Universidade Federal Rural de Pernambuco

Bacharelado em Sistemas de Informação

Migração de aplicações de análise de dados em lote para um cenário de grandes dados: estudo de caso da ferramenta scriptLattes

Victor Pereira Luna

Recife
2015

Victor Pereira Luna

Migração de aplicações de análise de dados em lote para um cenário de grandes dados: estudo de caso da ferramenta scriptLattes

Monografia apresentada como exigência para obtenção do grau de Bacharel em Sistemas de Informação da Universidade Federal Rural de Pernambuco.

Orientador: Glauco Gonçalves

Co-orientador: Victor Medeiros

Recife
2015

AGRADECIMENTOS

Agradeço primeiramente ao meu orientador Glauco Gonçalves, por orientar e conduzir este trabalho com tanto afinco. Também me proporcionou um grande aprendizado na elaboração deste trabalho. Agradeço também ao meu co-orientador Victor Medeiros, pelo compartilhamento de conhecimento e técnicas para solucionar os problemas que tivemos. Sem vocês dois esse trabalho não teria o mesmo resultado. Agradeço a minha namorada Marcella que me apoiou em diversos momentos difíceis durante a elaboração do trabalho e que teve bastante compreensão nos momentos de ausência. Agradeço a minha família, em especial a meu pai Fernando, minha mãe Albenildes e minha irmã Rebeca, pelo incentivo dado durante toda a minha vida escolar e acadêmica, me proporcionando uma excelente formação pessoal e profissional. Também agradeço a meus amigos do grupo UKM e Bons Meninos, os quais me proporcionaram excelentes momentos durante minha vida: Matheus, Rodrigo, Bruno, Raphael, Amanda, Aninha, Romulo, Waldemir, Augusto, Aaron e Bruninho. Agradeço a Aninha, que me ajudou com excelentes traduções no decorrer do trabalho.

RESUMO

Um dos desafios mais importantes da computação hoje é o desenvolvimento de métodos, ferramentas e técnicas para lidar com a massa de dados que cresce a cada dia no mundo. Grandes empresas como Google, Amazon, Oracle, Apache, IBM, junto com a comunidade de software livre e aberto vêm trabalhando no desenvolvimento de software que tenha a capacidade de processar e gerenciar todas essas informações de forma mais eficiente.
Assim, o principal desafio das aplicações que trabalham analisando dados é o de completar o processamento das grandes massas em tempo hábil. Este desafio é ainda mais proeminente no caso de aplicações para a análise de dados em lote, já que muitas destas aplicações foram concebidas para lidar com massas de dados muito menores do que aquelas disponíveis no cenário atual.
Este trabalho apresenta um estudo sobre o problema da migração de aplicações de análise de dados em lote, por meio do estudo de caso com a ferramenta scriptLattes, mostrando as soluções e os ganhos obtidos no cenário de grandes dados. A ferramenta foi analisada em profundidade e suas principais limitações de desempenho foram investigadas e solucionadas através de ferramentas de computação distribuída. Os resultados obtidos mostram como aplicações deste tipo podem ser modificadas para escalar e atender o cenário de grandes dados.

Palavras-chave: Big data, Análise de dados em lote, Hadoop, Memcached.

ABSTRACT

Current information systems research is moving towards development of methods, tools and techniques to deal with the growing amount of stored data in the world. Companies like Google, Amazon, Oracle, Apache, IBM, and the free and open source software community are developing software and tools to process and manage this huge amount of data in an efficient fashion.

The main challenge of applications for data analysis today is to completely process the big data in a timely manner. This challenge is even more prominent in the case of applications for batch data analysis, since such applications were designed to cope with datasets smaller than the available ones.

In this monograph, we investigate the problem of migrate an application for batch data analysis to the big data scenario, which was accomplished through the case study of scripLattes. The main performance limitations of this tool were analyzed and solved through tools for distributed computing. The results show that such type of application can scale to suit big data scenarios.

Keywords: Big data, Batch data analysis, Hadoop, Memcached.

Palavras-chave: Big data, análise de dados em lote, aplicações legadas, hadoop, memcached.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE CÓDIGOS

1 INTRODUÇÃO

1.1 Contextualização

A facilidade de troca de informações que o desenvolvimento da internet ofereceu ao mundo e o barateamento do armazenamento de dados foram primordiais para o aumento na quantidade de informações geradas, armazenadas e trocadas diariamente por pessoas, empresas e softwares. Segundo um estudo realizado em 2011 pela *International Data Corporation* (IDC), a previsão do volume de dados gerados na internet em 2015 deve chegar a 8 zettabytes (Gantz and Reinsel 2011). Esse número representa um aumento considerável em um curto intervalo de tempo, levando em consideração que o valor medido em 2011 foi de 1,7 zettabytes. Ao mesmo tempo, observou-se também um grande avanço no desenvolvimento de software para atender as constantes demandas do mercado para comunicação.

Com a chegada dos smartphones e de tecnologias como o 3G e 4G, os usuários passaram a utilizar a internet na maioria dos lugares que frequentam. As empresas se emanciparam e passaram a atuar por todo o mundo. Um estudo de como essa emancipação provocou esse aumento foi realizado em 2010 (Economist 2010) e mostrou que a empresa Walmart gera cerca de um milhão de transações por hora.

Com o passar do tempo, notou-se que os softwares desenvolvidos e as tecnologias atuais não estavam sendo suficientes para tratar em tempo hábil a quantidade de dados gerados, ou seja, o poder de processamento das aplicações que trabalhavam analisando dados não era proporcional à velocidade em que os dados eram gerados. A partir dai, criou-se uma preocupação em torno de como migrar as aplicações de análise e processamento de dados para trabalhar com novas tecnologias criadas para a melhoria no processamento desses dados.

Alguns trabalhos têm mostrado como as novas tecnologias têm sido utilizadas para prover soluções para processamento de dados em aplicações que estão inseridas em um cenário de grande massa de dados.

O trabalho de (Bernardes 2014) promove um estudo aprofundado sobre a arquitetura do software Hadoop, explicando em detalhes o funcionamento do seu sistema de arquivo distribuído (HDFS) e seu modelo de programação MapReduce. Ao fim da pesquisa é proposta uma arquitetura de aplicação Big Data voltada para

um estudo de caso real, o qual envolve a extração e análise de publicações de redes sociais com foco voltado para políticas públicas.

Já (Silva and Macêdo) estudam o caso de dados de monitoramento de tráfego de veículos em uma cidade e procura apresentar os principais algoritmos de mineração de dados para cenários de grandes massas de dados (*BigData*[1]) considerando a utilização de ambientes de computação em nuvem e o software Hadoop, além de apresentar boas práticas para o desenvolvimento de soluções neste contexto.

Contudo, migrar aplicações existentes para cenários de grandes massas de dados ainda é uma tarefa complexa que envolve o mapeamento cuidadoso das limitações e necessidades da aplicação em termos de processamento, memória, acesso ao disco e rede, bem como a posterior adequação da ferramenta para o uso das tecnologias de computação e armazenamento distribuído disponíveis para o tratamento de grandes massas de dados (Gomes 2011).

Este trabalho então apresenta a investiga o problema da migração de aplicações de processamento de dados em lote por meio da análise e solução de um estudo de caso com o scriptLattes*,* no intuito de melhorar o seu desempenho em um cenário que exige grande processamento de dados.

O scriptLattes é um software livre criado por Jesús P. Mena-Chalco e Roberto M. Cesar-Jr, com o intuito de facilitar o trabalho de compilação ou sumarização de produções bibliográficas de um grupo de pesquisadores, que eram realizadas manualmente na plataforma Lattes, demandando um grande esforço mecânico e suscetível a falhas. Segundo (J. Mena-Chalco and Cesar Junior 2011), a aplicação está sendo usada por diversas instituições de ensino e pesquisa no país em razão da necessidade de levantar, explorar e analisar de maneira automática currículos de pesquisadores cadastrados na Plataforma Lattes. Desta forma, universidades, centros de pesquisa, órgãos de fomento e agentes governamentais podem obter, de modo automático, informação atualizada sobre a produção intelectual de pesquisadores do seu interesse consolidadas na forma de gráficos e métricas.

Apesar de ter o potencial de ser empregada como ferramenta para tomada de decisão em diversas instituições e departamentos, a ferramenta tem sido empregada apenas em cenários com centenas de pesquisadores (cf. (J. P. Mena-Chalco, Digiampietri, and Oliveira 2012) e (J. Mena-Chalco and Cesar Junior 2011)),

[1] Apesar do termo inglês *BigData* ser mais popular informalmente em português, optamos pelo uso do termo grandes massas dados posto que os trabalhos acadêmicos aqui referenciados tem optado por este uso.

apresentando limitações quando submetida à análise de casos com milhares ou dezenas de milhares de professores. Note que esta é a escala adequada quando se trata de um órgão governamental que precise analisar pesquisadores de um estado ou região, ou mesmo, ao avaliar pesquisadores em uma área a nível nacional. Tal limitação ocorre por causa da quantidade de recursos memória demandada pela ferramenta e do tempo necessário para conclusão do processamento.

