sequencial.

A terceira forma de realizar consultas na base de dados gerenciada pelo *middleware* é através de operações de travessias paralelas. Este método de travessia foi desenvolvido para tirar proveito da distribuição dos dados, permitindo a execução de operações em paralelo quando possível.

A travessia em paralelo desenvolvida no Graphene utiliza a busca em largura como ordem de visitação dos vértices. A busca em largura garante que todos os nós de um determinado nível são visitados antes de passar para o próximo nível. Em um ambiente distribuído, a busca em largura é uma operação paralelizável, uma vez que visitar o conjunto de vértices em uma determinada profundidade é uma operação que pode

ser executada ao mesmo tempo por múltiplos nós, onde cada nó fica responsável por visitar os vértices armazenados por ele.

A busca em largura distribuída implementada no Graphene segue uma abordagem síncrona, baseada no BSP (*Bulk Synchronous Parallel*). Conforme descrito em maiores detalhes na seção 2.3.2, o BSP é baseado em três etapas principais: processamento local, troca de mensagens entre os nós e barreira de sincronização.

Existem algumas críticas quanto à utilização, de um modo geral, do modelo BSP em programação paralela. Um dos argumentos é que a presença de barreiras de sincronização atrasa a computação. Um segundo argumento utilizado é que o tempo de computação gasto em cada *superstep* é definido pelo tempo de execução do nó mais lento, uma vez que os outros nós precisam aguardar o término desta computação antes de passar para a etapa seguinte. No contexto de algoritmos de grafos, porém, o BSP é amplamente utilizado. Algoritmos síncronos baseados no BSP são utilizados por ferramentas de processamento de grafos em grande escala como o Pregel (MALEWICZ *et al.*, 2010) e outras ferramentas semelhantes. Quando o diâmetro do grafo é pequeno, poucos pontos de sincronização são necessários ao executar uma busca em largura distribuída a partir de um vértice de origem usando o modelo BSP. O problema da falta de balanceamento computacional entre os nós, por sua vez, pode ser resolvido por meio de uma estratégia de fragmentação mais uniforme que consiga equilibrar o conjunto de vértices entre os nós participantes evitando sobrecarregar um nó específico durante a travessia.

A implementação da busca em largura paralela executada pelo Graphene utilizou como base os algoritmos descritos em (YOO *et al.*, 2005) e (MUNTÉS-MULERO *et al.*, 2010). O primeiro trabalho propõe uma busca em largura paralela e distribuída no BlueGene/L, que é um sistema com arquitetura altamente paralela desenvolvido pela IBM. Já o segundo trabalho propõe estratégias de fragmentação de grafos para execução eficiente de busca em largura em um ambiente de memória distribuída. A busca em largura é uma operação importante em grafos porque muitas outras operações como, por exemplo, encontrar o menor caminho entre dois vértices, detectar comunidades (NEWMAN & GIRVAN, 2004), dentre outras dependem do processamento eficiente da busca em largura. Ambos os trabalhos utilizam o mesmo algoritmo de busca em largura para a fragmentação 1D.

O algoritmo de busca em largura utilizado pelo Graphene precisou ser adaptado por conta de algumas diferenças entre a busca em largura executada pelo Graphene e a busca executada pelos trabalhos citados acima. A busca em largura do Graphene retorna um conjunto de caminhos que atendem a um determinado critério, percorrendo, para isso, arestas específicas e armazenando todos os caminhos encontrados. Todos os caminhos percorridos também são avaliados por uma função definida pelo usuário, que decide qual o comportamento a ser assumido pela busca daquele ponto em diante. Esta função de avaliação pode incluir ou excluir um caminho do resultado da busca e também realizar podas, caso o usuário não deseje continuar a busca a partir de um determinado ponto. Ao contrário da funcionalidade fornecida pela busca em largura paralela presente no Graphene, o algoritmo utilizado pelos trabalhos acima citados sempre percorre todos os nós a partir da origem e apresenta como resposta somente a ordem em que os vértices são visitados.

Apesar de amplamente utilizados, os algoritmos de busca em largura síncronos baseados em BSP não são a única forma conhecida de realizar a busca em largura em grafos de forma paralela. Outra abordagem que envolve a execução de múltiplas travessias de forma concorrente é abordado por (ULLMAN & YANNAKAKIS, 1990). Esta abordagem envolve identificar e executar a travessia a partir de vértices de alto grau de conectividade e é particularmente útil em grafos cujo diâmetro é grande enquanto a estratégia BSP é mais utilizada em grafos que apresentam um diâmetro pequeno.

Ao contrário da busca em largura, a busca em profundidade não é uma operação de travessia comumente paralelizada. Isso ocorre devido à dificuldade em paralelizar esta operação, uma vez que na busca em profundidade é necessário expandir um ramo do grafo completamente antes de percorrer o próximo. A grande dependência entre os dados oferece um obstáculo ao surgimento de algoritmos paralelos para esta finalidade.

A operação de busca em profundidade foi provada como uma operação inerentemente sequencial em (REIF, 1985), de forma que não seria possível a criação de um algoritmo paralelo para esta finalidade. Posteriormente, em (FREEMAN, 1991) foi apresentada uma discussão sobre a existência de algoritmos paralelos para determinadas classes de problemas. Este trabalho mostra que a busca em profundidade para grafos planares e não direcionados pertence à classe de problemas NC e, de um modo mais geral,

a busca em profundidade em grafos não direcionados pertence à classe de problemas RNC. Um problema é dito NC se existirem constantes c e k tal que o problema pode ser resolvido em tempo $O(log^c\ n)$ usando $O(n^k)$ processadores paralelos. Até o momento, não temos conhecimento de nenhum trabalho que utilize na prática algoritmos paralelos para execução de busca em profundidade em grafos. Por esse motivo, a travessia em profundidade em paralelo não é uma operação suportada pelo Graphene. Se o usuário deseja executar uma busca em profundidade sequencial, esta pode ser realizada programaticamente através das operações fornecidas pelo próprio *middleware*, através da linguagem Gremlin ou ainda através da utilização do pacote de algoritmos fornecidos pelo Furnace[10].

Para realização da travessia em paralelo foi necessário criar uma nova operação, denominada *Traversal*. Esta operação não está prevista na API do Blueprints e, portanto, é uma funcionalidade específica do *middleware* desenvolvido. A funcionalidade de travessia desenvolvida no Graphene foi inspirada na travessia executada pelos principais SGBDG, tais como Sparksee, Neo4j e InfiniteGraph.

Além da busca em largura, outra operação muito comum em banco de dados de grafos é a de encontrar o menor caminho entre dois vértices. Muitas vezes, o algoritmo de busca em largura é utilizado como base para o cálculo de menor caminho entre dois vértices quando não existem pesos associados às arestas. No Graphene, um algoritmo distribuído de menor caminho foi implementado. Este algoritmo tem como base a operação de busca em largura descrita acima. Maiores detalhes sobre os algoritmos de travessia e caminho mínimo implementados estão descritos na seção 3.2 deste capítulo.

3.1.4. Utilitários

O Graphene também oferece uma série de funcionalidades utilitárias para auxiliar os usuários nas operações comuns em banco de dados de grafos. Uma solução para ajudar

[10] Furnace - https://github.com/tinkerpop/furnace

na importação de dados para banco distribuído foi implementada. Uma segunda funcionalidade implementada foi a exportação da estrutura do grafo armazenado para o formato DOT[11].

A funcionalidade de importação de dados utiliza operações primitivas do Blueprints para criação dos vértices, arestas, índices e propriedades necessárias para o funcionamento do banco de dados distribuído. Este utilitário recebe como entrada uma função de fragmentação e o caminho de um arquivo de texto que contém a descrição da estrutura de um grafo. Ele é responsável pela criação dos índices, indexação de elementos, representação de arestas reais e virtuais e todos os passos necessários para que a base importada possa ser utilizada como um fragmento participante no Graphene.

O DOT é uma linguagem de texto simples que permite descrever um grafo. É um padrão bem conhecido e utilizado por diversas ferramentas de grafos, tanto para operações associadas à visualização quanto para execução de cálculos ou consultas na estrutura do grafo. O *middleware* é capaz de exportar o grafo armazenado para o formato DOT através de uma classe utilitária fornecida junto com o Graphene. O formato DOT pode ser utilizado por ferramentas como o Graphviz (GANSNER & NORTH, 2000), por exemplo, para visualização de grafos.

3.2. Detalhes de Implementação

Esta seção descreve em maiores detalhes as decisões de implementação adotadas na construção do Graphene.

O Graphene foi desenvolvido na linguagem de programação Java, versão 1.6, utilizando a versão 2.1.0 do Blueprints (BLUEPRINTS, 2014). A comunicação é realizada através API Java RMI (*Remote Method Invocation*), que implementa chamadas de procedimentos remotos (RPC). Objetos passados como parâmetros e retornados como resposta passam por processos de serialização e desserialização automaticamente a cada

[11] DOT - http://www.graphviz.org/Documentation.php

chamada. A compatibilidade com a especificação do Blueprints trouxe algumas vantagens com relação à compatibilidade com outras ferramentas do ecossistema como:

- Compatibilidade com as principais ferramentas e tecnologias de banco de dados: compatível com a grande maioria dos SGBDG e também com SGBD de diferentes tipos como MongoDB[12], Oracle NoSQL[13], SQL/JDBC, JPA (*Java Persistence API*).
- Gremlin[14]: utilização de uma linguagem específica de domínio para consulta e travessia de banco de dados de grafos de propriedade.
- Pipes[15]: arcabouço de fluxo de dados que permite a definição de uma sequência de passos computacionais encadeados que transformam uma entrada de dados em uma saída previamente estabelecida.
- Frames[16]: ferramenta de mapeamento entre objetos e entidades no banco de dados de grafos. Permite que o usuário possa manipular objetos de alto nível sem se preocupar em como as entidades serão representadas no banco de dados.
- Furnace[17]: pacote de algoritmos de grafos. Oferece suporte aos principais algoritmos de grafos como busca em largura, profundidade, menor caminho, dentre outros. Oferece também a possibilidade de realizar computações centradas em vértice baseadas no modelo proposto pelo Pregel. O Furnace ainda é uma implementação em andamento e não possui uma versão oficial disponível para ser utilizada.

[12] MongoDB - http://www.mongodb.org/

[13] Oracle NoSQL - http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html

[14] Gremlin - https://github.com/tinkerpop/gremlin/wiki

[15] Pipes - https://github.com/tinkerpop/pipes

[16] Frames - https://github.com/tinkerpop/frames

[17] Furnace - https://github.com/tinkerpop/furnace

- Rexter[18]: permite expor o acesso a base de dados a partir de um servidor que utiliza uma interface RESTful. Operações de escrita e leitura da base de dados podem ser realizadas a partir do protocolo HTTP e seus métodos: GET, POST, PUT e DELETE.
- Compatibilidade com o padrão RDF: torna possível a conversão entre o formato de grafo de propriedades e o padrão RDF (*Resource Description Framework*).
- Compatibilidade com ferramentas de terceiros aderentes ao Blueprints: inclui uma gama de soluções como ferramentas para visualização de grafos, obtenção de métricas (medidas de centralidade, PageRank) e execução de algoritmos (*clustering*, cálculo de fluxo, dentre outras)

Os principais componentes do Graphene são o módulo cliente, o coordenador central e os coordenadores locais. O módulo cliente é representado pela classe GrapheneGraph. Esta classe expõe as operações de grafo para o mundo externo ao *middleware* e representa o ponto de entrada para a aplicação do usuário. O módulo coordenador central é representado pela classe CoordenadorCentralService. Um grafo armazenado pelo Graphene tem apenas um único coordenador central. O coordenador central comunica-se com os coordenadores locais, que são representados pela classe CoordenadorBDGService. Um elemento centralizador responsável por fornecer a localização de serviços é representado pela classe RMIBindService. Diagramas de classe do Graphene podem ser encontrados no Apêndice A.

Para o funcionamento do Graphene, os módulos participantes devem ser inicializados da seguinte forma: inicialmente um serviço de registro local RMI é iniciado em um nó do cluster. Logo em seguida, o serviço RMIBindService é iniciado na mesma máquina onde o registro local RMI foi iniciado. Quando o RMIBindService é iniciado, se registra no registro local RMI executado no mesmo nó. Em seguida, os coordenadores locais são iniciados em nós remotos do cluster. Para inicializar um coordenador local é necessário passar como parâmetro o identificador do coordenador, que deve ser único, e a localização (nome ou endereço do nó do *cluster*) onde é executado o registro RMI

[18] Rexter - https://github.com/tinkerpop/rexster

(catálogo). Cada coordenador local também registra-se no catálogo RMI centralizado durante a sua inicialização. Por último, é iniciado o módulo coordenador central em um nó remoto, passando como parâmetro também a localização onde é executado o catálogo RMI. Durante a inicialização do coordenador central, este módulo identifica os coordenadores locais participantes e configura-se internamente para permitir a comunicação entre os módulos. Desta forma todos os módulos participantes do Graphene são inicializados e estão preparados para atender a solicitações de aplicações clientes. As aplicações clientes precisam conhecer apenas a localização do catálogo RMI central para utilizar a solução.

As aplicações clientes são responsáveis por informar a política de fragmentação a ser utilizada pelo Graphene. Caso o usuário opte por escrever uma função de fragmentação própria, isso pode ser feito criando uma nova classe que seja uma subclasse de *FragmentationPolicy*. A subclasse precisará implementar um único método, que recebe como parâmetro o identificador do vértice e precisa retornar um inteiro com o identificador do nó onde o vértice será alocado e de onde poderá ser recuperado posteriormente. O *middleware* não suporta neste momento a realocação de elementos, ou seja, uma vez definido o destino de um vértice, ele deverá permanecer no destino escolhido e não poderá ser movido para outro nó. Dessa forma, é necessário que a função definida pelo usuário seja implementada de forma que garanta que as saídas serão idênticas em todas as vezes que a mesma entrada for utilizada.

O Graphene oferece suporte à execução de múltiplas operações de diferentes clientes ou threads simultaneamente. O módulo coordenador local foi implementado seguindo uma estratégia produtor/consumidor por meio de uma fila compartilhada. A utilização do *middleware* por aplicações clientes gera mensagens aos coordenadores centrais que são repassadas aos coordenadores responsáveis por cada SGBDG. Quando chega ao módulo coordenador local, uma mensagem com a operação a ser realizada é gerada pelo módulo produtor e direcionada a uma fila de espera que será consumida pelo módulo consumidor do coordenador local. O módulo consumidor do coordenador local é o único responsável por realizar efetivamente o acesso ao banco de dados local. Quando múltiplos clientes ou threads de um mesmo cliente realizam operações no Graphene ao mesmo tempo, cada acesso é isolado em uma transação independente.

É importante realçar, porém, que no momento o Graphene não possui suporte a transações distribuídas (como o protocolo de efetivação em duas fases, por exemplo) implementado confiando, desta forma, no processamento da transação executado por cada SGBD de forma independente.

Os dados utilizados pelo Graphene para a gerência de elementos distribuídos são os atributos: __id, _partition e _destVertexId associados aos vértices e arestas do grafo. Algumas operações de grafos exigem a recuperação eficiente de elementos por meio de consultas realizadas sobre estes atributos. Por este motivo optou-se por indexar estas propriedades, agilizando o acesso aos elementos sem a necessidade de percorrer o grafo. Cada SGBD participante mantém uma estrutura de índices própria, com os elementos pelos quais ele é responsável em armazenar.

Para implementação da política de *cache* de elementos presente nos módulos coordenador central e coordenador local, a biblioteca Google Guava[19] foi utilizada. Esta biblioteca oferece algumas melhorias às principais operações presentes na linguagem Java e também inclui uma série de funcionalidades adicionais. A implementação do Graphene utilizou apenas as soluções de cache desta biblioteca.

3.2.1. Busca em Largura Distribuída

O algoritmo utilizado na implementação da busca em largura distribuída recebe como parâmetros de entrada um vértice de origem (v_s), direção (D) e etiqueta das arestas que serão percorridas (L) e uma função de avaliação chamada de *Evaluator* (Ev) definida pelo usuário. O seu objetivo é percorrer os caminhos de um grafo a partir de uma origem retornando aqueles caminhos que atendem a um determinado critério definido pelo usuário por meio da função de avaliação.

[19] Google Guava - https://code.google.com/p/guava-libraries/

O algoritmo distribuído da busca em largura implementado no Graphene é descrito a seguir está representado na Fig. 10. Este trecho de código é executado em paralelo por cada nó coordenador local participante.

Cada nó possui três conjuntos principais: $V_{visited}$, $V_{current}$ e V_{next}. O primeiro conjunto é responsável por armazenar todos os vértices locais que já foram visitados em algum passo da computação. O segundo conjunto armazena os vértices que serão visitados pela busca no passo corrente. Já o terceiro conjunto armazena os vértices que serão visitados por aquele nó no próximo passo. Como cada vértice é armazenado por um único coordenador local então não há interseção entre conjuntos de diferentes coordenadores. Inicialmente estes conjuntos estão vazios exceto pelo conjunto $V_{current}$ do nó responsável pelo vértice de origem, que é iniciado com o elemento v_s.

Após iniciadas as estruturas que serão utilizadas no restante da computação, o algoritmo se repete enquanto houver vértices a serem explorados, isto é, enquanto $V_{current}$ for diferente de vazio para o nó coordenador local. O conjunto de todos os vértices visitados por cada coordenador representado por $V_{allvisited}$ é conhecido por cada nó e é calculado pela união do conjunto $V_{visited}$ de cada nó. Cada coordenador é responsável por expandir (visitar) os vértices presentes em seu conjunto $V_{current}$.

O vértice atual $v \in V_{current}$ é recuperado do BDG e um vértice de fronteira que possui uma conexão com o vértice atual v é expandido. O caminho resultante é armazenado temporariamente na variável $caminho$. O novo caminho encontrado passa pela função de avaliação definida pelo usuário. A função de avaliação pode retornar quatro operações possíveis: *Include And Continue*, *Include And Prune*, *Exclude And Continue* e *Exclude And Prune*. Cada uma dessas operações corresponde a uma estratégia que será executada pela busca em largura que está sendo realizada quanto ao caminho encontrado. A palavra-chave *Include* indica que o caminho encontrado será incluído na resposta final da busca enquanto a palavra-chave *Exclude* não inclui o caminho na resposta. A palavra-chave *Continue* indica que a busca deve continuar do ramo atual em diante, expandindo os próximos nós, enquanto a palavra-chave *Prune* indica que a expansão do caminho encontrado não deve ser efetuada e uma poda será executada. Quando um caminho é podado, ele é adicionado à variável $caminhosPodados$. A

operação de poda pode reduzir consideravelmente o tempo total da busca em largura realizada, diminuindo o espaço de busca.

Se o retorno da função de avaliação optar por continuar a busca a partir desse ponto, então os sucessores do vértice atual devem ser expandidos. Para localizar os sucessores de v, as arestas cujo tipo está contido em L são percorridas na mesma direção D. O conjunto de todas as arestas sucessoras a v que obedecem a este critério formam o conjunto Es_v. O vértice atual v, os seus sucessores e as arestas percorridas para alcançá-los são incluídos no mapa compartilhado $paths$. A variável $paths$ é um mapa compartilhado entre todos os nós coordenadores, cujo acesso é sincronizado e que possui como chave o vértice atual e como valor o conjunto de arestas que conectam os sucessores ao vértice atual. Esta estrutura é mantida em memória principal durante a execução da busca corrente.

Os próximos vértices que serão visitados começam a ser computados pela busca. A variável $Neighbors$ armazena os vértices vizinhos sucessores a v. A variável $UnvisitedNeighbors$ armazena os vizinhos que ainda não foram visitados pela busca. $MyUnvisitedNeighbors$ armazena, dos vizinhos que não foram visitados, aqueles que são de responsabilidade do coordenador local corrente.

Para cada vértice vizinho que ainda não foi visitado, é verificado se ele é armazenado pelo BDG local ou se é de responsabilidade de outro BDG. Se for armazenado pelo coordenador local, então o vértice é adicionado em seu conjunto V_{next}. Se for de responsabilidade de outro coordenador local, então ele é enviado via troca de mensagens para o nó responsável.

No passo anterior, os vértices vizinhos foram calculados e enviados para os responsáveis. A próxima etapa é finalmente adicionar o vértice atual v ao conjunto de vértices visitados $V_{visited}$ o que finalmente indica o final do processamento do vértice atual.

Esta etapa se repete até que todos os vértices de $V_{current}$ sejam processados por cada coordenador local. Quando isso acontece os coordenadores encontram um ponto de sincronização que corresponde a uma barreira do modelo BSP. É necessário que todos os

coordenadores locais terminem de processar seu conjunto $V_{current}$ antes de prosseguir com os próximos passos da busca.

Quando todos os coordenadores locais terminam de processar seu conjunto $V_{current}$ e atingem o ponto de sincronização, começa uma etapa em que cada um deles verifica o conjunto de mensagens recebidas. Cada vértice deste conjunto é adicionado ao seu conjunto V_{next}, exceto aqueles que já foram visitados em algum passo anterior.

Finalmente, o conjunto $V_{current}$ é definido com os vértices que serão visitados na próxima etapa da busca, que são aqueles que foram armazenados temporariamente em V_{next}. Um novo ponto de sincronização garante que todos os nós terminaram de processar as mensagens recebidas e estão prontos para prosseguir a busca em largura, explorando os vértices do próximo nível e recomeçando, desta forma, o algoritmo, que se repete enquanto houver vértices a serem processados pela busca.

Algoritmo 1 Busca em Largura Distribuída

```
1: função BFS DISTRIBUIDA(v_s, D, Ev, L)
2:     V_visited = ∅; V_next = ∅;
3:     se v_s ∈ V_i então V_current = {v_s};
4:     senão V_current = ∅;
5:     fim se
6:     ponto de sincronizaçao;
7:     enquanto pelo menos 1 nó possui V_current ≠ ∅ faça
8:         V_allvisited = ⋃ V_visited;          ▷ recupera V_visited de cada nó vizinho
9:         para cada v ∈ V_current faça
10:            recuperar v do BDG local;
11:            calcular caminho entre fronteira atual e v;
12:            se E_v(caminho) = Include_And_Continue então
13:                resultadoBusca ← caminho;
14:            senão se E_v(caminho) = Include_And_Prune então
15:                resultadoBusca ← caminho;
16:                caminhosPodados ← caminho;
17:            senão se E_v(caminho) = Exclude_And_Prune então
18:                caminhosPodados ← caminho;
19:            fim se
20:            se E_v(caminho) decidiu continuar a busca a partir desse ponto então
21:                Es_v = arestas vizinhas a v com tipo ∈ L na direçao D;
22:                para cada e_sucessor ∈ Es_v faça
23:                    v_sucessor = vértice adjacente a v interligado por e_sucessor;
24:                    paths(v) ← (e_sucessor, v_sucessor);
25:                fim para
26:                Neighbors = vértices vizinhos a v com etiqueta ∈ L na direçao D;
27:                UnvisitedNeighbors = Neighbors \ (V_current ∪ V_visited);
28:                MyUnvisitedNeighbors = UnvisitedNeighbors ∩ V_i;
29:                para cada u ∈ UnvisitedNeighbors faça
30:                    se u ∈ MyUnvisitedNeighbors então V_next = V_next ∪ {u};
31:                    senão enviar mensagem para o nó responsável por u;
32:                    fim se
33:                fim para
34:            fim se
35:            V_visited = V_visited ∪ {v};          ▷ adiciona v ao conjunto de visitados
36:        fim para
37:        ponto de sincronizaçao;
38:        M = conjunto de vértices recebidos por mensagens de outros nós;
39:        para cada u ∈ M faça
40:            V_next = V_next ∪ {u} \ (V_visited ∪ V_current);
41:        fim para
42:        V_current = V_next; V_next = ∅;
43:        ponto de sincronizaçao;
44:    fim enquanto
45:    retorna resultadoBusca;
46: fim função
```

Fig. 10 – Pseudocódigo do algoritmo de busca em largura distribuída implementado no Graphene.

3.2.2. Busca por Caminhos Mínimos

O algoritmo distribuído para determinação de menor caminho recebe como entrada o vértice de origem v_s, um vértice de destino v_d, a direção em que as arestas são percorridas D e o conjunto de etiquetas de arestas por onde a busca deve seguir L. Não existe uma função de avaliação definida pelo usuário. O algoritmo de menor caminho implementado utiliza duas operações de busca em largura executadas por cada nó ao mesmo tempo: uma busca em largura partindo de v_s na direção de D tentando alcançar o vértice de destino e outra busca em largura partindo de v_d na direção contrária a D tentando alcançar o vértice de origem.

As duas operações de busca em largura que são executadas no algoritmo de caminhos mínimos são, cada uma, bastante parecidas com o algoritmo descrito na Fig. 10. A busca em largura com direção ao vértice de destino conta com os conjuntos: $V_{visitedForward}$, $V_{currentForward}$ e $V_{nextForward}$ enquanto a busca em direção a origem conta com os conjuntos: $V_{visitedBackward}$, $V_{currentBackward}$ e $V_{nextBackward}$. As duas buscas se comportam de forma independente, onde cada uma possui o seu conjunto próprio de variáveis para controle. Uma nova barreira de sincronização separa a busca em direção ao destino da busca em direção à origem. A condição de parada para o algoritmo de caminhos mínimos é: 1. encontrar uma interseção entre os conjuntos $V_{visitedForward}$ e $V_{visitedBackward}$ ou 2. $V_{currentForward} = \emptyset$ ou $V_{currentBackward} = \emptyset$ e nenhuma interseção entre as buscas foi encontrada. O primeiro caso indica que um caminho foi encontrado entre v_s e v_d enquanto o segundo caso ocorre quando este caminho não existe.

A única variável compartilhada entre as duas buscas executadas pelo algoritmo de caminhos mínimos é a variável $paths$, responsável por armazenar os vértices e arestas percorridos no decorrer da busca. O algoritmo inicia com a execução da busca em largura da origem para o destino. A busca em largura em direção ao destino visita cada vértice $v \in V_{currentForward}$, incluindo os vértices e arestas percorridos na variável $paths$. Os próximos vizinhos são calculados e colocados em $V_{nextForward}$ ou então são enviados para o nó correspondente via troca de mensagens. O conjunto de vértices que devem ser visitados na próxima iteração recebido via troca de mensagens com outros nós são colocados em $V_{nextForward}$. Após esta etapa, existe o novo ponto de sincronização que

separa as duas buscas. A segunda busca tem início e se comporta de forma semelhante à primeira, diferenciando apenas pelo vértice de origem e a direção em que as arestas são percorridas. Este processo se repete alternando entre as duas buscas.

