BANCO DE DADOS VS MAPREDUCE EM ALGORITMOS PARA GRAFOS

Fabiano da Silva Fernandes

Março / 2015

Dissertação de Mestrado em
Ciência da Computação

BANCO DE DADOS VS MAPREDUCE EM ALGORITMOS PARA GRAFOS

Esse documento corresponde à dissertação de mestrado apresentada à Banca Examinadora no curso de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista.

Campo Limpo Paulista, 28 de Março de 2015.

Fabiano da Silva Fernandes

Prof. Dr. Eduardo Javier Huerta Yero (Orientador)

Faculdade Campo Limpo Paulista

Programa de Mestrado em Ciência da Computação

"Banco de Dados vs MapReduce em Algoritmos para Grafos"

Fabiano da Silva Fernandes

Dissertação de Mestrado apresentado ao Programa de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Eduardo Javier Huerta Yero
(Orientador –FACCAMP)

Prof. Dr. Luis Mariano Del Val Cura
(FACCAMP)

Prof. Dr. Marcelo Paiva Guimarães
(FACCAMP)

Prof. Dr. Flávio Marques Azevedo
(USP)

Campo Limpo Paulista, 28 de março de 2015.

FICHA CATALOGRÁFICA

Dados Internacionais de Catalogação na Publicação (CIP)

Câmara Brasileira do Livro, São Paulo, Brasil

Fernandes, Fabiano da Silva
Banco de dados vs MapReduce em algoritmos para grafos / Fabiano da Silva Fernandes. Campo Limpo Paulista, SP: FACCAMP, 2015.

Orientador: Profº. Dr. Eduardo Javier Huerta Yero
Dissertação (mestrado) – Faculdade Campo Limpo Paulista – FACCAMP.

1. Grafos. 2. Banco de dados paralelo. 3. MapReduce. 4. Banco de dados orientado a grafos. 5. NetworkX. I. Huerta Yero, Eduardo Javier. II. Faculdade Campo Limpo Paulista. III. Título.

CDD-005.75

À minha esposa, Monica Priscila S. Fernandes

que sempre me apoiou

AGRADECIMENTOS

À Deus, pelo seu amor e misericórdia.

Ao meu professor orientador, Eduardo Javier Huerta Yero, por sua paciência e orientação que enriqueceram este trabalho.

Ao grande amigo, Francisco Sanches Banhos Filho, pela sua enorme contribuição para elaboração deste trabalho.

Aos professores, funcionários e colegas do Programa de Mestrado em Ciência da FACCAMP.

À minha família, pelo apoio incondicional que sempre me deu ao longo desses anos.

À minha esposa, minha grande inspiração.

RESUMO

Nos últimos anos tem havido uma explosão sem precedentes na quantidade de dados disponíveis digitalmente. Conjuntos de dados deste porte apresentam grandes desafios na sua captura, processamento, armazenamento, transmissão e visualização, o que tem provocado uma concentração dos esforços da comunidade acadêmica nesta área de pesquisa. O presente trabalho foca no caso particular em que os dados são modelados através de grafos, como comumente acontece quando o problema modelado envolve entidades discretas que se relacionam entre si (e.g. redes sociais). Para isso, este trabalho analisa a efetividade de quatro tecnologias para tratar com grafos: modelo MapReduce, um Sistema de Gerenciamento de Bancos de Dados Relacionais Paralelo (SGBD paralelo), um Banco de Dados Orientado a Grafos e como referência adicional, o comparativo inclui uma solução processamento sequencial, uma biblioteca especializada em grafos chamada NetworkX que privilegia o processamento em memória. O problema escolhido para este comparativo foi o cálculo do raio e diâmetro de um grafo, pois ele necessita do cálculo da distância entre todos os pares de nós do grafo, que é um problema computacionalmente complexo. Experimentos feitos na plataforma Grid'5000 em uma infraestrutura *shared-nothing* indicam que os SGBDs paralelos estão melhores equipados para resolver este tipo de problemas quando executados em grande quantidade de computadores.

Palavras-chave: Grafos, Banco de dados pararelo; MapReduce; Banco de dados orientado a grafos, NetworkX;

ABSTRACT

Recent years have seen an unprecedented explosion in the amount of data available digitally. Capturing, processing, storing, transmitting and visualizing datasets this large pose enormous challenges, which has generated a concentration of the efforts of the academic community in this specific research topic. This work focuses on the particular case where these datasets can be modeled as graphs, as is commonly the case when the modeled problem involves discreet entities related to each other (e.g. social networks). To this effect, it examines the effectiveness of four technologies: the MapReduce model, Parallel Relational Database Management Systems (Parallel DBMS), Graph-oriented databases and a sequential library for manipulating graphs that focuses on processing in RAM. The problem chosen was the calculation of the radius and diameter of a graph, since it involves the calculation of the distance between all pairs of nodes of the graph, which is a computationally complex problem. Experiments conducted in the Grid'5000 platform indicate that the parallel DBMSs are best equipped to solve such problems when executed on a large number of computers.

Key-words: Graphs, Parallel relational database; MapReduce; Graph-oriented database.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
BFS	*Breath-First Search*
DDL	*Data Definition Language*
DFS	*Depth-First Search*
DML	*Data Manipulation Language*
E/S	Entrada e saída
GPLv3	*GNU Public License version 3*
HDFS	*Hadoop Distributed File System*
HTML	*HyperText Markup Language*
NoSQL	*Not only SQL*
PC	*Personal Computer*
RPC	*Remote Procedure Call*
SGBD	Sistema de Gestão de Banco de Dados
SQL	*Structured Query Language*

1. CAPÍTULO 1 - INTRODUÇÃO

Nos últimos anos tem-se visto uma explosão sem precedentes na quantidade de dados disponíveis digitalmente. Estudos patrocinados pela empresa EMC estimam que o mundo gerou 130 *exabytes* (10^{18} *bytes*) de dados em 2005, 1.227 exabytes em 2010, 2.837 *exabytes* em 2012 e prevê que sejam criados 40.026 *exabytes* de informação digital em 2020 (EMC2, 2014). Hoje, o volume de dados chega a 4.8 trilhões de anúncios online, 294 bilhões de e-mails diários, 100 *terabytes* (10^{12} *bytes*) de dados atualizados diariamente no Facebook e 230 milhões de mensagens diárias enviadas pelo Twitter (IBM, 2012). Áreas das ciências tradicionais, como astronomia, física, meteorologia, pesquisa genômica e biologia também geram *exabytes* de dados (Smith et al., 2012).

A coleta e processamento de grandes conjuntos de dados vêm sendo chamados de Big Data (Zhaohui, 2014). Os desafios apresentados por esta nova era podem ser descritos em 3 dimensões, representadas na (Gartner, 2011).

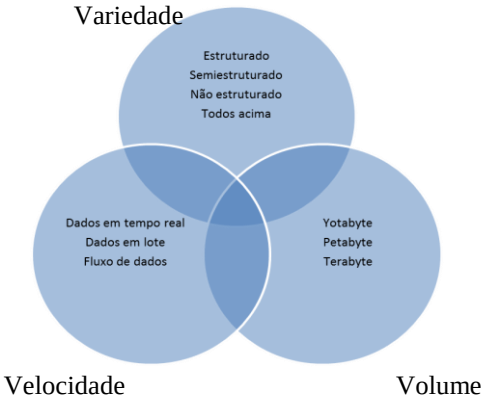


Figura 1.1 As três dimensões de Big Data
Adaptado de (Gartner, 2011)

- **Volume:** diz respeito à quantidade de dados gerados.
- **Velocidade:** diz respeito à velocidade com que os dados estão sendo produzidos e devem ser processados.

- **Variedade:** os dados podem ter características bem diferentes entre si (texto, dados de sensores, áudio, vídeo, imagens, arquivos de log, dados de medição, páginas *html*, etc).

Apesar das enormes possibilidades advindas do uso destes dados, sua extração, transmissão, armazenamento, processamento e visualização apresenta enormes desafios computacionais. Considerando o grande volume a ser analisado e que o processamento está além da capacidade de qualquer máquina individual, faz-se necessária a utilização de modelos de computação paralela (Lin & Dyer, 2010).

Dentre os problemas resolvidos usando técnicas de Big Data, destacam-se aqueles cujos dados podem ser representados por grafos. Os grafos são normalmente utilizados para modelar relacionamentos entre um conjunto finito de entidades. Projetos de instalações elétricas e mecânicas, serviços de utilidade pública (água, luz, gás e telefone), redes de drenagem, malhas viárias, redes sociais e circuitos eletrônicos, dentre outros, podem ser representados por grafos (Barroso, 2014). O processamento de grafos apresenta desafios que fazem com que o uso das técnicas de Big Data não seja trivial, devido à interdependência entre os vértices do grafo. Nesse sentido, o desafio torna-se ainda maior quando os grafos alcançam milhões de vértices e arestas.

Existem várias tecnologias para tratar problemas de processamento de grandes massas de dados. Essas tecnologias podem ser divididas em dois grupos: Sistemas de Gerenciamento de Banco de Dados (SGBD) e Sistemas de Fluxo de Dados (Herodotou, 2012). Dentre os SGBDs destacam-se aqueles que usam processamento paralelo, tais como Teradata (Teradata, 2014), HP Vértica (HP Vertica, 2014) e Pivotal Greenplum (Pivotal Greenplum, 2014). Já o exemplo mais conhecido dos Sistemas de Fluxo de Dados é o *Google MapReduce* (Dean & Ghemawat, 2008), que tornou-se muito popular nos ambientes empresariais e acadêmicos.

No caso particular de grafos, as tecnologias baseadas em bancos de dados incluem os bancos de dados orientados a grafos, como *InfiniteGraph* (InfiniteGraph, 2014), *HypergraphDB* (Anon., 2014) e *Neo4J* (Neo Technology, 2006), que se propõem garantir desempenho e escalabilidade para grafos de grande porte. Outras soluções, classificadas como sistemas de fluxo de dados, incluem o *Graphlab* (Low et al., 2012), um *framework* para aprendizagem de máquina, o *Dryad* (Isard et al., 2007), um sistema de propósito geral para aplicações com paralelismo de dados, assim como o *Apache*

Hama (Apache Hama, 2014) e o *Apache Giraph* (Apache Giraph, 2014), inspirados no *Google Pregel* (Malewicz et al., 2010) e que são baseados no modelo de programação paralela *Bulk Synchronous Parallel* (BSP).

Dentre as soluções apresentadas no parágrafo anterior, nenhuma tem gerado tanto interesse quanto *MapReduce* (Zhao, Y et al., 2014). Apesar da ampla adoção, não há consenso na comunidade científica sobre a validade deste modelo para tratar com grafos. Alguns trabalhos comparam *MapReduce* com o *BSP* (Kajdanowicz, T. et al., 2014) para algoritmos em grafos, e concluem que os sistemas baseados em BSP tem melhor desempenho nestes casos que os baseados em MapReduce. Por outro lado, Stonebraker já sugeriu em (Stonebraker, M. et al., 2010) que os bancos de dados paralelos baseados no modelo relacional ofereciam melhor desempenho para muitos dos problemas que são atualmente resolvidos com *MapReduce*, mas sem tratar especificamente de algoritmos para grafos. Para tratar especificamente com grafos há bancos de dados como o Neo4J que, apesar de altamente otimizados, não costumam oferecer a possibilidade de processamento paralelo e, portanto, acabam tendo sua escalabilidade questionada.

O objetivo deste trabalho é apresentar uma comparação entre quatro tecnologias para resolver o problema do cálculo de medidas de centralidade de um grafo (i.e. raio e diâmetro). O cálculo de medidas de centralidade foi escolhido por exigir o cálculo da distância entre todos os pares de vértices do grafo, um problema computacionalmente complexo, em particular para grafos grandes.

As tecnologias comparadas neste trabalho incluem uma solução baseada no modelo *MapReduce*, um algoritmo para Bancos de Dados Paralelos relacionais e outro que utiliza um Banco de Dados Orientado a Grafos além de, como referência adicional, uma solução de processamento sequencial que utiliza uma biblioteca especializada em grafos chamada *NetworkX*.

Para representar o modelo *MapReduce* no comparativo utiliza-se o algoritmo HEDA (Nascimento & Murta, 2012). O HEDA, primeiramente, calcula o menor caminho entre todos os pares de vértices e, posteriormente, extrai o raio e o diâmentro do grafo. Para o caso do Banco de Dados Paralelo, o Banco de Dados Orientados a Grafos e a solução sequencial foram gerados algoritmos equivalentes.

O trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta conceitos e definições importantes da teoria de grafos que serão utilizados nos capítulos subsequentes. O Capítulo 3 introduz os conceitos sobre bancos de dados e o modelo relacional. O Capítulo 4 aborda o movimento NoSQL e os bancos de dados orientados a grafos. O Capítulo 5 descreve o modelo *MapReduce* e sua implementação mais conhecida, o *framework* Hadoop. O Capítulo 6 apresenta o algoritmo HEDA, que utiliza o modelo *MapReduce* e os algoritmos equivalentes que calculam a centralidade em grafos para o Banco de Dados Paralelo e Banco de Dados Orientado Grafos, além da biblioteca *NetworkX*. No Capítulo 7, encontra-se o planejamento dos experimentos. O Capítulo 8 apresenta os resultados e análise do desempenho e, por fim, o Capítulo 9 apresenta as conclusões obtidas neste trabalho e sugestões para possíveis trabalhos futuros.

2. CAPÍTULO 2 - GRAFOS

Este capítulo apresenta algumas das definições básicas da teoria de grafos que serão úteis para entender o restante do trabalho, tais como o problema do menor caminho, o conceito de centralidade em grafos e algumas das medidas de centralidade existentes.

2.1 Conceitos básicos de grafos

Um grafo $G = (V, E)$ é uma estrutura matemática constituída de um conjunto V, finito e não vazio de *n* vértices, e um conjunto E de *m* arestas, que são pares de elementos de V (Nicoletti et al., 2011).

Em um grafo *G*, dois vértices *v1* e *v2* são *adjacentes* (ou vizinhos) se existe uma aresta *e1*= (*v1*, *v2*) em *E*. O número de arestas que incide em um vértice é denominado *grau do vértice*. A Figura 2.1 apresenta um exemplo de um grafo com cinco vértices e sete arestas.

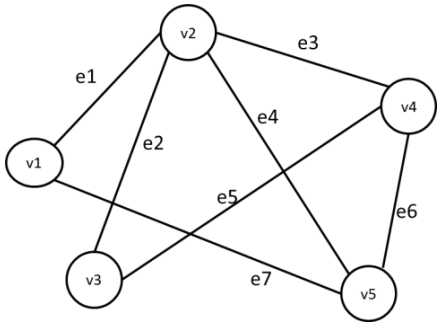


Figura 2.1 Exemplo de grafo com cinco vértices e sete arestas

Fonte: Elaborada pelo autor

Uma aresta é chamada de dirigida (orientada) se for representada por um par ordenado *v1* e *v2* $\in V$. Uma aresta é chamada de não dirigida (não orientada) se for representada por um par de vértices com sentido não definido. A Figura 2.2(a) demostra um grafo cujas arestas são dirigidas e a Figura 2.2(b) com arestas não dirigidas.

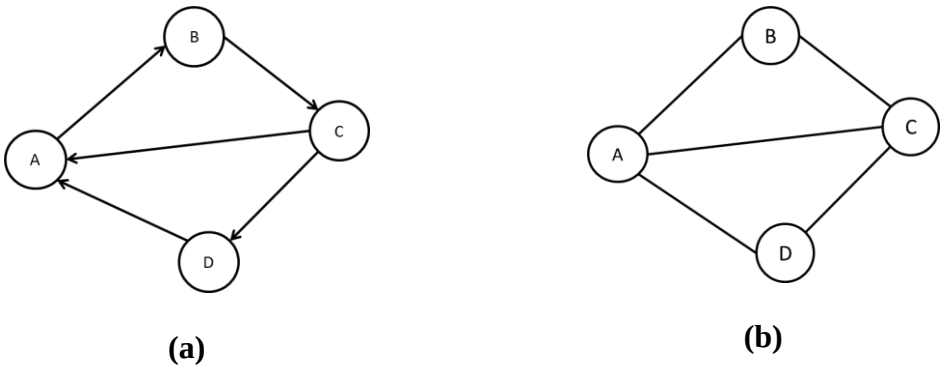


Figura 2.2 Exemplo de um grafo dirigido e não dirigido Fonte: Elaborada pelo autor

Um grafo é chamado de ponderado se cada aresta $e \in E$ tiver um número associado a ela, chamado peso de e, representado como $p(e)$. A soma dos pesos de todas as arestas de um grafo determina o peso dele, notado por $p(G)$. A Figura 2.3 ilustra um grafo ponderado de cinco vértices.

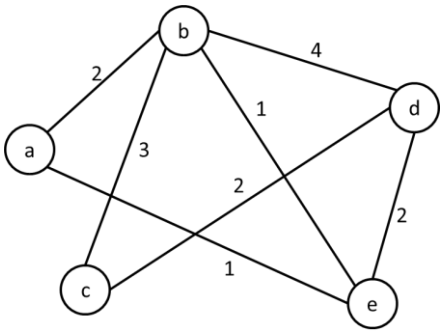


Figura 2.3 Grafo ponderado de cinco vértices Fonte: Elaborada pelo autor

Um caminho entre dois vértices *v1* e *vk* é uma sequência de vértices de G. Um caminho em um grafo pode ser representado pela sequência: $v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k$, onde $ei=(v_{i-1},v_i)$.. Se existe um caminho entre os vértices *v1* e *vk* de um grafo, diz-se que *v1* alcança *vk*. Se um vértice de um grafo alcança todos os demais, então o grafo é conexo, caso contrário, é desconexo (Bondy & Murty, 1976).

2.2 Representações computacionais de um grafo

Desde o ponto de vista computacional, grafos são comumente representados usando listas ou matrizes (Cormen et al., 2004). A seguir são apresentadas algumas das formas de representação mais usadas.

- **Lista de Adjacência**: os vértices são armazenados como registros e cada vértice armazena uma lista de vértices adjacentes. Esse tipo de estrutura de dados permite a adição de dados sobre os vértices.
- **Lista de Incidência**: os vértices e arestas são armazenados como registros. Cada vértice armazena suas arestas incidentes e cada aresta armazena seus vértices incidentes. Essa estrutura de dados permite a adição de dados sobre vértices e arestas.
- **Matriz de Adjacência**: uma matriz de duas dimensões, em que as linhas representam os vértices de origem e as colunas representam os vértices de destino. A entrada (*i*, *j*) da matríz é igual ao peso da aresta se o *i-ésimo* vértice e *o j-ésimo* vértice estão conectados e 0 em caso contrário. Os dados sobre arestas e vértices devem ser armazenados em outra estrutura.
- **Matriz de Incidência**: uma matriz bidimensional do tipo Booleana, em que as linhas representam os vértices e as colunas representam as arestas. As entradas indicam se a aresta de cada linha é incidente ao vértice de cada coluna.

A Figura 2.4 apresenta um grafo não direcionado de cinco vértices e sua respectiva representação utilizando lista de adjacência e matriz de adjacência

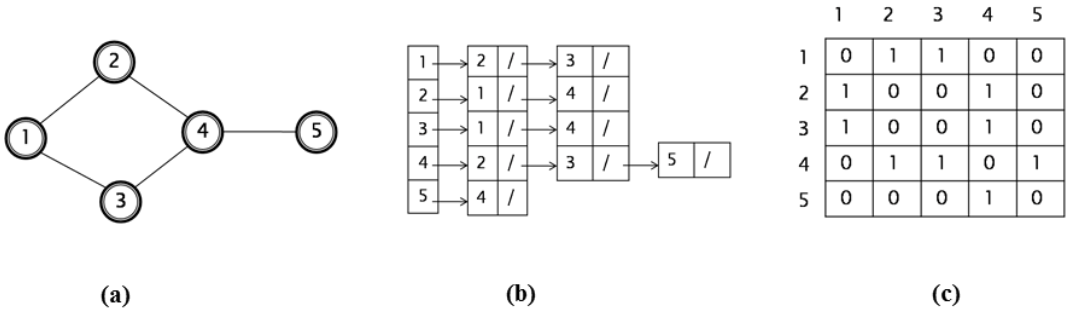


Figura 2.4 (a) Um grafo não direcionado, (b) a representação através de uma lista de adjacência, (c) a representação do grafo através de uma matriz de adjacência

Fonte: Elaborada pelo autor

2.3 O problema do menor caminho em grafos

Dado um grafo $G = (V, E)$ o menor caminho entre dois vértices *v1* e *vk* é aquele cuja soma dos pesos de suas arestas é a menor possível. A distância entre dois vértices denotada por $d(v1, vk)$ é definida pelo menor caminho que une o vértice *v1* ao vértice *vk* (Freeman, 1979).

Dado um grafo direcionado ponderado $G = (V, E)$, o peso de um caminho que inicia-se em *v1* e termina em *vk* é a soma dos pesos das arestas do caminho e definido por:

$$p(c) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$$

O caminho mais curto é, então, definido por:

$$d(v1, vk) = \begin{cases} \min\{p(c): v1 \rightarrow vk\}, & se\ existir\ um\ caminho\ de\ v1\ a\ vk; \\ \infty, & caso\ contrário \end{cases}$$

Para grafos não ponderados e não dirigidos, o peso do menor caminho é a quantidade de arestas que separa o vértice de origem do vértice de destino. O caminho mais curto do vértice *v1* ao vértice *vk* pode, portanto, ser definido como qualquer caminho *c* com peso $p(c) = d(v1, vk)$.

O problema de menor caminho pode apresentar algumas variações (Cormen et al., 2004):

- **Menor caminho a partir de uma única origem**: o problema consiste em identificar o menor caminho de um único vértice para um ou todos os outros vértices.
- **Menor caminho entre todos os pares de vértices**: o problema consiste em identificar o menor caminho entre todos os pares de vértices de um grafo.

2.4 Medidas de centralidade em grafos

O estudo de centralidade em grafos data da década de 50 no contexto de redes sociais (Galaskiewicz & Wasserman, 1993), e está relacionado ao interesse em se estimar o quão importante é um elemento participante de uma rede. Este problema aparece com bastante frequência no mundo real, como, por exemplo, ao precisar identificar locais favoráveis à implantação de uma determinada facilidade. O termo "facilidade" pode ser entendido como a localização de escolas, postos de saúde, quartéis de bombeiros militares, aterros sanitários, penitenciárias, etc.. O valor da centralidade, portanto, deve ser capaz de expressar a influência que o vértice exerce sobre os seus pares. A partir dessa motivação, várias medidas foram propostas com o objetivo de avaliar quantitativamente as propriedades que possam representar tal importância (Freeman, 1979).