Devido a isso, a ferramenta foi escolhida como um estudo de caso para a utilização de conceitos, métodos e ferramentas de um cenário de grande processamento de dados, no intuito de identificar seus problemas e posteriormente solucioná-los. Para a resolução dos problemas, foram escolhidas as ferramentas Hadoop e Memcached, que serão abordadas com mais detalhes durante este trabalho, para resolver os gargalos no tempo de execução e no armazenamento em memória da ferramenta, respectivamente.

1.2 Objetivos

Este trabalho de conclusão de curso tem como principal objetivo investigar o problema de migrar aplicações para análise de dados em lote para o cenário de grandes massas de dados por meio de um estudo de caso com a ferramenta scriptLattes. Especificamente, este trabalho pretende:

- Investigar as limitações do scriptLattes ao cenário de grandes dados em termos de consumo de recursos e tempo de processamento;
- Tratar as limitações observadas por meio de técnicas e ferramentas para distribuição de computação;
- Avaliar, apresentar e discutir os resultados obtidos pela distribuição da computação;
- Condensar as lições aprendidas em um método a ser seguido para adaptar outras ferramentas do mesmo tipo ao mesmo cenário.

1.3 Organização da monografia

Este trabalho está estruturado em cinco capítulos:

- **Capítulo 1:** Introdução – Apresentação e objetivos do trabalho.

- **Capítulo 2:** Ferramentas utilizadas – Descrição do software utilizado no estudo de caso e as ferramentas utilizadas.

- **Capítulo 3:** Implementação da solução – Apresentação da integração das ferramentas Hadoop e Memcached.

- **Capítulo 4:** Resultados – Resultado dos testes realizados em cima das soluções Hadoop e Memcached.

- **Capítulo 5:** Conclusão – Dificuldades encontradas e lições aprendidas.

2 FERRAMENTAS UTILIZADAS

Neste capítulo serão descritos a arquitetura e o funcionamento das ferramentas utilizadas no trabalho. Primeiramente, na seção 2.1, será apresentada a ferramenta a ser usada no caso de teste, e posteriormente as ferramentas de computação distribuída que foram utilizadas para melhoria das limitações identificadas no scriptLattes serão apresentadas nas seções 2.2 e 2.3. Finalmente, as ferramentas de aferição (*profiling*) de desempenho de código utilizadas para identificação dos gargalos na aplicação são apresentadas nas seções 2.4 e 2.5.

2.1 ScriptLattes

O scriptLattes é um programa desenvolvido utilizando a linguagem de programação Python, e é uma ferramenta de software livre pioneira na prospecção de conjuntos de dados acadêmicos provenientes da base Currículos Lattes (J. Mena-Chalco and Cesar Junior 2011). Tal programa permite a criação de relatórios acadêmicos de forma automática, a partir das informações cadastradas na plataforma Lattes do CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

Os “Currículos Lattes” são considerados um padrão nacional de avaliação (Amorin 2003). Esses currículos representam principalmente um histórico das atividades acadêmicas e científicas de pesquisadores brasileiros e estrangeiros que atuam ou atuaram no país.

O scriptLattes recebe como entrada um arquivo texto de contendo informação indicativa dos Currículos Lattes que formam a base de pesquisadores a ser investigada. Os pesquisadores da base de investigação não necessitam possuir qualquer correlação, a priori. Além de um identificador único do Currículo Lattes do pesquisador (obrigatório), a entrada ainda pode conter o nome do pesquisador, o período de permanência do membro no grupo (que serve filtrar publicações de um determinado período. e.g. 1992-2008) e um rótulo identificador (utilizado na visualização do grafo de colaborações). Um exemplo de um arquivo de entrada com 14 pesquisadores é mostrado na Figura 1.

```
4575931307749163 , Carlos Hitoshi Morimoto         , 1999-HOJE                          , Professor
0131770792108992 , Joao Eduardo Ferreira           , 1999-2000 & 2002-HOJE              , Professor
0362417828475021 , Junior Barrera                  , 1992-2008                          , Professor
5416099300504556 , Kelly Rosa Braghetto            ,                                    , Professor
0926213060635986 , Marcel Parolin Jackowski        ,                                    , Professor
0644408634493034 , Nina Sumiko Tomita Hirata       ,                                    , Professor
5251389003736909 , Paulo A. Vechiatto de Miranda   ,                                    , Professor
1647118503085126 , Roberto Hirata Junior           ,                                    , Professor
2240951178648368 , Roberto Marcondes Cesar Junior  ,                                    , Professor
9283304583756076 , Ronaldo Fumio Hashimoto         ,                                    , Professor
4727357182510680 , Jesus P. Mena-Chalco            , 1995-1999 & 2003-2005 & 2008-HOJE  , Colaborador
1228255861618623 , David Correa Martins Junior     ,                                    , Colaborador
1660070580824436 , Fabricio Martins Lopes          ,                                    , Colaborador
2837012019824386 , Andréa Britto Mattos            ,                                    , Aluno
```

Figura 1 – Lista de Ids do scriptLattes (Fonte: (“Script Lattes”))

A partir da lista de entrada, o scriptLattes baixa automaticamente os currículos Lattes por meio de um *crawler*, compila as listas de produções, tratando apropriadamente as produções duplicadas e similares. Após a compilação das listas, a ferramenta gera relatórios em formato HTML com listas de produções e orientações devidamente separadas. Além disso, a ferramenta gera também um mapa da distribuição geográfica dos pesquisadores, um grafo de coautoria e o grau de colaboração (AuthorRank)[2] entre os pesquisadores selecionados.

Para calcular o grau de colaboração entre os pesquisadores de um grupo a ferramenta utiliza o algoritmo proposto por Xiaoming Liu, denominado *AuthorRank*, que é uma adaptação do algoritmo *PageRank* para realizar a busca de páginas relevantes (Liu 2005).

Na Figura 2 é mostrado um exemplo de como pode ser realizada a representação de um grafo de coautoria. O primeiro (mais à esquerda) é um grafo de colaboração sem pesos, onde as arestas representam apenas a ligação de colaboração. O segundo (ao meio) representa um grafo de colaboração com pesos, onde o peso é representado pelo número de produções acadêmicas realizadas em conjunto entre os nós. E por último, o grafo onde os pesos das arestas são normalizados a partir do total de produções acadêmicas realizadas em conjunto, onde é dado um valor, a partir do execução do *AuthorRank*, representando a importância dessa relação para com o conjunto de pesquisadores.

A figura mostra que para o autor M2, M1 é 75% importante em sua produção bibliográfica, porque M1 tem 75% de participação de coautoria. Porém, para M1, M2 é apenas 50% importante para sua produção bibliográfica. Outro exemplo é se observarmos que M1 possui 100% de colaboração na produção bibliográfica de M4.

[2] O grau de colaboração é uma medida do impacto ou relevância de um pesquisador em um grafo de colaborações. O nome inglês, AuthorRank, é utilizado simultaneamente tanto para a medida como para o algoritmo que a calcula.

Entretanto, para M1, M4 é apenas 33% importante em sua produção bibliográfica. Assim, conforme apontam (J. Mena-Chalco and Cesar Junior 2011), a normalização proposta por X. Liu faz com que as relações entre pesquisadores que produziram mais publicações tenha um peso maior, dando mais importância na produção realizada em colaboração com outro.

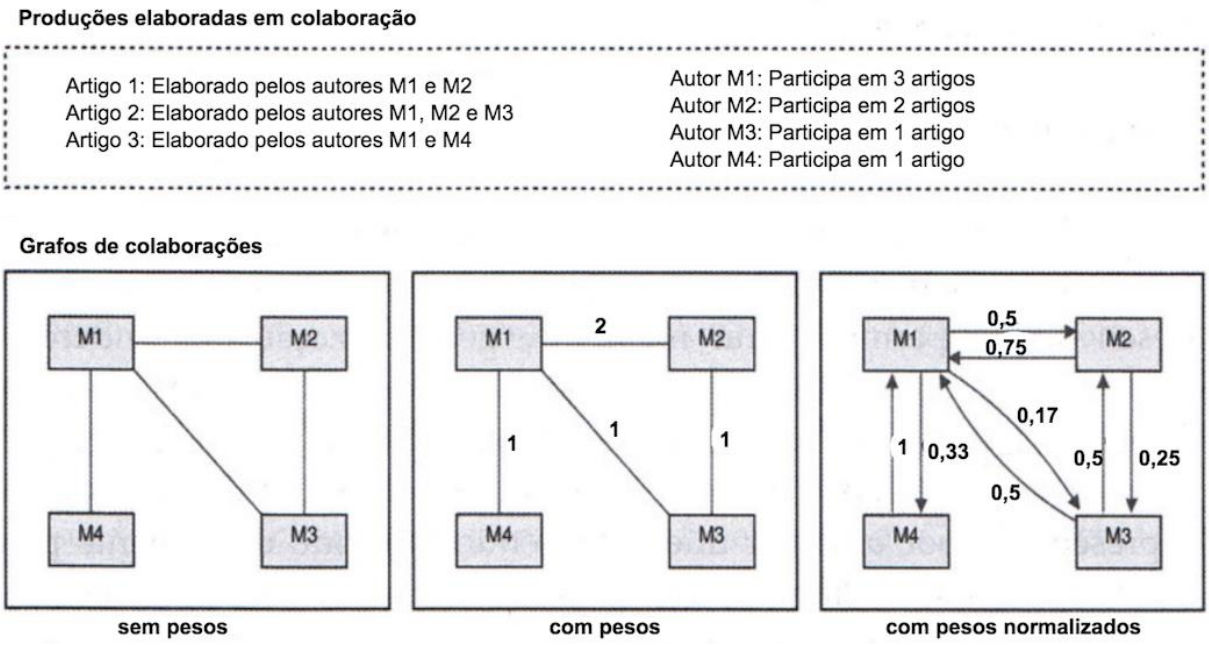

Figura 2 - Exemplo de grafos de colaboração (Fonte: (J. Mena-Chalco and Cesar Junior 2011))