Quando uma interseção é detectada entre $V_{visitedForward}$ e $V_{visitedBackward}$, uma nova busca se inicia no vértice de origem em direção ao destino utilizando os elementos armazenados na variável $paths$. Esta busca retorna todos os caminhos encontrados entre v_s e v_d. Os caminhos retornados correspondem finalmente aos caminhos mínimos entre os vértices de origem e destino procurados.

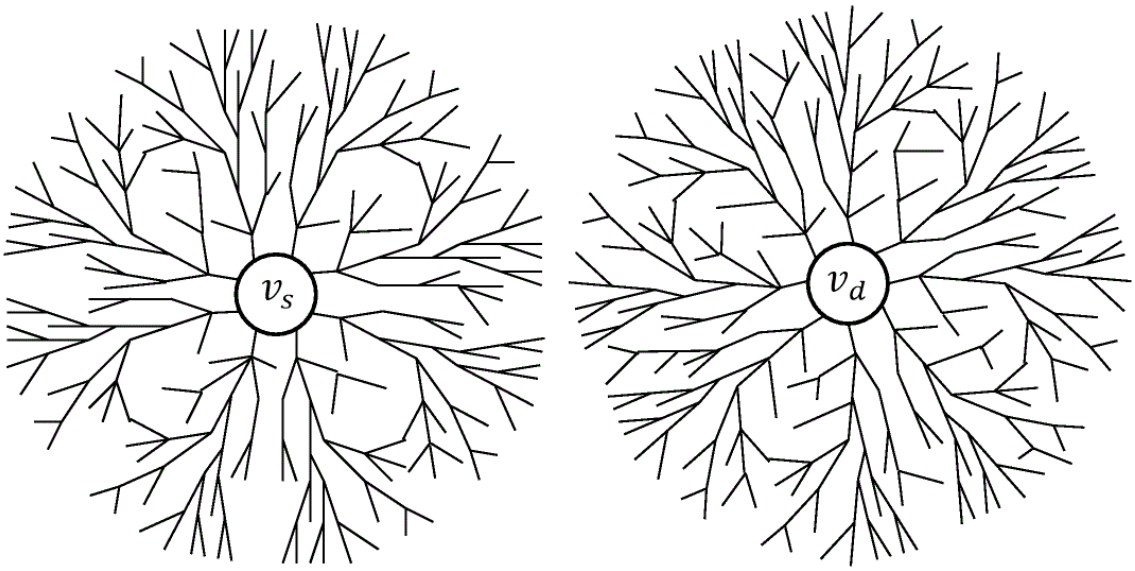


Fig. 11 – Representação da busca de caminhos mínimos bidirecional entre os vértices v_s e v_d implementada no Graphene. Adaptado de (RUSSELL & NORVIG, 2010).

A principal motivação para a implementação de uma busca bidirecional entre os vértices de origem e destino foi o fato de que este método permite reduzir a complexidade de tempo e espaço da busca quando comparada à busca em largura tradicional. A área de dois círculos pequenos é menor do que a área de um círculo grande com origem em v_s e que chega até v_d conforme representado na Fig. 11 (RUSSELL & NORVIG, 2010).

3.3. Criação de um BDG distribuído a partir de BDG centralizados já existentes

Bases de dados de grafos preexistentes podem ser interligadas através do Graphene de forma bastante simples. Uma grande variedade de SGBD também é suportada bastando apenas que a solução de banco de dados utilizada seja compatível com a especificação Blueprints e suporte a indexação de propriedades associadas a vértices e arestas.

Para interligar um conjunto de bases de dados através do Graphene basta gerar identificadores únicos para os vértices e arestas dos grafos já existentes. Uma nova propriedade com chave igual a "__id" deve ser criada para cada elemento do grafo e seu valor deve ser o novo identificador único associado àquele elemento. Este será o identificador a ser utilizado ao se referenciar um elemento pelo Graphene. Todos os elementos devem ser marcados também com a propriedade "_partition" e o valor "original" uma vez que inicialmente não haverá arestas virtuais entre os fragmentos.

Durante a geração de um identificador único, o usuário deve adotar uma estratégia que permita localizar posteriormente o nó onde está armazenada a base de dados original à qual o vértice pertencia. O usuário deve escrever uma classe filha de *FragmentationPolicy* implementando o método *vertexDestination* para informar ao *middleware* a localização do vértice dado o seu identificador. A aplicação cliente deve utilizar um *GrapheneGraph* passando como parâmetro a política de fragmentação que foi criada. Estas etapas já são suficientes para utilização do banco de dados como um fragmento participante do Graphene. A partir deste ponto, o Graphene já é capaz de localizar os elementos de todos os BDG participantes, criar novos elementos que serão automaticamente alocados nos BDG responsáveis de acordo com a política de fragmentação implementada e criar arestas entre elementos armazenados em BDG originalmente distintos.

As modificações que uma base de dados já existente deve sofrer para participar do Graphene são bastante simples e pouco intrusivas. Da mesma forma, a conversão de um BDG distribuído utilizado pelo Graphene para uma solução centralizada é muito simples bastando para isso remover todos os elementos que possuem a propriedade

“_partition” igual ao valor “virtual”. Esta alteração pode ser realizada rapidamente, uma vez que todos os elementos virtuais são indexados pelo Graphene, o que torna muito rápida a sua localização e remoção, caso necessário.

Capítulo 4 – Resultados Experimentais

Para testar o Graphene em diferentes cenários, optou-se por realizar a importação de grafos de aplicações reais para uma base de dados distribuída. O SGBDG utilizado junto a cada coordenador local foi o Neo4j, versão 1.7.2. Diversas consultas que ocorrem com frequência em banco de dados de grafos foram submetidas ao Graphene e o tempo de execução de cada uma foi coletada. As mesmas consultas também foram submetidas a uma base Neo4j centralizada para auxiliar a comparação entre os resultados obtidos a partir da base centralizada e da base distribuída gerenciada pelo Graphene.

Grafos naturais originados a partir de aplicações reais foram escolhidos para os experimentos. Três bases de dados distintas foram utilizadas: WikiVote, Epinions e WebStanford. Todos os arquivos de entrada para geração das bases foram obtidos a partir do SNAP[20]. O tamanho das bases de dados do WikiVote, Epinions e WebStanford centralizadas no Neo4j são, respectivamente 16, 85 e 397 *megabytes*.

O WikiVote é uma base gerada a partir de uma votação ocorrida na rede social Wikipedia[21] para eleger novos usuários que receberão o perfil de administrador da comunidade. Os usuários candidatos a administradores recebem votos de outros usuários. Os vértices do WikiVote representam usuários da Wikipedia e as arestas direcionadas indicam que um usuário votou em outro. As características principais do grafo WikiVote são descritos na Tabela 1.

[20] SNAP – Stanford Network Analysis Project - http://snap.stanford.edu/index.html

[21] Wikipedia - http://www.wikipedia.org/

Vértices	7115
Arestas	103689
Componentes Fracamente Conectados (CC)	24
Vértices no maior CC	7066 (99,3%)
Arestas no maior CC	103663 (~ 100%)
Componentes Fortemente Conectados (CFC)	5816
Vértices no maior CFC	1300 (18,3%)
Arestas no maior CFC	39456 (38,1%)
Coeficiente de agrupamento	0,1409
Diâmetro da rede	7
Diâmetro efetivo do percentil 90	3,8

Tabela 1 – Características principais do grafo da base de dados do WikiVote

O Epinions[22] é uma rede social em que os usuários são capazes de realizar avaliações sobre produtos e serviços. Um usuário cadastrado nesta rede social pode selecionar um grupo de usuários em quem confia, criando uma rede de confiança entre os usuários da comunidade. A rede de confiança é utilizada para determinar as avaliações mais relevantes para aquele usuário. Os vértices da base de dados extraída do Epinions representam os usuários da rede e as arestas direcionadas representam as relações de confiança. As características principais da base de dados Epinions são descritas na Tabela 2.

[22] Epinions - http://www.epinions.com/

Vértices	75879
Arestas	508837
Componentes Fracamente Conectados (CC)	2
Vértices no maior CC	75877 (~ 100%)
Arestas no maior CC	508836 (~100%)
Componentes Fortemente Conectados (CFC)	42176
Vértices no maior CFC	32223 (42,5%)
Arestas no maior CFC	443506 (87,2%)
Coeficiente de agrupamento	0.1378
Diâmetro da rede	14
Diâmetro efetivo do percentil 90	5

Tabela 2 – Características principais do grafo da base de dados do Epinions

O WebStanford é uma base de dados de grafos que representa ligações entre páginas da *web* da Universidade de Stanford[23]. Os vértices representam os documentos *web* e as arestas representam as ligações entre os documentos. As características principais da base de dados WebStanford são descritas na Tabela 3.

[23] Stanford University - http://www.stanford.edu/

Vértices	281903
Arestas	2312497
Componentes Fracamente Conectados (CC)	365
Vértices no maior CC	255265 (90,6%)
Arestas no maior CC	2234572 (96,6%)
Componentes Fortemente Conectados (CFC)	29914
Vértices no maior CFC	150532 (53,4%)
Arestas no maior CFC	1576314 (68,2%)
Coeficiente de agrupamento	0.5976
Diâmetro da rede	674
Diâmetro efetivo do percentil 90	9,7

Tabela 3 – Características principais do grafo da base de dados do WebStanford

As operações mais frequentes em banco de dados de grafos e geralmente utilizadas em benchmarks destas ferramentas são descritas por (CIGLAN *et al.*, 2012) e (PRABHAKARAN *et al.*, 2012). Dentre estas operações, podemos citar: recuperar um conjunto de vértices a uma distância N de um vértice de origem (N-saltos), recuperar o conjunto de vértices que possuem uma propriedade com um valor previamente especificado e a executar de travessias a partir de um vértice de origem. Consultas simulando estas operações foram submetidas ao Graphene.

O ambiente utilizado para a execução dos experimentos foi a plataforma Grid'5000 (CAPPELLO *et al.*, 2005). O Grid'5000 é uma infraestrutura computacional mantido pelo INRIA[24] e CNRS[25] que fornece um instrumento científico para o estudo de sistemas paralelos e distribuídos em grande escala. O *cluster* utilizado para a execução dos experimentos possui 64 nós. Cada nó é equipado com 2 CPU Intel Xeon L5420 (4

[24] INRIA – Instituto Nacional de Pesquisas em Ciência da Computação e Automação

[25] CNRS - Centro Nacional de Pesquisa Científica

núcleos por CPU), 32 GB de memória e um espaço de armazenamento local de 320 GB. Os nós do *cluster* são interligados por uma rede Gigabit Ethernet.

Para avaliar o comportamento do Graphene em múltiplos cenários, cada base de dados foi fragmentada e distribuída em 2, 4, 8 e 16 fragmentos. Cada fragmento foi alocado a um nó computacional. O coordenador central foi alocado em um novo nó do *cluster*. A aplicação cliente que submete as operações ao Graphene também foi isolada em um nó à parte. Cada experimento foi executado três vezes e o valor considerado como resultado é aquele obtido pela média das três execuções.

Duas estratégias de fragmentação foram avaliadas nos testes: a fragmentação baseada em Espalhamento e a estratégia de fragmentação multi nível METIS (KARYPIS & KUMAR, 1998). O objetivo desta comparação foi tentar quantificar mais precisamente o quanto a estratégia de fragmentação adotada pode influenciar no desempenho de consultas a bases de dados de grafos distribuídas. Para a estratégia de fragmentação de Espalhamento, a política padrão do Graphene foi utilizada. No caso da fragmentação METIS, a base de dados foi fragmentada fora do Graphene e a localização do vértice foi incorporada ao seu identificador. Uma política de fragmentação personalizada foi implementada para retornar a localização de acordo com o identificador. Este processo é descrito em maiores detalhes a seguir, na seção 4.1, que detalha o experimento de importação dos dados. No Apêndice B é possível encontrar uma tabela com informações detalhadas sobre os fragmentos encontrados utilizando ambas estratégias de fragmentação.

As próximas seções descrevem os experimentos executados. Na seção 4.1 é descrito o experimento de importação das bases de dados. Em 4.2 é descrito o experimento da consulta N-Saltos. Em 4.3 são apresentados os resultados da consulta de recuperação de elementos baseado em propriedades com determinada chave e valor. Em 4.4 é descrito o resultado da consulta de travessia executada com o algoritmo distribuído descrito em 3.2.1. A seção 4.5 descreve o resultado da consulta por caminhos mínimos executada com o algoritmo descrito em 3.2.2. A seção 4.6 descreve o experimento onde múltiplos usuários executam operações no Graphene ao mesmo tempo. A seção 4.7 descreve o experimento da utilização do Graphene com bases de dados preexistentes. A seção 4.8 traz uma análise sobre os principais resultados obtidos.

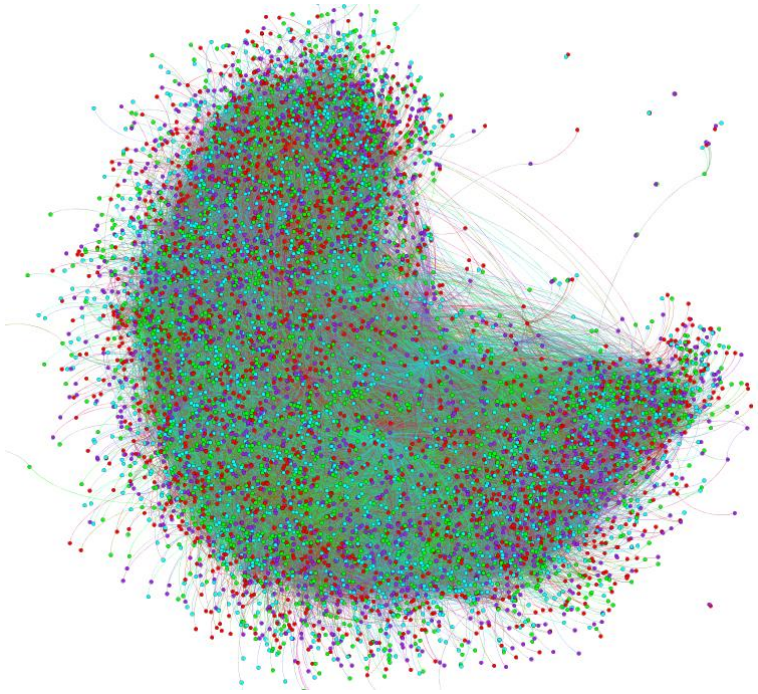


Fig. 12 – Visualização da estrutura do grafo formado a partir da base de dados do WikiVote distribuída em quatro fragmentos de acordo com a fragmentação baseada em Espalhamento. A coloração dos vértices representa o fragmento de destino deste elemento.

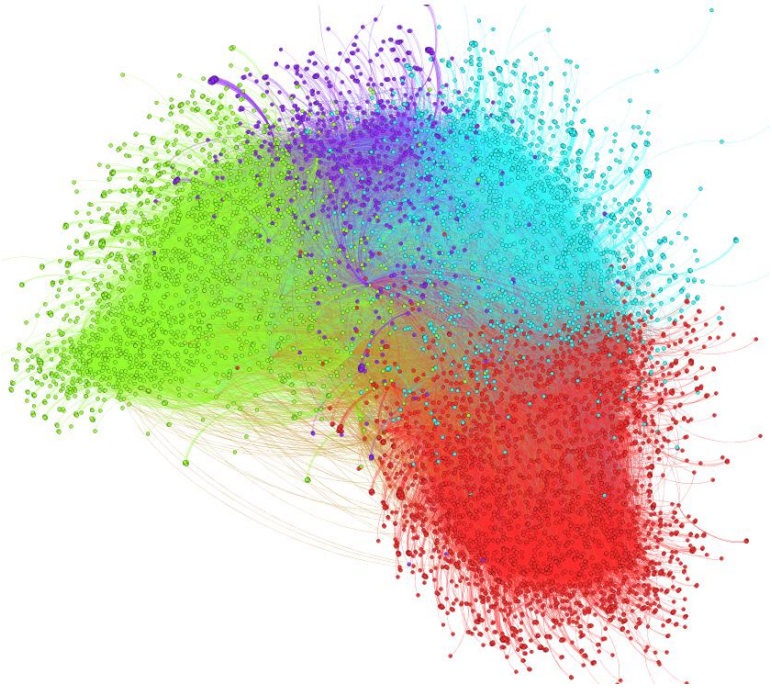


Fig. 13 - Visualização da estrutura do grafo formado a partir da base de dados do WikiVote distribuída em quatro fragmentos de acordo com a fragmentação METIS. A coloração dos vértices representa o fragmento de destino deste elemento.

4.1 Importação dos Dados

No experimento de importação dos dados foram comparados o desempenho da solução centralizada Neo4j e a solução distribuída proposta pelo Graphene para realização da carga de dados das bases analisadas. Os arquivos de entrada obtidos através do SNAP são arquivos de texto que contêm a descrição das arestas do grafo. No experimento centralizado, a carga de dados foi realizada através da classe *Neo4jBatchGraph*, que fornece uma implementação eficiente Neo4j para a inserção de dados em lote em um banco Neo4j. Os experimentos executados no Graphene com as bases distribuídas, por sua vez, utilizaram um utilitário do *middleware* para a inserção em lote através da classe *CargaDadosUtil*. Este utilitário utiliza um empacotador *(wrapper)* que faz parte da especificação do Blueprints para realização da inserção de elementos em lote de forma eficiente.

O tempo contabilizado para a carga de dados corresponde ao tempo gasto na leitura do arquivo de texto de origem, inserção dos elementos e indexação de propriedades associadas aos vértices e arestas. No ambiente distribuído, esta carga foi realizada em paralelo e o tempo considerado foi o do fragmento mais demorado. No caso da fragmentação baseada em Espalhamento, o mesmo arquivo de origem foi copiado para todos os nós que, durante a leitura, realizaram a inserção apenas dos elementos pelos quais eram responsáveis. No caso da fragmentação METIS, a estratégia foi diferente. Neste caso, a etapa da fragmentação do grafo ocorreu fora do Graphene, através de um módulo desenvolvido para esta finalidade.

No caso da fragmentação METIS, a primeira etapa do processo realizado foi a utilização de um módulo desenvolvido em Python para a leitura do arquivo de entrada. O grafo foi representado em memória com a utilização da biblioteca NetworkX (HAGBERG *et al.*, 2008). Esta biblioteca fornece um utilitário que funciona como um invólucro para realização de fragmentação de grafos através do METIS. Após encontrar os fragmentos, cada vértice tem o seu identificador alterado incluindo um sufixo "_" que é concatenado com o número do fragmento que foi atribuído pelo METIS. Uma função de fragmentação personalizada foi implementada no Graphene para utilizar o sufixo do identificador do vértice para identificar o nó responsável por ele. Um arquivo de texto final que contempla os vértices com identificador modificado é gerado por este módulo e

carregado em cada nó do cluster durante a importação dos dados. O tempo contabilizado na importação de dados do METIS inclui a etapa de pré-processamento do arquivo de entrada e geração dos fragmentos e a etapa de leitura do arquivo final e efetiva geração da base de dados final. O tempo de importação das bases de dados são representados na Fig. 14 para a base do WikiVote, na Fig. 15 para a base do Epinions e na Fig. 16 para a base de dados do WebStanford.

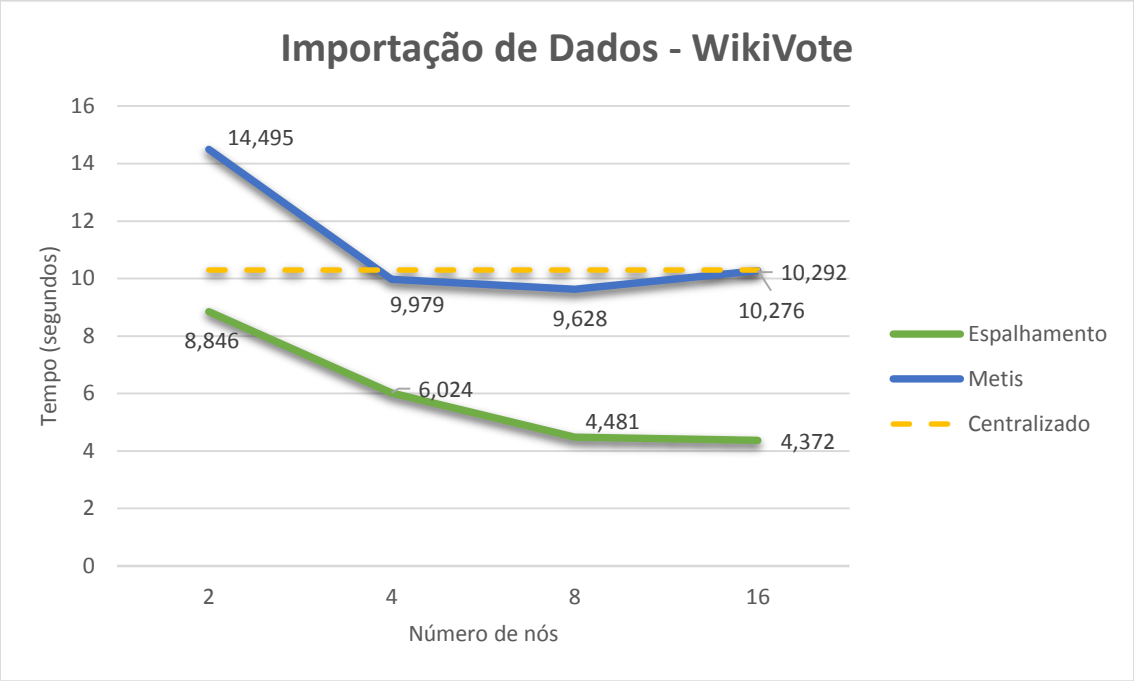


Fig. 14 – Tempo (em segundos) para a importação dos dados do WikiVote de acordo com a fragmentação de Espalhamento e METIS.

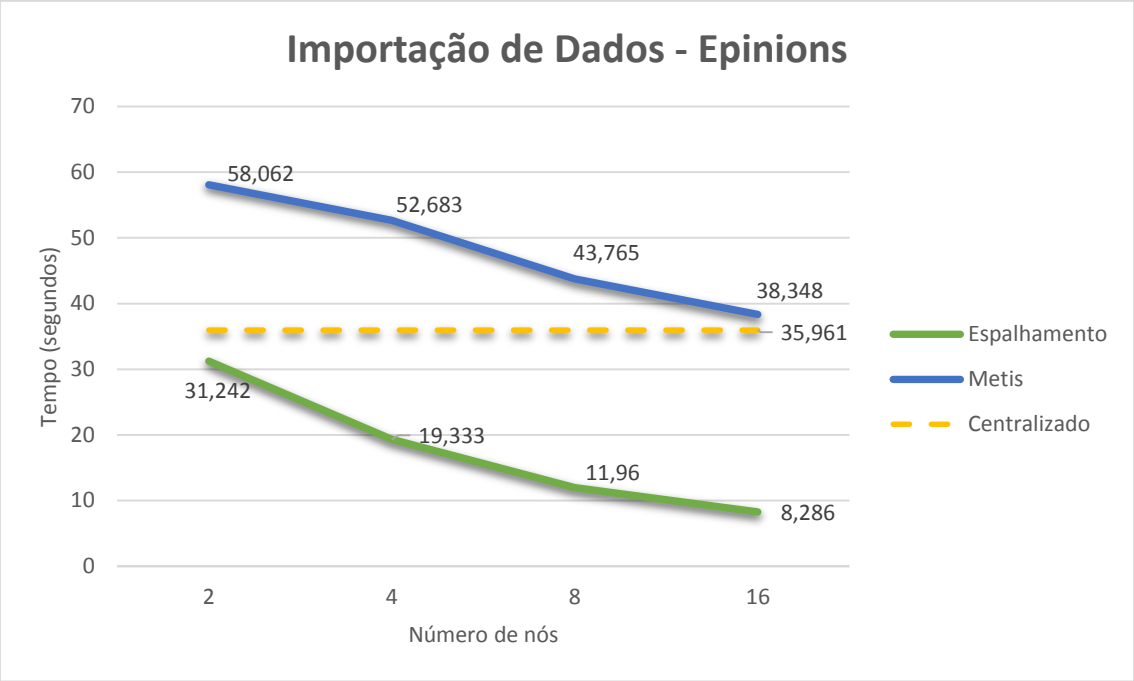


Fig. 15 - Tempo (em segundos) para a importação dos dados do Epinions de acordo com a fragmentação de Espalhamento e METIS.

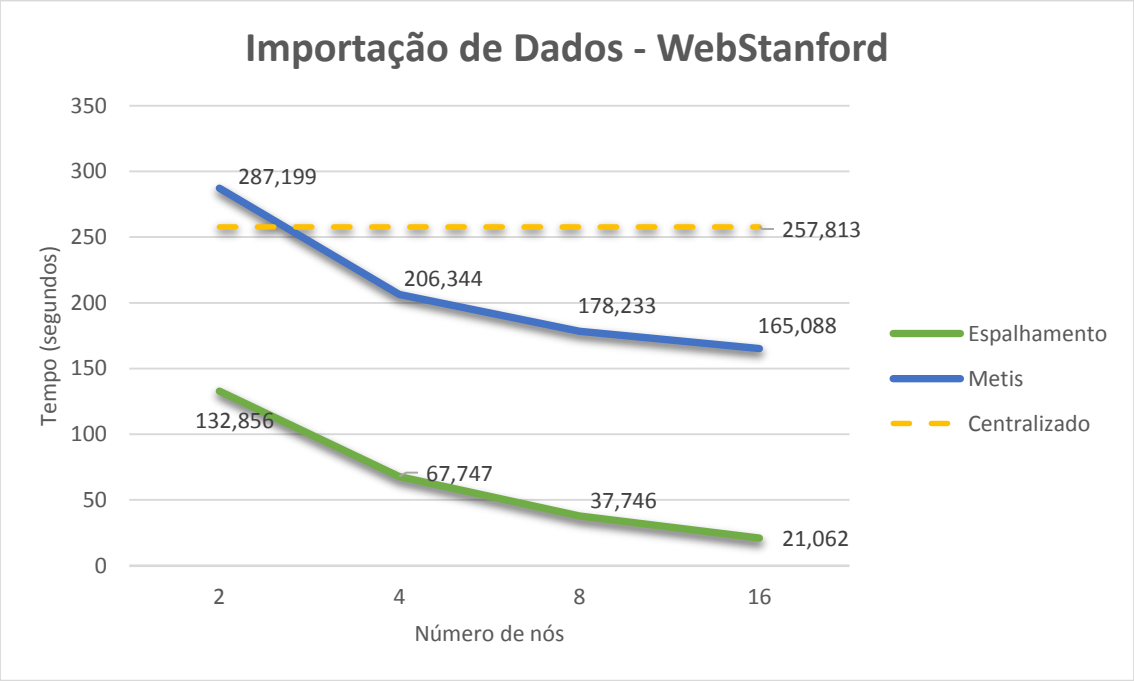


Fig. 16 - Tempo (em segundos) para a importação dos dados do WebStanford de acordo com a fragmentação de Espalhamento e METIS.