A centralidade de intermediação (***betweeness***) é baseada na frequência em que um vértice pertence ao menor caminho entre os outros vértices do grafo. Segundo essa definição, para se determinar a centralidade de intermediação de um vértice v_k num grafo com n vértices, tudo o que precisamos fazer é somar todas as intermediações parciais de v_k. Vale destacar que para grafos enormes, tanto o procedimento de identificação quanto o de contagem dos menores caminhos exigem um custo computacional considerável.

A centralidade de intermediação é talvez a mais frequentemente utilizada, mas não é a única baseada no cálculo de menores caminhos. A centralidade de proximidade (***closeness***) é baseada na soma das distâncias de um vértice em relação aos demais vértices do grafo. O inverso dessa soma fornece a noção de *closeness*, ou seja, uma medida da proximidade desse vértice em relação aos outros: quanto maior o valor do *closeness* mais central é o vértice no grafo. A centralidade de *closeness* só é mensurável em grafos conexos, uma vez que em grafos não conexos existe pelo menos um par de vértices v e u tal que $d(v, u) = \infty$. *N*esse caso, não há um caminho entre os vértices v e u e, portanto, o valor calculado para a centralidade de *closeness* no vértice v não representaria a sua real relevância no grafo.

A centralidade baseada no grau de um vértice (***degree***) diz respeito à quantidade de vértices adjacentes que o mesmo possui, ou a quantidade de vértices alcançáveis por um caminho de comprimento igual a 1. Esta medida de centralidade baseia-se na

percepção de que quanto maior o grau de um vértice, menor será sua distância média até os outros vértices do grafo.

Semelhante à centralidade de *closeness*, existe a centralidade baseada na excentricidade (***eccentricity***) de um vértice do grafo. (Hage & Harary, 1995). O cálculo da excentricidade é baseado na solução do problema do menor caminho entre todos os pares de vértices. A excentricidade de um vértice v é o valor máximo dos menores caminhos entre v e todos os demais vértices do grafo. Assim, denotamos a excentricidade de um vértice por: $e(v) = \max\{d(v,u)\ \forall\ u\ \in V\}$.

O diâmetro do grafo é definido como o valor máximo das excentricidades de todos os vértices. Assim, $D(g) = \max\{e(v)\ \forall\ v\ \in V\}$. O raio do grafo é o mínimo dos valores de excentricidade. Desse modo, o raio é denotado por $R(g) = \min\{e(v)\ \forall\ v\ \in V\}$. Os vértices cuja excentricidade é igual ao diâmetro são considerados vértices periféricos do grafo. Os vértices cuja excentricidade é igual ao raio são considerados vértices centrais do grafo (Nicoletti et al., 2011).

A Figura 2.5 mostra um grafo G conexo e não dirigido com vértices V = {v1, v2, v3, v4, v5} e arestas E = {e1, e2, e3, e4, e5, e6, e7}. Para calcular a excentricidade de cada vértice é necessário encontrar o menor caminho entre todos os pares de vértices do grafo. Considerando o grafo G da Figura 2.5, a **Erro! Fonte de referência não ncontrada.** contém as menores distâncias entre todos os vértices e suas respectivas excentricidades.

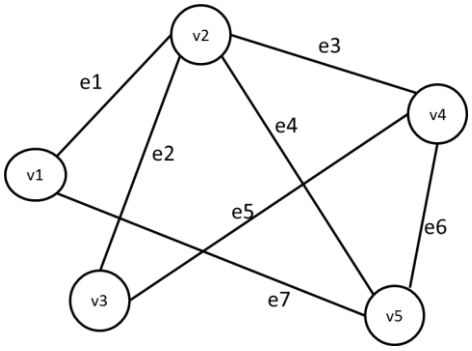


Figura 2.5 Exemplo de um grafo conexo e não dirigido com vértices

Fonte: Elaborada pelo autor

Tabela 2.1 Menores distâncias entre todos os vértices e excentricidade de cada vértice
Fonte: Elaborada pelo autor

v1 $v1 \rightarrow v2 = 1$ $v1 \rightarrow v3 = 2$ $v1 \rightarrow v4 = 2$ $v1 \rightarrow v5 = 1$	*v2* $v2 \rightarrow v1 = 1$ $v2 \rightarrow v3 = 1$ $v2 \rightarrow v4 = 1$ $v2 \rightarrow v5 = 1$	*v3* $v3 \rightarrow v1 = 2$ $v3 \rightarrow v2 = 1$ $v3 \rightarrow v4 = 1$ $v3 \rightarrow v5 = 2$
v4 $v4 \rightarrow v1 = 2$ $v4 \rightarrow v2 = 1$ $v4 \rightarrow v3 = 1$ $v4 \rightarrow v5 = 1$	*v5* $v5 \rightarrow v1 = 1$ $v5 \rightarrow v2 = 1$ $v5 \rightarrow v3 = 2$ $v5 \rightarrow v4 = 1$	$e(v1) = 2$ $e(v2) = 1$ $e(v3) = 2$ $e(v4) = 2$ $e(v5) = 2$

De acordo com a Tabela 2.1, o conjunto das excentricidades dos vértices do grafo *G* é: $e = (2, 1, 2, 2, 2)$. Diante desses dados, é possível definir as medidas de centralidade como diâmetro e raio. O diâmetro *D* é o máximo valor do conjunto *e*, ou seja, $D(G) = 2$. Por sua vez, o raio é o mínimo valor do conjunto, ou seja, $R(G) = 1$. O centro do grafo é formado por todos os vértices que possuem excentricidade igual ao raio, neste caso, formado pelo vértice *v2*. A periferia é formada por todos os vértices que possuem excentricidade igual ao diâmetro e, nesse caso, é formada pelos vértices *v1, v3, v4 e v5*.

Neste trabalho utilizaremos o cálculo do diâmetro e raio de um grafo como base para a comparação entre as tecnologias escolhidas. Como mostrado acima, estes cálculos dependem do cálculo da distância entre todos os pares de vértices do grafo, um problema computacionalmente complexo que nos permitirá avaliar os pontos fortes e fracos de cada solução comparada.

2.5 Considerações finais

Neste capítulo foram apresentados os principais conceitos sobre grafos que darão suporte a este trabalho. Definimos o problema do menor caminho e algumas medidas de centralidade em grafos. Por fim, também é apresentada a métrica conhecida como excentricidade, que será utilizada no restante deste trabalho.

O próximo capítulo apresenta conceitos sobre banco de dados e o modelo relacional e informações acerca das arquiteturas paralelas, paralelismo em consultas e

particionamento de dados. Também é apresentado o banco de dados paralelo escolhido para realização deste trabalho.

3. CAPÍTULO 3 - BANCOS DE DADOS

Neste capítulo são apresentados alguns dos conceitos mais relevantes sobre bancos de dados relacionais. Mostram-se também alguns tipos de arquiteturas paralelas para bancos de dados relacionais, o funcionamento do paralelismo nas consultas e algumas estratégias para o particionamento dos dados em um banco de dados paralelo. Por fim, descreveremos brevemente o banco de dados maciçamente paralelo escolhido para este trabalho.

3.1 Conceitos

Um banco de dados é definido como uma coleção organizada de dados (Elmasri & Navathe, 2007). A finalidade de um banco de dados é de armazenamento e organização dos dados referentes a uma determinada área. Embora seja comum utilizar o termo banco de dados para se referir a todo o sistema de banco de dados, a expressão diz respeito apenas aos dados.

O sistema de gerência de banco de dados (SGBD) é um software que permite a definição, criação, consulta, alteração e administração de bancos de dados. Em essência, o SGBD é um software que permite que usuários e outras aplicações possam interagir com um banco de dados.

Outra importante função de um SGBD é o gerenciamento de transações. Uma transação pode ser definida como um conjunto de operações que possuem uma função específica para a aplicação do sistema de banco de dados, em outras palavras uma transação representa um conjunto de operações de leitura ou escrita que são realizadas no banco de dados. (Özsu & Valduriez, 2011). A execução de transações em um SGBD deve obedecer algumas propriedades a fim de garantir o correto funcionamento do sistema e a respectiva consistência dos dados. Estas propriedades são chamadas de propriedades ACID e são definidas a seguir:

- **Atomicidade**: cada transação deve ser executada por completo, caso contrário o banco de dados deve permanecer inalterado;

- **Consistência**: cada transação deve levar o banco de dados de um estado consistente a outro;
- **Isolamento**: a execução de várias transações simultâneas deve ser equivalente à execução serial em alguma ordem dessas mesmas transações;
- **Durabilidade**: as alterações feitas por uma transação concluída devem ser preservadas mesmo em caso de falha no sistema.

Diversos modelos de dados foram propostos desde o surgimento dos SGBDs, os quais se diferenciam pelos conceitos adotados para representação dos dados. Inicialmente foram propostos os modelos hierárquicos e de rede. No modelo hierárquico os dados são representados como registros, organizados em árvores e relacionados por meio de associações do tipo pai-filho. O modelo em rede foi proposto como uma extensão ao modelo hierárquico, onde não existe o conceito de hierarquia e um mesmo registro pode estar envolvido em várias associações (Elmasri & Navathe, 2007).

O modelo relacional, proposto por Codd (Codd, 1970) consolidou-se como substituto natural do modelo hierárquico. O modelo relaciona usa a noção de bancos de dados separados em tabelas, também denomida “relação”, nela, cada coluna representa um campo ou atributo da relação, e cada linha representa um registro. Cada registro é identificado de forma única mediante “chaves primárias”, que são formadas por um sobconjunto dos atributos do registro que garantem sua unicidade. As tabelas podem ser relacionadas ou ligadas umas as outras mediante o uso das chaves primárias.

Alguns SGBDs do modelo relacional suportam uma linguagem de consulta, que permite pesquisar e derivar informações a partir dos dados armazenados no banco de dados. No caso dos SGBDs relacionais, a linguagem mais popular para consultas é a denominada *Structured Query Language* (SQL).

3.2 Arquiteturas paralelas para Banco de Dados

Um SGBD paralelo é aquele que utiliza o processamento paralelo para melhorar seu desempenho, sendo capaz de executar um grande número de transações simultâneas utilizando vários processadores.

A arquitetura de um SGBD paralela deve levar em consideração os pontos listados a seguir: (Özsu & Valduriez, 2011):

- **Comunicação**: quando existem vários dispositivos combinados para processar uma tarefa comum, eles trocam diversos tipos de informação, tais como resultados intermediários e mensagens de sincronização;
- **Vazão do disco**: um ponto crítico no processamento de dados de um banco de dados é o gargalo de *E/S* (entrada e saída de dados) imposto pela velocidade dos discos;
- **Capacidade de processamento**: para consultas complexas o problema pode não ser mais o gargalo de E/S, mas sim a própria capacidade de processamento.

A arquitetura de um SGBD paralelo deve ter por objetivo explorar vários nós computacionais para aumentar a vazão do disco e a capacidade de processamento sem permitir que a comunicação entre eles introduza novos gargalos no sistema. É desejável, também, que a distribuição de carga seja uniforme entre os diversos elementos para maximizar o ganho de processamento. Stonebraker sugere as seguintes classificações para os bancos de dados paralelos (Stonebraker, 1986).

- **Memória Compartilhada (Shared Memory):** É uma arquitetura fortemente acoplada em que todos os processadores, dentro de um único sistema, possuem acesso à mesma memória, tal como apresentado na Figura 3.1. Esta arquitetura é extremamente eficiente na comunicação entre os processadores, uma vez que todos eles podem acessar ao catálogo de dados, porém, têm o alto custo devido à complexidade da interconexão entre os processadores e os módulos de memória. Dentre os exemplos mais conhecidos, destacam-se os mainframes (IBM3090, Bull's DPS8) e multiprocessadores simétricos (Sequent, Encore).

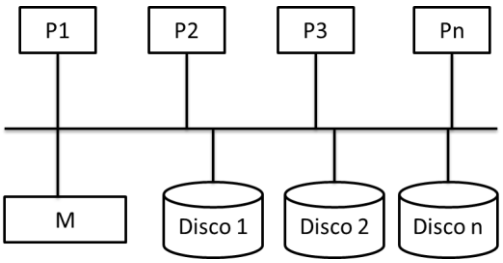


Figura 3.1 Arquitetura de memória compartilhada

Fonte: Elaborada pelo autor

- **Compartilhamento de Disco (Shared Disk):** É uma arquitetura de baixo acoplamento em que os processadores compartilham um conjunto comum de unidades de discos, como mostrado na Figura 3.2 A memória é acessada somente pelo respectivo processador. Essa arquitetura se destaca pelas vantagens de baixo custo, alta escalabilidade e disponibilidade por facilitar a migração de sistemas centralizados. Sua principal desvantagem decorre da complexidade computacional para sincronizar os dados entre os discos. Há também a possibilidade de uma sobrecarga no tráfego da rede, uma vez que os dados devem ser transferidos dos discos para as memórias locais de cada processador sempre que acessados. Alguns computadores construídos de acordo com essa arquitetura são o *IBM Parallel Sysplex* e *VAX/VMS Clusters* rodando *Oracle Rdb*.

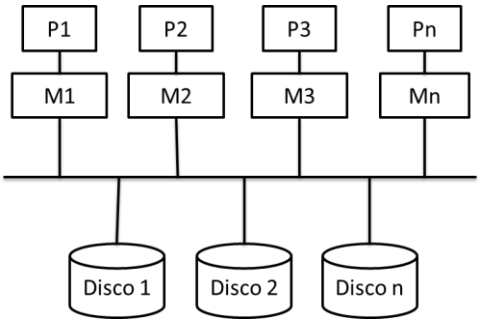


Figura 3.2 Arquitetura de compartilhamento de disco Fonte: Elaborada pelo autor

- **Sistema sem compartilhamento (Shared Nothing):** Nesta arquitetura, o processador tem acesso exclusivo à sua memória principal e sua unidade de disco, como mostrado na Figura 3.3. A comunicação é realizada entre os nós através de troca de mensagens. Essa arquitetura é também conhecida como Processamento Maciçamente Paralelo (*Massively Parallel Processing - MPP*), pois cada nó pode ser visto como um computador independente, com a sua própria base de dados e *software*. As grandes virtudes dessa arquitetura são: baixo custo, alta disponibilidade e escalabilidade. Entretanto, nela a comunicação entre processadores é o fator limitante, devido à necessidade de acesso a dados não locais. Dentre os exemplos mais conhecidos destacam-se: *ParAccel*

Analytic Platform, *Teradata Aster Data*, *Pivotal Greenplum Database* e *HP Vertica*.

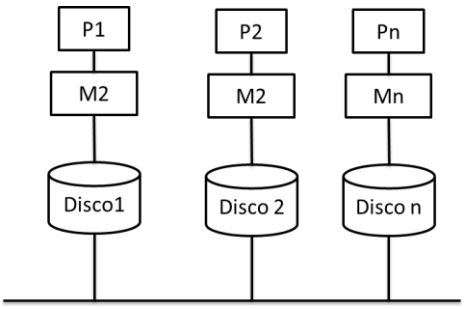


Figura 3.3 Arquitetura de sistema sem compartilhamento

Fonte: Elaborada pelo autor

Para pequenos sistemas de banco de dados paralelos (menos de 20 processadores), a arquitetura de memória compartilhada pode oferecer melhor desempenho em virtude de um melhor balanceamento de carga. Entretanto, as arquiteturas de disco compartilhado e sem compartilhamento superam o modelo de memória compartilhada tanto em disponibilidade como em escalabilidade, sendo que a arquitetura sem compartilhamento consegue maior grau de expansão que as outras duas arquiteturas (Ray, 2009).

3.3 Paralelismo de consultas em SGBD paralelo

A execução de consultas em um SGBD paralelo pode explorar três níveis de paralelismo: interconsultas, intraconsultas e paralelismo de operador (Özsu & Valduriez, 2011). O paralelismo interconsultas permite a execução concorrente de múltiplas consultas geradas por transações de diversos usuários. A sua principal aplicação é melhorar o sistema de processamento de transações. Os processadores têm de realizar algumas tarefas como bloqueio e *log* (registro diário), de forma coordenada, com o objetivo de assegurar que dois processadores não atualizem o mesmo dado, de modo independente, ao mesmo tempo.

O paralelismo intraconsultas consiste na execução paralela das diferentes operações que compõem uma única consulta para diminuir seu tempo de resposta. Nesse tipo de paralelismo, os planos de execução são criados em forma de árvores e várias subárvores que podem ser executadas em paralelo.

O paralelismo de operador é alcançado através da distribuição dos dados do banco de dados processadores disponíveis para o SGBD paralelo, assim, as operações podem ser executadas simultaneamente sobre os fragmentos dos dados (DeWitt & Gray, 1992). Há duas maneiras de paralelismo de operador:

- **Intraoperador**: Os algoritmos de cada operação podem ser paralelizados, permitindo que excecuções complexas sejam realizadas por inúmeros processadores, reduzindo o tempo de resposta de uma operação. A dificuldade está na necessidade de desenvolver novos algoritmos e otimizá-los para cada tipo de arquitetura.
- **Interoperador**: Uma consulta é dividida em várias operações simples e cada parte alocada em processadores diferentes. Desse modo, o tempo de resposta da consulta pode ser reduzido. O paralelismo interoperação pode ser implantado de duas formas: paralelismo independente, cujas consultas não dependem necessariamente uma das outras; e paralelismo *pipeline*, cujo resultado produzido por uma operação é redirecionado para as operações que precisam deles.

3.4 Particionamento de dados em SGBD paralelo

Em um SGBD paralelo, a distribuição dos dados deve ser realizada de modo a maximizar o desempenho do sistema, que pode ser medido pela combinação da quantidade total de trabalho realizado e o tempo de resposta das consultas individuais. O particionamento determina a divisão dos dados dentre os discos com base no critério estabelecido sobre os valores de um ou mais atributos de uma relação. A escolha do tipo de particionamento deve levar em consideração diversos fatores, como, por exemplo, os custos de inicialização e finalização do sistema, custo de comunicação, balanceamento de carga e integralização dos resultados. Existem três estratégicas básicas de particionamento de dados, apresentadas a seguir (Ray, 2009):

- **Particionamento cíclico:** Essa é a forma mais simples de particionamento dos dados. Ela consiste em distribuir os registros de forma sequencial entre as unidades de armazenamento. Para garantir a

distribuição uniforme, ela envia o *i-ésimo* registro ao disco numerado como (*i mod n*), considerando-se *n* o número de discos disponíveis.

- **Particionamento *hash*:** Neste tipo de particionamento, escolhe-se um ou mais atributos de uma relação e aplica-se uma função *hash* para distribuição dos dados. A relação é particionada de acordo com o valor da função de *hash* aplicada sobre o atributo, de forma a relacionar cada registro a uma unidade de amazenamento. Basicamente, se a função *hash* retorna *i*, o registro é alocado no disco *Di*.
- **Particionamento por faixa de valores:** Nessa forma de particionamento, os dados são agrupados de acordo com os valores de um ou mais atributos. Esse tipo de partição é adequado para situações em que os atributos de particionamento apresentam valores contínuos, como por exemplo, tempo, mês e ano.

3.5 Pivotal Greenplum Database

Para calcular as medidas de centralidade utilizando um SGBD paralelo, escolhemos o *Pivotal Greenplum Database* (Greenplum). A escolha se deve ao fato de ele possuir uma versão aberta para testes, desenvolvimento e pesquisa. O *Greenplum* é uma solução construída para analisar e armazenar grandes volumes de dados. É um banco de dados de processamento paralelo e usa uma arquitetura sem compartilhamento (EMC2, 2012).

O Greenplum é capaz de lidar com o armazenamento e processamento de grandes quantidades de dados, distribuindo a carga entre vários computadores, chamados de segmentos. Nessa arquitetura, cada segmento age como um sistema independente. O *Greenplum* distribui os dados e paraleliza as cargas de trabalho e de consultas em todos os segmentos disponíveis, levando o sistema a ter um desempenho significativamente melhor. A arquitetura do Greenplum, ilustrada na Figura 3.4 pode ser dividida em várias camadas:

Figura 3.4 Arquitetura do Greenplum Fonte: (Pivotal Greenplum, 2014)

A camada (1) compreende as aplicações responsáveis pelas consultas e carregamento dos dados nos segmentos. A camada (2) consiste no nó principal, denominado mestre, que contém o estado de todos os segmentos. O mestre armazena um catálogo global constituído pelo conjunto de tabelas do sistema e possui metadados sobre o banco de dados em si. A camada (3) é a interface de rede através da qual o *Greenplum* se comunica. Ela é chamada de *GNet Software Interconnect* e garante a comunicação rápida entre o nó mestre e todos os segmentos do da rede. A camada (4) contém todos os segmentos que estão ligados a *GNet Software Interconnect* formando um *cluster* de computadores. Cada segmento contém uma parte dos dados. A camada (5) é fornecida para o carregamento de dados em paralelo a partir de fontes externas.

Para garantir a distribuição de dados entre os segmentos, o *Greenplum* utiliza uma função *hash* para particionamento dos dados. O Greenplum suporta também o particionamento por faixa de valores (divisão de dados com base em um intervalo numérico, como data ou preço), ou particionamento por lista valores (divisão de dados com base em uma lista de valores, tais como o território de vendas ou linha de produto), ou uma combinação de ambos os tipos (EMC2, 2012).

Além das operações típicas de banco de dados, (seleções, junções e uniões), o *Greenplum* tem um tipo de operação adicional chamado “movimento”. Uma operação de movimento envolve mover tuplas entre os segmentos durante o processamento das consultas. Para atingir o paralelismo máximo durante a execução da consulta, o *Greenplum* entrega a cada segmento uma cópia do plano de consulta, permitindo a execução paralela. O plano de consultas também é dividido em fatias. A fatia é uma

parte do plano que pode ser trabalhada de forma independente. Um plano de consulta é fatiado sempre que uma operação de movimento ocorre, como mostra a Figura 3.5.

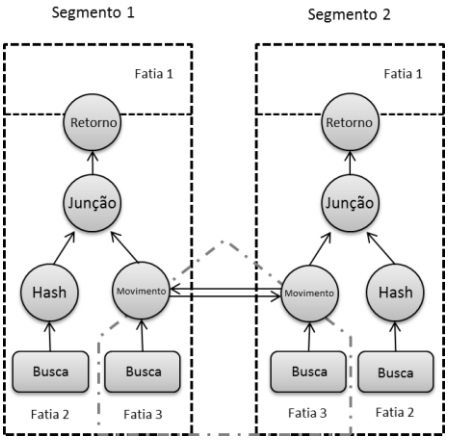


Figura 3.5 Fatiamento do Plano de Consulta Fonte: (Pivotal Greenplum, 2014)

O *Greenplum* cria alguns processos para lidar com o trabalho de uma consulta. No nó mestre, o processo de trabalho de consulta é chamado de distribuidor de consulta (DQ). O DQ é responsável pela criação e envio do plano de consulta, agregar e apresentar os resultados finais. Em cada nó segmento é criado um processo de trabalho chamado de "executor de consulta" (EC). O EC é responsável por completar a sua parte do trabalho e comunicar seus resultados intermediários para outros processos ou para o distribuidor de consulta.