2.2 Hadoop

Hadoop, criado por Doug Cutting em 2008 (White 2012), é uma das soluções mais utilizadas para análise e processamento de dados em cenários de grandes massas de dados.Esta solução utiliza como modelo de programação o MapReduce, juntamente com um sistema de arquivo distribuído chamado *Hadoop Distributed File System* (HDFS) para viabilizar o processamento de grandes volumes de dados em clusters.

O HDFS executa jobs que realizam a leitura, processamento e escrita de arquivos distribuídos e o seu grande diferencial é a alta capacidade de tolerância a falhas e o baixo custo de hardware que é requerido, fazendo com que ele se diferencie dos demais sistemas de arquivos distribuídos no mercado.

O Hadoop utiliza o modelo de programação MapReduce para a execução de programas. Este modelo de programação é baseado em duas primitivas da programação funcional: Map e Reduce. A execução MapReduce é feita a partir da função Map que recebe uma lista de pares chave-valor em sua entrada, realiza o

processamento da implementação do Map feita pelo usuário e gera uma saída que também é uma lista de pares chave-valor intermediários. Ao final de cada tarefa Map, um nó mestre do cluster coleta os dados da saída e, em seguida, ordena os resultados a partir da chave. Todas as chaves são divididas entre todas as tarefas Reduce. Para os pares que possuem as mesmas chaves, o nó mestre os designa para a mesma tarefa Reduce. A função Reduce recebe como entrada todos os valores de uma mesma chave, executa a implementação da função Reduce criada pelo usuário e gera como saída pares chave-valor, formando o resultado do processo MapReduce. O funcionamento das funções Map e Reduce podem ser acompanhadas pela Figura 3.

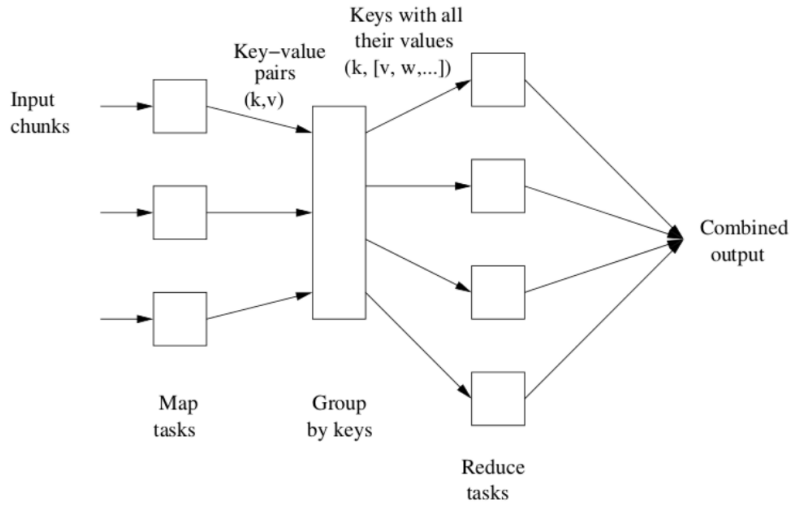

Figura 3 - Funcionamento das funções Map e Reduce (Fonte: (Rajaraman and Ullman 2012))

A arquitetura MapReduce é Mestre-Escravo. O nó mestre cria um número de tarefas Map e tarefas Reduce e realiza a alocação das tarefas para os nós Escravos. Como dito anteriormente, cada tarefa Map cria um arquivo intermediário que é usado como entrada na função Reduce, por isso é desejável limitar a quantidade de tarefas Reduce, caso contrário o número de arquivos intermediários poderá ser muito grande. Outra tarefa do nó Mestre é de armazenar o estado de cada tarefa Map e Reduce que pode possuir os estados: em espera, em execução ou finalizada. O nó Escravo reporta ao Mestre quando termina uma tarefa, a partir daí uma nova tarefa é escalonada pelo Mestre ao Escravo.

Outra característica do Hadoop é o seu processo de tolerância a falhas, onde o processo responsável pela alocação das tarefas (JobTracker) fica aguardando o término da execução do Map, realizado pelo (TaskTracker). Caso não haja um retorno ou o retorno vier com falha, o JobTracker é responsável por realocar a

mesma tarefa para um outro TaskTracker de um outro Escravo. Sempre que as tarefas são completadas, o TaskTracker informa o resultado para o JobTracker, que altera o estado da tarefa específica e o realoca em uma outra tarefa. Com isso, o processamento do MapReduce é completado quando todas as tarefas de Map e de Reduce são completadas.

2.3 Memcached

Memcached é um sistema de cache de objetos open-source que utiliza a memória RAM para o armazenamento de dados distribuídos em um cluster de máquinas. Utiliza como forma de armazenamento a estrutura de dados chave e valor, onde o tamanho máximo da chave é de 250 bytes e o tamanho máximo do valor é de 1MB.

Tem sido largamente utilizado em sites como Facebook, Twitter e Youtube, para acelerar o carregamento das páginas web dinâmicas que possuam conexões com banco de dados. O Facebok, por exemplo, hospeda a maior instalação de Memcached do mundo, utilizando 800 servidores de Memcached e obtendo um reservatório de 28 TB de memória, que permite uma taxa de acerto de cache de 99% (Issa and Figueira 2012).

Em clusters Memcached, não há uma comunicação entre os servidores, e sim a arquitetura cliente-servidor, sendo que o cliente utiliza uma biblioteca em uma linguagem de programação suportada (Python, Java, PHP e C), que possui um algoritmo que computa o hash da chave para descobrir qual o servidor em que o dado está guardado.

Na figura 4, podemos observar que a aplicação cliente se comunica com o Memcached de duas formas: utilizando a função Get, que é responsável por buscar um determinado valor armazenado através de uma chave; e por meio da função Set, que realiza o armazenamento de um determinado valor, utilizando uma chave identificadora. O balanceamento dos dados nos servidores é realizado através de um algoritmo que calcula um hash através da chave passada pelo usuário. Depois que o hash é calculado, a biblioteca Memcached no cliente procura em uma lista de servidores qual o responsável por aquele hash. A partir daí ele realiza uma requisição ao servidor solicitando o valor da chave passada pelo usuário.

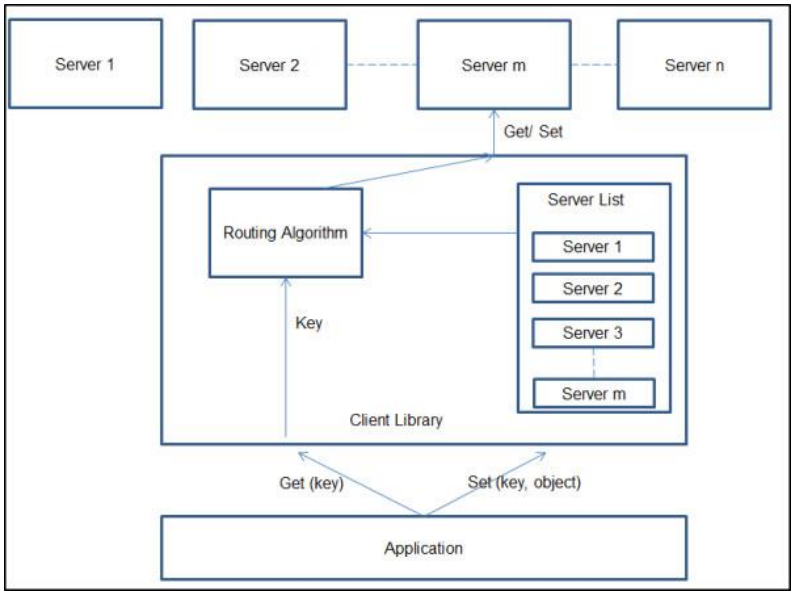

Figura 4 – Funcionamento do Memcached (Fonte: (Issa and Figueira 2012))

2.4 Line Profile

Line Profile é uma ferramenta desenvolvida em Python, por Robert Kern e mais cinco programadores entusiastas da linguagem. A ferramenta oferece suporte para programas escritos na linguagem Python, em versões igual ou superior a 2.7. Foi disponibilizado para download no *Python Package Index* (PyPI), que é um dos gerenciadores de pacotes da linguagem Python. Segundo dados levantados no *PypPI*, a ferramenta possui um alto grau de aceitação na comunidade e só no mês de janeiro de 2015 foram realizados 5770 downloads .