Na carga de dados distribuída, a fragmentação baseada em Espalhamento teve o melhor desempenho. A fragmentação METIS foi penalizada pelo tempo de pré-processamento do arquivo de entrada para encontrar os fragmentos de cada vértice. A fatia de tempo responsável pela etapa de pré-processamento foi considerável e levou a um desempenho pior do que o centralizado em alguns cenários específicos. Avaliamos que o número de fragmentos em que se deseja fragmentar o grafo tem pouca influência no tempo de execução do algoritmo de fragmentação METIS. Independentemente do número de fragmentos gerados, o tempo gasto na etapa de pré-processamento foi aproximadamente **4,57** segundos para o WikiVote, **27,19** segundos para o Epinions e **128,05** segundos para o WebStanford. Este tempo inclui a leitura do arquivo original, a execução do algoritmo de fragmentação e a escrita do arquivo final com os fragmentos detectados que foi utilizado de fato por cada nó para a importação dos dados.

Apesar de gerar fragmentos de melhor qualidade minimizando o número de arestas de corte entre os fragmentos, a fragmentação METIS também gera alguns fragmentos não balanceados. Alguns fragmentos podem conter um grande número de vértices e arestas enquanto outros fragmentos recebem menos elementos. Esta característica afetou diretamente a importação dos dados, pois verificamos que alguns nós do cluster responsáveis por fragmentos maiores levaram um tempo bastante superior aos nós responsáveis por fragmentos menores. Por exemplo, na importação METIS de 2 fragmentos da base de dados WebStanford, o nó responsável pelo fragmento 1 (menor) levou **89,778** segundos para realizar a importação dos dados enquanto o nó responsável pelo fragmento 2 (maior) levou **159,228** segundos. Este comportamento não foi verificado na fragmentação de Espalhamento, onde o tempo de carga de cada nó foi aproximadamente igual, como esperado. A falta de balanceamento entre os fragmentos prejudicou o desempenho da fragmentação METIS em alguns cenários, uma vez que apenas o tempo do nó mais lento foi considerado neste experimento. Apesar disso, a importação dos dados em paralelo realizada por cada nó se mostrou vantajosa para a maioria dos cenários quando comparada à carga de dados centralizada.

4.2 Consulta N-Saltos

A consulta N-saltos tem como objetivo recuperar o conjunto de vértices vizinhos que estão a uma profundidade máxima N a partir do vértice de origem. O experimento executado no Graphene definiu aleatoriamente o vértice de origem para cada uma das bases analisadas. Desta forma, os vértices de identificadores 1000, 4567 e 59693 foram escolhidos ao acaso para a origem da consulta N-saltos nas bases de dados WikiVote, Epinions e WebStanford, respectivamente. Também foi definido que o valor de N seria 2, ou seja, esta consulta retorna os vizinhos imediatos ao vértice de origem e os vizinhos de cada vizinho imediato formando o conjunto de todos os vértices alcançados com 2 saltos.

No caso do WikiVote, para o vértice de origem escolhido foi possível alcançar 1219 vértices com 2 saltos, enquanto no Epinions 1890 vértices e no WebStanford 166 vértices foram alcançados. A Fig. 17 descreve os resultados para a base do WikiVote, a Fig. 18 apresenta os resultados para o Epinions e a Fig. 19 apresenta os resultados para a base WebStanford.

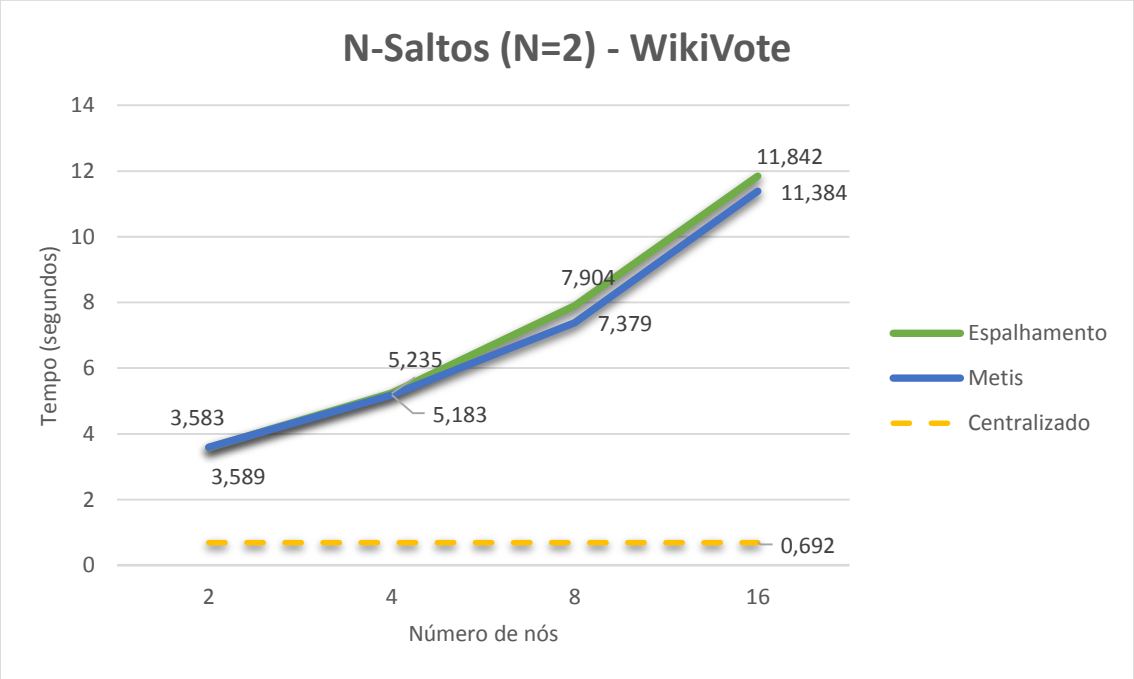


Fig. 17 – Tempo (em segundos) para a execução da consulta de N-hops, obtendo o conjunto dos nós vizinhos ao redor do nó de origem 1000 com até 2 saltos (N=2).

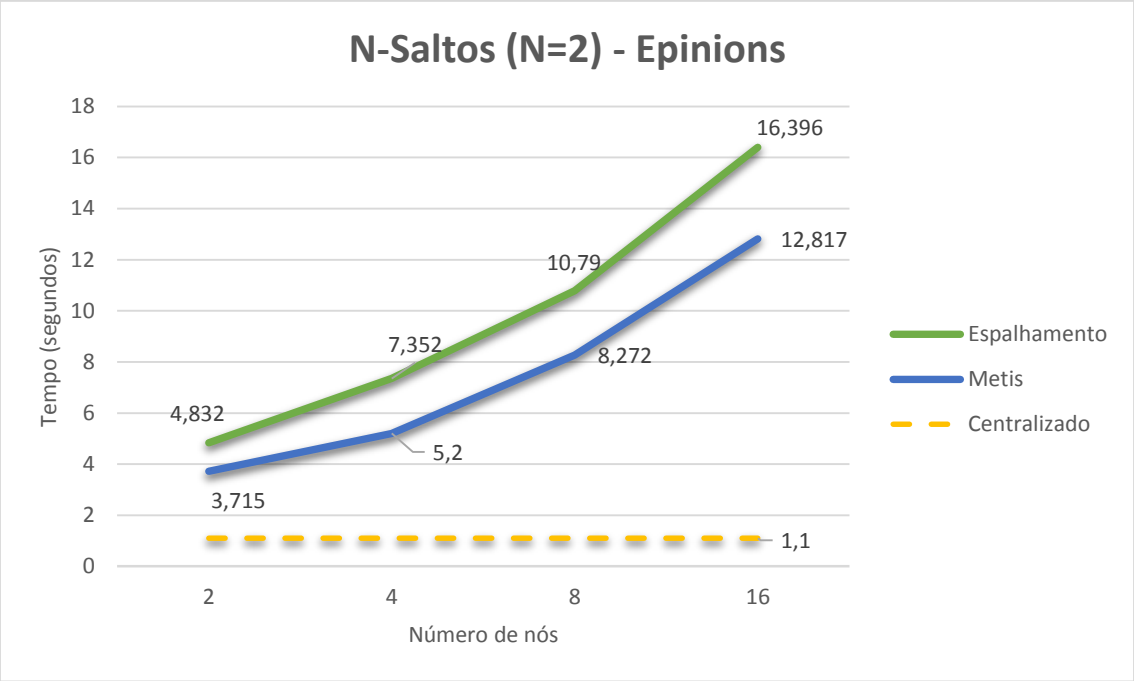


Fig. 18 - Tempo (em segundos) para a execução da consulta de N-hops, obtendo o conjunto dos nós vizinhos ao redor do nó de origem 4567 com até 2 saltos (N=2).

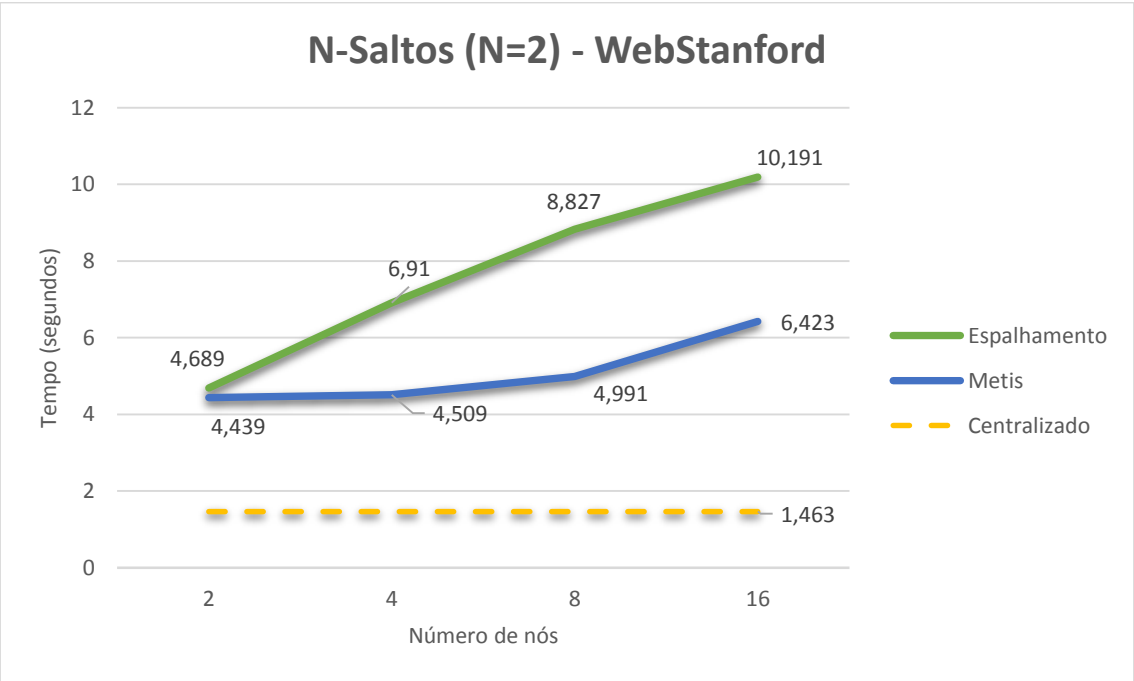


Fig. 19 - Tempo (em segundos) para a execução da consulta de N-hops, obtendo o conjunto dos nós vizinhos ao redor do nó de origem 59693 com até 2 saltos (N=2).

Para a consulta N-hops a fragmentação METIS teve um desempenho superior ao da fragmentação por Espalhamento. Isso se deve ao fato da comunicação entre os nós ter sido reduzida. Para a consulta N-saltos, é vantajoso que grande parte da vizinhança de um vértice esteja no mesmo fragmento do vértice de origem, uma vez que a necessidade de consultar outros nós da rede para realização da consulta é reduzida neste cenário. É possível observar uma piora no tempo de execução da consulta em função do aumento do número de fragmentos para a maioria dos casos observados, principalmente quando a fragmentação de Espalhamento foi utilizada.

4.3 Busca de Elementos por Propriedades

A busca de elementos por propriedades é um tipo de consulta que visa a recuperar elementos de um grafo que satisfaçam a predicados de seleção baseados em valores estabelecidos para chaves que representem suas propriedades. Estas buscas são mais frequentes em SGBDG que suportam a indexação de propriedades associadas aos elementos. Se o banco de dados de grafo possuir uma tabela de índices para a propriedade buscada, então os índices são utilizados para recuperação dos elementos. Caso contrário, uma travessia sequencial é executada para cada vértice ou aresta do grafo, filtrando manualmente os elementos desejados.

O Neo4j suporta a criação de índices associados às propriedades dos elementos. Por este motivo, os experimentos de busca por propriedades que foram executados fizeram uso de índices.

A busca por elementos baseado em propriedades funciona da seguinte forma no Graphene: o coordenador local responsável pela operação envia uma requisição aos outros coordenadores solicitando que estas bases façam uma busca local procurando por elementos que satisfaçam ao predicado especificado. Cada coordenador local realiza então uma busca em sua base local e retorna uma lista com os elementos que foram encontrados para o coordenador responsável pela operação. O coordenador responsável pela operação aguarda que todos os coordenadores locais tenham executado sua consulta local e tenham devolvido a lista de elementos. Quando isso acontece, o coordenador

responsável pela operação devolve a lista de elementos final ao coordenador central que por sua vez a devolve à aplicação cliente.

A busca de elementos por propriedades foi dividida em dois experimentos diferentes: o primeiro experimento buscou por propriedades comuns associadas aos vértices do grafo. Nesta consulta foram retornados 928 vértices para a base do WikiVote e 11111 vértices para as bases do Epinions e WebStanford. O resultado deste experimento pode ser visualizado na Fig. 20, Fig. 21 e Fig. 22. O segundo experimento buscou por uma propriedade que estava associada a um único vértice. O segundo experimento foi executado para ajudar a medir o impacto de enviar a consulta para cada coordenador local. Este experimento exclui também o tempo necessário para combinar os resultados antes de retornar a resposta para a aplicação cliente.

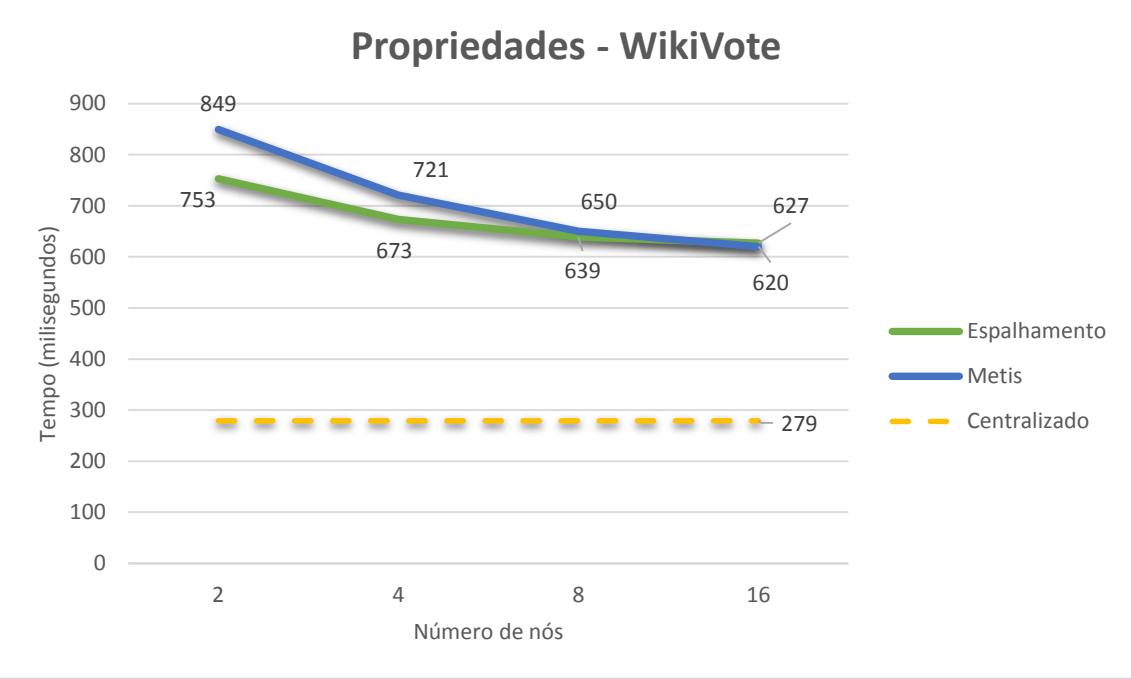


Fig. 20 – Tempo (em milissegundos) para a execução da consulta que recupera um conjunto de vértices baseado em propriedades para a base de dados do WikiVote.

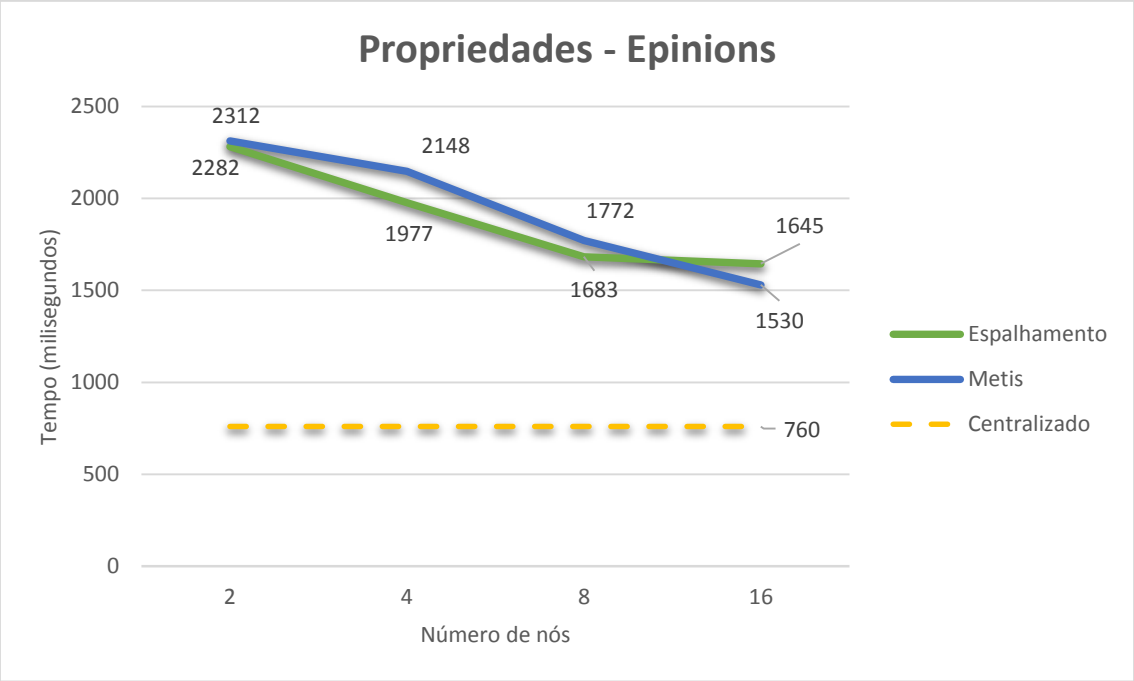


Fig. 21 - Tempo (em milissegundos) para a execução da consulta que recupera um conjunto de vértices baseado em propriedades para a base de dados do Epinions.

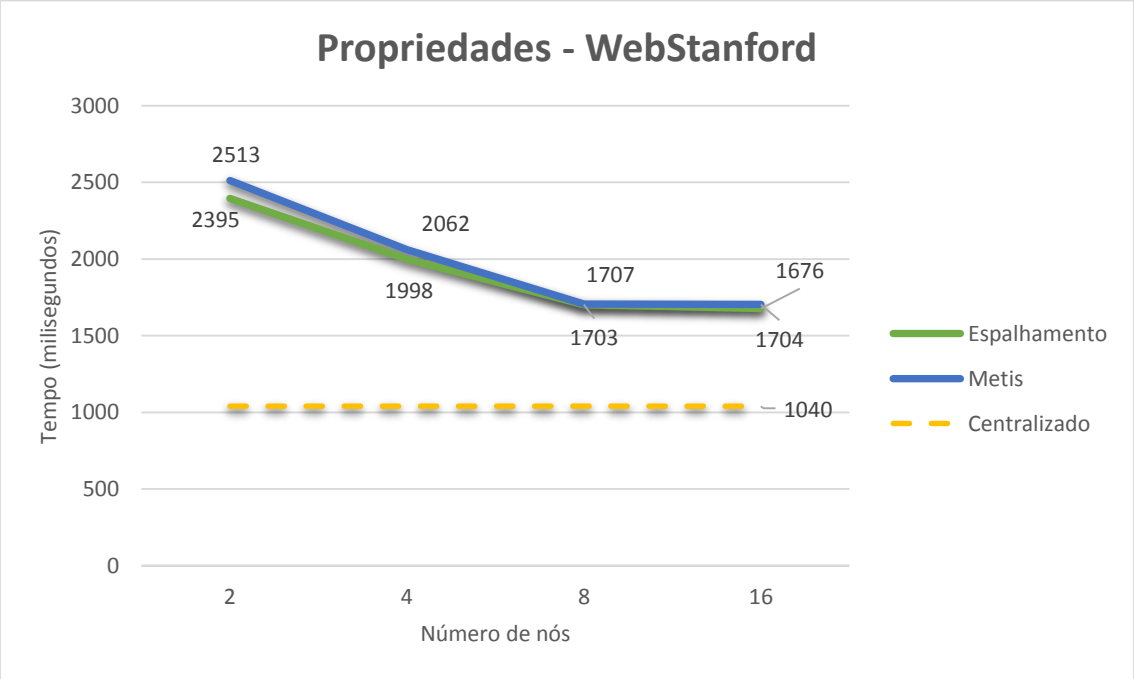


Fig. 22 - Tempo (em milissegundos) para a execução da consulta que recupera um conjunto de vértices baseado em propriedades para a base de dados do WebStanford.

Para as consultas baseadas em propriedades a escolha da estratégia de fragmentação teve pouca influência sobre o desempenho, uma vez que ambas as estratégias apresentaram um desempenho bastante próximo na maioria dos cenários. Dividir a base em um número maior de fragmentos também se mostrou vantajoso até certo ponto. A partir de um determinado ponto, o tempo de execução da consulta tende a se manter constante mesmo dividindo a base em um maior número de fragmentos devido ao acréscimo no custo de comunicação entre os nós envolvidos.

O segundo experimento relacionado à busca por propriedades foi executado buscando por uma propriedade associada a um único elemento. O desempenho obtido está representado pelos gráficos presentes na Fig. 23, Fig. 24 e Fig. 25.

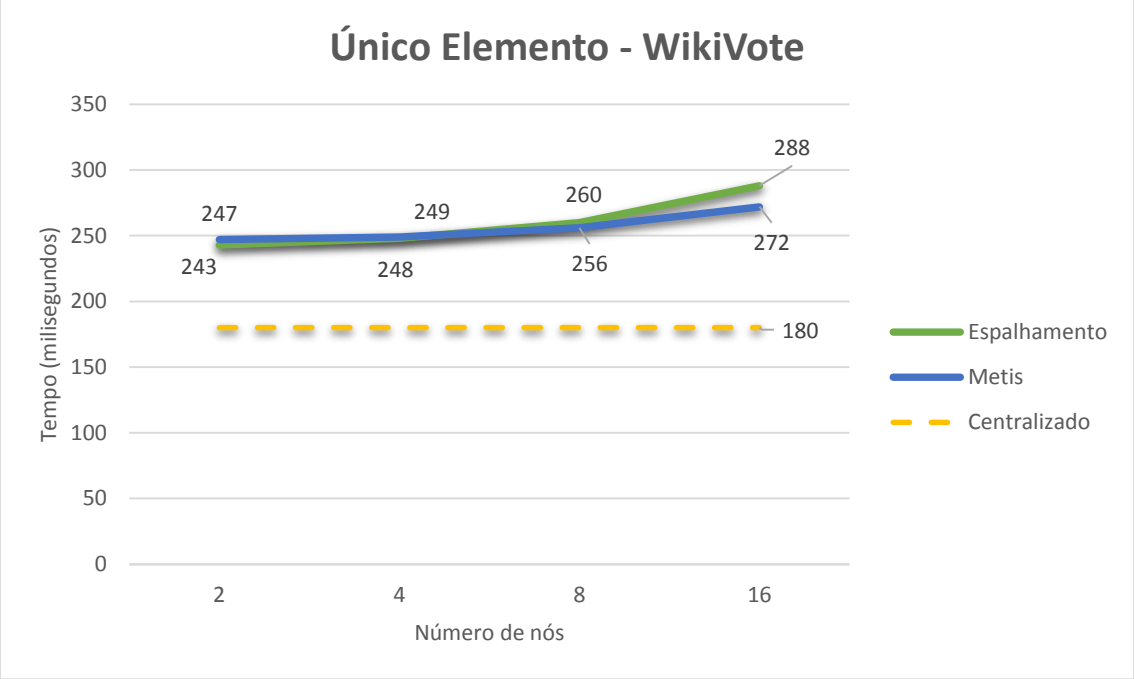


Fig. 23 - Tempo (em milissegundos) para a execução da consulta que recupera um único vértice buscando por uma propriedade para a base de dados do WikiVote.