3.6 Considerações finais

Este capítulo apresentou os conceitos sobre banco de dados e o modelo relacional. Apresentaram-se as diferentes arquiteturas paralelas para os bancos de dados existentes, além das estratégias disponíveis para paralelizar consultas e particionar os dados entre os nós computacionais que compõem o sistema. Por fim, introduzimos o *Pivotal Greenplum Database*, o banco de dados paralelo que escolhemos para realização deste trabalho.

O próximo capítulo apresenta os bancos de dados *NoSQL*, orientado a grafos e descreve o banco de dados *Neo4J*, escolhido para realização deste trabalho.

4. CAPÍTULO 4 - BANCOS DE DADOS NoSQL

Este capítulo apresenta um breve histórico dos bancos de dados *NoSQL* e os classifica de acordo com as suas características mais relevantes. O capítulo aprofunda também no funcionamento do *Neo4J*, um banco de dados orientado a grafos que foi escolhido para este trabalho.

4.1 NoSQL

A recente explosão na quantidade de dados disponíveis e na quantidade de usuários que geram e consomem esses dados levaram à procura de alternativas aos bancos de dados relacionais. Em geral, é consenso na comunidade científica que o modelo relacional não é adecuado para tratar com dados não estruturados ou semi-estruturados (e.g. logs, páginas web, relacionamentos em redes sociais, vídeos, etc). Para atender a essas novas exigências, soluções para gerenciamento de banco de dados começaram a serem propostas, como, por exemplo, os bancos de dados *NoSQL* (Leavitt, 2010).

Existem definições muito diferentes para *NoSQL*. Em (Strozzi, 1998) foi usado o termo *NoSQL* para nomear um banco de dados leve, relacional e código fonte aberto que não possui uma interface *SQL*, mas usa vários comandos *UNIX* para consultar os dados. O termo foi reintroduzido em 2009 durante um evento denominado *NoSQL Meetup 2009*, cujo objetivo foi discutir bancos de dados distribuídos de código aberto. Desta vez, porém, o termo não se referia a um sistema particular ou à própria linguagem de consulta. Considerando que o termo, muitas vezes, leva a acreditar que os fabricantes são contra *SQL*, alguns autores entendem que a sigla significa “Não apenas *SQL*” (*Not Only SQL*).

Adequado ou não, o termo *NoSQL* descreve o crescente número de bancos de dados não relacionais que surgiram recentemente. Esses sistemas também são definidos como uma nova geração de sistemas de banco de dados que têm, pelo menos, algumas das seguintes propriedades (Cattel, 2010):

- O modelo de dados não é relacional, e em alguns casos, há ausência de esquema ou esquema flexível que permite tipos de dados variados, complexos e/ou semiestruturados;
- Escalabilidade em troca das propriedades ACID. Esses sistemas garantem a "consistência eventual" ou somente dentro de um único objeto (ou registro ou documento). Consistência eventual é a garantia de que, caso nenhuma nova atualização seja feita nos dados, eventualmente, todos os acessos retornarão o último valor atualizado;
- Os sistemas são projetados para escalar horizontalmente, onde a quantidade de dados pode variar em um espaço de tempo relativamente curto, fazendo com que a estrutura de *hardware* demandada tenha que se adaptar.

Os bancos de dados *NoSQL* possuem características comuns e podem ser classificados, segundo (Stonebraker, 2010) e (Cattel, 2010), da seguinte forma:

- **Sistemas baseados em armazenamento chave-valor:** onde existe uma coleção de chaves únicas e os valores são associados a chaves. Devido a sua simplicidade, é o modelo que possui maior escalabilidade dentre os modelos classificados como *NoSQL*. Alguns bancos de dados nessa categoria são *RIAK*, *Redis* e *MemcacheDB*
- **Sistemas orientados a documento:** armazena e organiza os dados como uma coleção de documentos em vez de tabelas estruturadas. Em geral, não possuem esquema, ou seja, os documentos armazenados não precisam possuir estrutura em comum. Bancos de dados populares nessa categoria são o *MongoDB* e o *CouchDB*.
- **Sistemas orientados a coluna:** característica também presente em banco de dados do modelo relacional, ela muda o paradigma de orientação a registros (tuplas) para orientação a atributos (colunas). O valor de cada coluna é armazenado em sequência, aumentando o desempenho da leitura de uma única coluna. Com esse modelo, o banco de dados pode carregar em sua memória apenas os valores das colunas que serão utilizados, evitando preenchê-la com dados que não serão utilizados. Os principais bancos de dados nessa categoria são: *Cassandra*, *HBase* e *Hytable*.

- **Sistemas orientados a grafo:** os dados são armazenados em vértices de um grafo cujas arestas representam o tipo de associação entre esses vértices. A ideia desse modelo é representar os dados e/ou o esquema dos dados como grafos, ou como estruturas que generalizem a noção de grafos. Diferentemente de outros tipos de bancos de dados *NoSQL*, os bancos de dados orientados a grafos são livres de indexação de adjacência, isso significa que todo elemento contém uma ligação direta com seus elementos relacionados. Essa característica torna esses bancos muito eficientes em operações de busca e travessia em grafos. Exemplos de bancos de dados nessa categoria são o *Neo4j* e *InfoGrid*.

4.2 Neo4J

O Neo4j é um banco de dados orientado a grafos escrito em Java (neo4j.org, 2014). Ele é categorizado como um banco de dados de navegação, o que significa que a navegação de um nó para outro é realizada através de um relacionamento. A *travessia* em um grafo significa visitar os nós pertencentes ao escopo da consulta de acordo com as condições impostas. (Neo Technology, 2006).

No Neo4j, vértices adicionados ao banco são chamados de nós e as arestas são nomeadas de "relacionamentos". Os valores atribuídos aos nós e relacionamentos são denominados de propriedades. Os nós, os relacionamentos e as propriedades são os blocos de construção do modelo de dados. Uma relação conecta dois nós e, opcionalmente, pode ser dirigida ou não dirigida. Propriedades são pares de chave-valor que são anexadas tanto aos nós quanto às relações, e podem ser um valor unitário ou uma matriz de valores. Combinando os nós, as relações entre eles e suas propriedades formam uma rede, que representa os dados do grafo. Os principais elementos arquiteturais do Neo4J e a relação entre eles são mostrados a seguir:

- *Graph Database (Neo4j)* controla um *grafo* e os *índices;*
- Um *grafo* registra seus dados em *vértices* que possuem *propriedades;*
- *Vértices* são organizados por *relações* que também possuem *propriedades;*
- *Travessia* navega por um *grafo* e identifica os caminhos com *vértices* ordenados;
- *Índices* mapeiam propriedades dos *vértices* ou *relações.*

O armazenamento dos dados no Neo4J é realizado em um único diretório. Ele pode também ser implantado de forma distribuída em um cluster de computadores, replicando os dados em todas as instâncias pertencentes ao *cluster*. Denominado Neo4j *High Availability (HA)*, modelo de replicação de dados distribuído (mestre-escravo) é demonstrado na Figura 4.1.

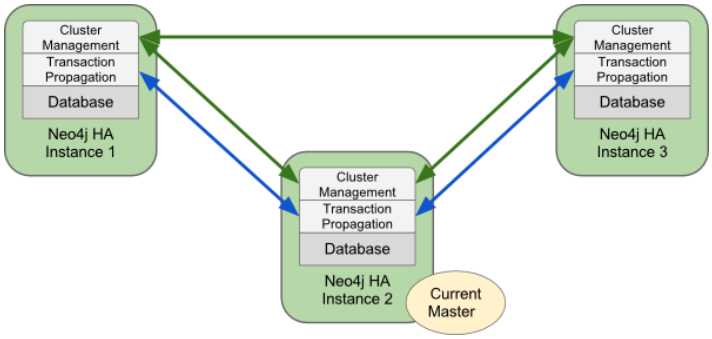


Figura 4.1 Arquitetura distribuída - Neo4J Fonte: (neo4j.org, 2014)

Na opção Neo4J HA, cada instância de banco de dados contém a lógica necessária para o gerenciamento e comunicação com os outros membros do cluster. Cada instância possui o próprio banco de dados, o gerenciador do cluster (*Cluster Management*) e o propagador de transações (*Transaction Propagation*). O componente gerenciador de cluster permanece ativo e conectado às instâncias. Ele controla e monitora quais outras instâncias estão disponíveis e/ou indisponíveis.

Quando o nó mestre apresenta algum problema como, por exemplo, falha de *hardware* ou interrupções na rede, automaticamente, o gerenciador de *cluster* utiliza um algoritmo de eleição que escolhe de forma aleatória e dinâmica outra instância para assumir a posição do nó mestre. Durante o processo de escolha, caso haja algum processo de escrita mal sucedido, a transação será revertida e novas operações serão bloqueadas até que um novo mestre esteja disponível. O componente propagador de transações é responsável pela replicação dos dados entre todas as instâncias no *cluster*. Quando uma operação de escrita for bem sucedida no master, ela será automaticamente sincronizada com as outras instâncias. Os nós escravos também podem ser configurados para que a sincronização ocorra de forma assíncrona, para isso, basta configurar o intervalo de tempo de sincronização dos dados (neo4j.org, 2014).

Neo4j não suporta consultas declarativas, *Stored Procedures* e *Triggers*, funcionalidades comumente achadas em bancos de dados relacionais. Em contrapartida, ele oference uma *API* em Java. Esta *API*, além de oferecer suporte à criação da estrutura do banco, permite a adição de nós, suas propriedades e seus relacionamentos. A *API* também oferece diversas classes para realizar a *travessia* entre os nós do grafo, possibilitando a recuperação dos dados persistidos. As principais classes utilizadas em uma *travessia* oferecida pela *API* do Neo4J são:

- ***Expanders***: define o critério da *travessia*, geralmente em termos de direção e tipo relacionamento;
- ***Order***: por exemplo, em profundidade ou em largura;
- ***Uniqueness***: a visita a um nó deve ser feita apenas uma vez;
- ***Evaluator***: o critério para continuar ou parar a travessia;
- ***Starting*** node: onde a *travessia* começará.

No Neo4J provê indexação dos vértices e arestas através da integração com o *framework Lucene*, uma biblioteca de busca e indexação de documentos, escrita na linguagem de programação Java (Lucene, 2014).

O Neo4J também inclui o *framework Cypher Query Language* (CQL), que permite escrever consultas através de uma linguagem formal. A CQL permite cláusulas encadeadas que se alimentam de resultados intermediários, recursos muito comuns aos bancos de dados relacionais e ao SQL. As cláusulas mais utilizadas para consultas são *match* e *where*. A cláusula *match* é usada para descrever a estrutura da pesquisa, principalmente, com base em relacionamentos, e a cláusula *where* é usada para adicionar restrições adicionais à consulta.

4.3 Considerações Finais

Neste capítulo, é apresentado brevemente os bancos de dados *NoSQL*, suas características e propriedades. Também é apresentado o banco de dados orientado a grafos Neo4j. O próximo capítulo apresenta o modelo *MapReduce* e o *framework Hadoop*.

5. CAPÍTULO 5 – O MODELO MAPREDUCE

Neste capítulo são apresentados e discutidos os principais conceitos sobre o modelo *MapReduce* e sua implementação mais conhecida, o *framework* Hadoop.

5.1 O Modelo *MapReduce*

O *MapReduce* (MR) é um modelo de programação criado para processar grandes conjuntos de dados em paralelo usando *clusters* de computadores. Do ponto de vista de programação, o modelo MR consiste em duas fases, apresentadas na Figura 5.1:

- *Map* – fase inicial, em que os dados individuais de entrada são processados em paralelo, sendo responsável por transformar a entrada em pares de registros <chave, valor>.
- *Reduce* – fase que faz o resumo e agregação, onde todos os registros gerados pelo *Map* com a mesma chave são agrupados e processados por uma única função.

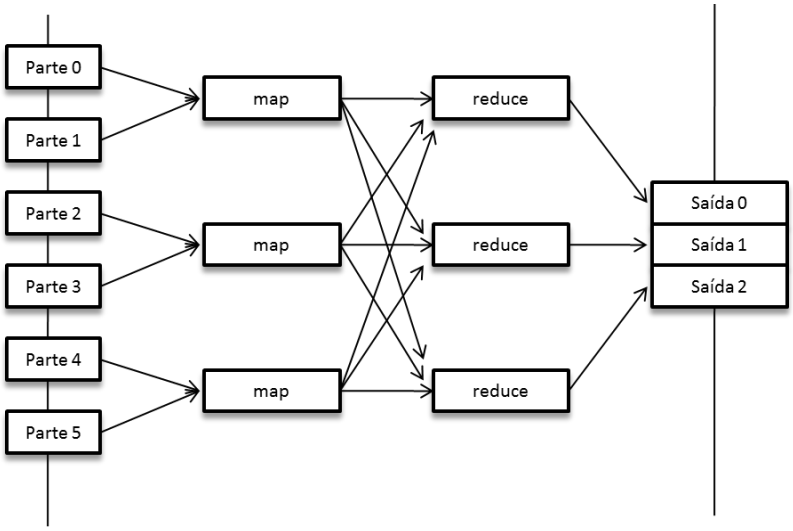


Figura 5.1 Modelo MapReduce Fonte: Elaborada pelo autor

Além da execução das funções *map* e *reduce* criadas pelo programador, quatro outras etapas são características da execução de uma aplicação *MapReduce*. Elas são conhecidas como *Input splitting, Shuffle, Sort e Combine*.

A etapa *Input splitting* consiste na leitura e divisão dos dados de entrada em partes iguais chamadas de *input split*, que são então utilizadas como entradas para uma invocação da função *map*. A etapa *Shuffle*, realizada a partir do momento em que cada tarefa *map* passa a produzir pares intermediários, é dividida em dois passos distintos: *particionamento* e *ordenamento*. No particionamento, os pares chave/valor são divididos em *R* partições, onde *R* corresponde à quantidade de instâncias da função *reduce* que serão executadas. Já na etapa de ordenamento, as chaves pertencentes a uma mesma partição são ordenadas para facilitar o processamento.

A etapa *Combine* consiste em um pré-processamento das listas de valores geradas pelas funções *map*. Esta etapa tem por objetivo reduzir o volume de comunicação entre os nós do *cluster*, agregando todos os valores associados a uma chave. Na maioria dos casos, a própria função *reduce* desempenha este papel.

```
Function Map(Integer chave, String valor)
      #chave: número da linha do arquivo
      #valor: texto da linha correspondente
      listaDePalavras = split(valor)
      for palavra in listaDePalavras:
            emit (palavra, 1)

Function Reduce(String chave, IntegerIterator valores)
      #chave: palavra emitida pela função map
      #valor: conjunto de valores emitido para a chave
      total = 0
      for v in valores:
            total = total + 1
      emit (chave, total)
```

Figura 5.2 Pseudocógido para aplicação contadora de palavras implementada usando o modelo MapReduce Fonte: Elaborada pelo autor

No exemplo do algoritmo da Figura 5.2, a função *reduce* conta a quantidade de elementos na lista de valores recebida como entrada (Karloff et al., 2010). Para auxiliar o entendimento, a Figura 5.3 mostra um exemplo concreto de funcionamento do código apresentado na Figura 5.2.

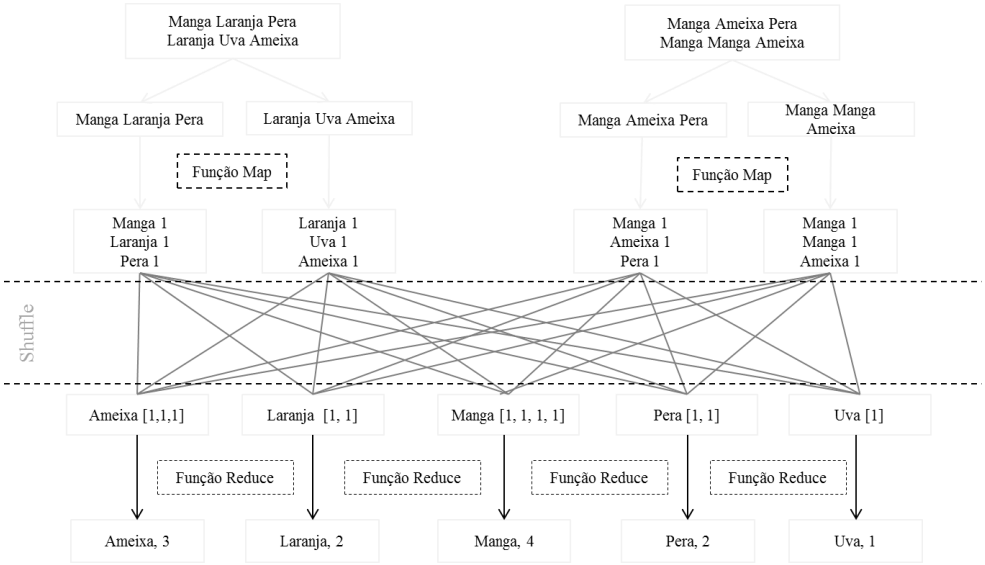


Figura 5.3 Exemplo de um contador de palavras usando o modelo MapReduce

Fonte: Elaborada pelo autor

5.2 Backup Tasks

Na proposta original do *MapReduce*, seus criadores identificaram um possível problema de desempenho causado por máquinas cuja performance está aquém dos demais nós do *cluster*. Essas máquinas são denominadas *stragglers*. Uma máquina pode tornar-se lenta por diversos problemas de *hardware* e *software*. Neste caso, as tarefas executadas em um *straggler* atrasam o processamento como um todo.

Para contornar esse problema no *MapReduce*, foram introduzidas as *Backup Tasks*, que consistem no procedimento de criar cópias em outras máquinas das tarefas em andamento e usar o resultado daquela que termina primeiro. Quando qualquer uma das cópias de uma tarefa é finalizada com sucesso, as demais são encerradas.

Para identificar as máquinas *straggler*, cada nó do *cluster* envia seu progresso e outras informações através do sinal de heartbeat (White, 2012). Para tarefas *map*, o progresso é calculado através da fração de dados já processados, enquanto que nas tarefas *reduce*, cada fase (cópia, ordenamento e redução) representa um terço do tempo de processamento.

5.3 Hadoop

Hadoop é uma plataforma de código livre, implementada em Java, cujo objetivo é executar programas desenvolvidos de acordo com o modelo *MapReduce*. Mantido e distribuído pela *Apache Foundation*, é uma implementação utilizada por um grande número de empresas, tais como Amazon, Adobe, Ebay, Facebook, Google, IBM, dentre outras. O projeto tem por objetivo "desenvolver software de código aberto para computação confiável, escalável e distribuída" (Hadoop, 2014).

O Hadoop inclui os seguintes módulos:

- ***Hadoop Common***: um conjunto de utilidades que fornece suporte aos outros subprojetos do Hadoop. Estão inclusas as bibliotecas para sistemas de arquivos, *Remote Procedure Call* (RPC) e serialização de objetos;
- ***Hadoop Distributed File System (HDFS)***: um sistema de arquivos distribuídos que oferece alto desempenho no acesso aos dados. Ele é responsável por armazenar dados de forma eficiente e tolerante a falhas. Para cumprir suas responsabilidades, ele cria múltiplas réplicas de blocos de dados e as distribui entre os nós do *cluster;*
- ***Hadoop YARN***: uma biblioteca de programação e gestão de recursos de *cluster*;
- ***Hadoop MapReduce***: um sistema baseado no Hadoop YARN para processamento paralelo de grandes conjuntos de dados.

O sistema de arquivos HDFS fornece os recursos para distribuição e redundância dos dados. A distribuição é feita em blocos de dados de tamanho configurável, criados e alocados pelo HDFS, responsáveis por armazená-los e, posteriormente, processá-los.

Os nós que executam o serviço do sistema distribuído HDFS são denominados *DataNodes*. A redundância é realizada da seguinte forma: para cada bloco alocado são criadas três réplicas distribuídas no sistema, de forma a aumentar a probabilidade de recuperação de dados em caso de falhas em algum *DataNode* (White, 2012). A arquitetura do Hadoop é apresentada de forma simplificada na

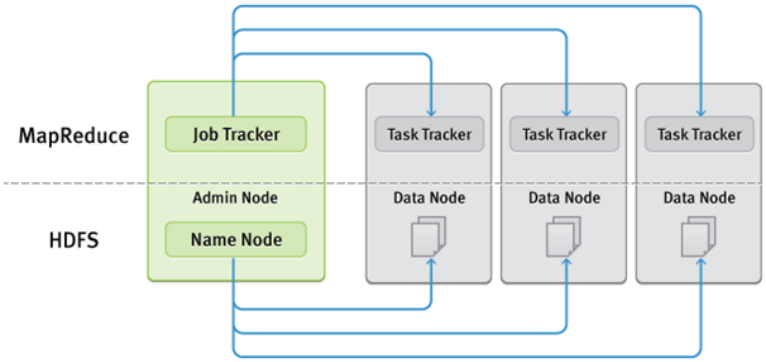


Figura 5.4 Arquitetura simplificada do Hadoop Fonte: (Hadoop, 2014)

O HDFS no Hadoop é controlado por dois nós computacionais. Eles executam os serviços denominados *NameNode* e *Secondary NameNode*. O *NameNode* tem a responsabilidade de armazenar todas informações relativas às localizações dos blocos de dados espalhados entres os *DataNodes.* O *NameNode* armazena as informações em um arquivo denominado *fsimage*. As atualizações do sistema de arquivos (adicionar/remover blocos) não são atualizadas diretamente no arquivo *fsimage*, em vez disso, são registradas em um arquivo *log*. Quando o *NameNode* é reinicialiazado, ele lê o arquivo log e aplica todas as alterações no arquivo *fsimage*, atualizando sistema de arquivos. O serviço *Secondary NameNode* tem a função de auxílio ao *NameNode* que realiza a leitura periódica das mudanças do sistema registradas no arquivo *log* e aplica no arquivo *fsimage*. Isso permite que o *NameNode* reinicialize mais rápido.

A execução das aplicações na plataforma Hadoop são realizadas pelo nó computacional denominado *JobTracker*, cuja função é gerenciar o processamento das funções alocando as tarefas *mapper* ou *reducer* nos *DataNode*, com base nas informações fornececidas pelo *NameNode*. O *JobTracker* também tem a responsabilidade de identificar falhas nos *DataNodes*, bem como realocar as tarefas atribuídas ao nó falho em outros nós *DataNode* que estejam funcionais. A comunicação

entre o *JobTracker* e os *DataNodes* é feita pelo serviço *TaskTracker*, que deve ser executado em todos os nós computacionais que mantêm o serviço *DataNode* em funcionamento.