Quando instalado, basta selecionar o trecho de código que há a necessidade de realizar o *profiling*, marcando o método com o *decorator profile*, como mostrado na figura de código 1.

```
@profile
def slow_function(a, b, c):
    ...
```

Código 1 – Exemplo do funcionamento do Line Profiler (Fonte: (“Line Profiler”))

Após esse procedimento, basta executar o programa. A figura de código 2 ilustra o resultado mostrado ao final da execução. A tela apresentada contêm informações linha a linha do tempo de execução e porcentagem do tempo total consumido por cada linha do programa. No caso exemplificado, pode-se concluir que a linha 151 do software consome 13% do tempo de execução do programa e que o tempo de processamento é bem distribuído ao longo deste trecho do programa.

```
Pystone(1.1) time for 50000 passes = 2.48
This machine benchmarks at 20161.3 pystones/second
Wrote profile results to pystone.py.lprof
Timer unit: 1e-06 s

File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc - IntGlob
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

Código 2 – Exemplo do resultado do Line Profiler (Fonte: (“Line Profiler”))

2.5 Memory Profiler

Memory Profiler é uma ferramenta desenvolvida em Python, por Fabian Pedregosa e Philippe Gervais, junto com 24 programadores espalhados pelo mundo. Esta ferramenta foi inspirada no Line Profile, apresentado na seção 3.4, e também oferece suporte para programas escritos na linguagem Python, em versões igual ou superior a 2.7. Foi disponibilizado também para download no *Python Package Index* (PyPI) e possui código aberto. Assim como o Line Profile, esta ferramenta também possui um alto grau de aceitação na comunidade e só no mês de janeiro de 2015 já possui cerca de 5049 downloads realizados.

Como foi inspirado na ferramenta Line Profile, seus métodos de execução são similares, bastando selecionar o trecho de código que há a necessidade de realizar a aferição, marcando o método com o decorator *profile*, como mostrado na figura de código 1.

Após esse procedimento, basta executar o programa. Ao final da execução a é apresentada uma tela contendo informações linha a linha sobre consumo de memória do programa, conforme mostra a figura de código 3.

```
Line #    Mem usage  Increment   Line Contents
================================================
     3                            @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

Código 3 - Exemplo do resultado do Memory Profile (Fonte: (“Memory Profile”))

No caso exemplificado, pode-se observar o que cada linha consome de memória e o total de consumo parcial. Dando uma visão do consumo de memória específico e geral do programa.

3 METODOLOGIA E IMPLEMENTAÇÃO DA SOLUÇÃO

No início deste trabalho, foi definido um método para investigar o problema de migração da ferramenta scriptLattes para o cenário de grandes dados, afim de conseguir sanar os problemas de tempo de processamento e consumo de memória previamente observados. Este método pode ser descrito por meio de quatro passos: identificação dos pontos de gargalo da ferramenta; escolha das soluções distribuídas adequadas para lidar com os gargalos; adequação da ferramenta para o ambiente distribuído; avaliação das modificações feitas em busca de indicações de melhorias.

De modo geral, o método definido segue uma abordagem prática e tem como objetivo averiguar, investigar e sanar os pontos de gargalo na ferramenta por meio de repetidas execuções e concomitante análise do código fonte. A partir de uma investigação preliminar feita com auxílio de ferramentas específicas identificam-se os possíveis pontos de gargalo, isto é, as funções que mais consomem tempo de processamento e as estruturas de dados que mais utilizam a memória. Estes pontos são analisados em detalhes e as soluções distribuídas mais adaptadas são elencadas. Este passo requer um minucioso estudo dos tipos dos gargalos (memória, processamento, acesso ao disco ou mesmo rede, para o caso de aplicações em lote que busquem sua informação através da rede) e das interações que possam existir entre eles, de modo que as soluções distribuídas escolhidas sejam adequadas ao problema em questão. Finalmente, as modificações são implementadas e avaliadas.

As seções a seguir descrevem como cada um destes passos foi realizado na prática. Para um melhor entendimento, o último passo do método, que visa avaliar as modificações feitas na ferramenta será abordado no próximo capítulo (capítulo 4).

3.1 Identificação dos pontos de gargalo

Como o primeiro procedimento visa identificar os pontos de gargalo na ferramenta, deve-se proceder esta identificação por meio de ferramentas adequadas a este fim. Considerando que ferramenta tem o propósito de analisar dados em lote a partir de arquivos no disco[3], e cujo código fonte é disponível, a solução adequada para este caso é o emprego de ferramentas de aferição (*profiling*) de tempo e

[3] Note que o scriptLattes opera buscando os currículos dos pesquisadores na base de dados Lattes, a Internet e armazenando-os em arquivos em disco para análise posterior. As avaliações feitas neste trabalho limitam-se ao uso da ferramenta no modo de leitura a partir de arquivos em disco.

memória. Para este caso, sendo Python a linguagem da ferramenta scriptLattes, escolheu-se o Time Profile e o Memory Profiler (abordados nos itens 2.4 e 2.5) para a identificação do problema de tempo de processamento e de consumo de memória, respectivamente.

Ao final do *profiling* de tempo, o resultado mostrou que havia um trecho de código que consumia bastante tempo do processamento. Este trecho de código foi identificado como o algoritmo *AuthorRank* que calcula o grau de colaboração entre os autores buscados. No item 2.1, há uma introdução ao algoritmo *AuthorRank*, explicando a importância dele na ferramenta. A figura a seguir contém a implementação desse algoritmo, tal como implementado na ferramenta, utilizando a linguagem de programação Python.

```
print "[CALCULANDO AUTHOR-RANK (PROCESSO ITERATIVO)]"
for index in range(0,iteracoes):
    self.vectorRank = self.calcularRanks(self.vectorRank)

def calcularRanks(self, vectorRank):
    vectorRankNovo = numpy.zeros( len(vectorRank), dtype=numpy.float32)
    d = 0.85
    for i in range(0, len(vectorRank)):
        soma = 0
        for j in range(0, len(vectorRank)):
                soma += vectorRank[j] * self.matriz[j , i]
                vectorRankNovo[i] = (1-d) + d*soma
    return vectorRankNovo
```

Código 4 – Algoritmo AuthorRank implementado em Python

No algoritmo é criado o vectorRank, que é um vetor de tamanho N, onde N é o número de pesquisadores buscados, e que contém os valores que representam a precedência de cada pesquisador no grupo de estudo. Esse valor é calculado a partir de uma matriz de colaboração de tamanho N x N, preenchida com a frequência de colaborações (coautoria) normalizada de um pesquisador para outro. O valor do AuthorRank é obtido por meio de um método iterativo que envolve 100 repetições para que o resultado tenha maior precisão. O detalhamento do cáculo para obter o vectorRank é explicado melhor no próximo item.

Para cada pesquisador que é adicionado ao lote de processamento, o custo computacional será multiplicado pelo total de pesquisadores. Isso faz com que o aumento no tempo de processamento do algoritmo seja não-linear e a busca por uma grande quantidade de pesquisadores se torne cada vez mais lenta neste cenário.

Já no Profiling de memória, observou-se que o problema estava na estrutura

de dados que armazenava as informações referente aos pesquisadores buscados. O programa possui uma classe chamada “Membro“ que contém todos os atributos de um pesquisador e para cada pesquisador buscado, o programa adiciona esse objeto “Membro“ em uma instância da classe “Grupo”, que funciona como uma coleção dos objetos membros (como mostra a figura de Código 5).

```
def carregarDadosCVLattes(self):
    indice = 1
    for membro in self.listaDeMembros:
        print "\n[LENDO REGISTRO LATTES: " + str(indice) + "o. DA LISTA]"
        indice += 1
        membro.carregarDadosCVLattes()
        membro.filtrarItemsPorPeriodo()
        print membro
```

Código 5 – Exemplo do carregamento de dados dos currículos

Porém, como o programa guarda todos os dados dos pesquisadores em memória, a busca com uma grande quantidade de pesquisadores irá exigir uma máquina que possua memória proporcionalmente grande para a execução do processo, caso contrário, a execução se torna impossível de ser realizada devido ao estouro de memória, ou inviável em termos de tempo de processamento caso se utilize memória virtual.

Foram realizados testes com quantidades variadas de pesquisadores, para averiguar o aumento proporcional de memória. Na figura 5, pode-se visualizar o gráfico do consumo de memória do trecho de código que realiza o agrupamento dos dados dos pesquisadores pela quantidade de pesquisadores buscados.