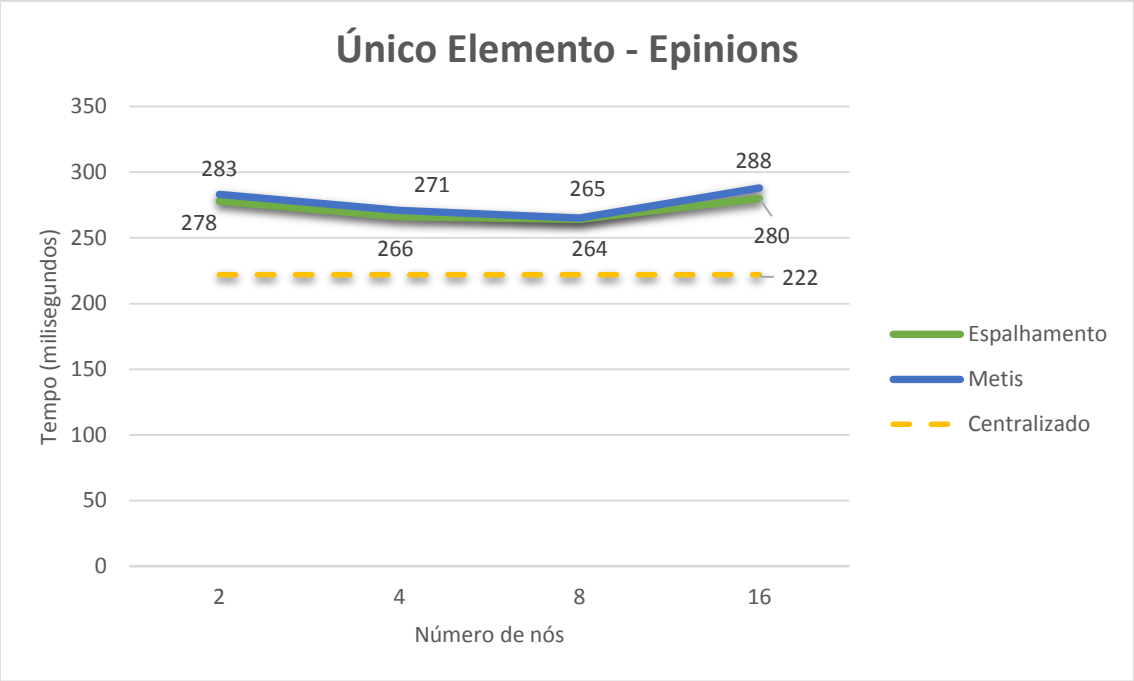


Fig. 24 - Tempo (em milissegundos) para a execução da consulta que recupera um único vértice buscando por uma propriedade para a base de dados do Epinions.

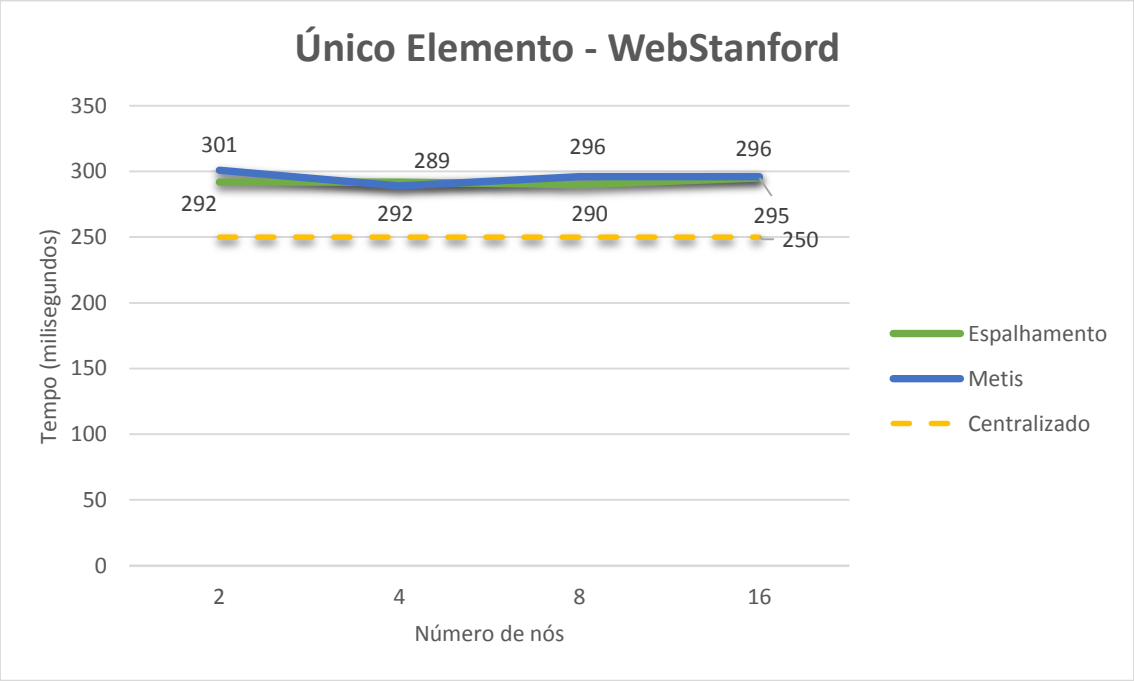


Fig. 25 - Tempo (em milissegundos) para a execução da consulta que recupera um único vértice buscando por uma propriedade para a base de dados do WebStanford.

O tempo de execução da busca por propriedade única se manteve aproximadamente constante com o aumento do número de nós. A diferença entre o tempo de execução das consultas nas bases de dados distribuídas gerenciadas pelo Graphene e a base centralizada foi menor do que no experimento anterior, o que demonstrou que grande parte do tempo de execução total da consulta é gasto com os processos de serialização e desserialização de objetos, troca de mensagens entre nós e com a combinação de resultados de múltiplos nós antes de calcular e retornar o resultado final da busca para a aplicação cliente.

4.4 Travessia

O experimento de travessia executado considerou um vértice de origem para cada base de dados e teve como objetivo retornar todos os caminhos que passassem por vértices com uma propriedade que foi pré-definida na consulta. A execução deste experimento utilizou o algoritmo de busca em largura distribuído descrito na seção 3.2.1.

Uma função de avaliação personalizada foi implementada no Graphene cuja lógica de avaliação permitia que a busca executasse uma poda ao encontrar um caminho com um vértice que não possuía a propriedade especificada. Da mesma forma, uma outra função de avaliação com a mesma lógica foi implementada ao executar a travessia na base de dados Neo4j centralizada.

Os vértices de origem para a travessia das bases de dados WikiVote, Epinions e WebStanford foram os mesmos utilizados para a origem da consulta N-Saltos da seção 4.2. Esta consulta retornou 353, 597 e 174 caminhos respectivamente para as bases utilizadas. Os resultados obtidos estão representados na Fig. 26, Fig. 27 e Fig. 28.

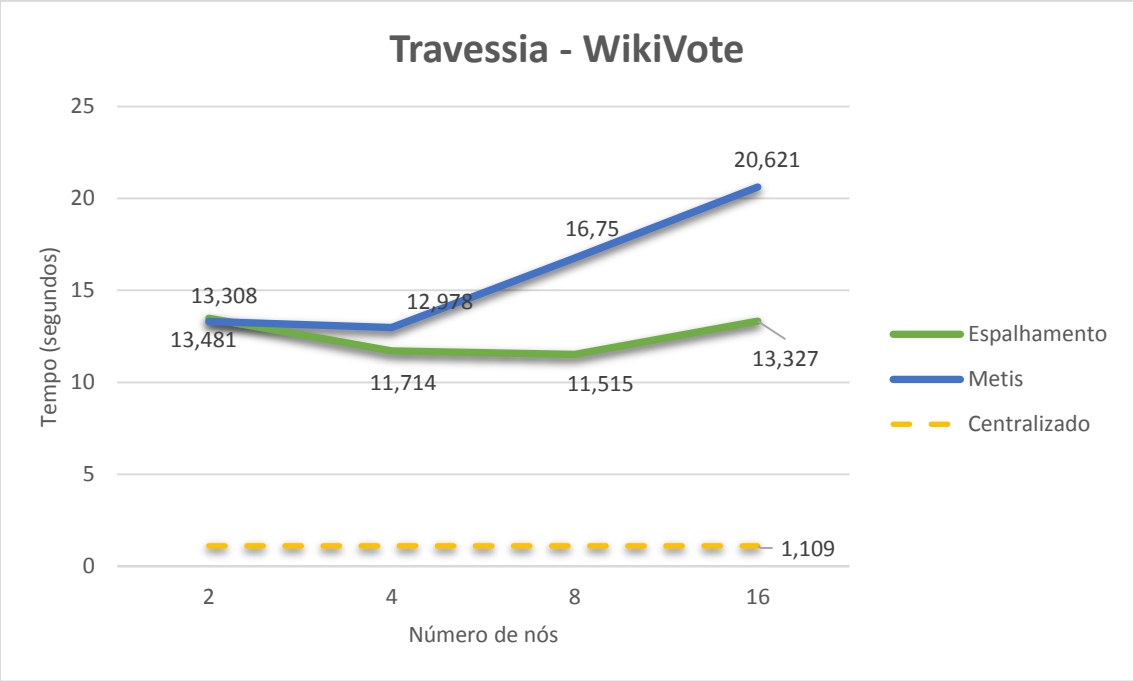


Fig. 26 – Tempo (em segundos) para a execução da consulta que recupera os caminhos que respeitam a função de avaliação especificada para a base de dados do WikiVote

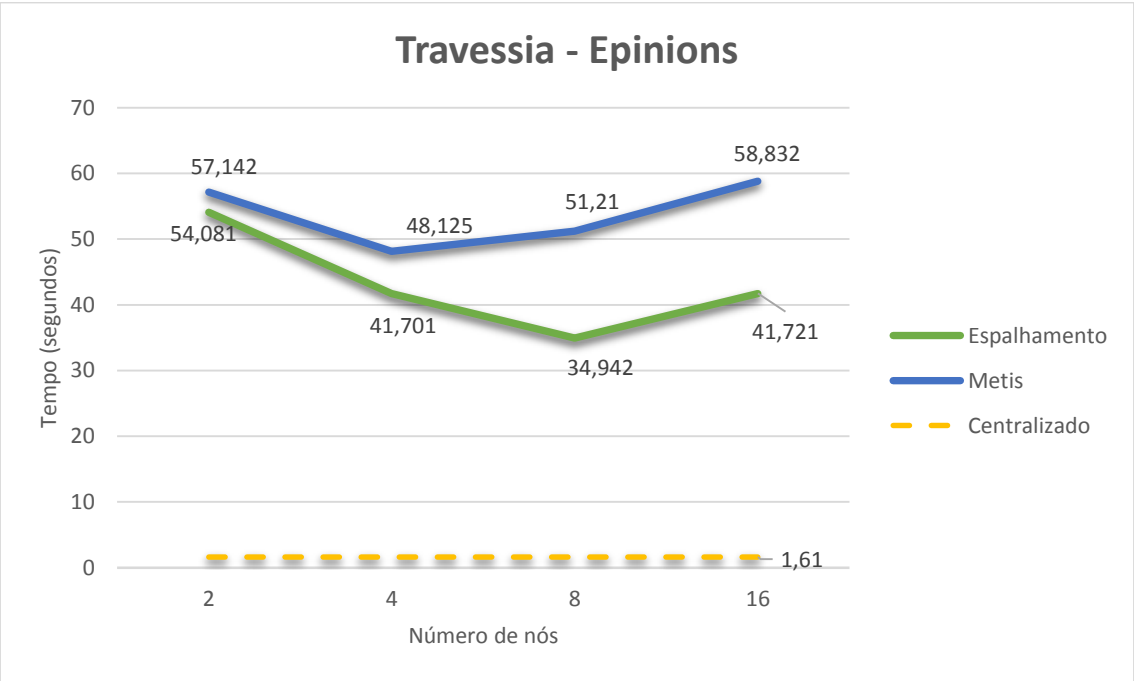


Fig. 27 - Tempo (em segundos) para a execução da consulta que recupera os caminhos que respeitam a função de avaliação especificada para a base de dados do Epinions

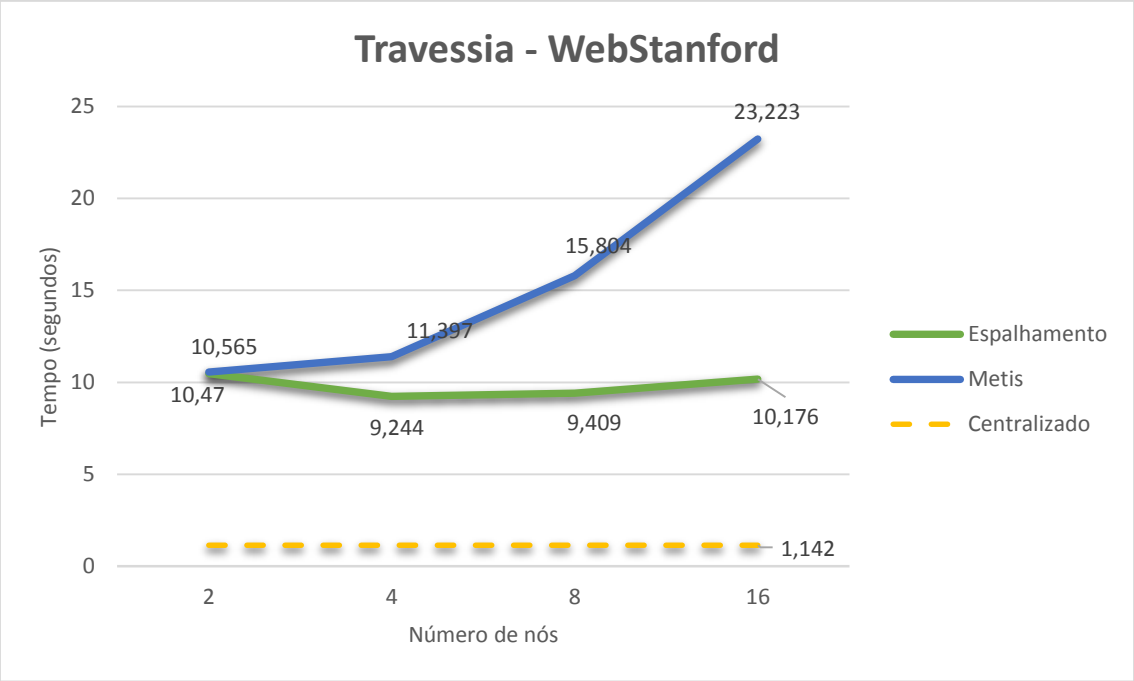


Fig. 28 - Tempo (em segundos) para a execução da consulta que recupera os caminhos que respeitam a função de avaliação especificada para a base de dados do WebStanford

Os resultados encontrados demonstram que a estratégia de fragmentação baseada em Espalhamento obteve um melhor desempenho quando comparado à fragmentação METIS em todos os cenários avaliados. Isso ocorreu porque o algoritmo de busca em largura distribuído implementado beneficiou a divisão do trabalho em múltiplos nós. No caso da fragmentação METIS, o fato das arestas de corte entre os fragmentos terem sido minimizadas contribuiu para a execução local e centralizada de boa parte da computação da travessia enquanto outros nós permaneceram ociosos, aguardando pela chegada de uma lista de vértices locais para serem visitados.

O tempo que cada nó permaneceu bloqueado durante a execução da operação da travessia foi uma métrica coletada para ajudar o entendimento entre a diferença de desempenho observada entre as estratégias de fragmentação baseada em Espalhamento e aquela baseada em minimizar arestas de corte (METIS). No Apêndice C é possível encontrar uma tabela que informa, para cada um dos cenários, o tempo que cada nó permaneceu bloqueado. O tempo de bloqueio do nó que permaneceu mais tempo

bloqueado em cada um dos cenários também foi coletado para a elaboração da Fig. 29, Fig. 30 e Fig. 31.

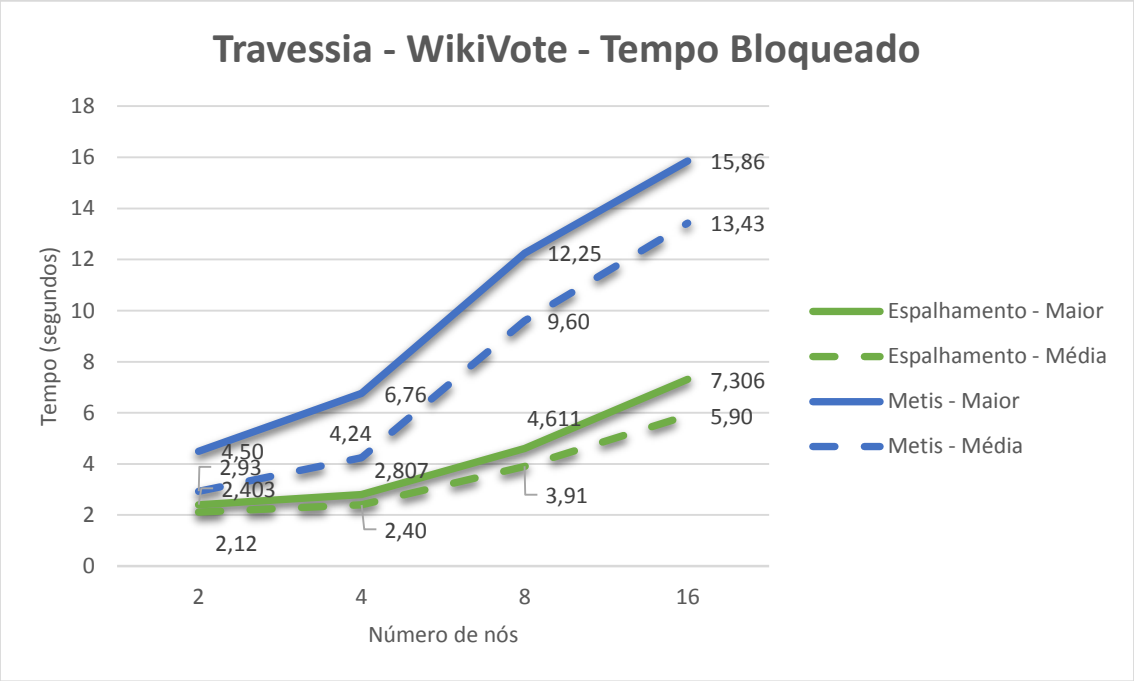


Fig. 29 – Maior tempo gasto por um nó em barreiras de sincronização BSP na travessia da base WikiVote

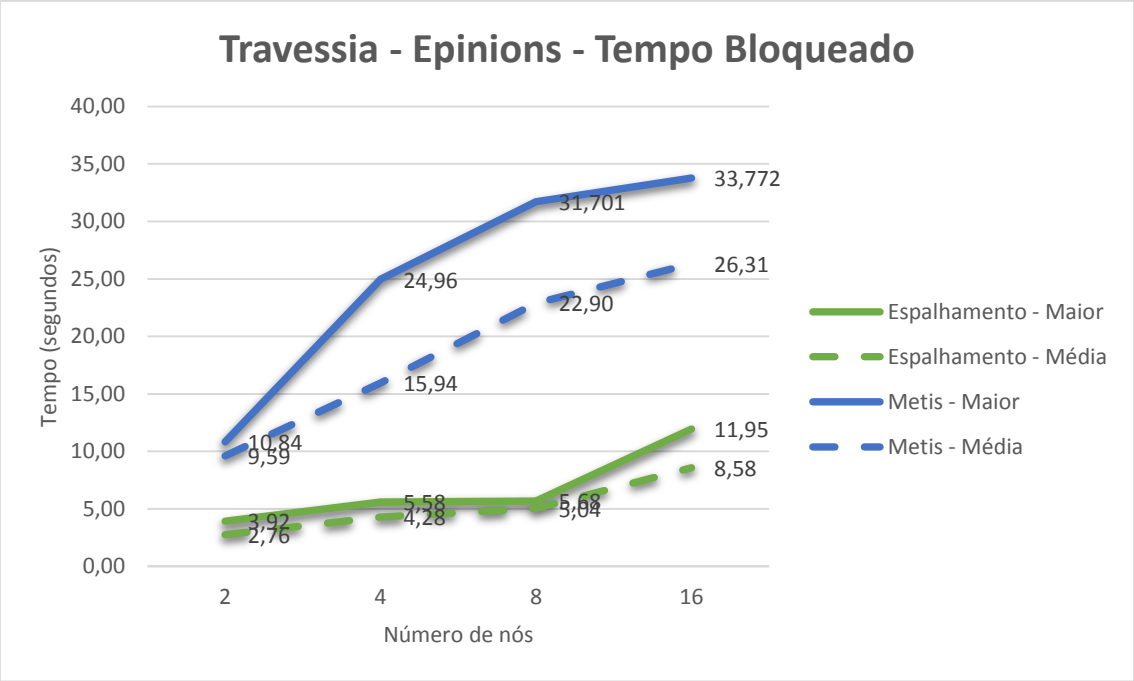


Fig. 30 - Maior tempo gasto por um nó em barreiras de sincronização BSP na travessia da base Epinions

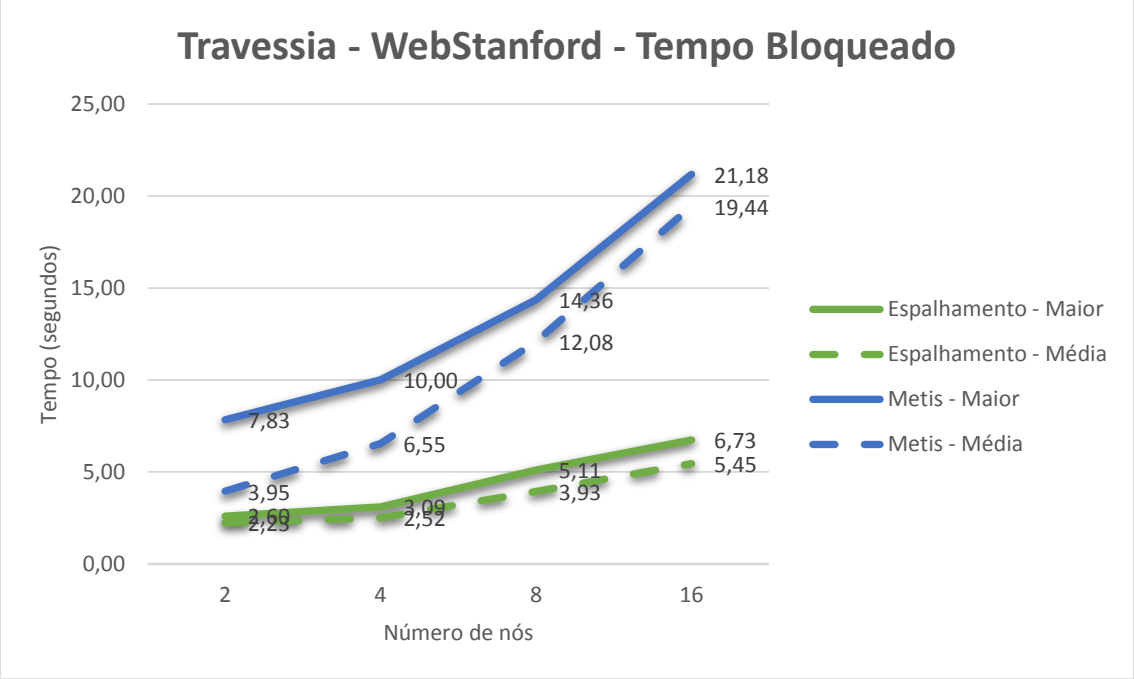


Fig. 31 - Maior tempo gasto por um nó em barreiras de sincronização BSP na travessia da base WebStanford

Os resultados apresentados para o maior tempo gasto por um nó em barreiras de sincronização BSP, em combinação com aqueles demonstrados no Apêndice C demonstram uma relação direta entre o tempo de execução total das consultas e a divisão da carga da travessia entre os nós envolvidos. A estratégia de fragmentação METIS, ao minimizar o número de arestas de corte entre os fragmentos envolvidos, acabou gerando um desbalanceamento de carga na execução da travessia. A fragmentação por Espalhamento, por outro lado, apresentou um desempenho mais uniforme com relação à divisão da carga de trabalho entre os nós envolvidos. O modelo de computação BSP, amplamente utilizado em arcabouços de processamento paralelo como o Pregel (MALEWICZ *et al.*, 2010), não foi capaz de tirar vantagem de uma maior localidade oferecida por técnicas de fragmentação mais sofisticadas como o METIS.

4.5 Caminhos Mínimos

Este experimento tem como objetivo retornar os caminhos mínimos existentes entre o vértice de origem v_s e o de destino v_d. Esta consulta foi executada através do algoritmo de caminhos mínimos baseado no modelo de computação BSP implementado no Graphene e descrito na seção 3.2.2.

Para a base de dados do WikiVote, os vértices de origem e destino escolhidos foram os vértices com identificadores 1000 e 3000. Para o Epinions a busca foi realizada entre os vértices 1677 e 1719. No WebStanford a origem escolhida foi o vértice de identificador 74238 e o destino foi o vértice 48185.

Para o WikiVote foram encontrados 33 caminhos mínimos entre a origem e o destino. Nos caminhos mínimos encontrados, o vértice de origem estava a uma distância de 3 vértices do destino. No Epinions foram encontrados 38 caminhos mínimos a uma distância de 2 vértices. No WebStanford apenas um caminho foi encontrado a 1 vértice de distância. Os resultados obtidos estão representados na Fig. 32, Fig. 33 e Fig. 34.