Em linhas gerais, arquitetura da plataforma Hadoop é baseada em dois princípios: a distribuição dos dados com confiabilidade, garantida pelo HDFS (*NameNode*, *Secondary NameNode*, *DataNodes*) e a distribuição do processamento com alta disponibilidade (*JobTracker*, *TaskTrackers*) (Lin & Schatz, 2010).

5.4 Considerações Finais

Este capítulo descreve o modelo de programação *MapReduce* e o funcionamento da implementação do *framework* Hadoop. O próximo capítulo apresenta os algoritmos sequenciais e paralelos que realizam o cálculo da medida de centralidades que foi utilizado neste trabalho.

6. CAPÍTULO 6 - CÁLCULO DE MEDIDAS DE CENTRALIDADE EM GRAFOS

Este capítulo apresenta os algoritmos utilizados para representar cada uma das tecnologias comparadas neste trabalho: Hadoop, SGBD paralelo e Neo4J e a biblioteca *NetworkX*. Para calcular a centricidade de um grafo, primeiramente, é necessário resolver o problema do menor caminho entre todos os pares de vértices. A excentricidade de um vértice é o valor máximo dos todos os menores caminhos. O raio é o mínimo dos valores de excentricidade e o diâmetro do grafo é o valor máximo das excentricidades.

6.1 Biblioteca *NetworkX*

Existem vários algoritmos que tratam o problema de menor caminho e, consequentemente, resolvem o problema da centralidade em grafos. De acordo com (Del Vechio et al., 2009), a escolha mais utilizada para o cálculo de menor caminho é o algoritmo que utiliza a estratégia de busca em largura *Breadth-first search (BSF).* Ele calcula as distâncias desde o vértice de origem até todos os vértices alcançáveis. O algoritmo produz uma árvore cuja raiz é o vértice de origem e que contém todos os vértices acessíveis, tal como mostra o pseudocódigo do Algoritmo 1. O algoritmo de busca em largura é o algoritmo mais simples e utilizado para pesquisas em grafos. A execução da busca em largura envolve um alto consumo de memória para manter a estrutura de dados para armazenar os vértices.

Para controlar seu andamento, o *BSF* utiliza as cores branca, cinza ou preta para mostrar o estado de cada vértice durante a execução. Os vértices que ainda não foram visitados são marcados na cor branca, os da cor cinza são os vértices que possuem algum vértice adjacente branco. Os vértices cinza representam a fronteira entre vértices descobertos e não descobertos. Os vértices marcados como pretos são aqueles que já tiveram todos os seus vértices adjacentes descobertos.

Algoritmo 1: Algoritmo de Busca em Largura

```
Entrada: O grafo original G e o vértice fonte s
para cada v ∈ V|G| − [s] faça
    cor[u] ← BRANCO;
    d[u] ← ∞;
    ∞[u] ← NULL;
fim
cor[s] ← CINZA;
d[s] ← 0;
∞[s] ← NULL;
Q ← 0;
ENFILEIRA(Q, s);
while Q ≠ 0 do
    u ←DESENFILEIRA(Q);
    para cada v ← Adj|u| faça
        if cor[v] = BRANCO then
            cor[v] ← CINZA;
            d[v] ← d[u] + 1;
            π[v] ← u;
            ENFILEIRA(Q, s);
        end
        cor[u] ← PRETO;
    fim
end
```

O algoritmo *BSF* é a base de implementação de muitas soluções que tratam de problemas em grafos. Este algoritmo também foi utilizado como base de diversas implementações da biblioteca *NetworkX*. A *NetworkX* é de código fonte aberto, escrita em liguaguem de programação *Python*, criada para analisar dados de epidemia e propagação de doenças, estudar estruturas das redes sociais, biológicas e de redes complexas.

A *NetworkX* é categorizada como uma biblioteca de processamento de dados em memória (Hagberg et al., 2008). De fácil instação, ela já é pré-compilada para funcionar em qualquer sistema operacional que tenha a linguagem *Python* instalada.

Na *NetworkX* exite diversas funções para cálculo estatísticos de rede e métricas como o raio, diâmetro, componentes conexos, coeficiente de agrupamento, entre outros. Além disso, possui geradores sintéticos de diversos tipos de grafos. A Figura 6.1 demostra uma parte da implementação do código fonte da *NetworkX* que realiza o cálculo do raio e diâmetro de um grafo baseado na estratégia de busca em largura.

```
    for n in G.nbunch_iter(v):

        if sp is None:
            length=networkx.single_source_shortest_path_length(G,n)
            L = len(length)
        else:
            try:
                length=sp[n]
                L = len(length)
            except TypeError:
                raise networkx.NetworkXError('Format of "sp" is invalid.')
        if L != order:
            msg = "Graph not connected: infinite path length"
            raise networkx.NetworkXError(msg)

        e[n]=max(length.values())

    if v in G:
        return e[v]  # return single value
    else:
        return e
```

Figura 6.1 Código fonte que implementa a busca em largura

Fonte: (Hagberg et al., 2008)

6.2 HEDA - *Hadoop-based Exact Diameter Algorithm*

O HEDA (*Hadoop-based Exact Diameter Algorithm*) é um algoritmo baseado no modelo *MapReduce* que calcula de forma exata o raio e o diâmetro de um grafo (Nascimento & Murta, 2012).

Ele faz o cálculo do menor caminho paralelamente, a partir de todos os vértices para todos os outros vértices do grafo, utilizando o algoritmo de *Dijkstra*. A implementação do HEDA utiliza dois arquivos, tal como mostra o Algoritmo 2.

Algoritmo 2: Algoritmo HEDA - Fluxo Principal

Entrada: Caminho do Arquivo de Arestas, Arquivo de Distâncias, Caminho do Arquivo de saída

1 $iteracao \leftarrow 1$;
2 $vetorArestas \leftarrow ArquivoArestas$;
3 **while** $PossuiVerticesAProcessar$ **do**
4 $\quad caminhoSaida \leftarrow CaminhoDeSaida$;
5 $\quad EncontrarMenorCaminho(vetorArestas, arquivoDistancias, caminhoSaida)$;
6 $\quad iteracao \leftarrow iteracao + 1$;
7 **end**

O primeiro arquivo contém o conjunto de vértices e suas respectivas arestas, enquanto o segundo contém as distâncias calculadas até o momento e o estado de processamento de cada vértice, sendo eles “EM PROCESSAMENTO” e “PROCESSADO”. O arquivo de arestas, em cada linha, possui o vértice de origem seguido por TAB e sua lista de vértices adjacentes separados por vírgula. Por exemplo, a entrada 1TAB 2,5 descreve o vértice de número 1, que tem arestas que o ligam aos vértices 2 e 5. O arquivo de distâncias possui a distância entre cada par de vértices e o estado de processamento (1 para "em processamento" e 2 para "processado"). Por exemplo, a entrada 1.5TAB1|1 indica que a distância atual entre os vértices 1 e 5 é de 2, e que o processamento para esse par de vértices já terminou. A Tabela 6.1apresenta ambos os arquivos de entrada do grafo da Figura 6.2.

Tabela 6.1 Exemplo de arquivos de entrada do algoritmo HEDA (Nascimento and Murta, 2012)

Formato do arquivo de arestas	Formato do arquivo de distâncias (estado inicial)
1TAB2,5	1.1TAB0|1
2TAB3,5	2.2TAB0|1
3TAB2,4	3.3TAB0|1
4TAB,2,3,5	4.4TAB0|1
5TAB,1,2,4	5.5TAB0|1

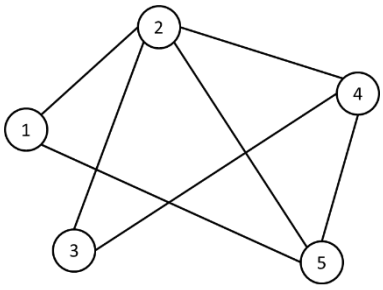


Figura 6.2. Exemplo de grafo: grafo I (HEDA) (Nascimento and Murta, 2012)

O algoritmo é dividido em duas etapas:

A primeira etapa tem como objetivo calcular o menor caminho entre todos os pares de vértices. Esta etapa recebe como parâmetros de entrada o arquivo de arestas, o arquivo de distâncias e o endereço no HDFS para gravação dos dados de saída. O

algoritmo faz o cálculo da distância entre um vértice origem e todos os outros de forma simultânea, na medida em que avança na descoberta de novos vértices como demostrado no Algorimo 3.

Na linha 6 cria-se uma instância do objeto de tipo *Node*, chamado *n1*. Na linha 7, é feita a validação do estado do objeto *n1*, caso ele esteja no estado 2, que indica que já foi processado, então é marcado e enviado para a fase *Reduce*. Na linha 11 é inicializado o laço que realiza a expansão a partir de todos os vértices presentes no grafo, considerando, em cada iteração, a existência de vértices cuja distância para os outros ainda não foi calculada. Na linha 15, um novo vértice é instanciado, *n2*, recebe a distância acumulada no vértice n1 acrescido de 1, seu estado é atribuído para 1 (em processamento), ele é enviado para a fase *Reduce* por meio do comando SAÍDA e o vértice é marcado como vértice visitado.

No final do laço, o objeto n1 tem seu estado atribuído para 2 (já processado), na linha 21. Na linha 23 e 24 o objeto n1 é marcado como visitados e enviado para a fase *Reduce*. A Tabela 6.2 apresenta as iterações do algoritmo e as alterações que ocorrem no arquivo de distância. Ao processar a entrada 1.1TAB0|1, na primeira iteração, aparecem as entradas 1.2TAB1|1 e 1.5TAB1|1, que correspondem às arestas (1,2) e (1,5) extraídas do arquivo de arestas. A entrada original passa a ser 1.1TAB1|2, para refletir o fato de que já foi processada. O algoritmo executa as iterações até que todas as entradas sejam marcadas com o estado de processado 2 (processado).

Algoritmo 3: Algoritmo HEDA - EncontrarMenorCaminho

```
   Entrada: Caminho do Arquivo de Arestas, Arquivo de Distâncias,
            Caminho do Arquivo de saída
 1 Classe Mapper
 2   método INICIAL (Array arestas);
 3     ARESTAS ←new array(arestas);
 4     VISITADOS ← ARRAYASSOCIATIVO;
 5     método MAP (chave, valor);
 6       Node n1 ← new Node(id chave, valor, ARESTAS[chave]);
 7       if n1.estado = 2 then
 8 |         VISITADOS[n1.verticeDestino] ← TRUE;
 9 |         SAIDA(n1.verticeDestino, n1.dados);
10       else
11 |         forall id a ∈ n1.listaArestas do
12 |   |         if a = n1.verticeDestino ou VISITADOS[a] = TRUE
   |   |         then
13 |   |   |         continue;
14 |   |         end
15 |   |         Node n2 ← new Node(a);
16 |   |         n2.atribuiDistancia(n1.distancia + 1);
17 |   |         n2.atribuiEstado(1);
18 |   |         SAIDA(n2.verticeDestino, n2.dados);
19 |   |         VISITADOS[a] ← TRUE;
20 |         end
21 |         n1.atribuiEstado(2);
22       end
23       VISITADOS[n1.verticeDestino] ← TRUE;
24       SAIDA(n1.verticeDestino, n.dados);

 1 Classe Reducer ;
 2   método REDUCE (id chave, P[p1, p2, ..., pn]);
 3     distancia ← INT − MAX;
 4     estado ← 0;
 5       while (P.leProximo()) do
 6 |         valor ← P.proximo;
 7 |         Node n1 ← new Node(chave, valor);
 8 |         if n1.distancia < distancia then
 9 |   |         distancia ← n1.distancia;
10 |         end
11 |         if n1.estado > estado then
12 |   |         estado ← n1.estado;
13 |         end
14       end
15       Node n2 ← new Node(chave);
16       n2.atribuiDistancia(distancia);
17       n2.atribuiEstado(estado);
18       SAIDA(n2.id, n2.dados);
```

A fase *Reduce* do Algoritmo 3 é responsável por identificar o menor caminho. Ela recebe como entrada os resultados enviados da fase *Map*. Na linha 5, é criado um

laço que percorre todos os vértices de uma mesma chave e, ao final, encontra a maior distância e a maior situação de uma determinada chave. Quando o laço termina, é armazenada a maior distância (linha 16) e o maior estado (linha 17). Em seguida, os dados são gravados no HDFS (linha 18).

Tabela 6.2 Exemplo de iterações do algoritmo HEDA (Nascimento and Murta, 2012)

Primeira iteração	Segunda iteração	Terceira iteração
1.1TAB0|2	1.1TAB0|2	1.1TAB0|2
1.2TAB1|1	1.2TAB1|2	1.2TAB1|2
1.5TAB1|1	1.3TAB2|1	1.3TAB2|2
2.2TAB0|2	1.4TAB2|1	1.4TAB2|2
2.1TAB1|1	1.5TAB1|2	1.5TAB1|2
2.3TAB1|1	2.2TAB0|2	2.2TAB0|2
2.4TAB1|1	2.1TAB1|2	2.1TAB1|2
2.5TAB1|1	2.3TAB1|2	2.3TAB1|2
3.3TAB0|2	2.4TAB1|2	2.4TAB1|2
3.2TAB1|1	2.5TAB1|2	2.5TAB1|2
3.4TAB1|1	3.3TAB0|2	3.3TAB0|2
4.4TAB0|2	3.1TAB2|1	3.1TAB2|2
4.2TAB1|1	3.2TAB1|2	3.2TAB1|2
4.3TAB1|1	3.4TAB1|2	3.4TAB1|2
4.5TAB1|1	3.5TAB2|1	3.5TAB2|2
5.5TAB0|2	4.4TAB0|2	4.4TAB0|2
5.1TAB1|1	4.2TAB1|2	4.2TAB1|2
5.1TAB1|1	4.3TAB1|2	4.3TAB1|2
5.2TAB1|1	4.5TAB1|2	4.5TAB1|2
5.4TAB1|1	5.5TAB0|2	5.5TAB0|2
	5.1TAB1|2	5.1TAB1|2
	5.1TAB1|2	5.1TAB1|2
	5.2TAB1|2	5.2TAB1|2
	5.3TAB2|1	5.3TAB2|2
	5.4TAB1|2	5.4TAB1|2

Na segunda etapa, o algoritmo processa o raio e o diâmetro a partir das medidas de excentricidade dos vértices. Esta etapa é uma tarefa mais simples. Sua função é buscar no HDFS os resultados da primeira etapa provenientes da terceira iteração da Tabela 6.2, percorrem todos os valores e verificam se o valor corrente é maior que a maior excentricidade já encontrada. Para cada conjunto de dados processado, o algoritmo grava no HDFS o identificador do vértice origem e o valor da excentricidade. Ao final, são calculados o radio e diâmetro a partir da excentridade de cada vértice. A

Tabela 6.3 apresenta o resultado de extração das excentricidades de cada vértice e os valores do raio e diâmetro do grafo.

Tabela 6.3 Exemplo resultado do algoritmo HEDA (Nascimento and Murta, 2012)

Extração das excentricidades de cada vértice	Cálculo do raio e do diâmetro do grafo
1TAB2 2TAB1 3TAB2 4TAB2 5TAB2	RaioTAB1 DiâmetroTAB2

Os resultados apresentados em (Nascimento & Murta, 2012) demostram que o algoritmo HEDA obteve resultados satisfatórios em relação à escalabilidade, *speedup* e eficiência.

6.3 O cálculo de centralidade em grafos com SGBD paralelo

O cálculo de centralidade em grafos com SGBD paralelo seguiu o mesmo princípio do algoritmo HEDA: primeiro, encontra o menor caminho entre todos os pares de vértices e, posteriormente, calcula o raio e o diâmetro.

Tabela 6.4 Tabelas utilizadas no SGBD paralelo para o cálculo da centralidade

Fonte: Elaborada pelo autor

Nome do atributo	Tipo do atributo
Orig	int
Dest	int
Weight	int

(a)

Nome do atributo	Tipo do atributo
Orig	int
Dest	int
distance	int
timestep	int

(b)

No processo para cálculo da centralidade do grafo foi necessário o uso de duas tabelas: Tabela 6.4(a), para armazenar o grafo, onde cada tupla representa o vértice de origem e seu vértice adjacente e seu respectivo peso, caso o peso não seja informado, a tabela terá como valor 1 como peso padrão; Tabela 6.4 (b), tabela para armazenar o vértice de origem, vértice de destino, sua respectiva distância e a quantidade de iterações necessárias para o cálculo da centralidade. Para facilitar a construção dos pseudocódigos das consultas iterativas, foi utilizada álgebra relacional, uma linguagem formal empregada em consultas em banco de dados relacionais. A Tabela 6.5 apresenta os operadores utilizados.

Tabela 6.5 Operadores da Álgebra Relacional Fonte: Elaborada pelo autor

Simbolo	Descrição
σ	A operação de Seleção usada para selecionar um subconjunto de tuplas de uma relação que satisfaça uma condição de seleção.
π	A operação de projeção seleciona certas colunas da relação e descarta outras.
$\bowtie$	O operador de junção natural combina as tuplas de duas relações que tem atributos comuns (mesmo nome), resultando numa relação que contém apenas as tuplas onde todos os atributos comuns apresentam o mesmo valor.

Na primeira parte do algoritmo que calcula o menor caminho entre todos os pares de vértices foram utilizadas consultas iterativas que avançam na descoberta de novos vértices adjacentes e computa o novo caminho somando ao peso existente. Um laço controla a execução das consultas e quando não houver novos menores caminhos para computar, as consultas iterativas são interrompidas. A cada iteração do laço é realizada a junção dos vértices de origem da tabela temporária com os vértices de destino do grafo somando o peso obtido. Quando a quantidade de pesos for menor do que a quantidade da iteração anterior, o laço é interrompido, significando que não há menores caminhos a serem computados.

Detalhamos o Algoritmo 4 da seguinte forma: as linhas (1), (2) e (3) criam as tabelas que irão armazenar os resultados temporários das iterações. As linhas (4), (5) e (6) inicializam as variáveis utilizadas nas consultas. Na linha (7), a tabela temporária *temp1* recebe o conteúdo do grafo e indica que se trata da primeira iteração atribuindo o valor 1 à coluna *timestep*.

Algoritmo 4: Cálculo do menor caminho entre todos os pares de vértices

Entrada: G = Grafo
Saída: Tabela contendo menor caminho entre todos os pares de vértices

1 $CREATE\ TEMP\ TABLE\ temp1;$
2 $CREATE\ TEMP\ TABLE\ temp2;$
3 $CREATE\ TEMP\ TABLE\ result;$
4 $count \leftarrow 1;$
5 $path \leftarrow 0;$
6 $check \leftarrow true;$
7 $temp1 \leftarrow \Pi_{(vOrig,vDest,weight,1)}(G);$ `/* set first step */`
8 **while** $check \neq false$ **do**
9 | **if** $mod(count, 2) = 0$ **then**
10 | | $oldtemp \leftarrow temp2;$
11 | | $newtemp \leftarrow temp1;$
12 | **else**
13 | | $oldtemp \leftarrow temp1;$
14 | | $newtemp \leftarrow temp2;$
15 | **end**
16 | $A \leftarrow \sigma_{G.Orig=oldtemp.Dest,G.Dest<>oldtemp.Orig}(G \bowtie oldtemp);$
17 | $B \leftarrow \sigma_{timestep=count}(A);$
18 | $oldtemp \leftarrow \Pi_{(oldtemp.Orig,G.Dest,distance+G.weight,timestep+1)}(B)$
19 | $newtemp \leftarrow \Pi_{(Orig,Dest,distance,timestep)}(\sigma_{timestep=count+1}(oldtemp))$
20 | $result \leftarrow \Pi_{(Orig,Dest,distance)}(oldtemp)$
21 | $oldtemp \leftarrow oldtemp - \sigma(oldtemp);$ `/* clear temp table */`
22 | $count \leftarrow count + 1;$
23 | $length \leftarrow \sigma_{count(*)}(newtemp))$
24 | **if** $lenght > path$ **then**
25 | | $path \leftarrow leght;$ `/* still has path to go */`
26 | | $check \leftarrow true;$
27 | **else**
28 | | $check \leftarrow false;$
29 | **end**
30 **end**
31 $return \leftarrow \Pi_{(Orig,Dest,MIN(distance))}(result)$

As linhas (8) até a (15) garantem que as tabelas *temp1* e *temp2* sejam usadas alternadamente para armazenar o estado atual e calcular os novos valores para as

distâncias entre os vértices em cada iteração. Se o valor da variável *count* for ímpar, as tabelas temporárias serão utilizadas na sequência: *temp1* e *temp2*, caso o valor seja par, a sequência é: *temp2* e *temp1*. Isso faz com que o uso das tabelas temporárias seja alternado de forma contínua: *temp1*, *temp2*, *temp1*, *temp2*, de modo que em cada iteração tenhamos disponíveis os resultados obtidos pela iteração anterior. As linhas (13) e (14) definem os nomes das tabelas que serão utilizadas na iteração.

Na linha (16) a junção entre a relação *G* e a relação temporária *oldtemp* é armazenada na variável *A*, . Esta junção é feita entre os vértices de origem de *G* com os vértices de destino de *oldtemp*, excluído os vértices de destino de *G* iguais aos vértices de origem de *oldtemp*, isso é necessário para evitar que se computem vértices que possuem arestas para ele mesmo.

Na linha (17) final retira-se da relação *A*, computada na linha (16), as tuplas com *timestep* igual ao valor a variável *count*. Na linha (18), é atribuída na tabela *oldtemp* a projeção da relação *B* somando-se o peso das arestas de *G* com as distâncias computadas de *B*, incrementado o valor *timestep* para ser utilizado na próxima iteração.

Na linha (19), os resultados computados até o momento são inseridos em uma nova tabela temporária, *newtemp*, que será utilizada na próxima iteração. A linha (20) armazena todos os resultados na tabela temporária *result*. Na linha (21), remove-se todo o conteúdo de *oldtemp* para reaproveitamento na próxima iteração. A linha (22) incrementa a variável *count* que determinará a sequência das tabelas temporárias para a próxima iteração.

Na linha (23), a variável *lenght* recebe a quantidade de tuplas existentes na tabela temporária *newtemp*. A quantidade de tuplas no final da iteração será a condição de parada. Quando em uma determinada iteração a quantidade de tuplas existentes em *newtemp* for maior que a quantidade de tuplas da iteração anterior, significa que houve menor caminho computado, portanto, o algoritmo deve continuar. Quando for o oposto, significa que não houve menores caminhos computados, determinando a parada das iterações na linha (27).