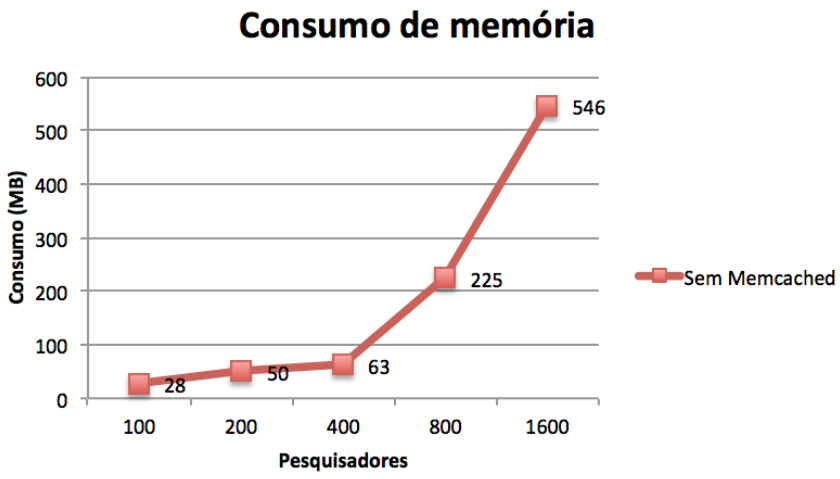

Figura 5 – Gráfico de consumo de memória do agrupamento dos dados realizado pelo scriptLattes

3.2 Escolha das ferramentas adequadas

Para solucionar o gargalo no tempo de processamento do scriptLattes gerado pelo algoritmo AuthorRank, foi realizado um estudo no algoritmo afim de conseguir paralelizá-lo de forma que o tempo de processamento diminuísse. Após esse estudo, foi identificado que a paralelização poderia ocorrer, pois na criação do vectorRank há uma sequência de cálculos que não possuem dependência de dados. A forma final desta paralelização será discutida com mais detalhes no item seguinte.

Sabendo como paralelizar o algoritmo, procedeu-se ao levantamento de ferramentas que poderiam atender este fim, e dentre as ferramentas investigadas decidiu-se empregar o software Hadoop por ser uma das soluções mais utilizadas para este tipo de processamento em cenários de grandes dados, devido às características de tolerância à falhas e auto-gerenciamento abordadas no item 2.2.

Para solucionar o gargalo de consumo de memória, foi utilizado o framework Memcached devido ao seu alto grau de aceitação em empresas do mercado mundial. Note-se que neste trabalho o memcached foi usado de forma diferente da usual, i.e., a utilização do memcached não teve como objetivo armazenar resultados de consultas de forma a melhorar o tempo de busca de informações , mas sim aumentar a capacidade de memória RAM em um processamento que já armazena todos os seus dados em memória. Com isso, toda a memória que seria armazenada em apenas uma máquina, passa a ser armazenada em um cluster de servidores, com o Memcached realizando o balanceamento do armazenamento nos nós do cluster.

3.3 Adequação da ferramenta às soluções encontradas

Para a adequação da ferramenta na utilização do Hadoop, foi necessário modificar o algoritmo AuthorRank, de modo a tornar seu processamento paralelizável em um cluster de máquinas. Para implementar a paralelização, foi necessário descobrir os pontos no algoritmo em que não há quaisquer dependências nos cálculos realizados .

O Código 6 ilustra o trecho de código em que esta independência ocorre. Nele, podemos ver que o cálculo do vectorRankNovo é feito a partir da soma do grau de colaboração de cada pesquisador com uma coluna da matriz de colaboração e, posteriormente, pela multiplicação da constante de amortecimento.

```
print "[CALCULANDO AUTHOR-RANK (PROCESSO ITERATIVO)]"
for index in range(0,iteracoes):
    self.vectorRank = self.calcularRanks(self.vectorRank)

def calcularRanks(self, vectorRank):
    vectorRankNovo = numpy.zeros( len(vectorRank), dtype=numpy.float32)
    d = 0.85
    for i in range(0, len(vectorRank)):
        soma = 0
        for j in range(0, len(vectorRank)):
                soma += vectorRank[j] * self.matriz[j , i]
                vectorRankNovo[i] = (1-d) + d*soma
    return vectorRankNovo
```

Código 6 – Trecho paralelizável do AuthorRank

A figura 6 mostra graficamente o cálculo do vectorRankNovo. Como podemos observar, primeiro é realizado o cálculo da soma S1 (que é o produto do vectorRank com a coluna correspondente ao autor na matriz de colaboração), depois o valor do S1 é utilizado na fórmula ((1 - 0,85) + (S1*0,85)) para dar origem ao N1, que é o valor que representa o grau de colaboração do pesquisador 1. Desse modo, podemos perceber que o cálculo de cada valor do novo vetor pode ser realizado paralelamente, desde que ao final das operações os resultados sejam agrupados em um novo vectorRank, que é exatamente o conceito do MapReduce trabalhado pelo Hadoop no processamento de dados distribuídos.

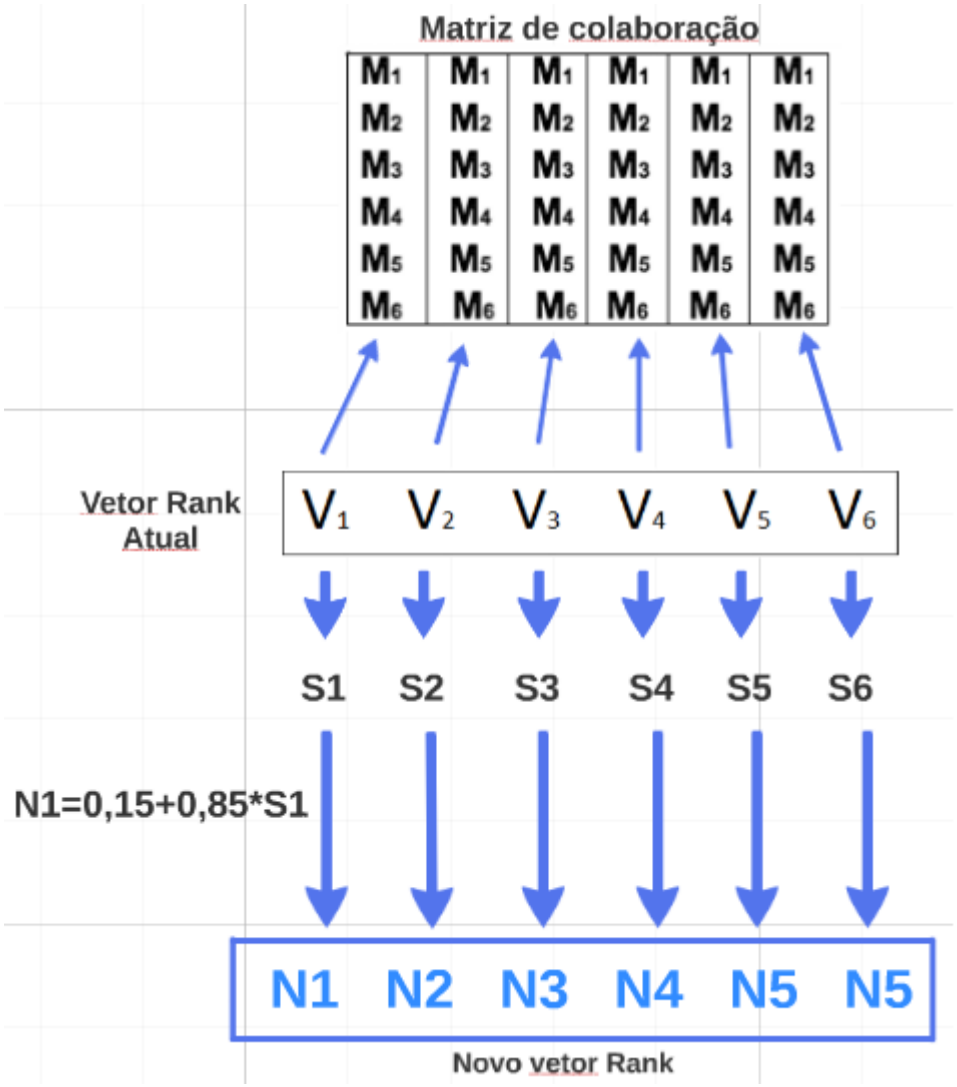

Figura 6 – Funcionamento do AuthorRank

Para a realização da alteração do modo de processamento do algoritmo, foi necessário primeiramente criar uma função para realizar o procedimento de criar os arquivos que irão para o HDFS do Hadoop para serem consumidos. Este procedimento é responsável por criar arquivos contendo 3 informações: o vetor rank atual; a coluna da matriz; e o índice em que vai ser adicionado o valor resultante do processamento. A figura de código 7, mostra a implementação dessa função em python na ferramenta scriptLattes.

```
def preparar_arquivos(self, vectorRank):
    shutil.rmtree('/home/hduser/sf_scriptLattesV8.08/hadoop/tmp')
    os.mkdir('/home/hduser/sf_scriptLattesV8.08/hadoop/tmp')
    for f in range(0, len(vectorRank)):
        file = open('/home/hduser/sf_scriptLattesV8.08/hadoop/tmp/file%d.txt' % (f+1), 'w')
        file.write("%d\n" % f)
        vector = map(str, vectorRank.tolist())
        file.write(",".join(vector) + "\n")
        col = map(str, self.matriz.getcol(f).toarray().tolist())
        file.write(",".join(col).replace('[', '').replace(']', '') + "\n")
        file.close()
```

Código 7 – Preparação dos arquivos para o HDFS

A figura mostra a criação dos arquivos no formato TXT para serem consumidos pelo HDFS. Note que a quantidade de arquivos que irão ser criados é igual a quantidade de pesquisadores buscados.