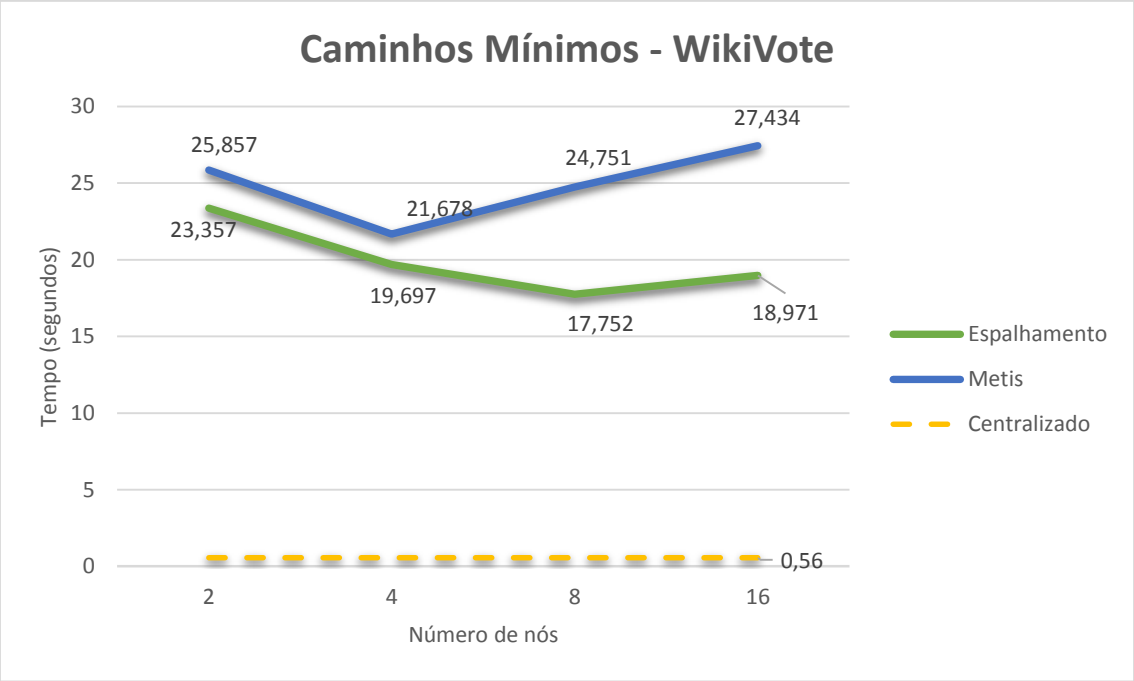


Fig. 32 - Execução da consulta de caminhos mínimos na base de dados do WikiVote

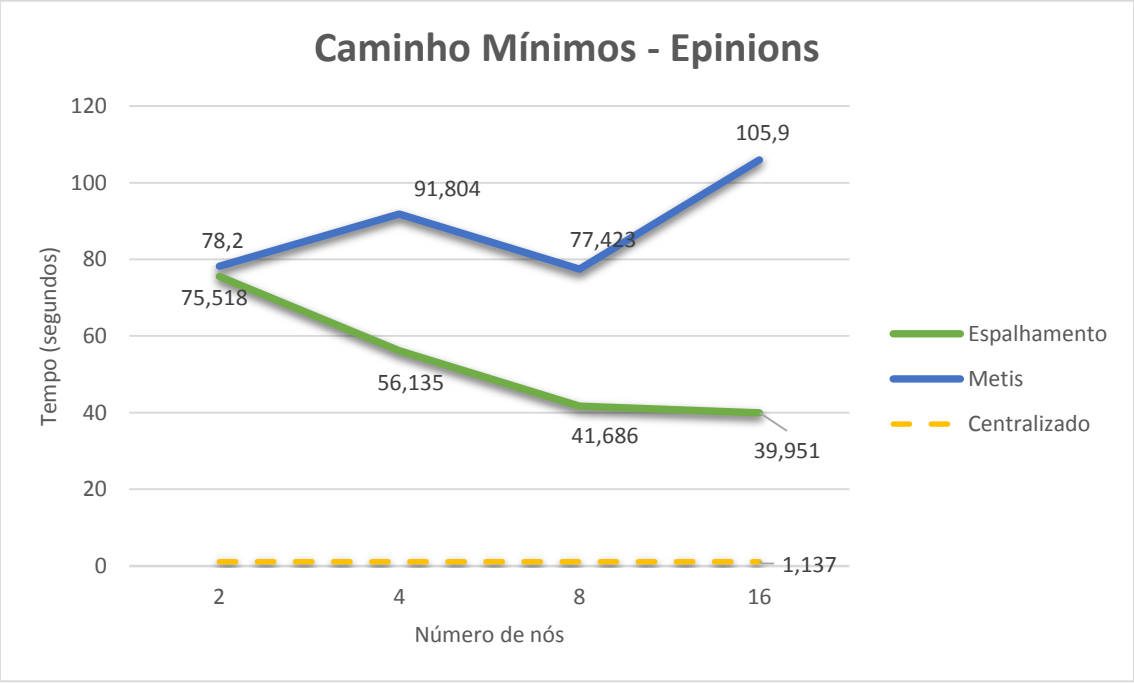


Fig. 33 - Execução da consulta de caminhos mínimos na base de dados do Epinions

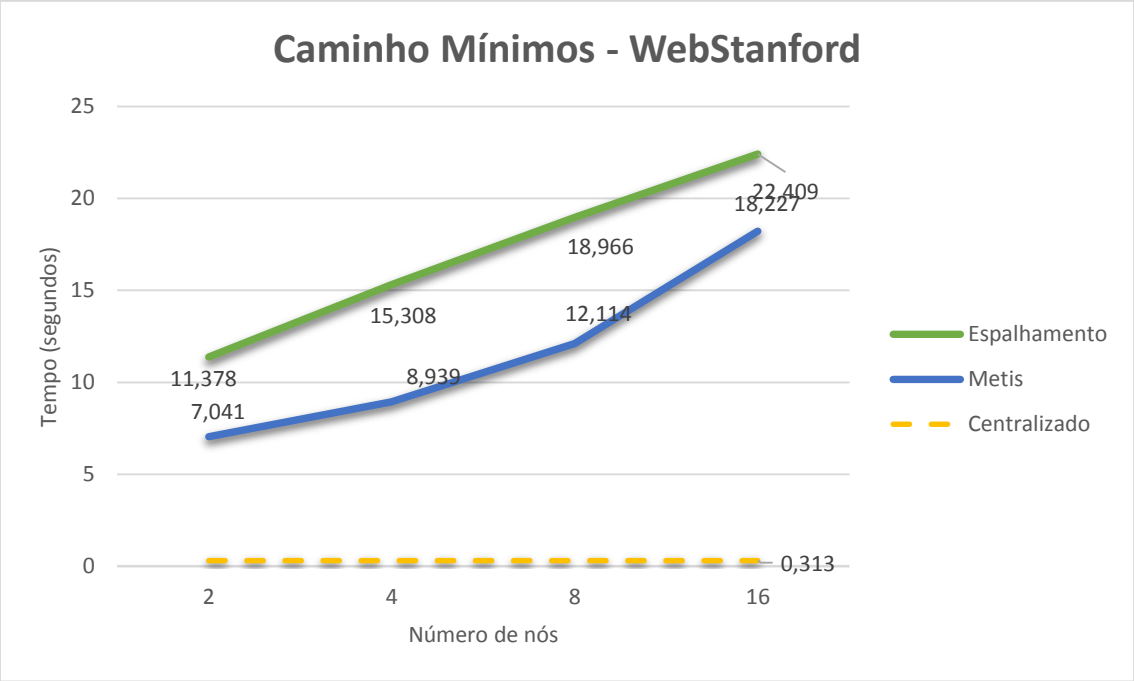


Fig. 34 - Execução da consulta de caminhos mínimos na base de dados do WebStanford

Para as consultas executadas nas bases WikiVote e Epinions é possível verificar uma melhora no tempo de execução para a fragmentação de Espalhamento até certo ponto (8 nós). O algoritmo de caminhos mínimos baseado no modelo BSP favoreceu a fragmentação de Espalhamento da mesma forma que ocorreu com a consulta da travessia descrita em 4.4. O tempo de bloqueio do nó que permaneceu mais tempo bloqueado em cada um dos cenários foi coletado e está representado na Fig. 35, Fig. 36 e Fig. 37.

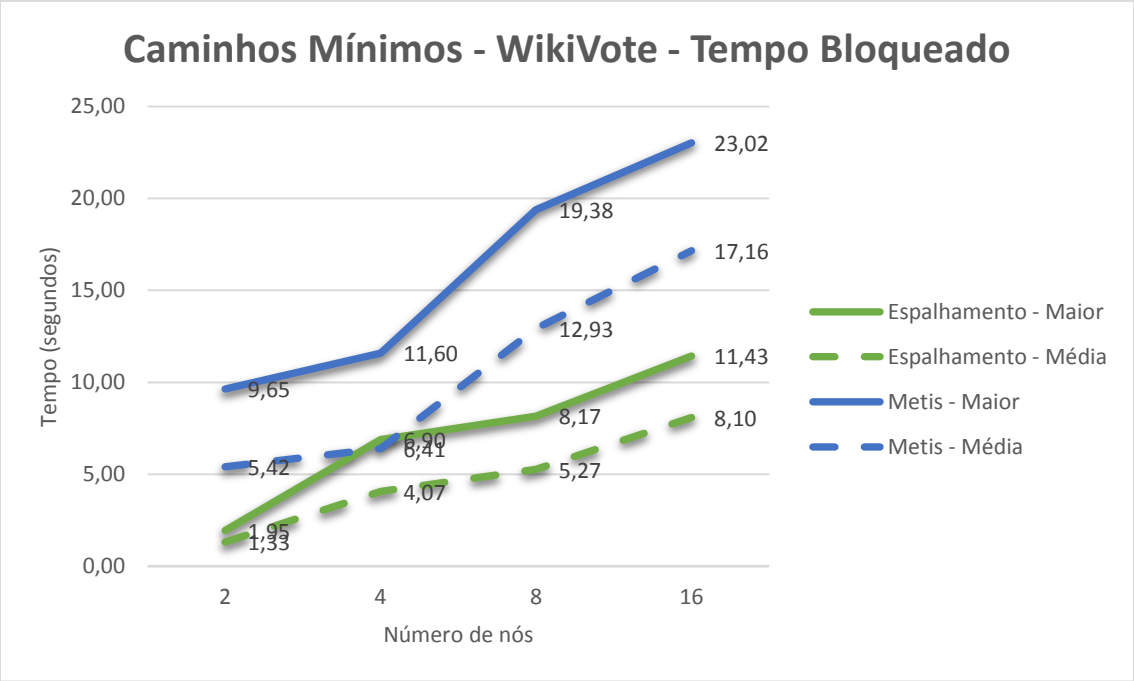


Fig. 35 - Maior tempo gasto por um nó em barreiras de sincronização BSP na consulta de caminhos mínimos da base WikiVote

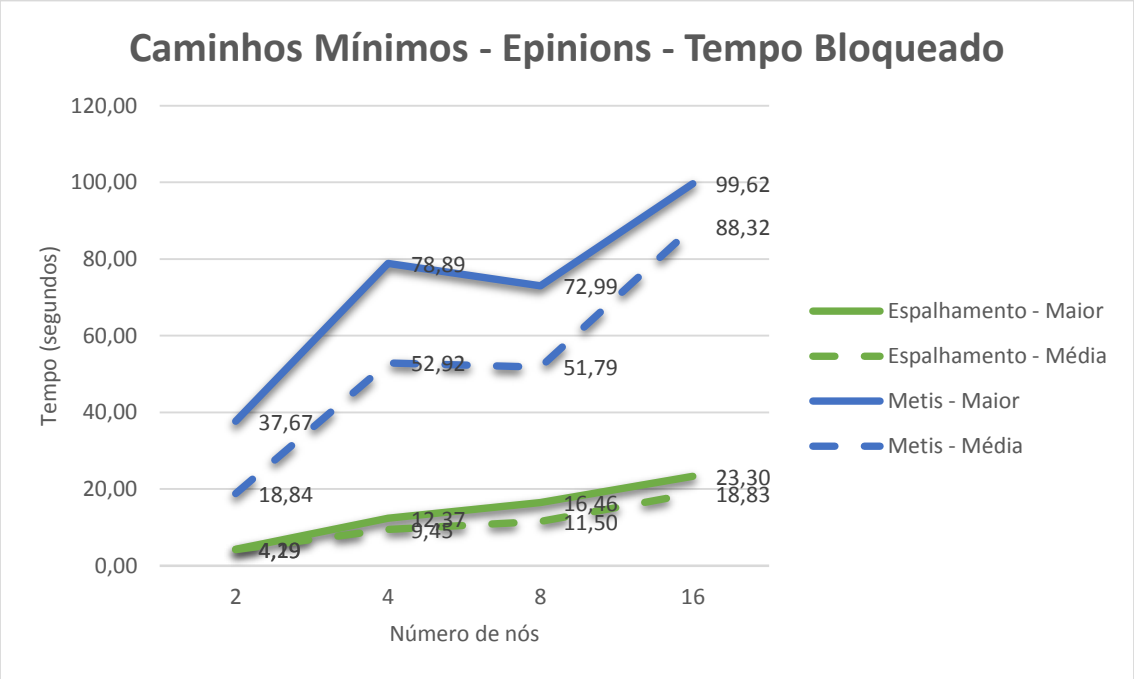


Fig. 36 - Maior tempo gasto por um nó em barreiras de sincronização BSP na consulta de caminhos mínimos da base Epinions

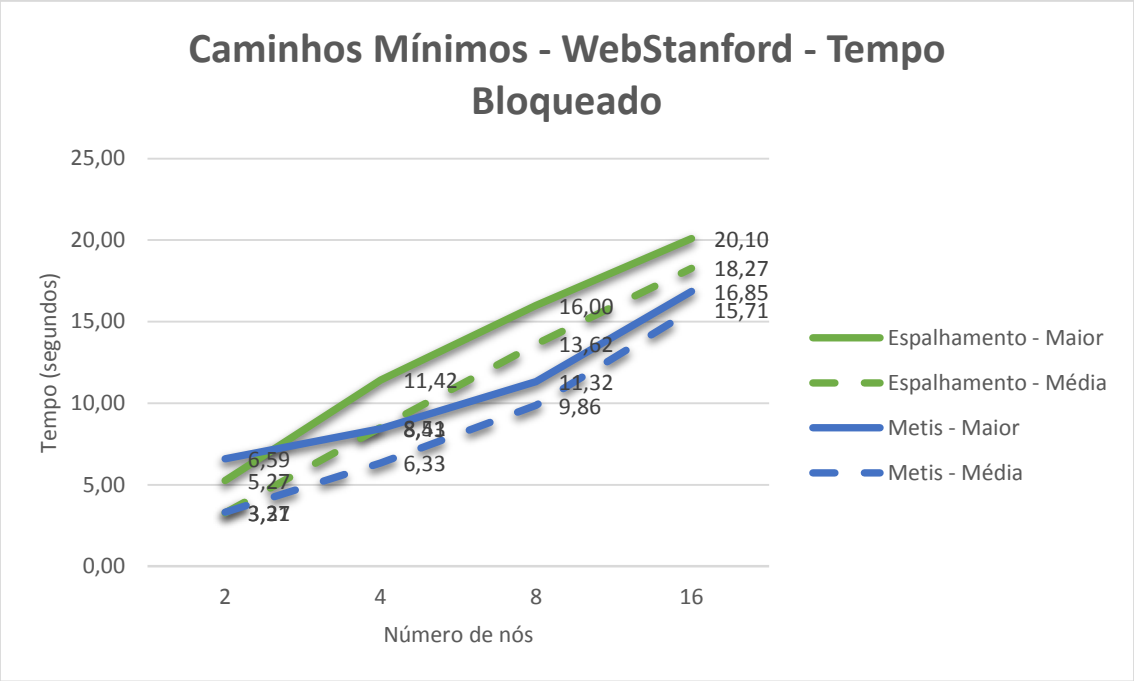


Fig. 37 - Maior tempo gasto por um nó em barreiras de sincronização BSP na consulta de caminhos mínimos da base WebStanford

A consulta executada na base de dados do WebStanford apresentou um desempenho ligeiramente melhor com estratégia de fragmentação METIS do que com a de Espalhamento. O METIS apresentou um melhor desempenho neste caso porque todos os vértices do caminho mínimo encontrado foram alocados no mesmo fragmento e puderam ser localizados na primeira iteração do algoritmo.

4.6 Múltiplos Usuários

Este experimento simulou a utilização do Graphene por múltiplos usuários executando simultaneamente operações sobre a base de dados. Neste experimento, a quantidade de nós coordenadores locais foi fixada em quatro nós e o número de clientes simulando operações de leitura foi aumentado gradativamente.

As operações de leitura simuladas representam operações comuns realizadas em banco de dados de grafos. Neste cenário, cada cliente simulado executa três operações:

uma travessia, uma consulta N-saltos e uma consulta para recuperar uma coleção de vértices baseado no valor de propriedades. As consultas submetidas são semelhantes àquelas descritas nas seções 4.4, 4.2 e 4.3, respectivamente, mudando apenas os vértices de origem escolhidos.

Um usuário no sistema sempre submete as três consultas, mas apenas uma consulta é submetida por vez por cada usuário. A ordem em que as consultas são submetidas por cada usuário é aleatória. Um intervalo aleatório uniformemente distribuído entre 100 ms e 1000 ms existe entre a submissão de consultas de um mesmo usuário. Todos os usuários simulados no sistema começam a submeter as suas consultas no mesmo instante. O tempo apresentado nos resultados corresponde àquele decorrido entre o início da submissão das consultas pelos usuários e o momento em que todas as consultas de todos os usuários foram atendidas. Os resultados obtidos estão representados na Fig. 38, Fig. 39 e Fig. 40.

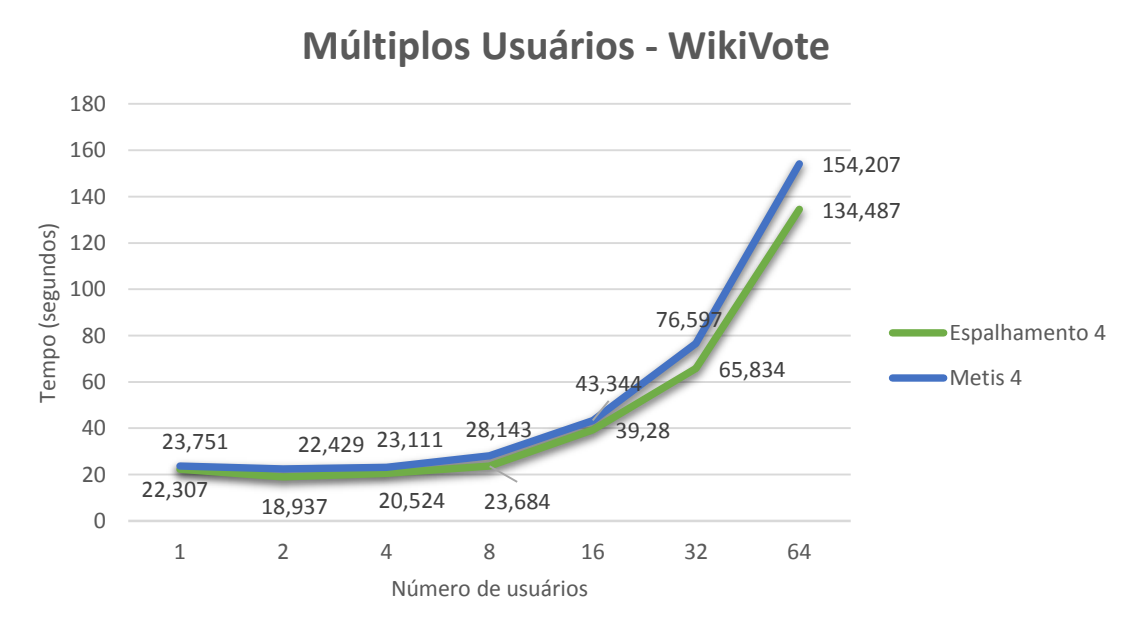


Fig. 38 – Tempo total necessário para atender múltiplos usuários submetendo consultas ao Graphene para a base de dados do WikiVote

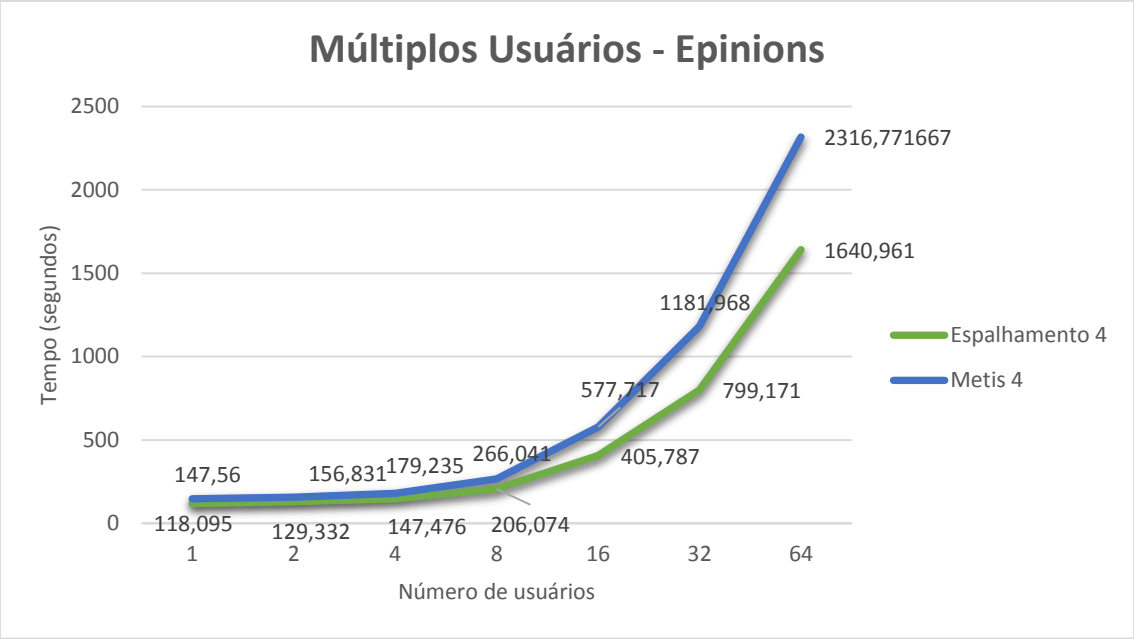


Fig. 39 - Tempo total necessário para atender múltiplos usuários submetendo consultas ao Graphene para a base de dados do Epinions

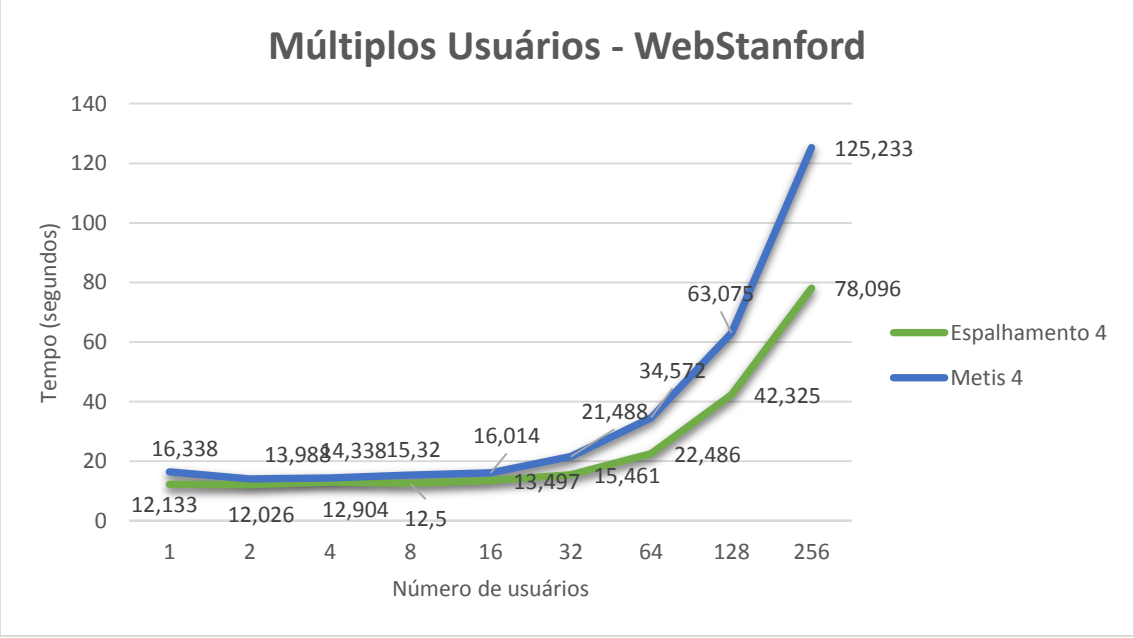


Fig. 40 - Tempo total necessário para atender múltiplos usuários submetendo consultas ao Graphene para a base de dados do WebStanford

Verificamos que a operação de travessia foi aquela que mais contribuiu para o tempo total da execução em todos os cenários por ser a operação mais custosa. O WebStanford apresentou um melhor desempenho comparado às outras duas bases porque a consulta de travessia realizada pelos usuários sobre esta base foi capaz de podar muitos caminhos na etapa inicial da busca. Esta característica contribuiu para o rápido processamento da travessia na base WebStanford e, como consequência, o sistema foi capaz de comportar um número maior de usuários submetendo consultas simultaneamente sem apresentar grande degradação de desempenho.

4.7 Agrupamento de Bases de Dados Preexistentes

Este experimento utilizou o Graphene para armazenar as três bases de dados: WikiVote, Epinions e WebStanford de forma que cada nó do *cluster* ficou responsável por gerenciar uma das bases integralmente. Este experimento utilizou cinco nós de forma que três nós foram alocados como coordenadores locais para gerenciar as bases citadas, um nó foi alocado como coordenador central e o último nó foi utilizado para simular as consultas submetidas a partir de uma aplicação cliente. As três bases de dados formaram um banco de dados distribuído gerenciado pelo Graphene neste experimento e não existia nenhuma aresta conectando vértices originalmente pertencentes a diferentes bases de dados.

As mesmas consultas N-saltos e aquelas baseadas em recuperação de elementos do grafo a partir de propriedades descritas nas seções 4.2 e 4.3 foram submetidas ao Graphene neste cenário. As mesmas consultas foram submetidas a bases de dados centralizadas e gerenciadas pelo Neo4j para fins de comparação. A diferença entre o tempo de execução das consultas deste cenário no Graphene e no Neo4j foram pequenas e estão representadas na Fig. 41 e Fig. 42 abaixo.

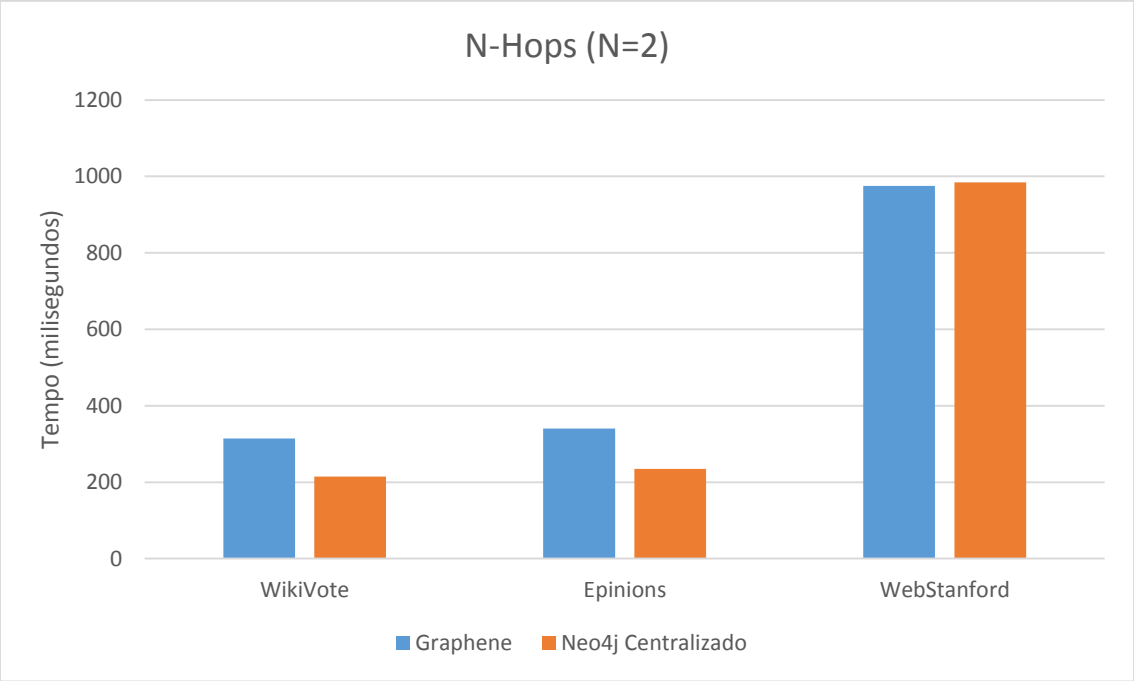


Fig. 41 – Comparação entre o tempo de execução no Graphene e no Neo4j centralizado para a consulta N-Hops.