Na linha (30), executa-se a projeção da relação de resultados temporários. Esta projeção restringe-se aos valores mínimos de distâncias entre os pares de vértices, caracterizando o menor caminho entre todos os pares de vértices do grafo *G*.

A segunda parte, a partir do resultado da primeira, extrai o conjunto de maiores distâncias de cada vértice, ou seja, a excentricidade, e finaliza a execução calculando o raio e o diâmetro demonstrados no Algoritmo 5. A linha (1) atribui em *A* a projeção das tuplas com o máximo valor de distância de cada vértice de origem e destino, ou seja, a excentricidade de cada par de vértices. No final, retorna-se o cálculo do diâmetro e raio, representado a centralidade do grafo.

Algoritmo 5: Extração do diâmetro e o raio do grafo

Entrada: D = Menor Caminho entre Todos os Pares de Vértices
Saída: diâmetro, raio

1 $A \leftarrow \Pi_{(Orig,Dest,MAX(distance))}(D)$
$return \leftarrow \Pi_{(MAX(distance),MIN(distance))}(A)$

Os algoritmos 4 e 5 completos encontram-se no Apêndice B deste trabalho.

6.4 Medidas de centralidade com o banco de dados orientado a grafos

O Neo4J oferece diversas possibilidades para cálculo do menor caminho. Neste trabalho explanamos a *API* de travessia para computar a excentricidade de um vértice como mostra a Figura 6.3.

```
1 Traverser traverser = nodeSrc.traverse(
2        Order.BREADTH_FIRST,
3        StopEvaluator.END_OF_GRAPH,
4        ReturnableEvaluator.ALL_BUT_START_NODE,
5        RelTypes.KNOWS,
6        Direction.BOTH);
```

Figura 6.3 Objeto traverser oferecido pela *API* do Neo4j

Fonte: Elaborada pelo autor

Os elementos mostrados na Figura 6.3 têm o seguinte significado:

1. Cria o objeto *traverser* e define o vértice de origem da travessia;
2. Determina o algoritmo usado para realizção da travessia, podendo ser Busca em Largura (*Breadth-First Search*) ou Busca em Profundidade (*Depth-First Search*);
3. Define o critério de parada, no nosso caso, a travessia deverá ocorrer até o final do grafo;

4. Define o retorno do objeto *traverser*, neste caso, retornar todos os vértices, exceto o vértice de origem;
5. Determina a propriedade de relacionamento que será utilizada para alcançar os vértices adjacentes;
6. A direção da travessia, neste caso, será em ambas as direções, pois se trata de um grafo não dirigido.

Depois de informar à função *traverser* os elementos do seu construtor, o objeto retornado por ela irá conter o menor caminho para todos os vértices do grafo. Para isso, basta realizar a iteração no objeto *traverser* e recuperar o tamanho do menor caminho para cada vértice, como mostrado na Figura 6.4.

```
for (Node nodes : traverser) {
      if (!(nodes == null)) {
            // Adiciona o menor caminho do vértice do origem
            // entre cada vértice de destino em um ArrayList
            allDistance.add(traverser.currentPosition().depth());
      }
}
```

Figura 6.4 Iterando objeto *traverser* para computar o menor caminho entre todos os pares de vértices Fonte: Elaborada pelo autor

O código fonte completo do cálculo da centralidade utilizando o Neo4j encontra-se no Apêndice A, juntamente com a descrição das *interfaces* e classes que foram utilizadas.

6.5 Considerações finais

Neste capítulo, apresentamos como cada tecnologia que comparamos aborda o cálculo da centralidade em grafos. Apresentamos o algoritmo HEDA baseado no modelo *MapReduce* e apresentamos os algoritmos equivalentes que desenvolvemos para o cálculo utilizando o SGBD paralelo e o banco de dados orientado a grafos , além de um algoritmo sequencial que será usado para o cálculo do *speedup* de cada uma das soluções testadas.

No próximo capítulo, descreveremos o projeto dos experimentos realizados.

7. CAPÍTULO 7 – METODOLOGIA E PROJETO DE EXPERIMENTOS

Este capítulo apresenta o processo metodológico na elaboração deste trabalho, incluindo a coleta dos dados e as métricas utilizadas para avaliar o desempenho de cada uma das tecnologias comparadas. Também descrevemos o ambiente computacional e os conjuntos de dados utilizados nos experimentos.

7.1 Desempenho

O fator que mede o desempenho de uma tecnologia de processamento de dados é denominado tempo de resposta. Tempo de resposta é o tempo medido entre o início do processamento da informação e a respectiva efetivação desta, seja ela uma atividade de leitura, escrita, ou de atualização ou de dados.

Otimizar o desempenho requer cuidados múltiplos, ora por parte da readequação do modelo de dados, ora por parte dos algoritmos que acessam os dados. Outra forma nem sempre eficiente e mais dispendiosa, porém de mais simples implementação é a melhoria dos recursos de computacionais através da otimização de processadores, aumento da memória física, redes mais eficientes no que tange à transferência dos dados e discos magnéticos mais rápidos (Özsu & Valduriez, 2011).

A avaliação de desempenho consiste na execução de procedimentos visando quantificar o comportamento do sistema, seja ele hardware ou software. A avaliação de desempenho pode ser compreendida de duas maneiras: *análise* e *predição de desempenho*.

A *análise de desempenho* tem por objetivo estudar o desempenho do sistema disponível, compreender seu comportamento e determinar um dimensionamento dos seus componentes de modo a identificar pontos de melhoria no desempenho.

A *predição* estuda o desepenho do sistema não disponível. A predição se preocupa em predizer o comportamento do sistema, e apresentar ao usuário se o desempenho do sistema é adequado ou não em relação ao esperado.

7.2 Medidas desempenho

A escolha de uma medida ou do conjunto de medidas de desempenho representa fator chave durante o processo de avaliação. Uma escolha equivocada de medida pode fazer com que os resultados e comparações sejam verdadeiros e/ou não demonstre o comportamento dos sistemas.

Dentre o conjunto de métricas de avaliação de desempenho, algumas se destacam como padrões por serem utilizadas na maioria das análises de desempenho. Quando o sistema alvo de avaliação de desempenho é um sistema de processamento sequencial, um determinado conjunto de métricas pode ser utilizado. São exemplos de medidas aplicáveis a sistemas sequencias: carga no sistema, porcentagem de falha de paginação, tempo de espera em operação de entrada e saída. Já quando o sistema alvo são sistemas de processamento paralelos, além das medidas utilizadas na avaliação de sistemas sequencias (devidamente modificadas) outras medidas devem ser empregadas para atender as características específicas. Exemplos dessas medidas são o *speedup* (aceleração) e a *eficiência*.

As medidas de desempenho podem ser classificadas em dependentes de velocidade e independete de velocidade (Özsu & Valduriez, 2011).

- As medidas dependentes de velocidade são aquelas de alguma forma estão ligadas à ou tempo de execução dos sistemas, sejam eles de processamento sequencial ou paralelos.
- As medidas independentes de velocidade, normalmente, são utilizadas para quantificar o desempenho dos sistemas e subsistemas específicos. Quando o alvo são sistemas de hardware, são utilizadas métricas que mostram a ocupação de registradores, falta de dados em cache, quantidade de memória livre.

7.2.1 *Speedup* (Aceleração)

O *speedup*, ou fator de aceleração, mede o quanto mais rápido é um sistema paralelo comparado à execução sequencial. É amplamente aplicado para descrever a escalabilidade de um sistema paralelo (Özsu & Valduriez, 2011). O *speedup* pode ser

relativo, dado pela razão entre o tempo de execução do algoritmo paralelo e um único processador pelo tempo de execução do mesmo algoritmo em *n* processadores. A fórmula para o *speedup* relativo é a seguinte:

$$Speedup\ relativo(n) = \frac{Tempo\ de\ execução\ em\ um\ processador}{Tempo\ de\ execução\ em\ n\ \text{processadores}}$$

Outra maneira de calcular o *speedup* é o conhecido como *speedup* absoluto que, em vez de utilizar o tempo de execução do algoritmo paralelo em um único processador, o cálculo leva em consideração o melhor algoritmo sequencial conhecido. A formula do *speedup* absoluto é a seguinte:

$$Speedup\ absoluto(n) = \frac{Tempo\ de\ execução\ sequencial}{Tempo\ de\ execução\ em\ n\ \text{processadores}}$$

O *speedup* pode ser linear, sublinear ou superlinear, como mostrado na Figura 7.1.

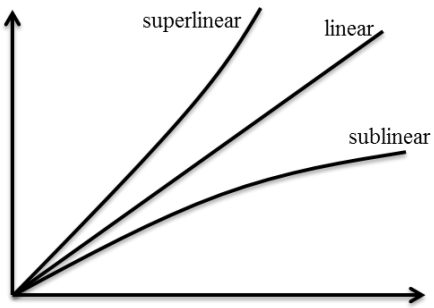


Figura 7.1 Speedup Superlinear, Linear e Sublinear Fonte: Elaborada pelo autor

- **superlinear**: indica que o *speedup* alcançado é maior do que o número de processadores utilizados;
- **linear**: indica que o *speedup* alcançado é igual ao número de processadores utilizados;

- **sublinear**: indica que o *speedup* alcançado é menor que o número de processadores utilizados utilizados.

O *speedup* superlinear pode ocorrer na realização de buscas que são terminadas no encontro de um elemento desejado. Nos sistemas paralelos, a pesquisa pode ser executada de forma eficaz em uma ordem diferente, o que pode implicar uma quantidade menor de dados pesquisados que, no caso, sequencial. Outra razão para *speedup* superlineares deriva-se do fato de que, em um sistema paralelo, há uma quantidade agregada de memória maior que em um único computador, resultando em uma necessidade menor de acessar memória secundária.

O *speedup* linear ocorre quando o aumento da capacidade computacional disponível produz uma redução equivalente no tempo de execução. Por exemplo, se ao dobrarmos a capacidade computacional o tempo de execução cai pela metade. Este cenário é raro, as curvas de *speedup* tendem a saturar na medida em que aumenta número de processadores; isso se dá ao fato de o tamanho do problema ser uma constante enquanto o número de processadores é aumentado, característica esta que implica na redução da quantidade de trabalho por processador.

Ressaltamos que o *speedup* calculado neste trabalho foi o **relativo**. O propósito desta escolha foi para analisarmos a variação dos tempos dos algoritmos paralelos de acordo com o número de computadores disponíveis, sendo esta a que tem tido a maior influência no processamento paralelo (Özsu & Valduriez, 2011).

7.2.2 Eficiência

A eficiência de um algoritmo paralelo representa o quão bem os recursos computacionais estão sendo usados. Em uma situação ideal, o *speedup* deveria ser linear em função da capacidade de processamento que se adicionou ao sistema, no entanto, isso nem sempre é possível. Nesses casos, a eficiência pode informar o quanto de esforço o sistema desperdiça em sincronização e comunicação. Ela pode ser calculada como:

$$E(n) = \frac{S(n)}{n}$$

Nesta fórmula, *S(n)* é o valor do *speedup* obtido usando *n* processadores. Quando o resultado da eficiência é igual a 1 (um), indica que o *speedup* é linear, já que todos os processadores estão sendo usados em plena capacidade. Devido a diversas fontes de perda de desempenho, a eficiência geralmente é menor que 1 (um). Eficiências maiores que 1 (um) são observadas nos casos de *speedup* superlineares (Rosário, 2012).

7.3 Avaliação do desempenho das tecnologias

Neste trabalho decidiu-se tomar como parâmetro de comparação entre as tecnologias as medidas em relação à velocidade de execução. Tal parâmetro foi escolhido, pois o tempo de execução de um sistema é considerado um fator de grande importância (Özsu & Valduriez, 2011).

Todas as tecnologias, SGBD paralelo (Greenplum), MapReduce (Hadoop), SGBD orientado a grafos (Neo4J) e o algoritmo de processamento sequencial (NetworkX), foram submetidos ao processamento de grafos de diferentes tamanhos para computar a medidade de centralidade de um grafo baseda na excentricidade de cada vértice.

Os tempos coletados de cada tecnologia são a média do tempo de três execuções diferentes, e foram calculadas usando o programa *time* do Linux. Foram consideradas três medidas de tempo, já que o ambiente computacional era dedicado para os testes, os tempos de execução obtidos nos testes individuais eram extremamente similares entre si.

As tecnologias de processamento sequencial (Neo4J e *NetworkX*) foram executadas em um único computador. As tecnologias paralelas (Greenplum e Hadoop) foram submetidas a várias combinações de quantidades de computadores.

Para que o foco da comparação recaísse apenas nas medidas de velocidades de execução de cada tecnologia, desconsideramos o tempo de carregamento dos dados.

A Tabela 7.1 apresenta um resumo das principais características de cada tecnologia comparada neste trabalho.

Tabela 7.1 Principais características de cada tecnologia

Tecnologia	Tipo de processamento	Sistema de arquivos	Persistência dos dados
Greenplum	Paralelo	Local distribuído via função Hash	Em disco
Hadoop	Paralelo	Compartilhado via *HDFS*	Em disco
Neo4J	Sequencial	Local	Em disco
Network	Sequencial	Local	Memória RAM

7.4 Ambiente Computacional

Os experimentos realizados utilizaram a infraestrutura computacional chamada Grid'5000, uma plataforma experimental reconfigurável, controlável e monitorável, dedicados a pesquisas relacionadas a sistemas paralelos e distribuídos de larga escala.

O Grid'5000 é composto de aproximadamente 1.200 máquinas (8000 núcleos) distribuídas em 11 locais situados, principalmente, na França, nas cidades de Bordeaux, Grenoble, Lille, Luxembourg, Lyon, Nancy, Nantes, Reims, Rennes, Sophia-Antipolis e Toulouse, como mostra a Figura 7.2. O *Grid'5000* abrange 19 laboratórios pelo país com o objetivo de proporcionar à comunidade um ambiente para realização de experimentos em todas as camadas de *software*, desde os protocolos de rede até as aplicações (Balouek et al., 2013).

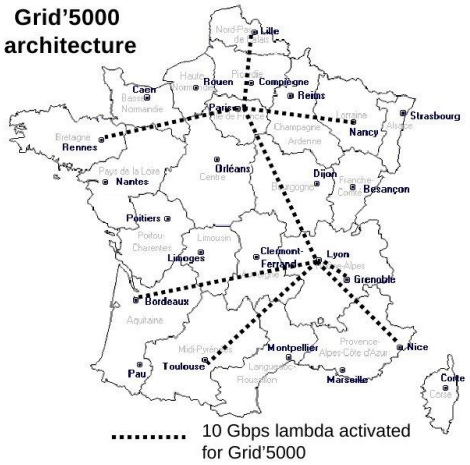


Figura 7.2 Distribuição geográfica dos clusters no Grid5000 (Balouek, Lèbre and Quesnel, 2013)

Os experimentos utilizaram dois *clusters* do conjunto de computadores oferecidos pelo Grid'5000:

- *Cluster paradent* disponível no site Rennes com 45 máquinas, cada máquina contendo dois processadores Intel Xeon L5420 com quatro núcleos físicos a 2.66 GHz, 32GB de memória RAM e um disco rígido de 350GB SATA II. Ao todo são 360 núcleos;
- *Cluster graphene* disponível no site Nancy com 120 máquinas, cada máquina contendo um processador Intel Xeon X3440 com quatro núcleos físicos a 2.53 GHz, 16GB de memória RAM, e um discos rígidos de 320GB SATA II. Ao todo são 480 núcleos físicos.

Foram utilizados dois serviços essenciais oferecidos pelo Grid'5000 durante os experimentos: 1) o agendador em lote OAR, para gerir os recursos da plataforma, possibilitando a reserva de quantidades arbitrárias de computadores por período de tempo específico. 2) Kadeploy, utilizado para replicar o sistema operacional e os *softwares* utilizados nos testes entre todos os computadores participantes.

Todas as máquinas utilizadas nos experimentos estavam interligadas por um *swicth* de velocidade de 1Gb/s e receberam o Sistema Operacional Linux CentOS 5.9, kernel 2.6.18-371.4.1.el5.x86 64, a *Java Virtual Machine* versão 1.7.0_51, Python versão 2.7.6, Python-NetworkX versão 1.9.0, Apache Hadoop versão 1.2.1 e o SGBD paralelo *Pivotal Greenplum Database*, versão 4.2.2.

7.5 Conjunto de dados

Os conjuntos de dados utilizados nos testes foram compostos por grafos reais e grafos sintéticos gerados pela biblioteca *NetworkX*. Os grafos sintéticos foram utilizados pois oferece a possibildade de execução de grafos com diferentes números de vértices e arestas.

7.5.1 Grafos Reais

Dentre os grafos reais utilizados existem dois que descrevem a rede de colaboração da *High Energy Physics Theory* - Biblioteca da Universidade de Cornell

(ca-HepPh e ca-HepTh). O terceiro grafo descreve a rede de colaboração científica entre coautores que publicaram artigos na categoria *Astro Physics* da revista eletrônica *arXiv*. Por fim, o quarto grafo, da loja virtual Amazon, representa relacionamentos entre produtos. Estes grafos estão disponíveis na base de dados da Universidade de Stanford, mantida pelo grupo de pesquisa *Stanford Network Analysis Project* (Stanford Large Nertwork Dataset Collection, 2013). Tabela 7.2 sumariza os grafos apresentados nesta seção.

Tabela 7.2 Grafos DataSet de Stanford (Stanford Large Nertwork Dataset Collection, 2013)

Nome do Grafo	Quantidade Vértices	Quantidade Arestas	Raio	Diâmetro
Cahepth	9.877	51.971	1	17
Cahepph	12.008	237.01	1	13
Astroph	18.772	396.161	1	14
Com-amazon	334.863	925.872	1	44

7.5.2 Grafos Sintéticos

Os grafos sintéticos utilizados nos testes foram gerados utilizando a biblioteca *NetworkX* (Hagberg et al., 2008), escrita em linguagem orientada a objetos *Python*. A biblioteca disponibiliza a função *gnm_random_graph(n, m)*, que gera aleatoriamente um grafo, sem arestas repetidas e pertencente ao conjunto de todos os possíveis grafos $G(V,E)$ com n vértices e m arestas, cujos vértices são nomeados com números naturais entre 0 e $(n-1)$. A biblioteca *NetworkX* também oferece a função *eccentricity(G,v)* que utilizamos para calcular a distância máxima a partir de v para todos os outros vértices de G.

Foram criados grafos conectados e não direcionados com diferentes números de vértices e arestas para verificar o comportamento de cada tecnologia. A Tabela 7.3 apresenta os grafos sintéticos que foram gerados para os testes.

Tabela 7.3 Grafos Sintéticos Fonte: Elaborada pelo autor

Qtde Vértices	Qtde Arestas
10.000	100.000
20.000	200.000
30.000	300.000
40.000	400.000
50.000	500.000

7.6 Organização dos dados no SGBD paralelo

No processamento de grafos usando o SGBD paralelo, a divisão dos dados de entrada implica em dividir o grafo entre todos os segmentos do *cluster*. Os algoritmos mais conhecidos para resolução desse problema são algoritmos de aproximação, considerados NP-Completos, que tendem a encontrar soluções próximas das consideradas ótimas (Chataigner et al., 2007). Podemos destacar o algoritmo de separação cartesiana (Heath & Raghavan, 1994), o algoritmo Kernighan-Lin (KL) (Kernighan & Lin, 1970), o algoritmo Fiduccia-Mattheyses (FM) (Fiduccia & Matteyses, 1982), o método multinível (Karypis & Kumar, 1995) e bisseção coordenada recursiva (Simon & Teng, 1991). Vale ressaltar que o algoritmo KL é um dos mais referenciados para a resolução desse problema, já que ele produz um bom particionamento em relação ao número de arestas cortadas.

Como o problema de particionamento de grafo não é trivial, adotamos neste trabalho uma estratégia “neutra”, que é a de dividir o grafo tentando equilibrar a quantidade de vértices em cada segmento. Optou-se por dividir o grafo pelos vértices de origem, utilizando função *hash*, de tal forma que os dados relacionados a um determinado vértice estivessem localizados no mesmo nó do *cluster*. A Figura 7.3 demostra a distribuição de um grafo de cinco vértices e sete arestas. As tabelas do banco de dados foram configuradas para utilizar orientação à linha, seguindo a recomendação do fabricante, que indica orientação à coluna apenas para casos em que há atualizações e inserções frequentes.

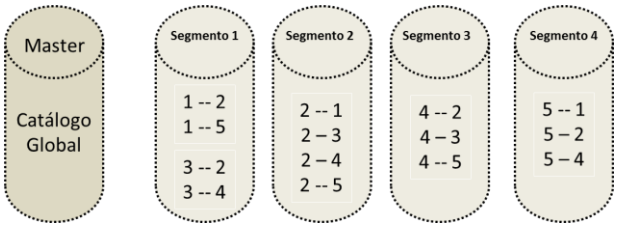


Figura 7.3 Distribuição Física do Grafo no SGBD paralelo Fonte: Elaborada pelo autor

7.7 Organização dos dados no Hadoop

No caso do algoritmo HEDA, não é possível especificar explicitamente como particionar os dados, como fizemos no SGBD paralelo. Em vez disso, o Hadoop HDFS particiona os arquivos em blocos, cria repetições para cada bloco e, em seguida, distribui aleatoriamente todos esses blocos entre os vários nós do *cluster*.

7.8 Organização dos dados no Neo4j e *NetworkX*

No caso do Neo4J e o algoritmo sequencial, os dados estão organizados em formato “lista de incidência” e são carregados no momento da execução. Como o Neo4J opera em *cluster* apenas para replicação dos dados para garantir a alta disponibilidade do banco de dados, todas as operações de cálculo da centralidade foram realizadas em um único computador.

7.9 Considerações finais

Este capítulo apresentou o processo metodológico utilizado na elaboração deste trabalho, o ambiente computacional e os conjuntos de dados utilizados na comparação entre o SGBD paralelo (Greenplum), *MapReduce* (Hadoop), o SGBD orientado a grafos (Neo4J) e a biblioteca *NetworkX*, utilizados para o cálculo da medidade de centralidade de um grafo baseda na excentricidade de cada vértice. O capítulo apresentou também as métricas que utilizadas para comparar o desempenho de cada tecnologia.

No próximo capítulo, é apresentado os resultados dos experimentos conduzidos neste trabalho.

8. CAPÍTULO 8 - RESULTADO DOS EXPERIMENTOS

Nesse capítulo é apresentado os resultados do tempo de execução, *speedup* e eficiência ao calcular medidade de centralidade de um grafo baseda na excentricidade de cada vértice, utilizando os grafos enumerados no capítulo anterior. Todos os dados coletados dos experimentos estão anexados no Apêndice C.