Para que o Hadoop utilize esses arquivos para calcular o AuthorRank, é necessário criar a função Map, responsável por calcular o valor do S1 e posteriormente o valor do N1, retornando o índice que representa o pesquisador no novo vetor rank e o N1 para adicionar o valor que representa seu AuthorRank. A figura de código 8, mostra a implementação da função Map, utilizando a linguagem de programação Python. A variável index representa o índice em que o valor vai ser adicionado no novo vetor rank. Depois disso, é calculada a multiplicação do vetor rank atual com a coluna da matriz, junto com o valor que representa o grau de colaboração e é retornado o índice do vetor junto com esse valor.

```
#!/usr/bin/env python
import sys

d = 0.85
soma = 0

index = sys.stdin.next().strip()
vectorRank = sys.stdin.next().split(',')
col = sys.stdin.next().split(',')
for i in range(len(vectorRank)):
    soma += float(vectorRank[i]) * float(col[i])

print "%s,%s" % (index, str((1-d) + d*soma))
```

Código 8 – Implementação da função Map em Python

Com a função Map pronta, foi implementada a função Reduce, que é responsável simplesmente por agrupar todos os resultado do Map em um único vetor. Na figura de código 9, podemos visualizar o código implementado. A variável line representa cada resultado realizado pelo Map, onde os valores são adicionados em um vetor e no final de tudo é retornado o novo vetor rank.

```
#!/usr/bin/env python
from operator import itemgetter
import sys

vectorRankNovo = []
for line in sys.stdin:
    line = line.strip()
    line = line.split(',')

    vectorRankNovo.insert(int(line[0]), line[1])

print ",".join(vectorRankNovo)
```

Código 9 – Implementação da função Reduce em Python

Para adequar a ferramenta na utilização do Memcached, foi necessária uma mudança na estrutura do armazenamento dos dados dos currículos. Criando primeiramente uma classe chamada “Memcached”, como mostra a figura de código 6, que se conecta ao cluster utilizando a biblioteca memcached para Python e ficará sendo usada para a manipulação dos dados de entrada e saída da memória distribuida.

```
import memcache

class MemCached(object):
    _instance = None
    cluster = None
    def __new__(self, *args, **kwargs):
        if not self._instance:
            self._instance = super(MemCached, self).__new__(self, *args, **kwargs)
        return self._instance

    def __init__(self):
        self.cluster = memcache.Client(['192.168.0.111:11211'], debug=1)

    def getMembro(self, key):
        return self.cluster.get('membro%d' % key)

    def addMembro(self, key, membro):
        self.cluster.set('membro%d' % key, membro)
```

Código 10 – Classe criada para manipulação dos dados no Memcached

Posteriormente, foi feito um levantamento em todo o código do programa, afim de analisar todas as partes que teriam que ser adaptadas à nova estrutura e assim vislumbrar o impacto que a mudança acarretaria.

Ao final do processo, foi observado a necessidade de haver primeiramente uma mudança na estrutura da inserção dos objetos ‘Membro’ na coleção ‘Grupo’,

deixando de ser armazenado em uma variável do código e passando a ser armazenado na memória distribuída. Para que isso ocorresse, foi realizada uma mudança no construtor da classe Grupo, responsável por montar a lista dos membros buscados. As figuras de código 11 e 12, mostra o trecho antes e depois da alteração, respectivamente.

```
self.listaDeMembros.append(Membro(idSequencial, identificador, nome, periodo, rotulo, \
                        self.itemsDesdeOAno, self.itemsAteOAno, self.diretorioCache))
```

Código 11 – Trecho de código antes da alteração do Memcached

```
novo_membro = Membro(idSequencial, identificador, nome, periodo, rotulo,\
                 self.itemsDesdeOAno, self.itemsAteOAno, self.diretorioCache)

self.listaDeMembros.append(idSequencial)

self.cache.addMembro(idSequencial, novo_membro)
```

Código 12 – Trecho de código depois da alteração do Memcached

Consequentemente todos os trechos de código que consumiam a lista de membros tiveram de ser alteradas, devido ao fato de que agora somente é guardado um identificador do membro na lista de membros, e o mesmo identificador serve para obter o objeto Membro a partir do Memcached. A segunda mudança na estrutura foi nos trechos que iteravam sobre a lista para realizar seus procedimentos. A figura de código 13, ilustra esta alteração que foi realizada em todo o programa, modificando os trechos de código que iteram sobre a lista de pesquisadores.

```
def carregarDadosCVLattes(self):
    indice = 1
    for key in self.listaDeMembros:
        membro = self.cache.getMembro(key)
        print "\n[LENDO REGISTRO LATTES: " + str(indice) + "o. DA LISTA]"
        indice += 1
        membro.carregarDadosCVLattes()
        membro.filtrarItemsPorPeriodo()
        self.cache.addMembro(membro.idMembro, membro)
        print membro
```

Código 13 – Alteração da estrutura de iteração dos membros da pesquisa

Diante deste cenário, o programa passou a armazenar uma lista de

identificadores que servem para buscar o membro que foi armazenado no Memcached, e todas as iterações armazenam em memória apenas um membro por vez.

4 RESULTADOS

4.1 Tratamento do gargalo de processamento

A paralelização do algoritmo AuthorRank por meio do software Hadoop, tal como discutido no capítulo 3, faz com que todo o processamento que não possui dependência de dados possa ser realizado em múltiplas máquinas. Este benefício, contudo, traz um custo de comunicação e controle inerente devido aos processos realizados pelo Hadoop para garantir a distribuição das tarefas e a tolerância à falhas. Para que os dados fiquem disponíveis para os escravos, há um custo devido ao gerenciamento e transporte desses dados para o HDFS, como também podemos observar um custo devido às garantias de tolerância a falhas do Hadoop, que realizam a reexecução das tarefas sempre que ocorrer uma falha.

Os testes de desempenho apresentados nesta seção foram realizados em 6 máquinas virtuais distribuídas em 4 máquinas físicas, de tal forma que 2 máquinas físicas possuíam 1 máquina virtual cada e as outras 2 máquinas físicas possuíam 2 máquinas virtuais cada, com sistema operacional Ubuntu 12.04, um processador com 2 cores, 1.7GB de memória RAM, 8GB de HD, 48MB de memória de vídeo. A versão do Hadoop utilizada foi a 2.5. O ambiente de rede configurado entre as máquinas foi através de um roteador TP-LINK modelo TL-WR720N, cujas máquinas realizavam a comunicação através do modo bridge. As máquinas acessam a rede via cabo e via rede sem fio (wi-fi). A figura 7 ilustra este cenário.

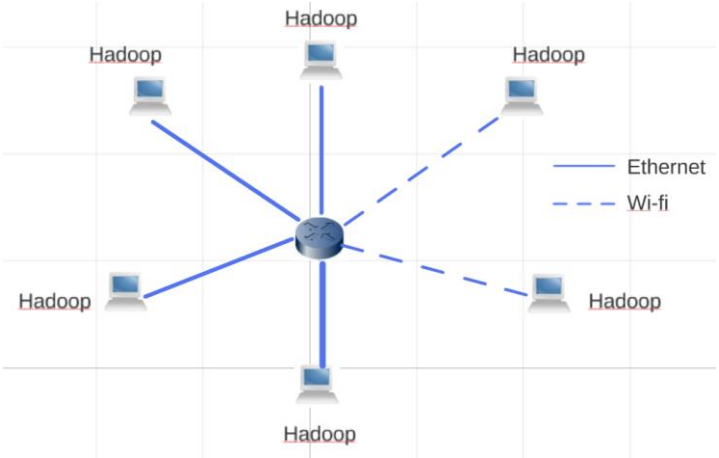

Figura 7 - Cenário utilizado na execução do Hadoop

A execução do teste utilizou uma amostra de 3 mil pesquisadores e os testes foram realizados em dois casos. No primeiro, considerado o caso-base, foi executado um

teste sem a utilização do MapReduce, isto é, o AuthorRank foi executado em apenas uma máquina virtual e o tempo médio das iterações do algoritmo AuthorRank foi contabilizado a partir de 4 iterações do algoritmo. No segundo caso, foram executados testes com a utilização do MapReduce e variando o número de escravos pertencentes ao cluster (de 2 até 6 escravos), afim de indicar o ganho obtido pelo aumento do número de máquinas, conforme mostra a figura 8.