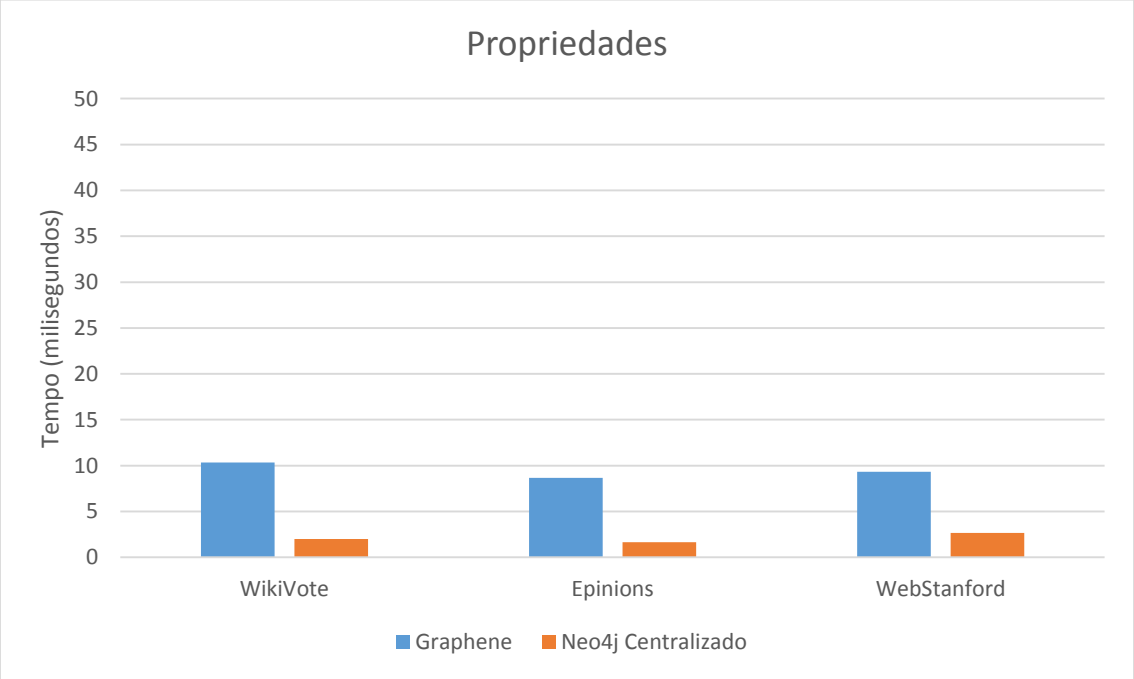


Fig. 42 – Comparação entre o tempo de execução no Graphene e no Neo4j centralizado para a consulta de recuperação de elementos baseado em propriedades.

4.8 Análise dos Resultados dos Experimentos Executados

As principais conclusões obtidas a partir dos experimentos executados são que a solução centralizada apresentou um desempenho superior aquele apresentado pela solução distribuída. Este fato ocorreu principalmente devido aos altos custos de comunicação entre os nós envolvidos.

Não foi possível realizar testes com bases de dados muito grandes devido a algumas restrições no *cluster* onde os experimentos foram executados. As bases de dados utilizadas são pequenas e cabem totalmente em memória. Esta característica favoreceu o ambiente centralizado quando comparado ao distribuído, contribuindo para um menor tempo de resposta as consultas submetidas.

É possível verificar, a partir dos resultados dos experimentos de travessia e caminhos mínimos, que a abordagem BSP amplamente utilizada pelos arcabouços de processamento paralelo não consegue tirar proveito de estratégias de fragmentação mais inteligentes como o METIS. Os resultados apresentaram um desempenho superior da estratégia de Espalhamento quando comparada a estratégia METIS. A investigação de outros métodos capazes de tirar maior proveito da redução do número de arestas de corte fornecida por algoritmos de fragmentação inteligentes como o METIS corresponde a um importante trabalho futuro.

Capítulo 5 – Conclusões e Trabalhos Futuros

Esta dissertação descreveu o Graphene, um protótipo de *middleware* para a criação e utilização de bancos de dados de grafos distribuídos. Características de sua arquitetura foram descritas, bem como detalhes dos algoritmos utilizados para execução de operações sobre grafos. Procurou-se aproveitar o ambiente distribuído para implementação destas operações. As principais consultas de banco de dados de grafos foram submetidas ao Graphene e os resultados mostrados e avaliados.

A utilização do Graphene apresenta algumas vantagens quando comparada a outras soluções de SGBDG distribuídos como o Titan (TITAN, 2014) e o OrientDB (ORIENTDB, 2014). O Titan é um SGBDG distribuído que utiliza outras formas de armazenamento para a representação da estrutura do grafo. Os dados que compõem o grafo são atualmente acomodados em outros modelos *NoSQL* como modelos baseados em chave e valor ou família de colunas. O Titan também não permite um maior controle sobre a estratégia de distribuição dos dados. O Graphene, por outro lado, permite que o usuário consiga controlar a distribuição dos dados entre os nós computacionais presentes e utiliza preferencialmente SGBDG como solução de armazenamento em todos os nós para tirar proveito do mecanismo eficiente de representação de adjacências fornecido por estas ferramentas. Apesar disso o Graphene pode ser utilizado com outros SGBD compatíveis com o *Blueprints*.

O Graphene pôde ser utilizado com sucesso para execução das principais operações em bases de dados de grafos da mesma forma que as soluções de SGBDG tradicionais. Ele também demonstrou um bom desempenho nos experimentos que simulavam sua utilização por múltiplos usuários executando operações simultaneamente. O desempenho observado ao utilizar o Graphene comparado àquele observado ao utilizar soluções de SGBDG nativos e centralizados foi menor devido aos custos de comunicação envolvidos. As bases de dados utilizadas nos testes foram pequenas devido a restrições quanto ao espaço disponível no ambiente em que os testes foram executados, limitando a execução de experimentos para bases de dados maiores. Este aspecto favoreceu o ambiente centralizado em comparação a distribuição realizada pelo Graphene.

O Graphene é uma camada de software capaz de gerenciar grandes bases de dados que, devido ao seu tamanho, não são capazes de serem armazenados ou explorados por um SGBDG centralizado. Eventualmente as bases de dados de grafos gerenciadas por SGBDG podem crescer e ultrapassar os limites computacionais de um único nó. A possibilidade de distribuição da base de dados de grafos em múltiplos nós oferecida pelo Graphene fornece uma solução para o armazenamento e processamento de operações OLTP em grandes bases de dados.

Os experimentos executados nesta dissertação também mostraram que o modelo de computação BSP pode não ser apropriado para tirar proveito de técnicas de fragmentação mais sofisticadas como o METIS (KARYPIS & KUMAR, 1998). Este modelo foi mais favorável à técnica de fragmentação de Espalhamento. Um importante trabalho futuro é pesquisar e analisar outras técnicas e algoritmos distribuídos para a execução das travessias de forma a conseguir tirar melhor proveito de fragmentos de melhor qualidade e com menor número de arestas de corte.

Trabalhos futuros incluem também executar experimentos com bases de dados maiores, que não podem ser armazenadas de forma centralizada, proporcionar um maior suporte à atualização dos dados no Graphene e o suporte a transações distribuídas. Atualmente, o Graphene é voltado para a execução de operações que envolvem apenas leituras de dados. A inserção e atualização de novos elementos na base é suportada e pode ser feito a qualquer momento mesmo durante a execução de uma consulta, mas neste caso os resultados podem ser imprevisíveis. Além disso, a política de *cache* implementada no coordenador central e coordenadores locais invalida as entradas da memória quando uma alteração é detectada na estrutura do grafo, o que força novos acessos ao disco ao buscar um determinado elemento.

Referências:

ABOU-RJEILI, A., KARYPIS, G., 2006. "Multilevel Algorithms for Partitioning Power-law Graphs". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society. 2006. pp. 124.

BADER, D.A., BERRY, J., AMOS-BINKS, A., et al., 2009, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation". In: *Georgia Institute of Technology, Tech. Rep.*

BADER, D.A., MADDURI, K., 2008. "SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE. Abril 2008. pp. 1–12.

BARGUÑÓ, L., MUNTÉS-MULERO, V., DOMINGUEZ-SAL, D., et al., 2011. "ParallelGDB". In: *Proceedings of the 15th Symposium on International Database Engineering & Applications - IDEAS '11*. New York, New York, USA: ACM Press. 2011. pp. 162.

BARRETT, B.W., BERRY, J.W., MURPHY, R.C., et al., 2009. "Implementing a portable Multi-threaded Graph Library: The MTGL on Qthreads". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. Maio 2009. pp. 1–8.

BLUEPRINTS, 2014. Disponível em: <https://github.com/tinkerpop/blueprints/wiki>.

BU, Y., HOWE, B., BALAZINSKA, M., et al., 2012, "The HaLoop approach to large-scale iterative data analysis". In: *The VLDB Journal*. v. 21, pp. 169–190.

BULUC, A., GILBERT, J.R., 2011, "The Combinatorial BLAS: design, implementation, and applications". In: *International Journal of High Performance Computing Applications*. v. 25, pp. 496–509.

BULUÇ, A., MADDURI, K., 2011. "Parallel breadth-first search on distributed memory systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*. Distributed, Parallel, and Cluster Computing; Mathematical Software; Performance. New York, New York, USA: ACM Press. 22 Abril 2011. pp. 1.

BULUC, A., MEYERHENKE, H., SAFRO, I., et al., 2013, "Recent Advances in Graph Partitioning". In: *CoRR*. v. abs/1311.3, pp. 1–36.

CAPPELLO, F., CARON, E., DAYDE, M., et al., 2005. "Grid'5000: a large scale and highly reconfigurable grid experimental testbed". In: *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.* IEEE. 2005. pp. 8 pp.

CHANG, F., DEAN, J., GHEMAWAT, S., et al., 2008, "Bigtable". In: *ACM Transactions on Computer Systems*. v. 26, pp. 1–26.

CHEN, R., WENG, X., HE, B., et al., 2010. "Large graph processing in the cloud". In: *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*. New York, New York, USA: ACM Press. 2010. pp. 1123.

CHEVALIER, C., PELLEGRINI, F., 2009, "PT-Scotch: A tool for efficient parallel graph ordering". In: *Parallel Computing*. v. 34, pp. 6–8.

CHING, A., KUNZ, C., 2011, "Giraph: Large-scale graph processing infrastructure on Hadoop". In: *Hadoop Summit*. v. 6, pp. 2011.

CIGLAN, M., AVERBUCH, A., HLUCHY, L., 2012. "Benchmarking Traversal Operations over Graph Databases". In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE. Abril 2012. pp. 186–189.

CODD, E.F., 1970, "A relational model of data for large shared data banks". In: *Communications of the ACM*. v. 13, pp. 377–387.

DEAN, J., GHEMAWAT, S., 2008, "MapReduce". In: DANIEL, L Purich (ed.), *Communications of the ACM*. v. 51, pp. 107.

DECANDIA, G., HASTORUN, D., JAMPANI, M., et al., 2007. "Dynamo". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07*. New York, New York, USA: ACM Press. 2007. pp. 205.

DIEKMANN, R., PREIS, R., SCHLIMBACH, F., et al., 2000, "Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM". In: *Parallel Computing*. v. 26, pp. 1555–1581.

DOMINGUEZ-SAL, D., MARTINEZ-BAZAN, N., MUNTES-MULERO, V., et al., 2011. "A discussion on the design of graph database benchmarks". In: *Performance Evaluation, Measurement and Characterization of Complex Systems*. Springer. pp. 25–40.

EDIGER, D., JIANG, K., RIEDY, E.J., et al., 2013, "GraphCT: Multithreaded Algorithms for Massive Graph Analysis". In: *IEEE Transactions on Parallel and Distributed Systems*. v. 24, pp. 2220–2229.

ELNIKETY, E., ELSAYED, T., RAMADAN, H.E., 2011, "iHadoop: Asynchronous Iterations for MapReduce". In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. pp. 81–90.

FAUNUS, 2014. Disponível em: <http://thinkaurelius.github.io/faunus/>.

FIDUCCIA, C.M., MATTHEYSES, R.M., 1982. "A Linear-time Heuristic for Improving Network Partitions". In: *Proceedings of the 19th Design Automation Conference*. Piscataway, NJ, USA: IEEE Press. 1982. pp. 175–181.

FREEMAN, J., 1991. *Parallel algorithms for depth-first search*. Philadelphia, PA.

GANSNER, E.R., NORTH, S.C., 2000, "An open graph visualization system and its applications to software engineering". In: *Software: Practice and Experience*. v. 30, pp. 1203–1233.

GEIST, A., GROPP, W., HUSS-LEDERMAN, S., et al., 1996. "MPI-2: Extending the message-passing interface". In: *Euro-Par'96 Parallel Processing*. Springer-Verlag. 1996. pp. 128–135.

GHEMAWAT, S., GOBIOFF, H., LEUNG, S.-T., 2003, "The Google file system". In: *ACM SIGOPS Operating Systems Review*. v. 37, pp. 29.

GILBERT, S., LYNCH, N., 2002, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News*. v. 33, pp. 51.

GONZALEZ, J.E.J., LOW, Y., GU, H., et al., 2012. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association. 2012. pp. 17–30.

GREGOR, D., LUMSDAINE, A., 2005. "The Parallel BGL: A Generic Library for Distributed Graph Computations". In: *Parallel Object-Oriented Scientific Computing (POOSC)*. 2005. pp. 1–18.

GÜTING, R.H., 1994. "GraphDB: Modeling and Querying Graphs in Databases". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 1994. pp. 297–308.

HAGBERG, A.A., SCHULT, D.A., SWART, P.J., 2008. "Exploring network structure, dynamics, and function using {NetworkX}". In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA: s.n. 2008. pp. 11–15.

HAGEN, L., KAHNG, A.B., 1992, "New spectral methods for ratio cut partitioning and clustering". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. v. 11, pp. 1074–1085.

HENDRICKSON, B., LELAND, R., 1995. "A multilevel algorithm for partitioning graphs". In: *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. New York, New York, USA: ACM Press. 1995. pp. 28–es.

HUNT, P., KONAR, M., JUNQUEIRA, F.P., et al., 2010. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association. 2010. pp. 11.

INFINITEGRAPH, 2014. Disponível em: <http://www.objectivity.com/infinitegraph>.

KANG, U., TSOURAKAKIS, C.E., FALOUTSOS, C., 2009. "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations". In: *2009 Ninth IEEE International Conference on Data Mining*. IEEE. Dezembro 2009. pp. 229–238.

KARYPIS, G., KUMAR, V., 1995. "Multilevel Graph Partitioning Schemes". In: *Proc. 24th Intern. Conf. Par. Proc., III*. CRC Press. 1995. pp. 113–122.

KARYPIS, G., KUMAR, V., 1996. "Parallel multilevel k-way partitioning scheme for irregular graphs". In: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. New York, New York, USA: ACM Press. 1996. pp. 35–es.

KARYPIS, G., KUMAR, V., 1998, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing*. v. 20, pp. 359–392.

KARYPIS, G., KUMAR, V., 1999. "Multilevel k -way hypergraph partitioning". In: *Proceedings of the 36th ACM/IEEE conference on Design automation conference - DAC '99*. New York, New York, USA: ACM Press. 1999. pp. 343–348.

KERNIGHAN, B.W., LIN, S., 1970, "An Efficient Heuristic Procedure for Partitioning Graphs". In: *The Bell system technical journal*. v. 49, pp. 291–307.

KHAYYAT, Z., AWARA, K., ALONAZI, A., et al., 2013. "Mizan". In: *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. New York, New York, USA: ACM Press. 2013. pp. 169.

LANG, K., 2004. *Finding good nearly balanced cuts in power law graphs*. Acessado em: 25 Março 2014. Disponível em: <http://www.optimization-online.org/DB_FILE/2004/12/1023.pdf>.

LESKOVEC, J., LANG, K.J., DASGUPTA, A., et al., 2009, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters". In: *Internet Mathematics*. v. 6, pp. 29–123.

LIN, J., DYER, C., 2010, "Data-Intensive Text Processing with MapReduce". In: *Synthesis Lectures on Human Language Technologies*. v. 3, pp. 1–177.

LIN, J., SCHATZ, M., 2010, "Design Patterns for Efficient Graph Algorithms in MapReduce". In: *Genome*. pp. 78–85.

LOW, Y., GONZALEZ, J., KYROLA, A., et al., 2010, "GraphLab: A New Framework for Parallel Machine Learning". In: *The 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*. pp. 8–11.

LUGOWSKI, A., ALBER, D., BULUÇ, A., et al., 2012. "A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis". In: *Proceedings of the Twelfth SIAM International Conference on Data Mining (SDM12)*. 2012. pp. 930–941.

LUMSDAINE, A., GREGOR, D., HENDRICKSON, B., et al., 2007, "Challenges in Parallel Graph Processing.". In: *Parallel Processing Letters*. v. 17, pp. 5–20.

MALEWICZ, G., AUSTERN, M.H., BIK, A.J.., et al., 2010. "Pregel". In: *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*. New York, New York, USA: ACM Press. 2010. pp. 135.

MARTÍNEZ-BAZAN, N., MUNTÉS-MULERO, V., GÓMEZ-VILLAMOR, S., et al., 2007. "Dex: High-performance Exploration on Large Graphs for Information Retrieval". In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*. New York, NY, USA: ACM. 2007. pp. 573–582.

MONIRUZZAMAN, A.B.M., HOSSAIN, S.A., 2013, "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison". In: *International Journal of Database Theory and Application*. v. 6.

MUNTÉS-MULERO, V., MARTÍNEZ-BAZÁN, N., LARRIBA-PEY, J.-L., et al., 2010. "Graph Partitioning Strategies for Efficient BFS in Shared-nothing Parallel Systems". In: *Proceedings of the 2010 International Conference on Web-age Information Management*. Berlin, Heidelberg: Springer-Verlag. 2010. pp. 13–24.

NEO4J, 2014. Disponível em: <http://www.neo4j.org/>.

NEWMAN, M.E.J., GIRVAN, M., 2004, "Finding and evaluating community structure in networks". In: *Physical review E*. v. 69, pp. 026113.

ORIENTDB, 2014. Disponível em: <http://www.orientechnologies.com/orientdb/>.

OZSU, M.T., VALDURIEZ, P., 2011, *Principles of Distributed Database Systems*. 3 ed. Springer.

PRABHAKARAN, V., WU, M., WENG, X., et al., 2012. "Managing Large Graphs on Multi-cores with Graph Awareness". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association. 2012. pp. 4.

REIF, J.H., 1985, "Depth-first search is inherently sequential". In: *Information Processing Letters*. v. 20, pp. 229–234.

ROBINSON, I., WEBBER, J., EIFREM, E., 2013, *Graph Databases*. 1 ed., O'Reilly Media, Incorporated.

RODRIGUEZ, M.A., NEUBAUER, P., 2010, "The Graph Traversal Pattern". In: pp. 1–18.

RUSSELL, S.J., NORVIG, P., 2010, *Artificial Intelligence: A Modern Approach*. 3 ed. Prentice Hall. Prentice Hall series in artificial intelligence.

SADALAGE, P.J., FOWLER, M., 2012, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education.

SALIHOGLU, S., WIDOM, J., 2013. "GPS". In: *Proceedings of the 25th International Conference on Scientific and Statistical Database Management - SSDBM*. New York, New York, USA: ACM Press. 2013. pp. 1.

SHAO, B., WANG, H., LI, Y., 2013. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM. 2013. pp. 505–516.

SPARKSEE, 2014. Disponível em: <http://www.sparsity-technologies.com/>.

TITAN, 2014. Disponível em: <http://thinkaurelius.github.io/titan/>.

ULLMAN, J., YANNAKAKIS, M., 1990. "High-probability Parallel Transitive Closure Algorithms". In: *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM. 1990. pp. 200–209.

VALIANT, L.G., 1990, "A bridging model for parallel computation". In: *Communications of the ACM*. v. 33, pp. 103–111.

WALSHAW, C., CROSS, M., 2000, "Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm". In: *SIAM Journal on Scientific Computing*. v. 22, pp. 63–80.

XIN, R.S., GONZALEZ, J.E., FRANKLIN, M.J., et al., 2013. "GraphX". In: *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*. New York, New York, USA: ACM Press. 2013. pp. 1–6.

YAN, J., TAN, G., SUN, N., 2013, "GRE: A Graph Runtime Engine for Large-Scale Distributed Graph-Parallel Applications". In: *arXiv preprint arXiv:1310.5603*. pp. 12.

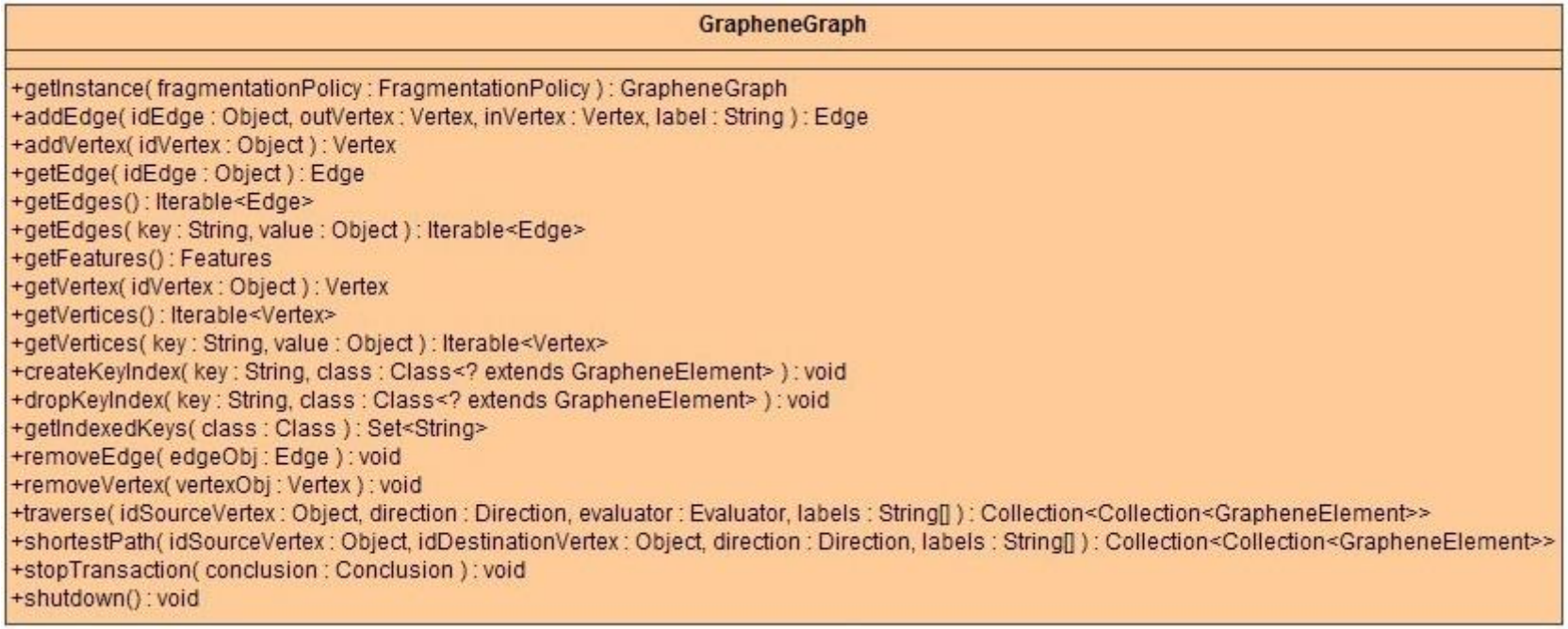
YANG, S., WANG, B., ZHAO, H., et al., 2009, "Efficient Dense Structure Mining Using MapReduce". In: *2009 IEEE International Conference on Data Mining Workshops*. pp. 332–337.