8.1 Grafos Reais

Esta seção mostra os tempos de execução obtidos ao processar os grafos reais apresentados na Tabela 7.2. Estes experimentos foram realizados no *cluster paradent*, que possui 45 computadores. Cada máquina contém dois processadores Intel Xeon L5420 com quatro núcleos físicos a 2.66 GHz, 32GB de memória RAM e um disco rígido de 350GB.

A decisão de executar os grafos reais neste conjunto de computadores está relacionada ao número de vértices e arestas pertencentes aos grafos. A execução das tecnologias paralelas necessitou de quantidades diferentes de computadores. Neste caso optamos em executar em um *cluster* que oferece a maior capacidade de processamento individual, pois nos caso de executar os grafos em uma quantidade mínima de computador despenderíamos de menor tempo para realizar os trabalhos.

A **Erro! Fonte de referência não encontrada.** mostra os resultados dos testes uando aumentamos gradativamente a quantidade de computadores no *cluster*. Este tipo de teste faz sentido apenas para as tecnologias que utilizam paralelismo, portanto apenas são comparadas o Greenplum com o Hadoop. Apesar de que os tempos de execução do Hadoop são menores quando utilizamos poucos computadores, o SGBD paralelo acaba obtendo melhores resultados na medida em que a quantidade de computadores aumenta, chegando a obter tempos 59,25% menores para o grafo Com-Amazon. Este fato indica que o Greenplum é capaz de aproveitar os recursos computacionais de forma mais eficiente, e que sua arquitetura é suficientemente escalável para aproveitar novos recursos computacionais quando eles são disponibilizados.

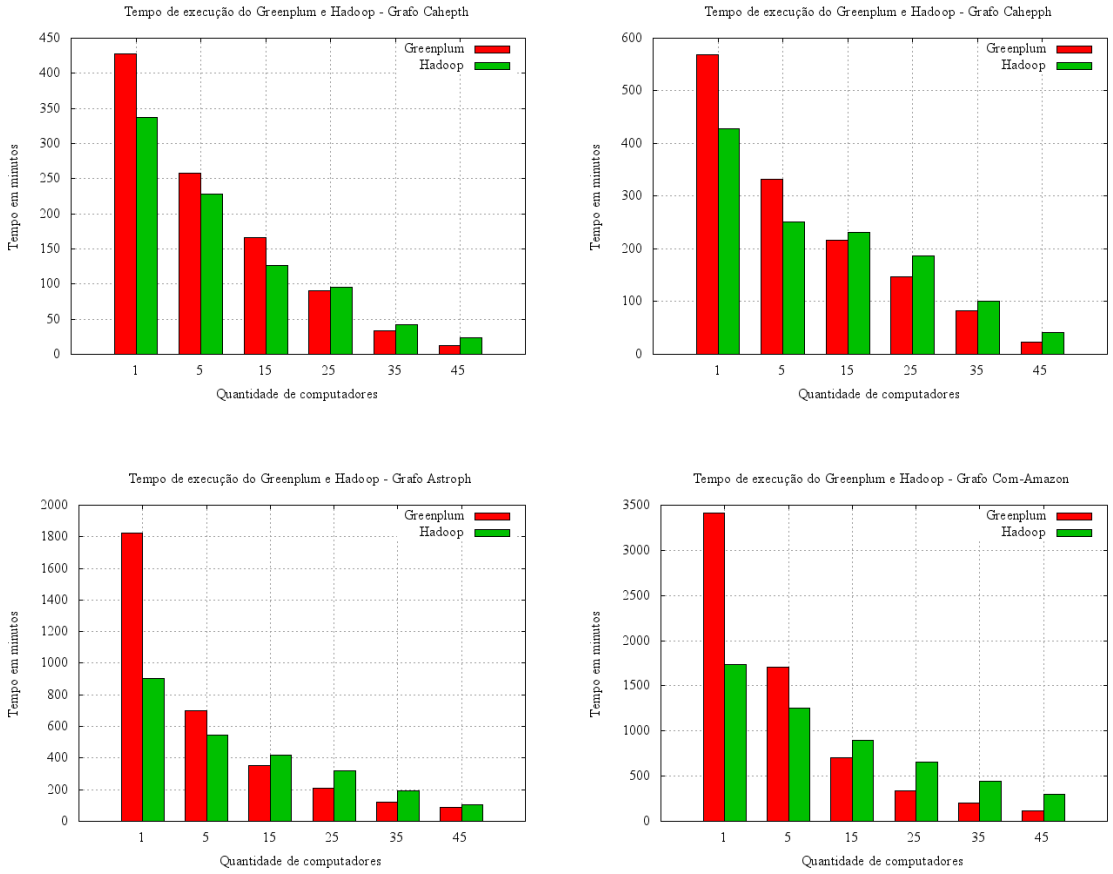


Figura 8.1 Tempo de execução Greenplum e Hadoop - grafos reais

Fonte: Próprio autor

A Figura 8.2 apresenta o resultado da execução dos grafos reais no *cluster paradent* com sua quantidade máxima, (45 computadores). Foram comparados os tempos de execução obtidos pelo Hadoop, Greenplum, Neo4J e o *NetworkX*. Ressaltamos que os tempos da *NetworkX* e o banco de dados Neo4j correspondem aos tempos de execução em único computador do *cluster* para que pudéssemos mensurar o ganho de tempo de execução das tecnologias paralelas em relação às tecnologias sequenciais. Outro aspecto que vale a pena ressaltar é que o *NetworkX* e o Neo4J não conseguiram computar os resultados do grafo *Com-Amazon* dentro do limite de tempo de reserva imposto pelo *Grid'5000*, que é de 60 horas.

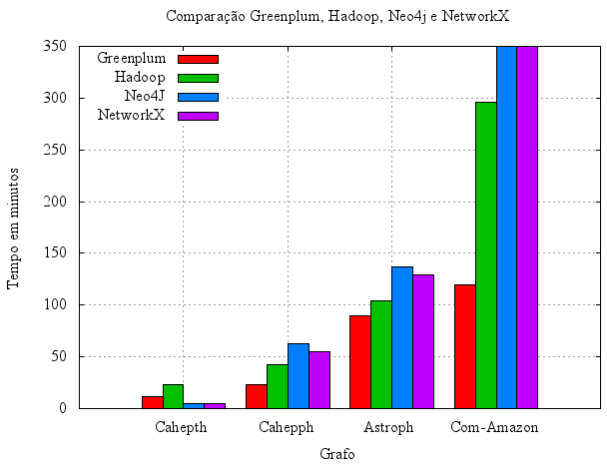


Figura 8.2 Comparação de desempenho entre Greenplum, Hadoop, Neo4j e *NetworkX*
Fonte: Próprio autor

Os resultados da Figura 8.2 evidenciam que as tecnologias sequenciais (Neo4J e *NetworkX*) obtém melhores resultados para grafos menores, o que é esperado já que as soluções paralelas incorrem em uma sobrecarga para gerenciar os recursos computacionais que somente é aceitável quando a quantidade de trabalho a ser realizado é grande o suficiente. Para grafos maiores, como é de se esperar, o Greenplum e o Hadoop obtêm melhores resultados, com o Greenplum apresentando os melhores tempos de execução, principalmente para o Com-Amazon, o maior grafo testado.

A Figura 8.3 mostra os resultados do *speedup* relativo, do Greenplum e do Hadoop para o mesmo conjunto de dados da Tabela 7.2. Nota-se que o SGBD paralelo apresenta *speedup* sublinear, mas tende a buscar a linearidade, valor próximo do ótimo, à medida que novos computadores são adicionados ao *cluster*. Isso pode ser explicado pelo fato de que, ao acrescentar máquinas no *cluster*, a quantidade de dados que um nó computacional deve processar é menor, permitindo que os dados em processamento possam ser carregados na sua totalidade (ou quase totalidade) na memória *RAM* dos computadores, o que implica na diminuição do acesso ao disco em cada computador participante. O Hadoop também apresenta valores de *speedup* sublinear, sofrendo pouca alteração em grande quantidade de computadores.

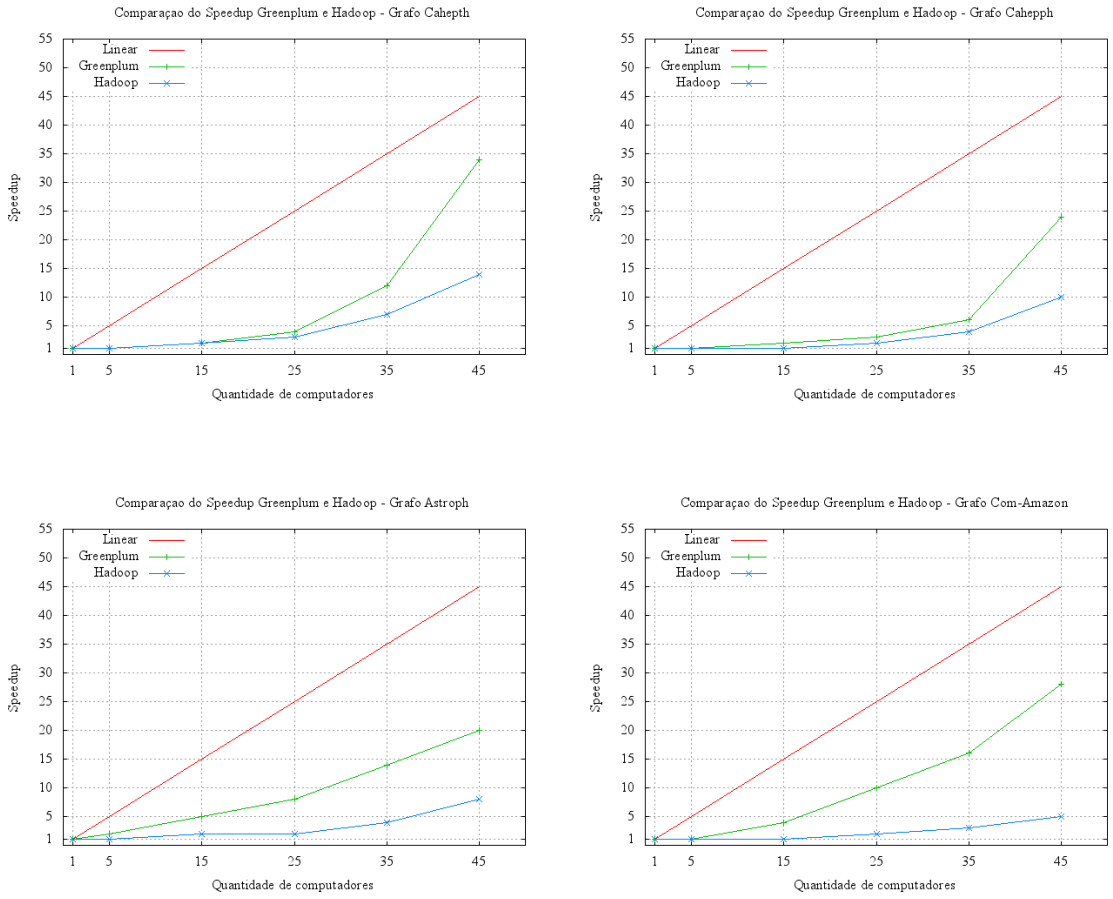
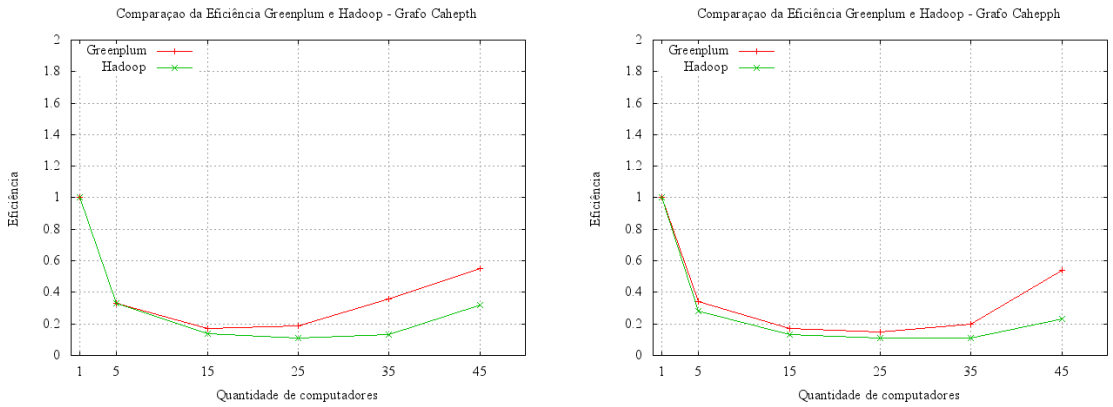


Figura 8.3 Comparação do *speedup* – Greenplum e Hadoop

Fonte: Próprio autor

A Figura 8.4 apresenta os resultados da eficiência do Greenplum comparado ao Hadoop. Os resultados mostram que o SGBD paralelo consegue ser mais eficiente, aproveitando melhor os recursos computacionais quando comparados ao Hadoop.



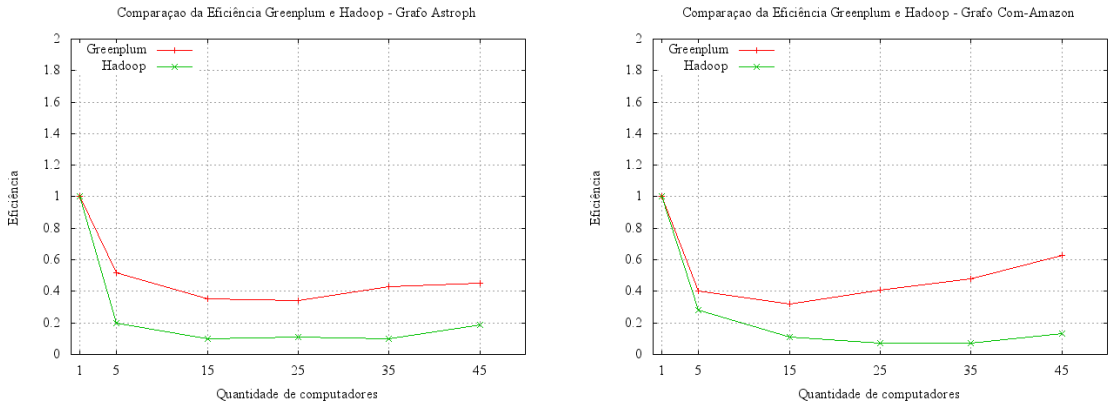


Figura 8.4 Comparação da eficiência – Greenplum e Hadoop

Fonte: Próprio autor

8.2 Grafos Sintéticos

Esta seção mostra os tempos de execução obtidos ao processar os grafos sintéticos. Por se tratar de um conjunto de dados maiores, optou-se por utilizar o c*luster graphene*, constituído por 120 computadores contendo um processador Intel Xeon X3440 com quatro núcleos físicos a 2.53 GHz, 16GB de memória *RAM*.

Nestes testes foram realizados utilizando somente a configuração máxima dos computadores do *cluster graphene* por questões operacionais. Este *cluster* oferece computadores de menor poder computacional, dificultando os testes. Executar grafos em quantidade mínima de computadores neste *cluster* não seria viável devido a restrição de tempo imposta pelo agendamento do Grid'5000.

8.2.1 Variação de arestas

A **Erro! Fonte de referência não encontrada.** sumariza os grafos utilizados este experimento. Os grafos gerados são sintéticos não direcionados com 20.000 vértices e quantidade de arestas entre 200.000 e 500.000. O aumento no número de arestas, neste caso, representa o acréscimo no número de relações entre uma quantidade fixa de vértices. A consequência imediata é a existência de novos caminhos entre os vértices, aumentando a exigência de processamento do menor caminho entre todos os pares de vértice do grafo.

Nesses grafos sintéticos, o raio e o diâmetro diminuem quando a quantidade de arestas aumenta e, portanto, a quantidade de iterações necessárias para chegar ao resultado também dimimui. Assim, ao aumentar a quantidade de arestas há duas forças opostas em ação: uma que aumenta a complexidade de cada iteração e outra que diminui a quantidade de iterações necessárias para realização do cálculo do raio e diâmetro.

Tabela 8.1 Variação do número de arestas em grafos sintéticos

Fonte: Próprio autor

Arestas	Raio	Diâmetro	Variação de arestas
200.000	6	7	-
300.000	5	6	1,5x
400.000	4	5	2,0x
500.000	4	5	2,5x

A **Erro! Fonte de referência não encontrada.** apresenta o resultado do tempo e execução dos grafos sintéticos da comparação do Greenplum em relação ao Hadoop, Neo4J e o *NetworkX*. Assim como ocorreu para os grafos reais, o tempo de execução do *NetworkX* e do banco de dados Neo4J foi obtido em um único computador do *cluster*. Os resultados evidenciam que o Greenplum consegue melhor aproveitamento dos recursos computacionais em comparação ao Hadoop.

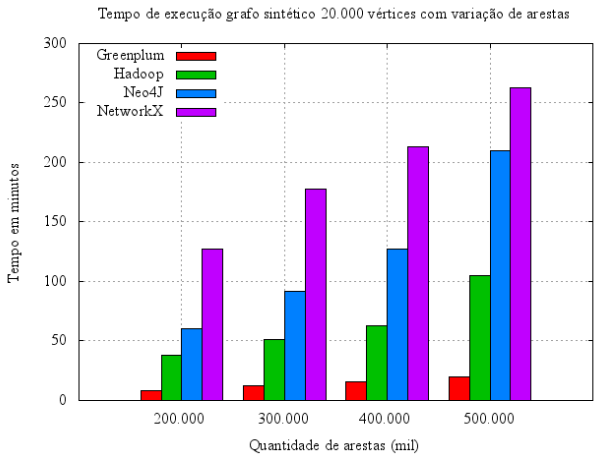


Figura 8.5 Comparação do tempo de execução entre banco de dados paralelo e HEDA com variação de arestas Fonte: Próprio autor

A Figura 8.6 mostra o *speedup* relativo da execução do Greenplum comparado ao Hadoop. Nota-se que o banco de dados paralelo apresenta um resultado melhor do *speedup* e que o resultado mantém-se quase que constante e independente da quantidade de arestas que o grafo possui.

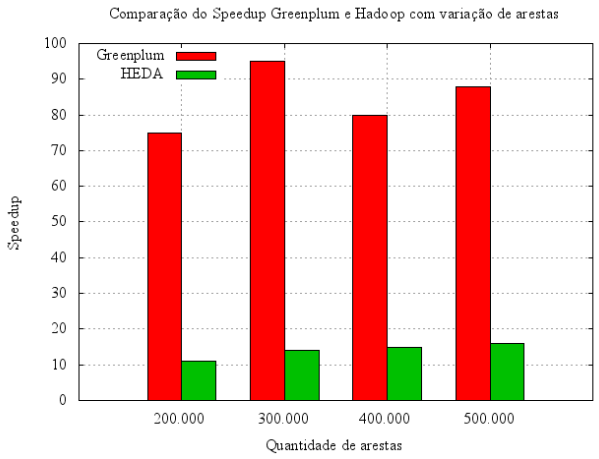


Figura 8.6 Comparação do *Speedup* entre Greenplum e Hadoop com variação de arestas Fonte: Próprio autor

8.2.2 Variação de vértices e arestas

A Tabela 8.2 resume os grafos sintéticos não direcionados com variação de vértices e arestas utilizados nos experimentos. Por padrão, a variação da quantidade de arestas manteve a proporção em 10 vezes a quantidade de vértices presente no grafo. O objetivo foi verificar o impacto que o aumento do número de vértices e do número de arestas entre eles em relação ao tempo de processamento.

Tabela 8.2 Variação de vértices e arestas em grafos sintéticos Fonte: Próprio autor

Vértices	Arestas	Raio	Diâmetro	Variação do grafo
10.000	100.000	4	3	-
20.000	200.000	5	6	2,0x
30.000	300.000	6	8	3,0x
40.000	400.000	7	11	4,0x
50.000	500.000	11	13	5,0x

A variação do tempo de execução com a variação do tamanho do grafo é maior se comparada à variação da quantidade de arestas. Ao aumentar a quantidade de vértices, mantendo a mesma proporção de relações arestas/vértices, estamos aumentando a quantidade de vértices distantes entre si ou, em outras palavras, o raio e o diâmetro do grafo.

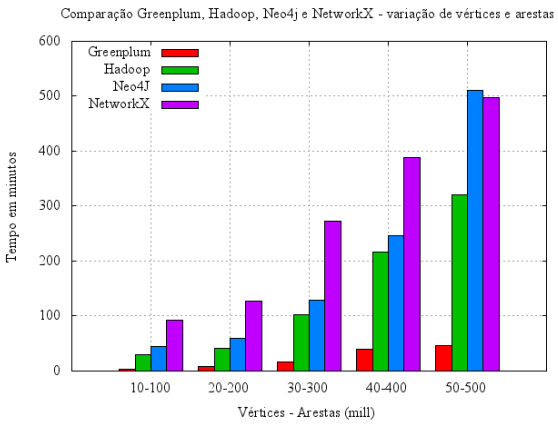


Figura 8.7 Comparação do tempo de execução entre SGBD paralelo e HEDA com variação de vértices e arestas Fonte: Próprio autor

A Figura 8.7 mostra o resultado do tempo de execução dos grafos sintéticos da Tabela 8.2 da comparação do banco de dados paralelo em relação ao Hadoop. Também foi adicionado o tempo de execução do Neo4J e algoritmo sequencial. Os resultados ratificam que o banco de dados paralelo consegue melhor aproveitamento dos recursos computacionais em relação ao Hadoop, mesmo resultado apresentado quando comparado ao processamento de grafos com variação de arestas. Também se percebe que algoritmo sequencial obteve melhor resultado em relação ao Neo4J para o grafo de 50.000 vértices e 500.000 arestas, um resultado surpreendente visto que o Neo4J é apresentado como uma solução particularmente eficiente para grafos maiores. .

A Figura 8.8 mostra o *speedup* relativo do Greenplum comparado ao Hadoop no processamento dos grafos com variaçãos de vértices.