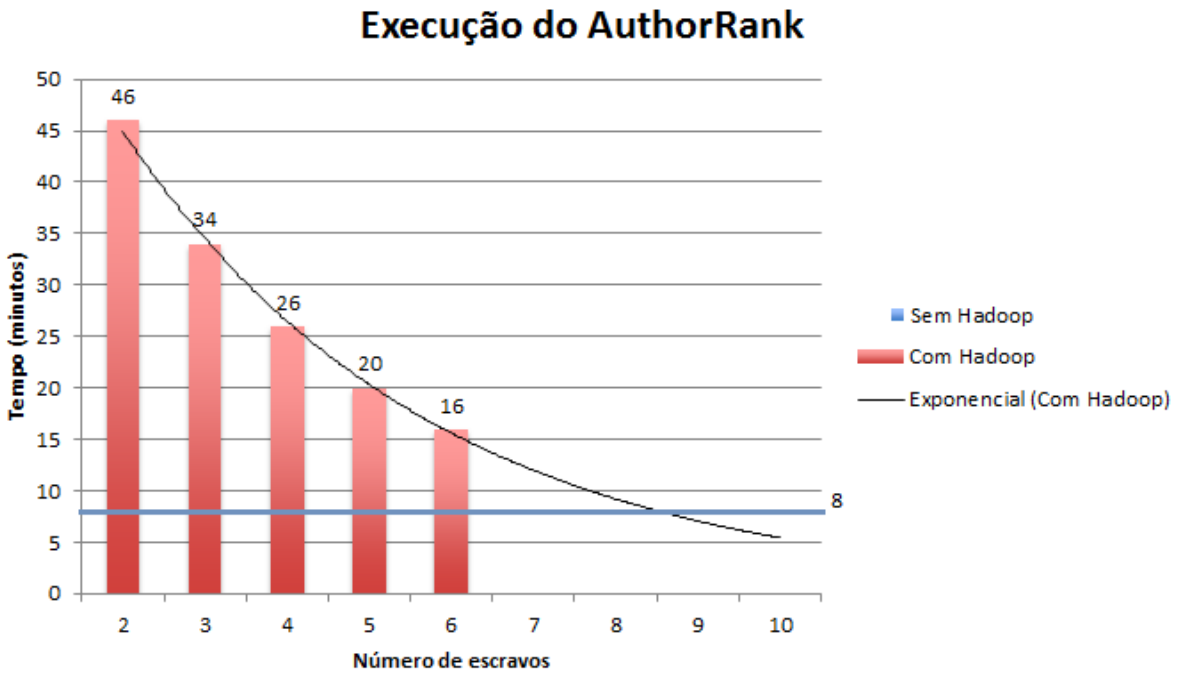

Figura 8 – Gráfico de tempo de processamento do *AuthorRank*

Analisando o gráfico, podemos observar que o *setup* que utilizou o Hadoop e 6 máquinas compondo o cluster, obteve um resultado duas vezes pior que o caso-base. Isso se deve aos custos de comunicação e tolerância a falhas do Hadoop, que foi projetado para a execução em grandes clusters (contendo centenas e algumas vezes até milhares de máquinas). Um outro fator que contribuiu foi a heterogeneidade da rede – com escravos utilizando a rede cabeada e outros a rede sem fio – quanto à inadequação das máquinas que fazem as vezes de servidores de máquinas virtuais neste caso.

Porém, também podemos observar uma diminuição do tempo de execução do algoritmo com o aumento do número de escravos no cluster. Essa diminuição do tempo na adição de um escravo no cluster pode ser observada através de uma regressão exponencial traçada no gráfico. Embora, não tenha sido possível a realização dos testes com um número maior de escravos, fazendo-se a regressão dos dados obtidos com base em um modelo exponencial decrescente, podemos

estimar que um cluster de apenas 9 máquinas fará o processamento de 3 mil pesquisadores em menos tempo utilizando o Hadoop.

4.2 Tratamento do consumo de memória

Na abordagem que envolveu o software Memcached para a diminuição do consumo de memória do scriptLattes, o fator preponderante foi a mudança na estrutura de armazenamento da ferramenta, que passou a armazenar uma lista dos pesquisadores em memória distribuída. A partir daí foram realizados dois tipos de testes, um para medir o resultado do consumo de memória gasta na execução da ferramenta, e outro para avaliar o impacto que a mudança ocasionaria em relação ao tempo gasto na execução.

Para a realização dos testes foram utilizadas 3 máquinas virtuais, distribuídas em 3 máquinas físicas, com sistema operacional Ubuntu 12.04, processador com 2 cores, 1.7GB de memória RAM, 8GB de HD, 48MB de memória de vídeo. A versão do Memcached utilizada foi a 1.4.22. O ambiente de rede configurado entre as máquinas foi através de um roteador TP-LINK modelo TL-WR720N, cujas máquinas realizavam a comunicação através do modo bridge. As máquinas acessam a rede via cabo e via rede sem fio (wi-fi). A figura 9 ilustra este cenário.

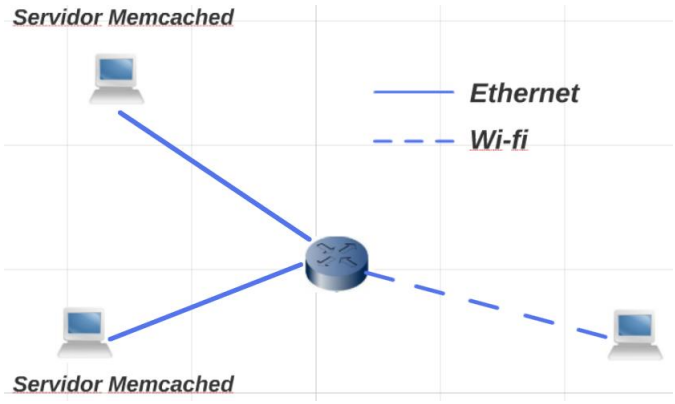

Figura 9 - Cenário utilizado na execução do Memcached

No processamento, uma máquina ficou responsável pela execução do scriptLattes, enquanto as outras duas foram utilizadas apenas para servir o armazenamento no memcached.

O objetivo principal do Memcached neste trabalho é escalar horizontalmente o armazenamento de dados em memória RAM, que antes era armazenado em apenas uma máquina, passando a ser armazenado em um cluster de máquinas.

Para os testes, foram realizadas execuções variando o número de pesquisadores buscados (100, 200, 400, 800, e 1600), para ter uma visão do comportamento do armazenamento de dados realizado pelo Memcached. Para aferir o consumo de memória utilizada pelo trecho de código que realizava o agrupamento dos dados dos pesquisadores, foi utilizada a ferramenta Memory Profiler abordada no item 3.5.

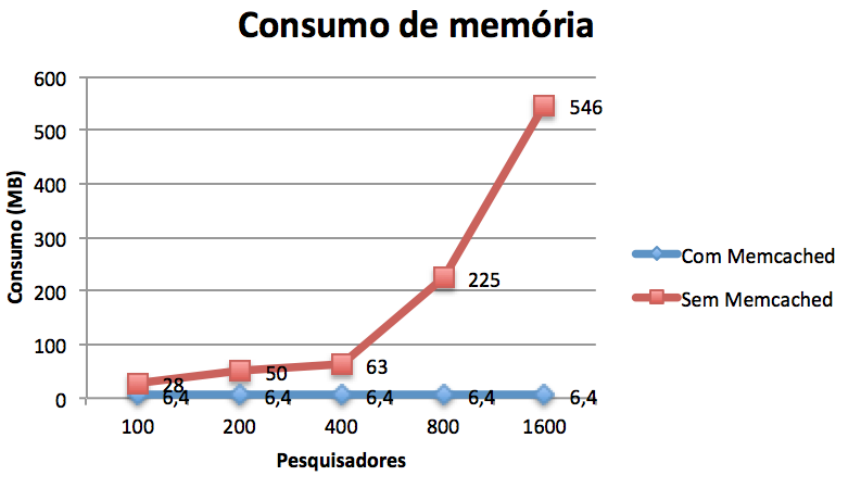

Figura 10 – Gráfico de consumo de memória do agrupamento dos currículos

Analisando o gráfico, podemos observar que o consumo de memória no cenário utilizando o Memcached foi praticamente o mesmo, sofrendo um aumento insignificante(os valores de 6,4MB foram arredondados), já que o programa armazena apenas uma lista com os identificadores dos pesquisadores, estando os dados dos pesquisadores distribuídos nas outras duas máquinas. Enquanto isso, o aumento de consumo de memória quando processado sem Memcached foi muito maior a partir de 400 pesquisadores, chegando a uma redução de 98,82% no consumo de memória RAM.

É importante frisar que há um custo de comunicação para o armazenamento de dados dos currículos dos pesquisadores nas máquinas que compõe o cluster, que são trafegados pela rede. Como no código do scriptLattes existem vários trechos de código que iteram sobre essa lista, pode-se esperar um aumento no tempo do processamento do programa quando comparado ao tempo de execução com armazenamento em memória local.