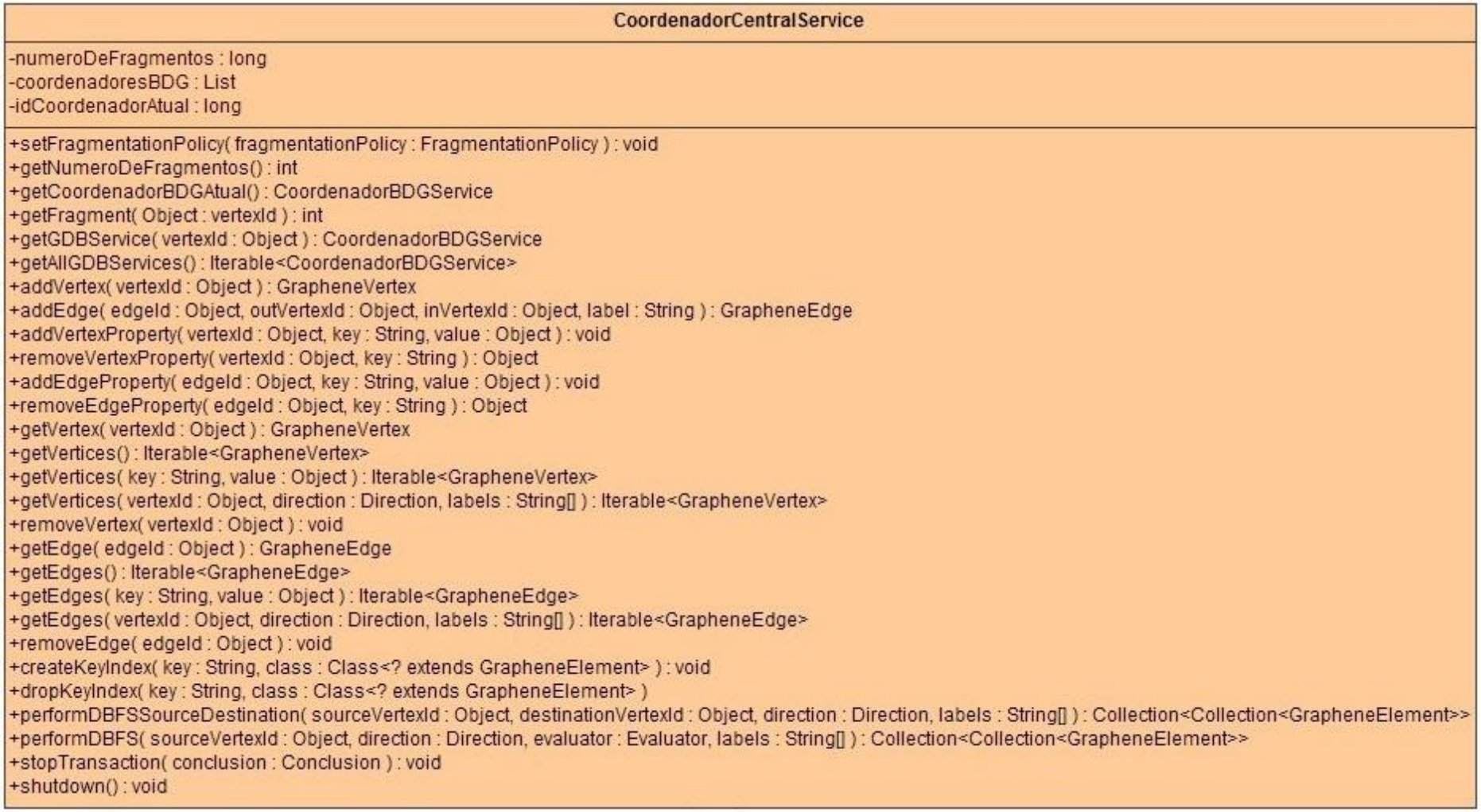
YOO, A, CHOW, E., HENDERSON, K., et al., 2005. "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L". In: *ACM/IEEE SC 2005 Conference (SC'05)*. IEEE. 2005. pp. 25–25.

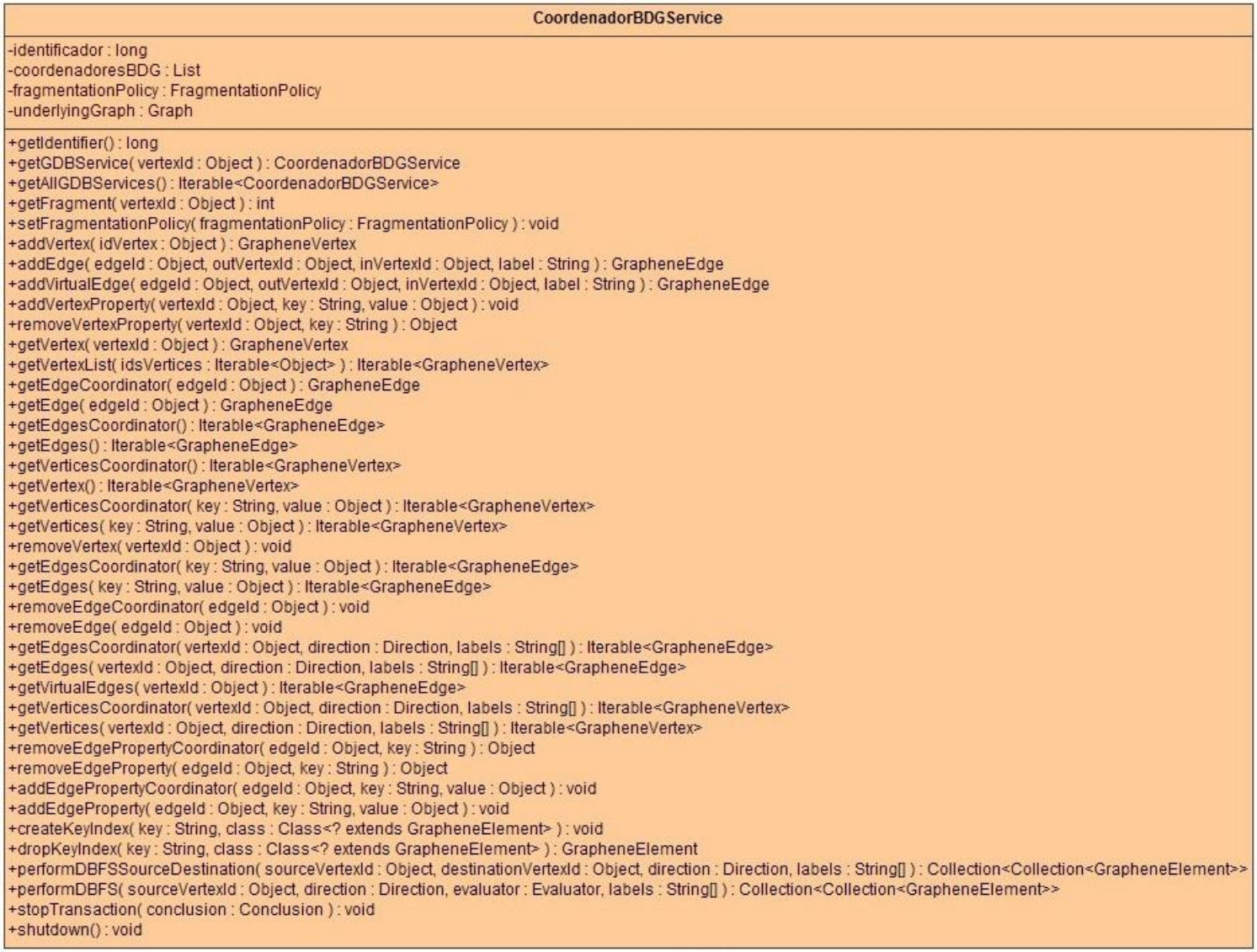
ZAHARIA, M., CHOWDHURY, M., DAS, T., et al., 2012. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. 2012. pp. 2.

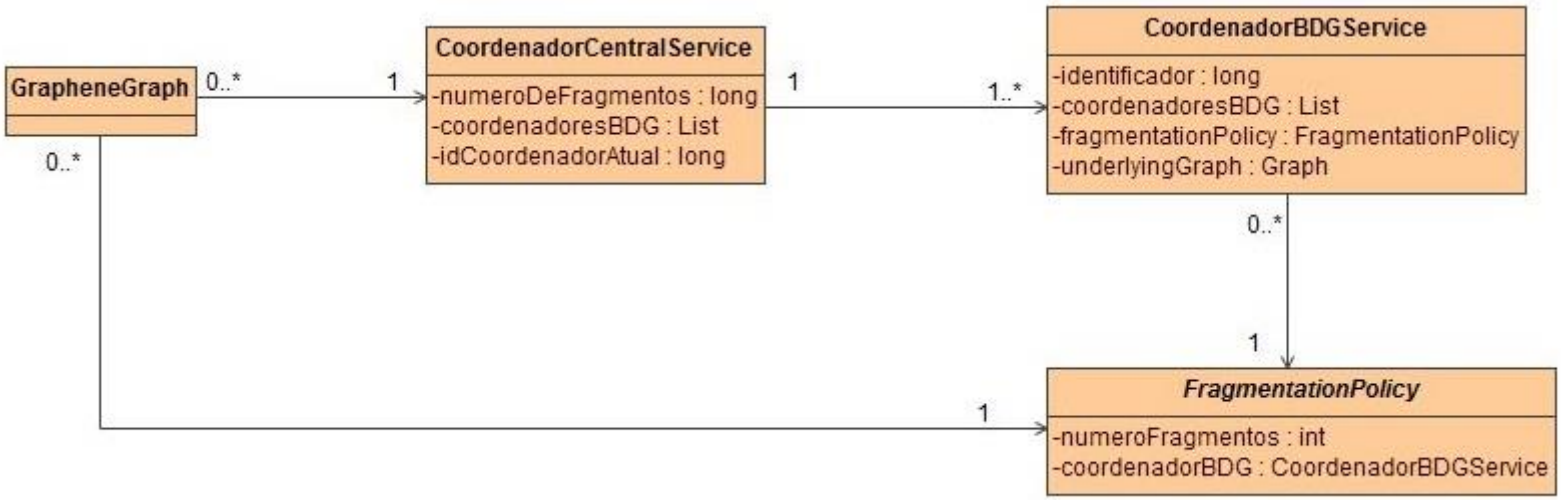
ZHANG, Y., GAO, Q., GAO, L., et al., 2012, "iMapReduce: A Distributed Computing Framework for Iterative Computation". In: *Journal of Grid Computing*. v. 10, pp. 47–68.

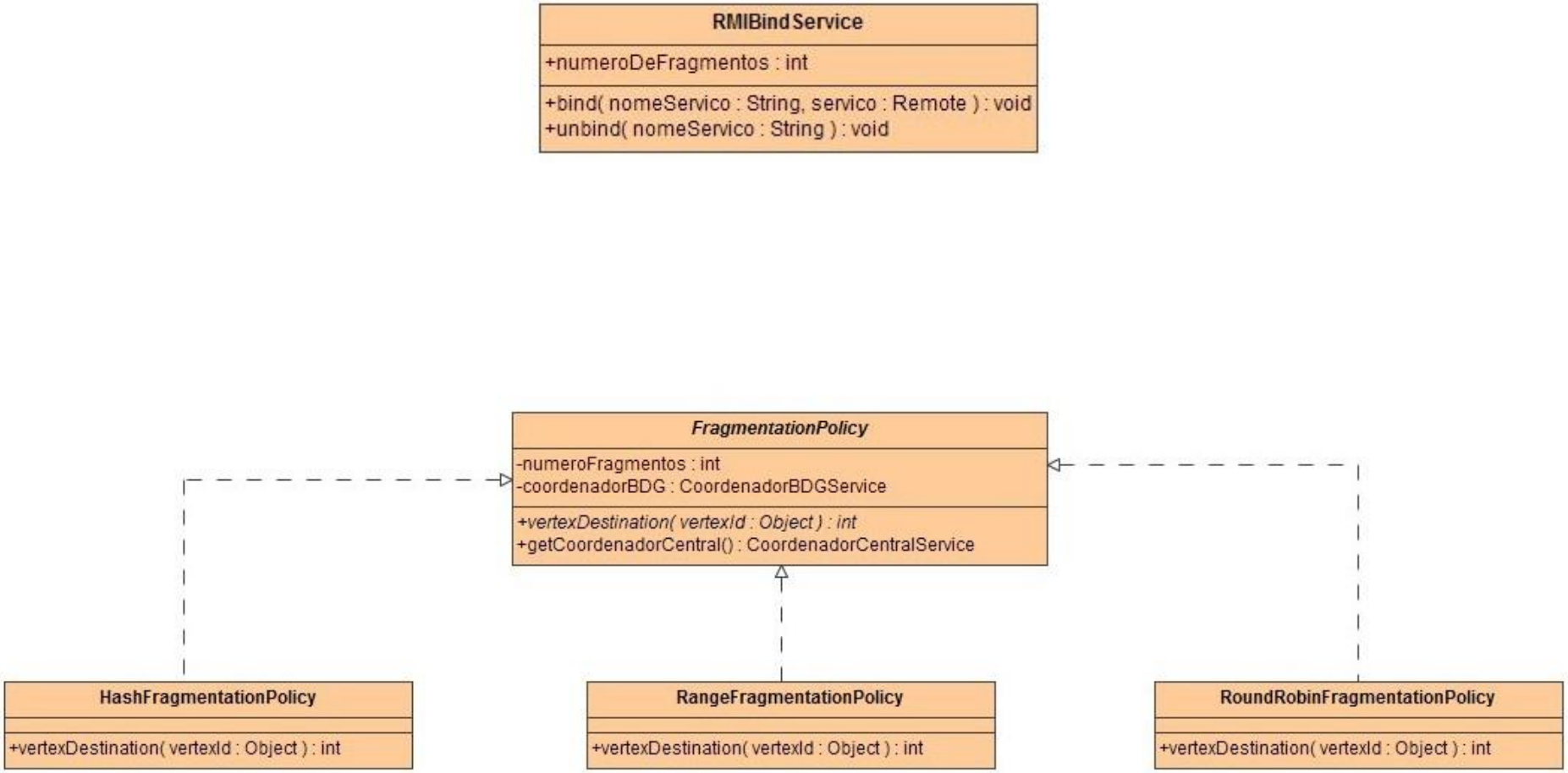
Apêndice A

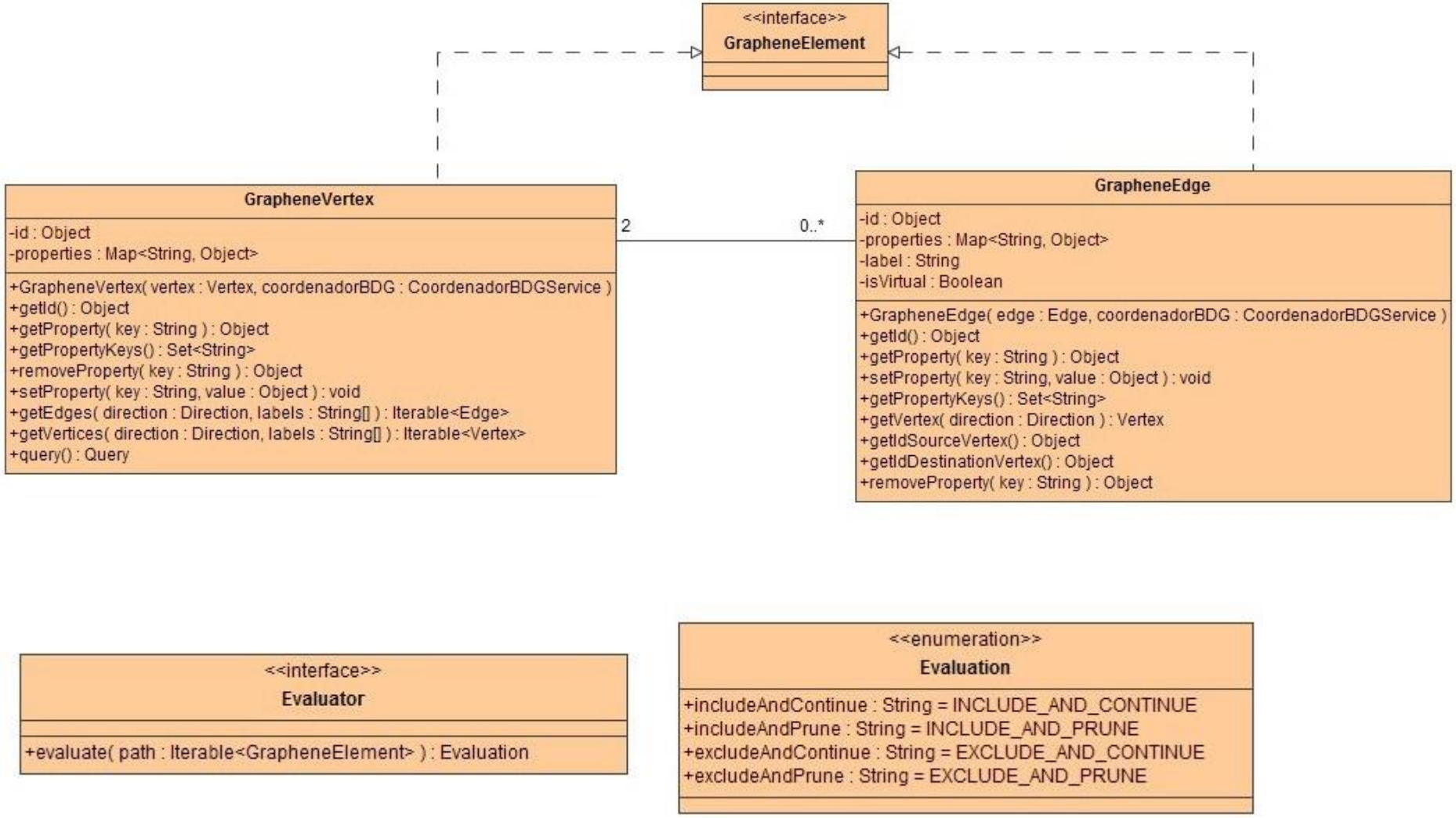

GrapheneGraph
+getInstance(fragmentationPolicy : FragmentationPolicy) : GrapheneGraph
+addEdge(idEdge : Object, outVertex : Vertex, inVertex : Vertex, label : String) : Edge
+addVertex(idVertex : Object) : Vertex
+getEdge(idEdge : Object) : Edge
+getEdges() : Iterable<Edge>
+getEdges(key : String, value : Object) : Iterable<Edge>
+getFeatures() : Features
+getVertex(idVertex : Object) : Vertex
+getVertices() : Iterable<Vertex>
+getVertices(key : String, value : Object) : Iterable<Vertex>
+createKeyIndex(key : String, class : Class<? extends GrapheneElement>) : void
+dropKeyIndex(key : String, class : Class<? extends GrapheneElement>) : void
+getIndexedKeys(class : Class) : Set<String>
+removeEdge(edgeObj : Edge) : void
+removeVertex(vertexObj : Vertex) : void
+traverse(idSourceVertex : Object, direction : Direction, evaluator : Evaluator, labels : String[]) : Collection<Collection<GrapheneElement>>
+shortestPath(idSourceVertex : Object, idDestinationVertex : Object, direction : Direction, labels : String[]) : Collection<Collection<GrapheneElement>>
+stopTransaction(conclusion : Conclusion) : void
+shutdown() : void

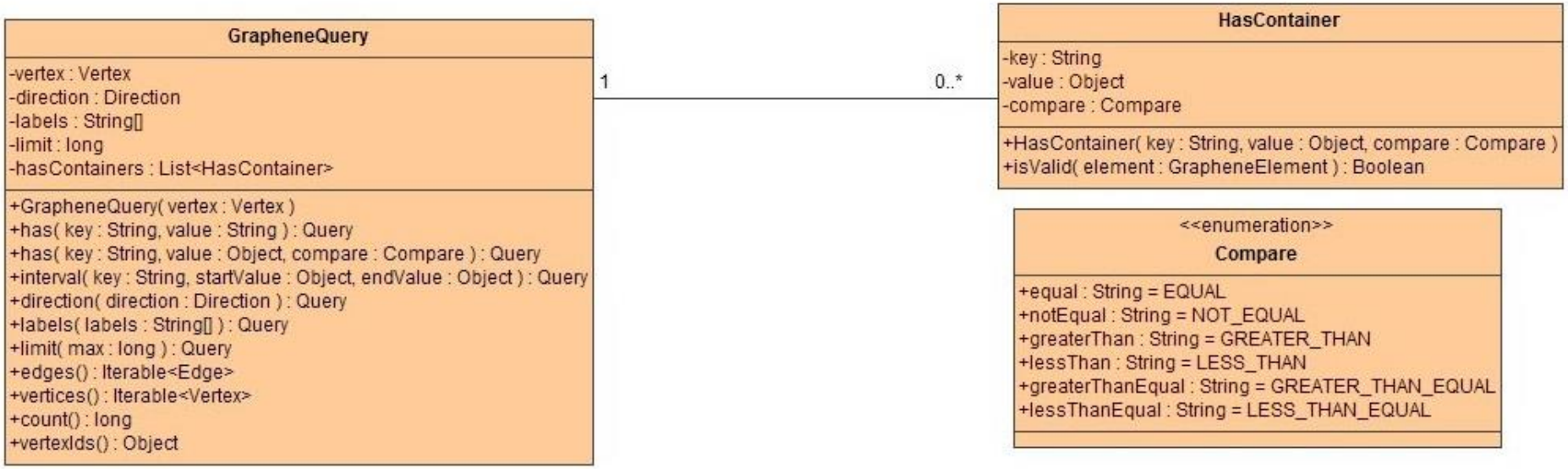

CoordenadorCentralService
-numeroDeFragmentos : long
-coordenadoresBDG : List
-idCoordenadorAtual : long
+setFragmentationPolicy(fragmentationPolicy : FragmentationPolicy) : void
+getNumeroDeFragmentos() : int
+getCoordenadorBDGAtual() : CoordenadorBDGService
+getFragment(Object : vertexId) : int
+getGDBService(vertexId : Object) : CoordenadorBDGService
+getAllGDBServices() : Iterable<CoordenadorBDGService>
+addVertex(vertexId : Object) : GrapheneVertex
+addEdge(edgeId : Object, outVertexId : Object, inVertexId : Object, label : String) : GrapheneEdge
+addVertexProperty(vertexId : Object, key : String, value : Object) : void
+removeVertexProperty(vertexId : Object, key : String) : Object
+addEdgeProperty(edgeId : Object, key : String, value : Object) : void
+removeEdgeProperty(edgeId : Object, key : String) : Object
+getVertex(vertexId : Object) : GrapheneVertex
+getVertices() : Iterable<GrapheneVertex>
+getVertices(key : String, value : Object) : Iterable<GrapheneVertex>
+getVertices(vertexId : Object, direction : Direction, labels : String[]) : Iterable<GrapheneVertex>
+removeVertex(vertexId : Object) : void
+getEdge(edgeId : Object) : GrapheneEdge
+getEdges() : Iterable<GrapheneEdge>
+getEdges(key : String, value : Object) : Iterable<GrapheneEdge>
+getEdges(vertexId : Object, direction : Direction, labels : String[]) : Iterable<GrapheneEdge>
+removeEdge(edgeId : Object) : void
+createKeyIndex(key : String, class : Class<? extends GrapheneElement>) : void
+dropKeyIndex(key : String, class : Class<? extends GrapheneElement>)
+performDBFSSourceDestination(sourceVertexId : Object, destinationVertexId : Object, direction : Direction, labels : String[]) : Collection<Collection<GrapheneElement>>
+performDBFS(sourceVertexId : Object, direction : Direction, evaluator : Evaluator, labels : String[]) : Collection<Collection<GrapheneElement>>
+stopTransaction(conclusion : Conclusion) : void
+shutdown() : void


CoordenadorBDGService
-identificador : long
-coordenadoresBDG : List
-fragmentationPolicy : FragmentationPolicy
-underlyingGraph : Graph
+getIdentifier() : long
+getGDBService(vertexId : Object) : CoordenadorBDGService
+getAllGDBServices() : Iterable<CoordenadorBDGService>
+getFragment(vertexId : Object) : int
+setFragmentationPolicy(fragmentationPolicy : FragmentationPolicy) : void
+addVertex(idVertex : Object) : GrapheneVertex
+addEdge(edgeId : Object, outVertexId : Object, inVertexId : Object, label : String) : GrapheneEdge
+addVirtualEdge(edgeId : Object, outVertexId : Object, inVertexId : Object, label : String) : GrapheneEdge
+addVertexProperty(vertexId : Object, key : String, value : Object) : void
+removeVertexProperty(vertexId : Object, key : String) : Object
+getVertex(vertexId : Object) : GrapheneVertex
+getVertexList(idsVertices : Iterable<Object>) : Iterable<GrapheneVertex>
+getEdgeCoordinator(edgeId : Object) : GrapheneEdge
+getEdge(edgeId : Object) : GrapheneEdge
+getEdgesCoordinator() : Iterable<GrapheneEdge>
+getEdges() : Iterable<GrapheneEdge>
+getVerticesCoordinator() : Iterable<GrapheneVertex>
+getVertex() : Iterable<GrapheneVertex>
+getVerticesCoordinator(key : String, value : Object) : Iterable<GrapheneVertex>
+getVertices(key : String, value : Object) : Iterable<GrapheneVertex>
+removeVertex(vertexId : Object) : void
+getEdgesCoordinator(key : String, value : Object) : Iterable<GrapheneEdge>
+getEdges(key : String, value : Object) : Iterable<GrapheneEdge>
+removeEdgeCoordinator(edgeId : Object) : void
+removeEdge(edgeId : Object) : void
+getEdgesCoordinator(vertexId : Object, direction : Direction, labels : String[]) : Iterable<GrapheneEdge>
+getEdges(vertexId : Object, direction : Direction, labels : String[]) : Iterable<GrapheneEdge>
+getVirtualEdges(vertexId : Object) : Iterable<GrapheneEdge>
+getVerticesCoordinator(vertexId : Object, direction : Direction, labels : String[]) : Iterable<GrapheneVertex>
+getVertices(vertexId : Object, direction : Direction, labels : String[]) : Iterable<GrapheneVertex>
+removeEdgePropertyCoordinator(edgeId : Object, key : String) : Object
+removeEdgeProperty(edgeId : Object, key : String) : Object
+addEdgePropertyCoordinator(edgeId : Object, key : String, value : Object) : void
+addEdgeProperty(edgeId : Object, key : String, value : Object) : void
+createKeyIndex(key : String, class : Class<? extends GrapheneElement>) : void
+dropKeyIndex(key : String, class : Class<? extends GrapheneElement>) : GrapheneElement
+performDBFSSourceDestination(sourceVertexId : Object, destinationVertexId : Object, direction : Direction, labels : String[]) : Collection<Collection<GrapheneElement>>
+performDBFS(sourceVertexId : Object, direction : Direction, evaluator : Evaluator, labels : String[]) : Collection<Collection<GrapheneElement>>
+stopTransaction(conclusion : Conclusion) : void
+shutdown() : void


GrapheneGraph
0..*
1
CoordenadorCentralService
-numeroDeFragmentos : long
-coordenadoresBDG : List
-idCoordenadorAtual : long
1
1..*
CoordenadorBDGService
-identificador : long
-coordenadoresBDG : List
-fragmentationPolicy : FragmentationPolicy
-underlyingGraph : Graph
0..*
1
0..*
1
FragmentationPolicy
-numeroFragmentos : int
-coordenadorBDG : CoordenadorBDGService


RMIBindService
+numeroDeFragmentos : int
+bind(nomeServico : String, servico : Remote) : void
+unbind(nomeServico : String) : void
FragmentationPolicy
-numeroFragmentos : int
-coordenadorBDG : CoordenadorBDGService
+vertexDestination(vertexId : Object) : int
+getCoordenadorCentral() : CoordenadorCentralService
HashFragmentationPolicy
+vertexDestination(vertexId : Object) : int
RangeFragmentationPolicy
+vertexDestination(vertexId : Object) : int
RoundRobinFragmentationPolicy
+vertexDestination(vertexId : Object) : int