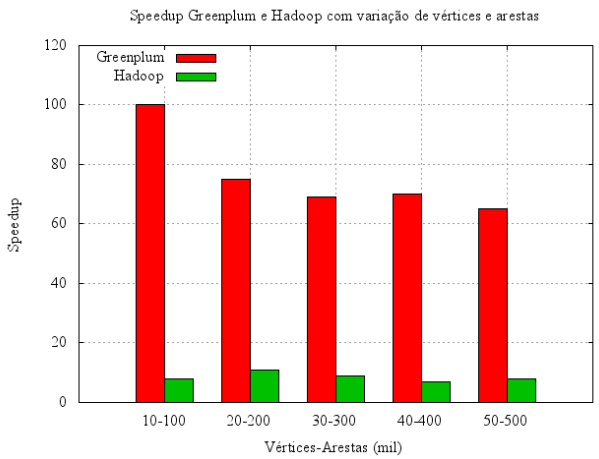


Figura 8.8 Comparação de Speedup entre o Greenplum e Hadoop com variação de vértices e arestas Fonte: Próprio autor

8.3 Análise dos resultados

A biblioteca *NetworkX* obteve melhor desempenho ao processar grafos menores comparado ao banco de dados orientado a grafo (Neo4J). Isso se deve ao fato de o *NetworkX* trabalhar com dados em memória *RAM*, enquanto o Neo4J trabalha com os dados em disco. O processo constante de leitura e escrita no disco, necessário para manter o banco consistente, faz com que o Neo4J perca em desempenho.

Quando comparamos as tecnologias paralelas com as tecnologias sequenciais, é notório que, com um grafo é menor, as tecnologias sequenciais conseguem melhores desempenho. Isso porque as tecnologias paralelas possuem custo muito alto para manter o ambiente computacional em funcionamento. Nos casos de grafos pequenos, o custo de comunicação e sincronização dos dados é muito maior do que o próprio processamento do grafo.

Para grafos maiores, as soluções baseadas em paralelismo obtiveram tempos melhores, como era de se esperar. Os resultados obtidos mostram que o SGDB paralelo utilizado neste comparativo (Greenplum) obteve tempos de execução consistentemente melhores do que a solução baseada no modelo *MapReduce* (Hadoop). Há várias razões que contribuiram para que isso ocorresse:

Parsing* dos dados:* o Greenplum processa os dados em formato de texto uma única vez, no momento do carregamento, e os armazena já convertidos em estruturas

otimizadas para cada tipo de dados. O Hadoop força todas as tarefas *map* e *reduce* a fazerem *parsing* dos dados em cada execução.

Tratamento de resultados intermediários: o Greenplum usa técnicas de *pipelining* para comunicar os resultados obtidos por um operador para o próximo. Esta técnica permite disponibilizar os resultados ao próximo operador assim que eles são produzidos pelo anterior. O Hadoop "materializa" esses resultados intermediários no HDFS, onde são escritos pelas tarefas *map* e lidos pelas tarefas *reduce*, incorrendo, portanto, em uma maior sobrecarga. Esta técnica, embora ajude na tolerância a falhas da plataforma, acaba impactando negativamente os tempos de execução.

Escalonamento: em um banco de dados paralelo, o plano de execução da consulta é traçado antes de começar sua execução. Dessa forma, é possível otimizar a distribuição de responsabilidades entre os nós do *cluster* de forma a diminuir o tráfego de dados entre eles e produzir os resultados mais rapidamente. No caso do Hadoop, o escalonamento de tarefas é feito em tempo de execução, com os nós do *cluster* recebendo novas tarefas na medida em que terminam as anteriores. Este mecanismo, embora seja excelente para tolerância a falhas e para se adaptar a mudanças no desempenho dos computadores participantes, não é o ideal para obter os melhores tempos de execução possíveis para um problema em particular.

De forma geral, os resultados obtidos mostram que o Hadoop é mais eficiente em conjunto menor de computadores e sacrifica desempenho para obter tolerância a falhas, adaptabilidade a mudanças no ambiente de execução e simplicidade na arquitetura. O Greenplum, entretanto, tem como foco extrair o melhor desempenho possível dos recursos computacionais disponíveis, ainda que ao custo de uma arquitetura mais complexa e rígida.

8.4 Considerações finais

Este capítulo apresentou os resultados do tempo de execução, *speedup* e eficiência das tecnologias comparadas neste trabalho para o cálculo da medida de centralidade. O SGBD paralelo mostrou ter melhor desempenho na medida em que aumenta o tamanho dos grafos processados e da infraestrutura utilizada.

9. CAPÍTULO 9 - CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresenta uma comparação entre o modelo *MapReduce*, um Sistema de Gestão de Banco de Dados Relacional Paralelo, um Banco de dados Orientado por Grafos no processamento de grafos e uma biblioteca de processamento sequencial denominada *NetworkX*. Utilizamos o algoritmo HEDA, que segue o modelo *MapReduce*, e desenvolvemos soluções equivalentes para o SGBD paralelo e o banco de dados orientado por grafos.

Os resultados mostram que, apesar do SGBD paralelo ser relativamente lento com poucos computadores, ele se torna a melhor opção para executar o cálculo da centralidade em grafos quando a quantidade de computadores disponíveis aumenta, com valores de *speedup* que tendem alcançar a linearidade.

Os resultados da comparação entre o SGBD paralelo e o modelo *MapReduce* evidenciam as diferenças arquiteturais. O *MapReduce* não oferece uma estrutura predefinida para os dados, o que obriga que seu "parsing" seja feito em tempo de execução. Sem uma estrutura de dados predefinida, cada usuário deve escrever um analisador personalizado, o que dificulta o compartilhamento de dados.

O SGBD paralelo oferece uma estrutura de dados definida e realiza a análise dos dados em tempo de carregamento, com isso, ele mantém as informações críticas para otimização de consultas, incluindo os índices existentes, tabelas particionadas e distribuição dos valores.

Para o *MapReduce* o aumento do tamanho do grafo ocasiona o aumento do processamento das partições de dados intermediários, aumentando a granularidade dos dados e a quantidade de tarefas *map* e *reduce*.

No caso do banco de dados orientado a grafos, ele trabalha em *cluster* de computadores apenas para garantir alta disponibilidade. Por este motivo, o processamento do cálculo da centralidade aconteceu de forma sequencial, não conseguindo aproveitar todos os recursos computacionais disponíveis. No entanto, ele obteve os melhores tempos de execução para grafos pequenos.

A partir dos resultados apresentados, fica evidente que, para grafos grandes, os SGBD paralelos são a melhor opção.

9.1 Trabalhos futuros

Durante este trabalho foram identificadas algumas possibilidades que não puderam ser implementadas ou testadas.

- **Heterogeneidade**: realizar os testes em ambientes heterogêneos, como desktop grid.
- **Avaliar outros parâmetros de desempenho**: por exemplo, o tempo de leitura e escrita de disco nas estações; uso de memória *RAM*; tráfego de dados na rede.
- **Ampliar a comparação com outras tecnologias**: tais como *Bulk Synchronous Parallel* (BSP) e *Message Passing Interface* (MPI).
- **Outros problemas:** que possam ser aplicados para aferir o desempenho das tecnologias, tais como: outras medidas de centralidade; *pagerank*; *counting triangles*; ancestral comum mais baixo; componentes conexos.

REFERÊNCIAS BIBLIOGRÁFICAS

Anon., 2014. *HyperGraphDB*. [Online] Available at: http://www.hypergraphdb.org [Accessed 20 Setembro 2014].

Apache Giraph, 2014. [Online] Available at: http://giraph.apache.org [Accessed 20 Setembro 2014].

Apache Hama, 2014. [Online] Available at: https://hama.apache.org/ [Accessed 20 Setembro 2014].

Balouek, D., Lèbre, A. & Quesnel, F., 2013. Flauncher and DVMS -- Deploying and Scheduling Thousands of Virtual Machines on Hundreds of Nodes Distributed Geographically. *Finalist of the IEEE International Scalable Computing Challenge (SCALE 2013)*.

Barroso, M.M.A., 2014. Aplicação de grafos em um problema de rede. *ABAKÓS - Instituto de Ciências Exatas e Informática*, Maio. pp.48-78.

Bondy, J.A. & Murty, U.S.R., 1976. *Graph Theory with Applications*. New York: North-Holland.

Cattel, R., 2010. *Relational Databases, Object Databases, Key-Value Stores, Document Stores, and Extensible Record Stores: A Comparison. ODBMs*. [Online] Available at: http://www.odbms.org/wp-content/uploads/2010/01/Cattell.Dec10.pdf [Accessed 10 Abril 2013].

Chataigner, F., Salgado, L.R.B. & Wakabayashi, Y., 2007. Approximation and Inaproximability Results on Balanced Connected Partitions of Graphs. *Discrete Mathematics and Theoretical Computer Science (DMTCS)*, 9, pp.177-92.

Codd, E.F., 1970. A relational model of data for large shared data banks. *Communications of the ACM 13 (6)*, pp.377–87.

Cormen, T., Leiserson, C., Rivest, R. & Stein, C., 2004. *Introduction to Algorithms*. New York: MIT Press.

Dean, J. & Ghemawat, S., 2008. MapReduce: Simplied Data Processing on Large Clusters. *ACM Digital Library*, pp.107-13.

Del Vechio, R., Lima, L., Galvão, D. & Loures, R., 2009. Medidas de Centralidade da Teoria dos Grafos aplicada a Fundos de Ações no Brasil. *XLI Simpósio Brasileiro de Pesquisa Operacional, Porto Seguro*.

DeWitt, D. & Gray, J., 1992. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, (35(6)), pp.85–98.

DeWitt, D. & Stonabraker, M., 2008. *MapReduce: A major step backwards*. [Online] Available at: http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.htm [Accessed 26 Novembro 2013].

Elmasri, R. & Navathe, S.B., 2007. *Fundamentals of Database Systems*. 3rd ed. Addison-Wesley: Redwood.

EMC2, 2012. *Greenplum® Database 4.2 System Administrator Guide*. [Online] Available at: http://media.gpadmin.me/wp-content/uploads/2012/11/GPSysAdminGuide.pdf [Accessed 15 Março 2013].

EMC2, 2014. *The Digital Universe in 2020*. [Online] Available at: http://www.emc.com/infographics/digital-universe-consumer-infographic.htm [Accessed 30 Julho 2014].

Fiduccia, C.M. & Matteyses, R.M., 1982. A linear time heuristic for improving network partitions. *19th Design Automaton Conference*, pp.175-81.

Freeman, L.C., 1979. Centrality in networks: I. Conceptual clarification. *Social Networks 1*, pp.215-39.

Galaskiewicz, J. & Wasserman, S., 1993. Social Network Analysis: Concepts, Methodology, and Directions for the 1990s. *Sociological Methods & Research 22*, pp.3-22.

Gartner, 2011. *Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data*. [Online] Available at: http://www.gartner.com/newsroom/id/1731916 [Accessed 10 Abril 2013].

Hadoop, 2014. *Welcome to Apache Hadoop*. [Online] Available at: http://hadoop.apache.org [Accessed 10 Abril 2014].

Hagberg, A.A., Schult, D.A. & Swart, P.J., 2008. Exploring network structure, dynamics, and function using NetworkX. *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pp.11–15.

Hage, P. & Harary, F., 1995. Eccentricity and centrality in networks. *Social Networks*, 17, pp.57-63.

Heath, M.T. & Raghavan, P., 1994. A Cartesian Parallel Nested Dissection Algorithm. *SIAM. J. Matrix Anal. & Appl.*, 16, pp.235–53.

Herodotou, H., 2012. Automatic Tuning of Data-Intensive Analytical Workloads. In *Ph.D. Dissertation*. Duke University - Department of Computer Science.

HP Vertica, 2014. [Online] Available at: http://www.vertica.com [Accessed 20 Setembro 2014].

IBM, 2012. *The Flood of Big Data*. [Online] Available at: http://www.ibmbigdatahub.com/infographic/flood-big-data [Accessed 10 Abril 2013].

InfiniteGraph, 2014. [Online] Available at: http://www.objectivity.com [Accessed 20 Setembro 2014].

Isard, M. et al., 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, (41), pp.59-72.

Kajdanowicz, T., Kazienko, P. & Indyk, W., 2014. *"Parallel processing of large graphs," Future Generation Computer Systems*. [Online] Available at: http://dx.doi.org/10.1016/j.future.2013.08.007 [Accessed 02 Março 2015].

Karloff, H., Suri, S. & Vassilvitskii, S., 2010. A model of computation for MapReduce. *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms. [S.l.]: Society for Industrial and Applied Mathematics*, pp.938–48.

Karypis, G. & Kumar, V., 1995. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *Technical Report Departament of Computer Science. University of Minnesota*.

Kernighan, B.W. & Lin, S., 1970. An efficient heuristic procedure for partitioning graphs. *Bell Sys. Tech. J*, pp.291-307.

Leavitt, N., 2010. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43 no.2, pp.12-14.

Lin, J. & Dyer, C., 2010. Data-Intensive Text Processing. *Synthesis Lectures on Human Language Technologies*, pp.1–177.

Lin, J. & Schatz, M., 2010. Design patterns for efficient graph algorithms in MapReduce. *Proceedings of the Eighth Workshop on Mining and Learning with Graphs. ACM*, pp.78–85.

Low, Y. et al., 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Journal Proceedings of the VLDB Endowment*, pp.716-27.

Lucene, 2014. [Online] Available at: http://lucene.apache.org/ [Accessed 29 Dezembro 2014].

Malewicz, G. et al., 2010. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp.135-46.

Nascimento, J.P. & Murta, C., 2012. Um algoritmo paralelo em Hadoop para Cálculo de Centralidade em Grafos Grandes. *XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. Ouro Preto: SBC.*.

Neo Technology, 2006. *The Neo Database – A Technology Introduction*. [Online] Available at: http://dist.neo4j.org/neo-technology-introduction.pdf [Accessed 23 Março 2014].

neo4j.org, 2014. *World's Leading Graph Database*. [Online] Available at: htp://neo4j.org [Accessed 22 Março 2014].

Nicoletti, M.C., Hruschka, J. & Estevam, R., 2011. *Fundamentos da teoria dos grafos para computação*. São Carlos: EduFSCar, 228p. (Apontamentos).

Özsu, M.T. & Valduriez, P., 2011. *Principles of Distributed Databases Systems*. 3rd ed. New York: Prentice Hall.

Pivotal Greenplum, 2014. [Online] Available at: http://www.pivotal.io/big-data/pivotal-greenplum-database [Accessed 20 Setembro 2014].

Ray, C., 2009. *Distributed Database Systems*. India: Pearson Education.

Rosário, D.A.N., 2012. *Escalabilidade paralela de um algoritmo de migração reversa no tempo (RTM) pré-empilhamento*. Natal - RN: Dissertação (Mestrado) -

Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e de Computação.

Simon, H.D. & Teng, S.H., 1991. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2, pp.135–48.

Smith, M., Szongott, C., Henne, B. & Von Voigt, G., 2012. Big data privacy issues in public social media. *Digital Ecosystems Technologies (DEST) 6th IEEE International Conference*, pp.1,6, 18-20.

Stanford Large Nertwork Dataset Collection, 2013. *Stanford Network Analysis Platform*. [Online] Available at: http://snap.stanford.edu/ [Accessed 13 Agosto 2013].

Stonebraker, M. et al., 2010. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53, pp.64-71.

Stonebraker, M., 1986. The case for shared nothing. *Database Engineering*, 1(1).

Stonebraker, M., 2010. *The "NoSQL"Discussion has Nothing to Do With SQL*. [Online] Available at: http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothingto-do-with-sql/fulltext [Accessed 10 Abril 2013].

Strozzi, C., 1998. *NoSQL Relational Database Management System*. [Online] Available at: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home%20Page [Accessed 17 Abril 2014].

Teradata, 2014. [Online] Available at: http://www.teradata.com [Accessed 20 Setembro 2014].

White, T., 2012. *Hadoop: The Definitive Guide*. 3rd ed. O'Reilly Media, Inc.

Zhao, Y et al., 2014. *Evaluation and Analysis of Distributed Graph-Parallel Processing Frameworks*. [Online] Available at: http://riverpublishers.com/journal/journal_articles/RP_Journal_2245-1439_333.pdf [Accessed 02 Março 2015].

Zhaohui, W., 2014. From Big Data to Data Science: A Multi-disciplinary Perspective. *Big Data Research*, 1, pp.1-66.

APÊNDICE

Apêndice A

Neste apêndice é apresentado o código fonte da classe escrita em Java para cálculo da centralidade em grafos utilizando o banco de dados orientado a grafos Neo4j e a descrição das interfaces e classes utilizadas.

Para tirar o máximo da *API* do Neoj4j, foi necessário adicionar a dependência *Maven* do Neo4j no *build path* do projeto Java e utilizar as *interfaces* e classes, das quais as mais importantes utilizadas foram (neo4j.org, 2014):

- ***GraphDatabaseService***: interface que provê o ponto de acesso principal para uma instância do Neo4j, definindo serviços básicos de criação do banco e de vértices, recuperação de vértices e arestas entre outros;
- ***EmbeddedGraphDatabase***: implementação de GraphDatabaseService para uso embutido em um programa Java, permitindo a criação e uso do banco em um diretório local;
- ***Index***: interface que define os serviços de criação e uso de índices baseados em pares chave e valor, que podem ser criados tanto para vértices quanto para arestas;
- ***RelationshipType***: interface para definir o tipo do relacionamento entre dois vértices. Essa abordagem permite que dois vértices possuam mais de um relacionamento, porém com tipos diferentes, indicando diferentes interações entre eles;
- ***Transaction***: interface que permite a manupulação de transações por meio de programação;
- ***Node***: interface que representa o vértice;
- ***Relationship***: interface que representa a aresta;
- ***PropertyContainer***: interface para trabalhar com propriedades dos vértices e arestas;
- ***GlobalGraphOperations***: classe que fornece serviços de operações globais no banco, como recuperar todos os vértices e todas as arestas.

```
package centrality;

/**
 * @author Fabiano da Silva Fernandes
 */

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Set;
import java.util.TreeSet;
import org.neo4j.graphdb.Direction;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Relationship;
import org.neo4j.graphdb.RelationshipType;
import org.neo4j.graphdb.ReturnableEvaluator;
import org.neo4j.graphdb.StopEvaluator;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.Traverser;
import org.neo4j.graphdb.Traverser.Order;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.neo4j.graphdb.index.Index;
import org.neo4j.tooling.GlobalGraphOperations;

public class Centrality {

	private static GraphDatabaseService db = null;
	private Transaction tx = null;
	private static final String DB_PATH = "/tmp/graphDB";
	private Index<Node> nodeIndex = null;
	private Relationship reelationship = null;

	public static final String NAME = "name";
	private static final String COST = "cost";
	private BufferedReader reader;

	/**
	 * Determina o relacionamento utilizado no grafo
	 */

	private static enum RelTypes implements RelationshipType {
		KNOWS
	}

	/**
	 * @param args
	 * the command line arguments args[0] graph file adjacency list
	 */
	public static void main(String[] args) {

		if (args.length != 1) {
			System.out.println("Expected 1 parameters");
			return;
		}

		Centrality centrality = new Centrality();
		// Inicializa o banco de dados
		centrality.startup();

		// Computa o tempo de carregamento do arquivo
		long start = System.currentTimeMillis();
		long duration = 0;
		centrality.loadGraph(args[0]);
		duration = System.currentTimeMillis() - start;
```

```
                System.out.println("Timing to loading graph: " + duration +
" ms");

                // Executa o cálculo da excentridiade
                centrality.run();

        }

        public void startup() {
                db                              =                           new
GraphDatabaseFactory().newEmbeddedDatabaseBuilder(DB_PATH)
                                .newGraphDatabase();
                tx = db.beginTx();
                nodeIndex = db.index().forNodes("nodes");
                shutdownDb();
                tx.success();
                tx.close();
        }

        public Set<String> loadGraph(String datasetDir) {
                Set<String> listAllNodes = new TreeSet<String>();
                tx = db.beginTx();
                try {
                        reader = new BufferedReader(new InputStreamReader(
                                        new FileInputStream(datasetDir)));
                        String line = null;
                        while ((line = reader.readLine()) != null) {
                                String[] lineDetails = line.split("\\t");
                                Node                    srcNode                 =
findOrCreateNode(lineDetails[0]);
                                Node                    dstNode                 =
findOrCreateNode(lineDetails[1]);
                                reelationship                                   =
srcNode.createRelationshipTo(dstNode,
                                                RelTypes.KNOWS);
                                reelationship.setProperty("relationship-
type", "knows");
                                reelationship.setProperty(COST, 1);

                        }

                        tx.success();
                } catch (Exception e) {
                        e.printStackTrace();
                        throw new RuntimeException(e.getCause());
                } finally {
                        tx.close();
                }

                return listAllNodes;
        }

        private static void shutdownDb() {
                Runtime.getRuntime().addShutdownHook(new Thread() {

                        @Override
                        public void run() {
                                db.shutdown();
                        }
                });
        }

        private void run() {
                ArrayList<Integer>      allExcentricity      =      new
ArrayList<Integer>();

                tx = db.beginTx();
```

```
                    Iterable<Node>                  allNodes                 =
GlobalGraphOperations.at(db).getAllNodes();
                    for (Node nodeSrc : allNodes) {
                          ArrayList<Integer>      allDistance      =      new
ArrayList<Integer>();
                          Traverser                traverser               =
nodeSrc.traverse(Order.BREADTH_FIRST,
                                        StopEvaluator.END_OF_GRAPH,
                                        ReturnableEvaluator.ALL_BUT_START_NODE,
      RelTypes.KNOWS,
                                        Direction.BOTH);

                          if (!(traverser == null)) {
                                for (Node nodes : traverser) {
                                        if (!(nodes == null)) {
                                                // Adiciona o menor caminho do
vértice do origem
                                                // para cada vértice de destino
em um ArrayList

      allDistance.add(traverser.currentPosition().depth());
                                        }

                                }

                          }

                          if (!allDistance.isEmpty()) {
                                // Retira as excentricidade de cada vértice
(máximo valor)

      allExcentricity.add(Collections.max(allDistance));
                          }

                    }
                    tx.close();

                    // Retorna as medidas de centralidade (raio e diâmetro)
                    System.out.println("Raio:"                              +
Collections.min(allExcentricity));
                    System.out.println("Diametro:"                          +
Collections.max(allExcentricity));

              }

              private Node findOrCreateNode(String nodeName) {
                    Node node = getNode(nodeName);
                    if (node == null) {
                          node = db.createNode();
                          node.setProperty(NAME, nodeName);
                          nodeIndex.add(node, NAME, node.getProperty(NAME));
                    }
                    return node;
              }

              public Node getNode(String name) {
                    return nodeIndex.get(NAME, name).getSingle();
              }
      }
```

Apêndice B

Neste apêndice é apresentado o código fonte do algoritmo escrito em PL/Phython para cálculo da centralidade em grafos utilizando o SGBD paralelo.