A figura 11 mostra o aumento do tempo de execução do scriptLattes, conforme o aumento do número de pesquisadores utilizados na entrada de dados do programa.

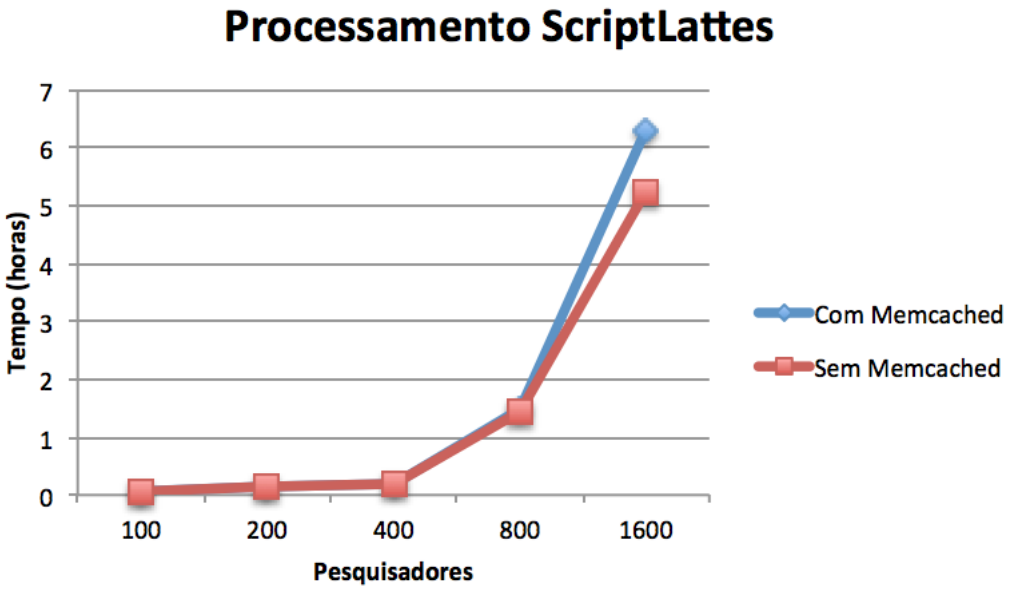

Figura 11 – Gráfico de tempo de processamento do scriptLattes com e sem Memcached

Ao analisar o gráfico, podemos observar que o crescimento do tempo com Memcached segue o mesmo padrão de quando não utilizamos o Memcached, devido ao fato de as duas execuções terem a mesma lógica, porém há uma diferença no tempo do processamento devido ao tempo necessário para o tráfego dos dados na rede na realização de operações de inserção e recuperação de dados no memcached. Por isso, a execução com o Memcached obteve um tempo um pouco maior em relação a execução sem Memcached, como mostra o gráfico.

5 CONCLUSÃO

5.1 Contribuições

Um dos principais problemas das aplicações que trabalham analisando dados específicos em lote é como realizar processamentos em um cenário de grandes massas de dados que exigem muitos recursos para se obter as informações desejadas em tempo hábil.

Este trabalho investigou o problema de migração de aplicações de análise de dados em lote em um contexto de grande processamento de dados, o qual mostrou a partir de uma metodologia aplicada em um estudo de caso, os ganhos obtidos neste cenário. Embora tenha sido executado em apenas um estudo de caso, o trabalho contribuiu para construir conhecimentos na análise e resolução de problemas neste cenário, apresentando técnicas e ferramentas para investigar as causas reais dos gargalos e as soluções e conceitos para a paralelização de algoritmos e distribuição de memória.

Como estudo de caso, este trabalho investigou pormenorizadamente a ferramenta scriptLattes e discutiu como lidar com suas limitações frente ao cenário de grandes dados. Uma contribuição direta deste estudo foi a paralelização do algoritmo AuthorRank, cuja implementação utilizando a plataforma Hadoop foi apresentada. Outra contribuição, foi a discussão e implementação de mecanismos para distribuição da memória do programa em cluster.

Este trabalho permitiu ainda consolidar um método para a migração de aplicações de análise de dados em lote para o cenário de grandes dados. Embora tenha sido testada apenas por meio deste estudo de caso, o método desenvolvido pode ser utilizado pela comunidade acadêmica e servir de base para novas investigações neste tema.

5.2 Dificuldades encontradas

Dentre as dificuldades encontradas no desenvolvimento deste trabalho, destaca-se a ausência de uma infraestrutura adequada para processamento em um cenário de grandes dados. Para os testes serem realizados, foi necessário a configuração de um cluster de máquinas, que não puderam ser adquiridas em uma quantidade ideal e tiveram que ser limitadas em 4 máquinas físicas, ocasionando uma diminuição na massa de dados trabalhada.

Em relação ao ambiente de rede utilizado, houve problemas devido ao fato de

o trabalho ser realizado em um ambiente inadequado, onde aconteciam interrupções de conexão e ocupação de parte da capacidade da rede por outros usuários, que ocasionava lentidão na comunicação entre as máquinas do cluster.

Outro fator foi o tempo de execução dos testes. Como o cenário trabalhado era de grande processamento de dados, o tempo de realização de cada teste tornou inviável a análise de dados de largas bases de pesquisadores. Principalmente, quando considerado o ambiente de testes utilizado. Isso dificultou a análise dos resultados e a tomada de decisão enquanto a metodologia era realizada.

5.3 Lições aprendidas

Este trabalho teve uma forte contribuição em relação a mudança no conceito de processamento para a resolução de problemas de grandes dados. Tornando mais visível a resolução de problemas como a paralelização de algoritmos e a distribuição do armazenamento de dados com o mínimo de impacto possível na aplicação.

Outro ponto importante no aprendizado, foi em relação a metodologia utilizada em todo o processo, desde a identificação dos pontos de gargalo da aplicação, às técnicas de profiling para identificar os pontos de melhoria, estudo da paralelização e distribuição dos dados e a aplicação de ferramentas como Hadoop e Memcached.

5.4 Trabalhos futuros

Para a evolução deste trabalho, bem como o aprimoramento de seus resultados, faz-se necessário um aumento na quantidade de máquinas disponíveis para a realização do processamento paralelo, visando um alcance maior na quantidade de dados processados na realização dos testes.

Considerando que a realização dos testes se basearam em um estudo de caso específico, pode-se usar os conceitos e métodos contidos neste trabalho para a realização da portabilidade de outros tipos de aplicações, realizando uma análise comparativa entre as aplicações e testando a metodologia aplicada neste trabalho.

Outro fator a ser explorado em um futuro trabalho diz respeito à implementação, avaliação e comparação das melhorias sugeridas neste trabalho por meio de outras ferramentas de processamento distribuído e de armazenamento de dados. Ferramentas como o *Message Passing Interface*[4] e Redis[5] podem facilmente

[4] http://www.open-mpi.org/

ser aplicadas neste trabalho para permitir o processamento distribuído no algoritmo AuthorRank e o armazenamento distribuído de dados em memória, respectivamente.

[5] http://redis.io/

REFERÊNCIAS

Amorin, Cristiane V. 2003. “Curriculum Vitae Organization – the Lattes Software Platform.”

Balancieri, Renato. 2005. “A Análise de Redes de Colaboração Científica Sob as Novas Tecnologias de Informação E Comunicação: Um Estudo Na Plataforma Lattes.”

Bernardes, Guilherme. 2014. “Desenvolvimento de Software No Contexto Big Data.”

Economist, The. 2010. “Data, Data Everywhere.” http://www.economist.com/node/15557443.

Gantz, John, and David Reinsel. 2011. “Extracting Value from Chaos.”

Issa, Joseph, and Silvia Figueira. 2012. “Hadoop and Memcached_ Performance and Power Characterization and Analysis - Springer.pdf.”

“Line Profiler.” https://github.com/rkern/line_profiler.

Maia, Maria de Fatima, and Sônia Elisa Caregnato. 2008. “Co-Autoria Como Indicador de Redes de Colaboração Científica.”

“Memory Profile.” https://github.com/fabianp/memory_profiler.

Mena-Chalco, Jesús, and Roberto Cesar Junior. 2011. “PROSPECÇÃO DE DADOS ACADÊMICOS DE CURRÍCULOS LATTES ATRAVÉS DE SCRIPTLATTES.”

Mena-Chalco, Jesús P., Luciano A. Digiampietri, and Leonardo B. Oliveira. 2012. “Perfil de Produção Acadêmica Dos Programas Brasileiros de Pós-Graduação Em Ciência Da Computação Nos Triênios 2004-2006 E 2007-2009.”

Rajaraman, A., and J.D. Ullman. 2012. *Mining of Massive Data- Sets*. Cambridge University Press.

“Script Lattes.” http://scriptlattes.sourceforge.net/input.html.

Silva, Ticiana L. C., and José Antônio F. Macêdo. “Análise Em Big Data E Um Estudo de Caso Utilizando Ambientes de Computação Em Nuvem.”

White, T. 2012. “Hadoop: The Definitive Guide. Third Edition. Beijing: O’Reilly.”