<<interface>>
GrapheneElement
GrapheneVertex
-id : Object
-properties : Map<String, Object>
+GrapheneVertex(vertex : Vertex, coordenadorBDG : CoordenadorBDGService)
+getId() : Object
+getProperty(key : String) : Object
+getPropertyKeys() : Set<String>
+removeProperty(key : String) : Object
+setProperty(key : String, value : Object) : void
+getEdges(direction : Direction, labels : String[]) : Iterable<Edge>
+getVertices(direction : Direction, labels : String[]) : Iterable<Vertex>
+query() : Query
2
0..*
GrapheneEdge
-id : Object
-properties : Map<String, Object>
-label : String
-isVirtual : Boolean
+GrapheneEdge(edge : Edge, coordenadorBDG : CoordenadorBDGService)
+getId() : Object
+getProperty(key : String) : Object
+setProperty(key : String, value : Object) : void
+getPropertyKeys() : Set<String>
+getVertex(direction : Direction) : Vertex
+getIdSourceVertex() : Object
+getIdDestinationVertex() : Object
+removeProperty(key : String) : Object
<<interface>>
Evaluator
+evaluate(path : Iterable<GrapheneElement>) : Evaluation
<<enumeration>>
Evaluation
+includeAndContinue : String = INCLUDE_AND_CONTINUE
+includeAndPrune : String = INCLUDE_AND_PRUNE
+excludeAndContinue : String = EXCLUDE_AND_CONTINUE
+excludeAndPrune : String = EXCLUDE_AND_PRUNE


GrapheneQuery
-vertex : Vertex
-direction : Direction
-labels : String[]
-limit : long
-hasContainers : List<HasContainer>
+GrapheneQuery(vertex : Vertex)
+has(key : String, value : String) : Query
+has(key : String, value : Object, compare : Compare) : Query
+interval(key : String, startValue : Object, endValue : Object) : Query
+direction(direction : Direction) : Query
+labels(labels : String[]) : Query
+limit(max : long) : Query
+edges() : Iterable<Edge>
+vertices() : Iterable<Vertex>
+count() : long
+vertexIds() : Object
1
0..*
HasContainer
-key : String
-value : Object
-compare : Compare
+HasContainer(key : String, value : Object, compare : Compare)
+isValid(element : GrapheneElement) : Boolean
<<enumeration>>
Compare
+equal : String = EQUAL
+notEqual : String = NOT_EQUAL
+greaterThan : String = GREATER_THAN
+lessThan : String = LESS_THAN
+greaterThanEqual : String = GREATER_THAN_EQUAL
+lessThanEqual : String = LESS_THAN_EQUAL

Apêndice B

Base de Dados - WikiVote					
		Vértices	Arestas	Arestas de Corte	Tamanho da Base (MB)
	Centralizado	7115 (100%)	103689 (100%)	0	16,0
Espalhamento 2	Frag. 1	3549 (49,88%)	51816 (49,97%)	26228 (50,61%)	11,0
	Frag. 2	3566 (50,11%)	51873 (50,02%)	25567 (49,28%)	11,0
Espalhamento 4	Frag. 1	1782 (25,04%)	25682 (24,76%)	19471 (75,81%)	6,0
	Frag. 2	1795 (25,22%)	25153 (24,25%)	18657 (74,17%)	5,9
	Frag. 3	1767 (24,83%)	26134 (25,20%)	19484 (74,55%)	6,1
	Frag. 4	1771 (24,89%)	26720 (25,76%)	20229 (75,70%)	6,2
Espalhamento 8	Frag. 1	888 (12,48%)	12191 (11,75%)	10924 (89,60%)	2,9
	Frag. 2	904 (12,70%)	12574 (12,12%)	10927 (86,90%)	2,9
	Frag. 3	903 (12,69%)	12938 (12,47%)	11234 (86,82%)	3,0
	Frag. 4	891 (12,52%)	12761 (12,30%)	11196 (87,73%)	3,0
	Frag. 5	894 (12,56%)	13491 (13,01%)	11659 (86,42%)	3,1

	Frag. 6	891 (12,52%)	12579 (12,13%)	11027 (87,66%)	2,9
	Frag. 7	864 (12,14%)	13196 (12,72%)	11563 (87,62%)	3,1
	Frag. 8	880 (12,36%)	13959 (13,46%)	12221 (87,54%)	3,2
Espalhamento 16	Frag. 1	489 (6,87%)	7864 (7,58%)	7330 (93,20%)	2,0
	Frag. 2	505 (7,09%)	6142 (5,92%)	5707 (92,91%)	1,6
	Frag. 3	499 (7,01%)	8364 (8,06%)	7742 (92,56%)	2,1
	Frag. 4	485 (6,81%)	7303 (7,04%)	6793 (93,01%)	1,8
	Frag. 5	475 (6,67%)	7293 (7,03%)	6771 (92,84%)	1,8
	Frag. 6	446 (6,26%)	5706 (5,50%)	5354 (93,83%)	1,5
	Frag. 7	424 (5,95%)	7266 (7,00%)	6820 (93,86%)	1,8
	Frag. 8	403 (5,66%)	7205 (6,94%)	6764 (93,87%)	1,8
	Frag. 9	399 (5,60%)	4327 (4,17%)	4170 (96,37%)	1,2
	Frag. 10	399 (5,60%)	6432 (6,20%)	6078 (94,49%)	1,7
	Frag. 11	404 (5,67%)	4574 (4,41%)	4350 (95,10%)	1,3
	Frag. 12	406 (5,70%)	5458 (5,26%)	5166 (94,65%)	1,5
	Frag. 13	419 (5,88%)	6198 (5,97%)	5807 (93,69%)	1,6

	Frag. 14	445 (6,25%)	6873 (6,62%)	6445 (93,77%)	1,7
	Frag. 15	440 (6,18%)	5930 (5,71%)	5546 (93,52%)	1,5
	Frag. 16	477 (6,70%)	6754 (6,51%)	6340 (93,87%)	1,7
METIS 2	Frag. 1	3664 (51,49%)	53760 (51,84%)	11268 (20,95%)	13,0
	Frag. 2	3451 (48,50%)	49929 (48,15%)	16788 (33,62%)	12,0
METIS 4	Frag. 1	1832 (25,74%)	16267 (15,68%)	3321 (20,41%)	4,1
	Frag. 2	1725 (24,24%)	27050 (26,08%)	13862 (51,24%)	6,7
	Frag. 3	1831 (25,73%)	38481 (37,11%)	19590 (50,90%)	9,4
	Frag. 4	1727 (24,27%)	21891 (21,11%)	13120 (59,93%)	5,8
METIS 8	Frag. 1	916 (12,87%)	13613 (13,12%)	9220 (67,72%)	3,7
	Frag. 2	863 (12,12%)	6562 (6,32%)	5748 (87,59%)	2,0
	Frag. 3	916 (12,87%)	23574 (22,73%)	13132 (55,70%)	5,9
	Frag. 4	863 (12,12%)	9872 (9,52%)	8655 (87,67%)	2,9
	Frag. 5	916 (12,87%)	23159 (22,33%)	13578 (58,62%)	5,9
	Frag. 6	915 (12,86%)	14641 (14,12%)	10798 (73,75%)	3,9

	Frag. 7	863 (12,12%)	3275 (3,15%)	2786 (85,06%)	1,2
	Frag. 8	863 (12,12%)	8993 (8,67%)	6959 (77,38%)	2,5
METIS 16	Frag. 1	458 (6,43%)	13274 (12,80%)	9552 (71,96%)	3,6
	Frag. 2	432 (6,07%)	4957 (4,78%)	4580 (92,39%)	1,6
	Frag. 3	431 (6,05%)	3157 (3,04%)	2734 (86,60%)	1,1
	Frag. 4	458 (6,43%)	7313 (7,05%)	5228 (71,48%)	2,1
	Frag. 5	431 (6,05%)	3340 (3,22%)	3103 (92,90%)	1,2
	Frag. 6	458 (6,43%)	6664 (6,42%)	6038 (90,60%)	2,0
	Frag. 7	432 (6,07%)	2556 (2,46%)	2264 (88,57%)	0,928
	Frag. 8	458 (6,43%)	13704 (13,21%)	11100 (80,99%)	3,7
	Frag. 9	434 (6,09%)	2587 (2,49%)	2255 (87,16%)	0,936
	Frag. 10	458 (6,43%)	15809 (15,24%)	11440 (72,36%)	4,2
	Frag. 11	457 (6,42%)	6627 (6,39%)	5357 (80,83%)	2,0
	Frag. 12	431 (6,05%)	2736 (2,63%)	2433 (88,92%)	0,988
	Frag. 13	431 (6,05%)	3782 (3,64%)	3400 (89,89%)	1,3
	Frag. 14	458 (6,43%)	6332 (6,10%)	4184 (66,07%)	1,8

	Frag. 15	430 (6,04%)	1682 (1,62%)	1582 (94,05%)	0,724
	Frag. 16	458 (6,43%)	9169 (8,84%)	7477 (81,54%)	2,6

Base de Dados - Epinions					
		Vértices	Arestas	Arestas de Corte	Tamanho da Base (MB)
	Centralizado	75879 (100%)	508837 (100%)	0	85
Espalhamento 2	Frag. 1	37938 (49,99%)	257192 (50,54%)	128530 (49,97%)	54
	Frag. 2	37941 (50,00%)	251645 (49,45%)	125994 (50,06%)	53
Espalhamento 4	Frag. 1	18970 (25,00%)	132990 (26,13%)	98681 (74,20%)	30
	Frag. 2	18973 (25,00%)	123020 (24,17%)	93192 (75,75%)	28
	Frag. 3	18968 (24,99%)	124202 (24,40%)	94467 (76,05%)	28
	Frag. 4	18968 (24,99%)	128625 (25,27%)	95626 (74,34%)	29
Espalhamento 8	Frag. 1	9483 (12,49%)	65666 (12,90%)	56974 (86,76%)	15
	Frag. 2	9483 (12,49%)	62756 (12,33%)	55157 (87,89%)	15
	Frag. 3	9482 (12,49%)	64976 (12,76%)	57205 (88,04%)	15
	Frag. 4	9485 (12,50%)	64472 (12,67%)	56341 (87,38%)	15
	Frag. 5	9487 (12,50%)	67324 (13,23%)	58673 (87,15%)	16
	Frag. 6	9490 (12,50%)	60264 (11,84%)	53055 (88,03%)	14
	Frag. 7	9486 (12,50%)	59226 (11,63%)	52129 (88,01%)	14

	Frag. 8	9483 (12,49%)	64153 (12,60%)	55780 (86,94%)	15
Espalhamento 16	Frag. 1	4806 (6,33%)	35407 (6,95%)	32806 (92,65%)	8,1
	Frag. 2	4904 (6,46%)	33475 (6,57%)	31276 (93,43%)	7,8
	Frag. 3	4978 (6,56%)	36418 (7,15%)	33868 (92,99%)	8,4
	Frag. 4	5017 (6,61%)	36862 (7,24%)	34282 (93,00%)	8,5
	Frag. 5	5013 (6,60%)	36581 (7,18%)	33860 (92,56%)	8,4
	Frag. 6	4970 (6,54%)	30618 (6,01%)	28665 (93,62%)	7,3
	Frag. 7	4889 (6,44%)	30522 (5,99%)	28620 (93,76%)	7,3
	Frag. 8	4786 (6,30%)	33440 (6,57%)	31098 (92,99%)	7,7
	Frag. 9	4677 (6,16%)	30259 (5,94%)	28541 (94,32%)	7,3
	Frag. 10	4579 (6,03%)	29281 (5,75%)	27730 (94,70%)	7,0
	Frag. 11	4504 (5,93%)	28558 (5,61%)	27154 (95,08%)	6,9
	Frag. 12	4468 (5,88%)	27610 (5,42%)	26116 (94,58%)	6,7
	Frag. 13	4474 (5,89%)	30743 (6,04%)	29041 (94,46%)	7,2
	Frag. 14	4520 (5,95%)	29646 (5,82%)	27929 (94,20%)	7,1
	Frag. 15	4597 (6,05%)	28704 (5,64%)	27020 (94,13%)	6,9

	Frag. 16	4697 (6,19%)	30713 (6,03%)	28890 (94,06%)	7,2
METIS 2	Frag. 1	36834 (48,54%)	176166 (34,62%)	56607 (32,13%)	47
	Frag. 2	39045 (51,45%)	332671 (65,37)	35106 (10,55%)	77
METIS 4	Frag. 1	18379 (24,22%)	53114 (10,43%)	31285 (58,90%)	17
	Frag. 2	19539 (25,75%)	272019 (53,45%)	40427 (14,86%)	62
	Frag. 3	18980 (25,01%)	81084 (15,93%)	36157 (44,59%)	23
	Frag. 4	18981 (25,01%)	102620 (20,16%)	46309 (45,12%)	28
METIS 8	Frag. 1	9280 (12,22%)	30271 (5,94%)	19041 (62,90%)	9,3
	Frag. 2	9281 (12,23%)	33518 (6,58%)	18431 (54,98%)	9,9
	Frag. 3	9268 (12,21%)	20443 (4,01%)	11645 (56,96%)	6,8
	Frag. 4	9465 (12,47%)	72153 (14,17%)	30330 (42,03%)	19
	Frag. 5	9769 (12,87%)	179128 (35,20%)	54537 (30,44%)	41
	Frag. 6	9278 (12,22%)	29823 (5,86%)	19763 (66,26%)	9,1
	Frag. 7	9769 (12,87%)	65418 (12,85%)	42287 (64,64%)	18
	Frag. 8	9769 (12,87%)	78083 (15,34%)	34940 (44,74%)	20

METIS 16	Frag. 1	4668 (6,15%)	14453 (2,84%)	8363 (57,86%)	4,5
	Frag. 2	4664 (6,14%)	19901 (3,91%)	11524 (57,90%)	5,9
	Frag. 3	4658 (6,13%)	14500 (2,84%)	8443 (58,22%)	4,6
	Frag. 4	4884 (6,43%)	32590 (6,40%)	17968 (55,13%)	8,9
	Frag. 5	4884 (6,43%)	46779 (9,19%)	21665 (46,31%)	12
	Frag. 6	4883 (6,43%)	34022 (6,68%)	17226 (50,63%)	9,1
	Frag. 7	4661 (6,14%)	10898 (2,14%)	6481 (59,46%)	3,7
	Frag. 8	4884 (6,43%)	95288 (18,72%)	45322 (47,56%)	23
	Frag. 9	4654 (6,13%)	13592 (2,67%)	9520 (70,04%)	4,4
	Frag. 10	4884 (6,43%)	57741 (11,34%)	29896 (51,77%)	15
	Frag. 11	4651 (6,12%)	16154 (3,17%)	11029 (68,27%)	5,1
	Frag. 12	4884 (6,43%)	87944 (17,28%)	42298 (48,09%)	22
	Frag. 13	4661 (6,14%)	18760 (3,68%)	13250 (70,62%)	5,7
	Frag. 14	4652 (6,13%)	15644 (3,07%)	9719 (62,12%)	4,7
	Frag. 15	4653 (6,13%)	14555 (2,86%)	9727 (66,82%)	4,9
	Frag. 16	4654 (6,13%)	16016 (3,14%)	10912 (68,13%)	5

Base de Dados - WebStanford					
		Vértices	Arestas	Arestas de Corte	Tamanho da Base (MB)
	Centralizado	281903 (100%)	2312497 (100%)	0	397
Espalhamento 2	Frag. 1	140951 (49,99%)	1157681 (50,06%)	552343 (47,71%)	268
	Frag. 2	140952 (50%)	1154816 (49,93%)	602941 (52,21%)	268
Espalhamento 4	Frag. 1	70478 (25%)	579820 (25,07%)	430186 (74,19%)	136
	Frag. 2	70476 (25%)	582625 (25,19%)	451199 (77,44%)	137
	Frag. 3	70473 (24,99%)	577861 (24,98%)	424736 (73,5%)	135
	Frag. 4	70476 (25%)	572191 (24,74%)	427468 (74,7%)	134
Espalhamento 8	Frag. 1	35249 (12,5%)	288204 (12,46%)	250634 (86,96%)	68
	Frag. 2	35244 (12,5%)	292636 (12,65%)	260827 (89,13%)	69
	Frag. 3	35234 (12,49%)	290582 (12,56%)	253130 (87,11%)	69
	Frag. 4	35229 (12,49%)	287063 (12,41%)	251935 (87,76%)	68
	Frag. 5	35229 (12,49%)	291616 (12,61%)	254119 (87,14%)	69
	Frag. 6	35232 (12,49%)	289989 (12,54%)	256034 (88,29%)	69

	Frag. 7	35239 (12,5%)	287279 (12,42%)	247957 (86,31%)	68
	Frag. 8	35247 (12,5%)	285128 (12,32%)	247718 (86,87%)	68
Espalhamento 16	Frag. 1	17872 (6,33%)	145351 (6,28%)	135009 (92,88%)	35
	Frag. 2	18137 (6,43%)	150248 (6,49%)	142324 (94,72%)	36
	Frag. 3	18320 (6,49%)	153810 (6,65%)	143559 (93,33%)	37
	Frag. 4	18399 (6,52%)	149595 (6,46%)	139610 (93,32%)	36
	Frag. 5	18361 (6,51%)	151319 (6,54%)	141908 (93,78%)	36
	Frag. 6	18211 (6,46%)	150503 (6,5%)	141377 (93,93%)	36
	Frag. 7	17972 (6,37%)	145060 (6,27%)	137440 (94,74%)	35
	Frag. 8	17680 (6,27%)	144718 (6,25%)	132386 (91,47%)	35
	Frag. 9	17377 (6,16%)	142853 (6,17%)	134490 (94,14%)	34
	Frag. 10	17107 (6,06%)	142388 (6,15%)	134400 (94,38%)	34
	Frag. 11	16914 (5,99%)	136772 (5,91%)	128265 (93,78%)	33
	Frag. 12	16830 (5,97%)	137468 (5,94%)	129721 (94,36%)	33
	Frag. 13	16868 (5,98%)	140297 (6,06%)	130979 (93,35%)	34
	Frag. 14	17021 (6,03%)	139486 (6,03%)	131487 (94,26%)	34

	Frag. 15	17267 (6,12%)	142219 (6,15%)	130407 (91,69%)	34
	Frag. 16	17567 (6,23%)	140410 (6,07%)	133982 (95,42%)	34
METIS 2	Frag. 1	136839 (48,54%)	783304 (33,87%)	19461 (2,48%)	195
	Frag. 2	145064 (51,45%)	1529193 (66,12%)	36252 (2,37%)	357
METIS 4	Frag. 1	72214 (25,61%)	608164 (26,29%)	36651 (6,02%)	145
	Frag. 2	72568 (25,74%)	731809 (31,64%)	83412 (11,39%)	174
	Frag. 3	68567 (24,32%)	520186 (22,49%)	6179 (1,18%)	124
	Frag. 4	68554 (24,31%)	452338 (19,56%)	20350 (4,49%)	111
METIS 8	Frag. 1	36296 (12,87%)	222952 (9,64%)	2331 (1,04%)	55
	Frag. 2	34210 (12,13%)	171877 (7,43%)	40006 (23,27%)	45
	Frag. 3	34841 (12,35%)	327150 (14,14%)	1184 (0,36%)	76
	Frag. 4	34209 (12,13%)	161206 (6,97%)	1936 (1,2%)	41
	Frag. 5	34234 (12,14%)	187761 (8,11%)	17042 (9,07%)	48
	Frag. 6	35546 (12,6%)	317059 (13,71%)	26010 (8,2%)	76
	Frag. 7	36296 (12,87%)	510026 (22,05%)	7673 (1,5%)	116

	Frag. 8	36271 (12,86%)	414466 (17,92%)	18432 (4,44%)	96
METIS 16	Frag. 1	17948 (6,36%)	102147 (4,41%)	14951 (14,63%)	26
	Frag. 2	17613 (6,24%)	96748 (4,18%)	5174 (5,34%)	24
	Frag. 3	17104 (6,06%)	101328 (4,38%)	7763 (7,66%)	26
	Frag. 4	17099 (6,06%)	98371 (4,25%)	2551 (2,59%)	25
	Frag. 5	17104 (6,06%)	153220 (6,62%)	11538 (7,53%)	37
	Frag. 6	18148 (6,43%)	138739 (5,99%)	17099 (12,32%)	34
	Frag. 7	18146 (6,43%)	190243 (8,22%)	29742 (15,63%)	46
	Frag. 8	18145 (6,43%)	251930 (10,89%)	17354 (6,88%)	58
	Frag. 9	18018 (6,39%)	97175 (4,2%)	5259 (5,41%)	25
	Frag. 10	17957 (6,36%)	81449 (3,52%)	12241 (15,02%)	22
	Frag. 11	17964 (6,37%)	83650 (3,61%)	4817 (5,75%)	22
	Frag. 12	17510 (6,21%)	234341 (10,13%)	564 (0,24%)	54
	Frag. 13	17103 (6,06%)	171591 (7,42%)	31205 (18,18%)	42
	Frag. 14	18148 (6,43%)	102009 (4,41%)	17395 (17,05%)	27
	Frag. 15	16491 (5,84%)	296747 (12,83%)	158588 (53,44%)	75

	Frag. 16	17405 (6,17%)	112809 (4,87%)	17900 (15,86%)	29

Apêndice C

Tempo Bloqueado por Nó na Operação de Travessia da Base WikiVote		
		Tempo Bloqueado (em segundos)
Espalhamento 2	Nó 1	1,828
	Nó 2	2,403
Espalhamento 4	Nó 1	2,807
	Nó 2	2,44
	Nó 3	2,681
	Nó 4	1,653
Espalhamento 8	Nó 1	4,487
	Nó 2	3,696
	Nó 3	4,365
	Nó 4	4,209
	Nó 5	4,21
	Nó 6	3,989
	Nó 7	4,611
	Nó 8	1,674
Espalhamento 16	Nó 1	5,531
	Nó 2	5,941
	Nó 3	5,919
	Nó 4	6,034
	Nó 5	6,077
	Nó 6	5,858
	Nó 7	5,955
	Nó 8	5,6

	Nó 9	7,306
	Nó 10	5,708
	Nó 11	6,799
	Nó 12	6,726
	Nó 13	6,213
	Nó 14	5,995
	Nó 15	7,041
	Nó 16	1,647
METIS 2	Nó 1	4,498
	Nó 2	1,362
METIS 4	Nó 1	6,763
	Nó 2	2,797
	Nó 3	1,353
	Nó 4	6,063
METIS 8	Nó 1	12,035
	Nó 2	12,253
	Nó 3	10,439
	Nó 4	9,359
	Nó 5	1,151
	Nó 6	8,506
	Nó 7	12,03
	Nó 8	11,026
METIS 16	Nó 1	14,012
	Nó 2	15,858
	Nó 3	15,499
	Nó 4	14,544
	Nó 5	15,098
	Nó 6	14,996
	Nó 7	11,452
	Nó 8	15,047
	Nó 9	13,278

	Nó 10	1,043
	Nó 11	13,808
	Nó 12	15,224
	Nó 13	13,989
	Nó 14	14,905
	Nó 15	15,84
	Nó 16	10,253

Tempo Bloqueado por Nó na Operação de Travessia da Base Epinions		
		Tempo Bloqueado (em segundos)
Espalhamento 2	Nó 1	3,923
	Nó 2	1,597
Espalhamento 4	Nó 1	3,52
	Nó 2	5,385
	Nó 3	5,583
	Nó 4	2,637
Espalhamento 8	Nó 1	4,213
	Nó 2	5,505
	Nó 3	5,684
	Nó 4	4,982
	Nó 5	4,79
	Nó 6	5,091
	Nó 7	4,48
	Nó 8	5,535
Espalhamento 16	Nó 1	8,481
	Nó 2	6,312
	Nó 3	5,683
	Nó 4	6,622
	Nó 5	8,062
	Nó 6	7,933
	Nó 7	9,15
	Nó 8	10,42
	Nó 9	7,912
	Nó 10	11,143
	Nó 11	11,946
	Nó 12	10,373
	Nó 13	8,497
	Nó 14	8,257
	Nó 15	8,401
	Nó 16	8,15

METIS 2	Nó 1	10,84
	Nó 2	8,342
METIS 4	Nó 1	24,96
	Nó 2	5,056
	Nó 3	20,092
	Nó 4	13,666
METIS 8	Nó 1	30,755
	Nó 2	27,891
	Nó 3	31,701
	Nó 4	20,519
	Nó 5	3,87
	Nó 6	28,974
	Nó 7	18,366
	Nó 8	21,088
METIS 16	Nó 1	33,772
	Nó 2	30,859
	Nó 3	30,498
	Nó 4	24,672
	Nó 5	23,607
	Nó 6	22,616
	Nó 7	31,53
	Nó 8	11,736
	Nó 9	30,404
	Nó 10	20,119
	Nó 11	28,718
	Nó 12	12,688
	Nó 13	29,56
	Nó 14	29,697
	Nó 15	30,523
	Nó 16	29,953

Tempo Bloqueado por Nó na Operação de Travessia da Base WebStanford		
		Tempo Bloqueado (em segundos)
Espalhamento 2	Nó 1	2,604
	Nó 2	1,856
Espalhamento 4	Nó 1	2,059
	Nó 2	2,921
	Nó 3	3,094
	Nó 4	1,987
Espalhamento 8	Nó 1	3,989
	Nó 2	5,111
	Nó 3	3,776
	Nó 4	3,512
	Nó 5	2,613
	Nó 6	3,6
	Nó 7	4,977
	Nó 8	3,9
Espalhamento 16	Nó 1	4,985
	Nó 2	6,731
	Nó 3	4,663
	Nó 4	4,625
	Nó 5	2,616
	Nó 6	5,273
	Nó 7	6,642
	Nó 8	5,07
	Nó 9	6,054
	Nó 10	5,818
	Nó 11	6,075
	Nó 12	5,495
	Nó 13	5,868
	Nó 14	5,174

	Nó 15	5,914
	Nó 16	6,154
METIS 2	Nó 1	7,831
	Nó 2	0,065
METIS 4	Nó 1	9,565
	Nó 2	4,631
	Nó 3	10,004
	Nó 4	1,986
METIS 8	Nó 1	14,361
	Nó 2	14,277
	Nó 3	14,336
	Nó 4	14,19
	Nó 5	13,661
	Nó 6	13,434
	Nó 7	12,199
	Nó 8	0,21
METIS 16	Nó 1	21,005
	Nó 2	20,741
	Nó 3	20,791
	Nó 4	20,953
	Nó 5	19,716
	Nó 6	21,004
	Nó 7	19,152
	Nó 8	0,263
	Nó 9	21,054
	Nó 10	21,181
	Nó 11	20,597
	Nó 12	21,086
	Nó 13	21,061
	Nó 14	21,047
	Nó 15	20,205

	Nó 16	21,16