Algoritmo 1

```
CREATE OR REPLACE FUNCTION all_shortest_path(graph_table text, result_table
text)
  RETURNS void AS
$BODY$

       # Cria uma cópia da tabela que representa do grafo para agregar
       # pesos e arestas duplicadas, caso exista.

       plpy.execute("DROP TABLE IF EXISTS G")
       plpy.execute("CREATE TEMP TABLE G ( v1 int, v2 int, weight int ) " +
                        "WITH (appendonly=true, orientation=column,
compresstype=quicklz) DISTRIBUTED BY (v1,v2)")
       plpy.execute("INSERT INTO G SELECT DISTINCT v1, v2, 1 FROM " +
graph_table + " GROUP BY v1,v2")

       #
       # -- O calculo é feito de forma iterativa expandindo as arestas através
       # -- de seus vértices adjacentes
       # -- É calculado o menor caminho através de junções entre as tabelas.
       # -- Referencia http://techportal.inviqa.com/2009/09/07/graphs-in-the-
       # -- database-sql-meets-social-networks/
       # -- Para aproveitamento de tabelas temporários o processo irá utilizar
       # -- as tabelas TEMP_1 e TEMP_2
       #

       new_temp = "TEMP_1"
       old_temp = "TEMP_2"
       temp_result = "TEMP_RESULT"

       # Remove as tabelas temporárias caso elas existam
       plpy.execute("DROP TABLE IF EXISTS " + new_temp)
       plpy.execute("DROP TABLE IF EXISTS " + old_temp)
       plpy.execute("DROP TABLE IF EXISTS " + temp_result)

       # Cria tabela temporária TEMP_1 e define o primeiro passo da iteração.
       # Também cria-se a tabela temporária que irá armazenar os resultados
       # temporários
       plpy.execute("CREATE TEMP TABLE " + new_temp + " (v1 int, v2 int,
distance int, timestep int) " +
                        "WITH (appendonly=true, orientation=column,
compresstype=quicklz) DISTRIBUTED BY (v1,v2)")
       plpy.execute("CREATE TEMP TABLE " + temp_result + " (v1 int, v2 int,
distance int, timestep int) " +
                        "WITH (appendonly=true, orientation=column,
compresstype=quicklz) DISTRIBUTED BY (v1,v2)")
       plpy.execute("INSERT INTO " + new_temp + " SELECT v1, v2, weight, 1
FROM G")

       is_path = 0
       checker = True
       count = 1

       # Inicia-se a iteração entre TEMP_1 e TEMP_2
       while  checker: # Evita loop infinito
```

```
                #
                # -- Calcula-se o id da tabela necessário para reuso.
                # -- Segue a order de uso. TEMP_1, TEMP_2, TEMP_1 e TEMP_2,
                # -- TEMP_1 e TEMP_2, TEMP_1 e TEMP_2
                #
            new_temp_id = count % 2
            if (new_temp_id == 0):
                old_temp_id = 2
            else:
                old_temp_id = 1

            old_temp = "TEMP_" + str(old_temp_id)
            new_temp = "TEMP_" + str(new_temp_id)


        # -- Adicionamos o caminho e o peso obtido através dos vértices
        # -- adjacentes e marcamos o valor para ser usado na próxima iteração

                plpy.execute("INSERT INTO " + old_temp + " " +
                    "SELECT DISTINCT " + old_temp + ".v1, G.v2, distance +
G.weight, timestep + 1 " +
                    "FROM G INNER JOIN " + old_temp + " ON G.v1 = " + old_temp
+ ".v2 AND G.v2 <> " + old_temp + ".v1 " +
                    "WHERE timestep = " + str(count))

        # -- Cria-se a próxima tabela temporária
        plpy.execute("CREATE TEMP TABLE " + new_temp + " (v1 int, v2 int,
distance int, timestep int) " +
                    "WITH (appendonly=true, orientation=column,
compresstype=quicklz) DISTRIBUTED BY (v1,v2)")

        # -- Insere na próxima tabela temporária somente os valores que serão
        # -- usados na próxima iteração
                plpy.execute("INSERT INTO " + new_temp + " " +
                    "SELECT DISTINCT v1, v2, distance, timestep FROM " +
old_temp + " WHERE timestep = " + str(count + 1))

        # -- Armazena todos os valores resultantes da iteração
                plpy.execute("INSERT INTO " + temp_result + " " +
                    "SELECT DISTINCT v1, v2, distance, timestep FROM " +
old_temp)

        # -- Remove a tabela que será usada novamente na próxima iteração
        plpy.execute("DROP TABLE " + old_temp)

        count = count + 1

        # -- Verifica se ainda há caminhos alcançaveis à partir de um vertice
        # -- Caso não exista mais nenhum caminho alcançável interrompe o
        # -- while
        # -- Ná há mais caminhos alcançaveis quando a quantidade de tuplas da
        # -- tabela que armazena os resultados não sofrer mais alteraçao de
        # -- tamanho

                fcount_t = plpy.execute("SELECT COUNT(*) c FROM " + new_temp)
        count_path = fcount_t[0]['c']
        if count_path > is_path:
            is_path = count_path
            plpy.info(' Ainda tem caminho para percorrer ')
        else:
            checker = False
            plpy.info(' Não há mais vértices alcançáveis para percorrer ')



        plpy.info(' iteration %d, table size %d' % (count,count_path))
```

```
        #
        # -- Após a iteração entre as tabelas temporárias, extraímos o menor
        # -- caminho
        #
        plpy.execute("DROP TABLE IF EXISTS " + result_table)
        plpy.execute("CREATE TABLE " + result_table + " (v1 int, v2 int,
distance int ) " +
                     "WITH (appendonly=true, orientation=column,
compresstype=quicklz) DISTRIBUTED BY (v1,v2)")

        plpy.execute("INSERT INTO " + result_table +
                        " SELECT v1,v2,min(distance) FROM " + temp_result +
                        " GROUP BY v1,v2")


$BODY$
  LANGUAGE plpythonu VOLATILE;
```

Algoritmo 2

```
CREATE OR REPLACE FUNCTION centrality(result_table text)
  RETURNS void AS
$BODY$

       SELECT Max(excentricity) AS diameter, MIN(excentricity) AS radius
       FROM   (SELECT v1,
                    Max(distance) AS excentricity
                    FROM  result_table GROUP BY v1)sq;


$BODY$
  LANGUAGE plpythonu VOLATILE;
```

Apêndice C

Neste apêndice são mostrados os dados referentes às figuras apresentadas no Capítulo 6, As seções estão organizadas de acordo com a ordem em que as figuras aparecem naquele capítulo.

A.1 - Dados da Figura 8.1

Tabela A.1: Algoritmos sequenciais - Tempo de execução, de uma máquina, em minutos. Grafos reais executados no *cluster paradent*.

Algoritmo	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
Sequencial (*NetworkX*)	426,11	425,64	425,02	425,59	0,5
Neo4j	354,49	355,16	355,88	355,18	0,7

Tabela A.2: Algoritmo HEDA - Tempo de execução em minutos. Grafos reais executados no *cluster paradent*.

Configuração do Cluster	Grafo	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
45 máquinas (360 núcleos)	Cahepth	23,12	23,33	23,25	23,23	0,1
	Cahepph	42,19	41,87	42,67	42,24	0,4
	Astroph	104,55	104,05	104,66	104,42	0,3
	Com-Amazon	295,54	296,92	295,69	296,05	0,8
35 máquinas (280 núcleos)	Cahepth	42,78	41,99	42,47	42,41	0,4
	Cahepph	100,72	100,39	100,02	100,38	0,4
	Astroph	193,46	193,33	194,43	193,74	0,6
	Com-Amazon	420,45	449,88	450,13	440,15	17,1
25 máquinas (200 núcleos)	Cahepth	95,88	96,16	96,59	96,21	0,4
	Cahepph	187,39	187,03	187,66	187,36	0,3
	Astroph	319,82	319,03	319,17	319,34	0,4
	Com-Amazon	651,87	650,65	650,79	651,10	0,7
15 máquinas (120 núcleos)	Cahepth	127,41	126,24	126,12	126,59	0,7
	Cahepph	232,02	232,77	232,42	232,40	0,4
	Astroph	421,45	420,63	420,02	420,70	0,7
	Com-Amazon	850	921,09	920,34	897,14	40,8
5 máquinas (40 núcleos)	Cahepth	228,46	227,76	228,19	228,14	0,4
	Cahepph	252,99	252,67	252,15	252,60	0,4
	Astroph	543,77	543,21	543,12	543,37	0,4
	Com-Amazon	1258,24	1259,07	1258,49	1258,60	0,4
1 máquina (8 núcleos)	Cahepth	337,13	337,67	337,56	337,45	0,3
	Cahepph	427,87	428,19	428,13	428,06	0,2
	Astroph	903,24	904,89	904,13	904,09	0,8
	Com-Amazon	1734,38	1733,79	1734,66	1734,28	0,4

Tabela A.3: SGBD Paralelo - Tempo de execução em minutos. Grafos reais executados no *cluster paradent.*

Configuração do Cluster	Grafo	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
45 máquinas (360 núcleos)	Cahepth	12,51	13,02	11,49	12,34	0,8
	Cahepph	23,39	24,12	23,14	23,55	0,5
	Astroph	91,21	89,78	90,78	90,59	0,7
	Com-Amazon	121,41	120,12	120,31	120,61	0,7
35 máquinas (280 núcleos)	Cahepth	34,45	33,65	34,10	34,07	0,4
	Cahepph	82,36	81,76	82,76	82,29	0,5
	Astroph	121,39	122,03	121,45	121,62	0,4
	Com-Amazon	202,45	201,23	202,04	201,91	0,6
25 máquinas (200 núcleos)	Cahepth	90,47	91,05	90,53	90,68	0,3
	Cahepph	147,54	148,12	148,01	147,89	0,3
	Astroph	211,28	212,1	211,4	211,59	0,4
	Com-Amazon	334,65	333,13	334,56	334,11	0,9
15 máquinas (120 núcleos)	Cahepth	166,57	166,35	165,1	166,01	0,8
	Cahepph	217,42	216,59	216,92	216,98	0,4
	Astroph	350,11	351,39	350,62	350,71	0,6
	Com-Amazon	707,39	708,14	707,12	707,55	0,5
5 máquinas (40 núcleos)	Cahepth	258,55	259,29	259,05	258,96	0,4
	Cahepph	332,19	331,78	332,03	332,00	0,2
	Astroph	702,44	703,15	702,31	702,63	0,5
	Com-Amazon	1708,17	1707,91	1707,03	1707,70	0,6
1 máquina (8 núcleos)	Cahepth	428,35	428,89	429,53	428,92	0,6
	Cahepph	568,37	569,23	569,54	569,05	0,6
	Astroph	1823,32	1822,98	1823,12	1823,14	0,2
	Com-Amazon	3415,33	3414,97	3414,96	3415,09	0,2

A.2 - Dados da Figura 8.3

Tabela A.4: SGBD Paralelo - *Speedup*.

Configuração do Cluster	Grafo	1° Execução	2° Execução	3° Execução	Speedup Médio	Desvio Padrão
45 máquinas (360 núcleos)	Cahepth	34,24	32,94	37,38	34,85	2,28
	Cahepph	24,30	23,60	24,61	24,17	0,52
	Astroph	19,99	20,30	20,08	20,13	0,16
	Com-Amazon	28,13	28,43	28,38	28,31	0,16
35 máquinas (280 núcleos)	Cahepth	12,43	12,75	12,60	12,59	0,16
	Cahepph	6,90	6,96	6,88	6,92	0,04
	Astroph	15,02	14,94	15,01	14,99	0,04
	Com-Amazon	16,87	16,97	16,90	16,91	0,05
25 máquinas (200 núcleos)	Cahepth	4,73	4,71	4,74	4,73	0,02
	Cahepph	3,85	3,84	3,85	3,85	0,00
	Astroph	8,63	8,59	8,62	8,62	0,02
	Com-Amazon	10,21	10,25	10,21	10,22	0,03
15 máquinas (120 núcleos)	Cahepth	2,57	2,58	2,60	2,58	0,02
	Cahepph	2,61	2,63	2,63	2,62	0,01
	Astroph	5,21	5,19	5,20	5,20	0,01
	Com-Amazon	4,83	4,82	4,83	4,83	0,00
5 máquinas (40 núcleos)	Cahepth	1,66	1,65	1,66	1,66	0,00
	Cahepph	1,71	1,72	1,72	1,71	0,00
	Astroph	2,60	2,59	2,60	2,59	0,00
	Com-Amazon	2,00	2,00	2,00	2,00	0,00

A.3 - Dados da Figura 8.4

Tabela A.5: SGBD Paralelo - Eficiência.

Configuração do Cluster	Grafo	1° Execução	2° Execução	3° Execução	Speedup Médio	Desvio Padrão
45 máquinas (360 núcleos)	Cahepth	0,10	0,73	0,83	0,55	0,40
	Cahepph	0,54	0,52	0,55	0,54	0,01
	Astroph	0,44	0,45	0,45	0,45	0,00
	Com-Amazon	0,63	0,63	0,63	0,63	0,00
35 máquinas (280 núcleos)	Cahepth	0,36	0,36	0,36	0,36	0,00
	Cahepph	0,20	0,20	0,20	0,20	0,00
	Astroph	0,43	0,43	0,43	0,43	0,00
	Com-Amazon	0,48	0,48	0,48	0,48	0,00
25 máquinas (200 núcleos)	Cahepth	0,19	0,19	0,19	0,19	0,00
	Cahepph	0,15	0,15	0,15	0,15	0,00
	Astroph	0,35	0,34	0,34	0,34	0,00
	Com-Amazon	0,41	0,41	0,41	0,41	0,00
15 máquinas (120 núcleos)	Cahepth	0,17	0,17	0,17	0,17	0,00
	Cahepph	0,17	0,18	0,18	0,17	0,00
	Astroph	0,35	0,35	0,35	0,35	0,00
	Com-Amazon	0,32	0,32	0,32	0,32	0,00
5 máquinas (40 núcleos)	Cahepth	0,33	0,33	0,33	0,33	0,00
	Cahepph	0,34	0,34	0,34	0,34	0,00
	Astroph	0,52	0,52	0,52	0,52	0,00
	Com-Amazon	0,40	0,40	0,40	0,40	0,00

A.4 - Dados da Figura 8.3

Tabela A.6: HEDA - *Speedup*.

Configuração do Cluster	Grafo	1° Execução	2° Execução	3° Execução	Speedup Médio	Desvio Padrão
45 máquinas (360 núcleos)	Cahepth	14,58	14,47	14,52	14,52	0,05
	Cahepph	10,14	10,23	10,03	10,13	0,10
	Astroph	8,64	8,70	8,64	8,66	0,03
	Com-Amazon	5,87	5,84	5,87	5,86	0,02
35 máquinas (280 núcleos)	Cahepth	7,88	8,04	7,95	7,96	0,08
	Cahepph	4,25	4,27	4,28	4,26	0,02
	Astroph	4,67	4,68	4,65	4,67	0,02
	Com-Amazon	4,13	3,85	3,85	3,94	0,16
25 máquinas (200 núcleos)	Cahepth	3,52	3,51	3,49	3,51	0,01
	Cahepph	2,28	2,29	2,28	2,28	0,00
	Astroph	2,82	2,84	2,83	2,83	0,01
	Com-Amazon	2,66	2,66	2,67	2,66	0,00
15 máquinas (120 núcleos)	Cahepth	2,65	2,67	2,68	2,67	0,02
	Cahepph	1,84	1,84	1,84	1,84	0,00
	Astroph	2,14	2,15	2,15	2,15	0,01
	Com-Amazon	2,04	1,88	1,88	1,94	0,09
5 máquinas (40 núcleos)	Cahepth	1,48	1,48	1,48	1,48	0,00
	Cahepph	1,69	1,69	1,70	1,69	0,00
	Astroph	1,66	1,67	1,66	1,66	0,00
	Com-Amazon	1,38	1,38	1,38	1,38	0,00

Tabela A.7: HEDA - Eficiência.

Configuração do Cluster	Grafo	1° Execução	2° Execução	3° Execução	Speedup Médio	Desvio Padrão
45 máquinas (360 núcleos)	Cahepth	0,32	0,32	0,32	0,32	0,00
	Cahepph	0,23	0,23	0,22	0,23	0,00
	Astroph	0,19	0,19	0,19	0,19	0,00
	Com-Amazon	0,13	0,13	0,13	0,13	0,00
35 máquinas (280 núcleos)	Cahepth	0,13	0,13	0,13	0,13	0,00
	Cahepph	0,12	0,11	0,11	0,11	0,00
	Astroph	0,10	0,10	0,10	0,10	0,00
	Com-Amazon	0,07	0,07	0,07	0,07	0,00
25 máquinas (200 núcleos)	Cahepth	0,11	0,11	0,11	0,11	0,00
	Cahepph	0,11	0,11	0,11	0,11	0,00
	Astroph	0,11	0,11	0,11	0,11	0,00
	Com-Amazon	0,07	0,07	0,07	0,07	0,00
15 máquinas (120 núcleos)	Cahepth	0,14	0,14	0,14	0,14	0,00
	Cahepph	0,14	0,13	0,13	0,13	0,01
	Astroph	0,10	0,10	0,10	0,10	0,00
	Com-Amazon	0,11	0,11	0,11	0,11	0,00
	Cahepth	0,33	0,33	0,33	0,33	0,00

5 máquinas	Cahepph	0,28	0,28	0,28	0,28	0,00
(40 núcleos)	Astroph	0,20	0,20	0,20	0,20	0,00
	Com-Amazon	0,28	0,28	0,28	0,28	0,00

A.5 - Dados da Figura 8.5

Tabela A.8: SGBD Paralelo - Tempo de execução de grafo de 20.000 vértices com variação de arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de arestas	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
200.000	8,20	8,17	8,25	8,21	0,0
300.000	12,13	12,17	12,25	12,18	0,1
400.000	16,56	16,74	16,89	16,73	0,2
500.000	20,46	20,41	20,5	20,46	0,05

Tabela A.9: HEDA - Tempo de execução de grafo de 20.000 vértices com variação de arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de arestas	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
200.000	37,2	39,45	38,5	38,38	1,1
300.000	50,56	51,45	51,32	51,11	0,5
400.000	63,16	62,51	64,12	63,26	0,8
500.000	105,07	106,22	106,13	105,81	0,64

Tabela A.10: Neo4J - Tempo de execução de grafo de 20.000 vértices com variação de arestas executados no *cluster graphene* com 1máquina (4 núcleos).

Número de arestas	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
200.000	60,04	60,32	60,07	60,14	0,2
300.000	92,98	92,33	92,66	92,66	0,3
400.000	127,87	127,33	127,44	127,55	0,3
500.000	210,09	210,31	210,39	210,26	0,2

Tabela A.11: Sequencial - Tempo de execução de grafo de 20.000 vértices com variação de arestas executados no *cluster graphene* com 1máquina (4 núcleos).

Número de arestas	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
200.000	127,33	126,98	127,15	127,15	0,2
300.000	178,44	178,09	178,32	178,28	0,2
400.000	213,47	212,89	213,15	213,17	0,3
500.000	263,27	263,16	263,02	263,15	0,1

A.6 - Dados da Figura 8.6

Tabela A.12: SGBD Paralelo - *Speedup* da execução de grafo de 20.000 vértices com variação de arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de arestas	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
200.000	75,77	75,81	75,21	75,60	0,3
300.000	96,32	95,92	95,20	95,81	0,6
400.000	81,23	80,30	79,73	80,42	0,8
500.000	88,69	88,84	88,40	88,64	0,2

Tabela A.13: HEDA - *Speedup* da execução de grafo de 20.000 vértices com variação de arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de arestas	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
200.000	11,40	11,08	11,16	11,22	0,2
300.000	14,27	14,04	14,09	14,13	0,1
400.000	15,99	16,14	15,74	15,96	0,2
500.000	16,36	17,14	17,15	16,88	0,5

A.7 - Dados da Figura 8.7

Tabela A.14: SGBD Paralelo - Tempo de execução de grafos com variação de vértices e arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de vértices	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
10.000	3,01	3,15	3,06	3,07	0,1
20.000	8,2	8,17	8,25	8,21	0,04
30.000	16,69	16,58	16,6	16,62	0,1
40.000	39,04	38,98	39,15	39,06	0,1
50.000	46,34	46,49	46,27	46,37	0,1

Tabela A.15: HEDA - Tempo de execução de grafos com variação de vértices e arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de vértices	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
10.000	30,87	29,41	28,23	29,50	1,3
20.000	40,56	41,45	41,32	41,11	0,48
30.000	104,54	102,34	103,55	103,48	1,1
40.000	216,71	218,3	217,45	217,49	0,8
50.000	320,32	319,56	322,45	320,78	1,5

Tabela A.16: Neo4J - Tempo de execução de grafos com variação de vértices e arestas executados no *cluster graphene* com 1máquina (4 núcleos).

Número de vértices	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
10.000	45,87	45,39	45,63	45,63	0,2
20.000	60,04	60,32	60,07	60,14	0,2
30.000	129,76	129,58	129,44	129,59	0,2
40.000	247,51	247,32	247,49	247,44	0,1
50.000	510,73	510,62	510,88	510,74	0,1

Tabela A.17: Sequencial - Tempo de execução de grafos com variação de vértices e arestas executados no *cluster graphene* com 1máquina (4 núcleos).

Número de vértices	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
10.000	93,19	93,31	93,45	93,32	0,1
20.000	127,33	126,98	127,15	127,15	0,2
30.000	272,27	272,54	272,13	272,31	0,2
40.000	389,49	389,02	389,28	389,26	0,2
50.000	498,41	498,83	498,18	498,47	0,3

A.8 - Dados da Figura 8.8

Tabela A.18: SGBD Paralelo - *Speedup* da execução de grafos com variação de vértices e arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de vértices	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
10.000	103,24	97,86	100,97	100,69	2,7
20.000	75,77	75,81	75,21	75,60	0,34
30.000	69,79	70,10	70,09	69,99	0,2
40.000	70,16	70,28	69,94	70,13	0,2
50.000	65,14	64,97	65,22	65,11	0,1

Tabela A.19: HEDA - *Speedup* da execução de grafos com variação de vértices e arestas executados no *cluster graphene* com 120 máquinas (480 núcleos).

Número de vértices	1° Execução	2° Execução	3° Execução	Tempo Médio	Desvio Padrão
10.000	8,12	8,48	9,14	8,58	0,5
20.000	11,40	11,08	11,16	11,22	0,17
30.000	9,42	9,63	9,49	9,51	0,1
40.000	7,56	7,50	7,53	7,53	0,0
50.000	8,80	8,83	8,74	8,79	0,0

Artigo Publicado

Fernandes, F. S. and Yero, E, J, H.. (2014) MapReduce vs Bancos de Dados Paralelos no cálculo de medidas de centralidade em grafos In: *XXXIV Congresso da Sociedade Brasileira de Computação - XIII – WPerformance: Workshop em Desempenho de Sistemas Computacionais e de Comunicação*. Brasília. p. 2009-2013.