CENTRO PAULA SOUZA GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Segurança da Informação

EDUARDO FERREIRA MENDES

COMPUTAÇÃO DISTRIBUÍDA COM APACHE HADOOP

Americana, SP
2015

CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO
Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Segurança da Informação

EDUARDO FERREIRA MENDES

mendesxeduardo@gmail.com

COMPUTAÇÃO DISTRIBUÍDA COM APACHE HADOOP

Trabalho de Conclusão de Curso desenvolvido em cumprimento Á exigência curricular do Curso de Tecnologia em Segurança da Informação, sob orientação do prof. Me. Rossano Pablo Pinto.

Área de concentração: Sistemas distribuídos

Americana, SP
2015

Mendes, Eduardo Ferreira

M49c Computação distribuída com Apache Hadoop. / Eduardo Ferreira Mendes. – Americana: 2015.

74f.

Monografia (Graduação em Tecnologia em Segurança da Informação). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.

Orientador: Prof. Me. Rossano Pablo Pinto

1. Sistemas distribuídos I. Pinto, Rossano Pablo II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.3.047

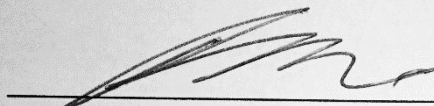
Eduardo Ferreira Mendes

Computação distribuída com Apache Hadoop

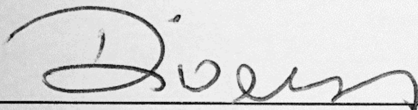
Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da informação pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Sistemas distribuídos

Americana, 10 de dezembro de 2015.

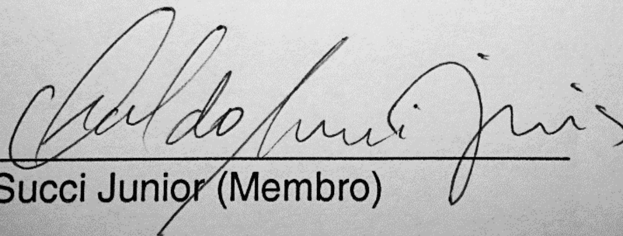
Banca Examinadora:



Rossano Pablo Pinto (Presidente)
Mestre
Fatec Americana



Diogenes de Oliveira (Membro)
Mestre
Fatec Americana



Osvaldo Succi Junior (Membro)
Mestre
Fatec Americana

AGRADECIMENTOS

Ao Garoa Hacker Clube, em especial ao Alexandre Souza e Luciano Ramalho por terem aceitado a empreitada de me ensinar Python em 2013;
A comunidade Python Brasil por estarem sempre dispostos a tirar qualquer dúvida a qualquer horário;
Aos professores José Luiz Zem e Osvaldo Succi por serem influenciadores diretos desse desafio;
Aos meus amigos Alexandre Makiyama, Ana Lúcia Peluzzo, Bruna Balthazar, Leonardo Hamam, Murilo Fujita, William Lopes e Wilson Santos por todo o auxílio oferecido durante o tempo de desenvolvimento desse projeto, principalmente nas últimas semanas;
Á minha família por ter oferecido auxilio e condições durante toda a minha permanência na graduação;
Ao Italo Muryllo Tosta por ter me indicado o caminho do processamento de linguagem quando tudo parecia impossível;
Ao Tony Berber Sardinha por ter compartilhado uma gama enorme de material;
Ao Grupo de Estudo Criptográficos de Americana pois muitas ideias implementadas durante nossos encontros foram abordadas nesse trabalho;
A Tainá Bueno por ser a pessoa mais paciente e motivadora existente no planeta terra;
Ao professor Rossano Pablo Pinto pela orientação e bom humor sempre, mesmo quando estávamos fora do prazo;
A Carlos Drummond Andrade e Fabio Altro por serem motivadores totalmente indiretos deste trabalho, mas direcionadores de tudo que é possível de se fazer usando apenas linguagem;
E por último e não menos importante, a todas as pessoas que desenvolvem ou se envolvem em projetos de software livre. Sem essa grande comunidade esse trabalho não existiria e talvez não existisse sentido em minha vida.

“*Peço desculpas pela turbulência e inconstância do site nos últimos dias. Ao que me parece, bizarros acúmulos de gelo projetam seu peso nos galhos da internet e caminhões transportando pacotes de informação derrapam por toda parte*”

(Andrew Tobias, 2007)

RESUMO

O presente trabalho detalha a construção de um cluster de computadores e sua aplicação para o processamento de análises de linguística de corpus. O cluster foi construído utilizando hardware comum e fazendo uso do framework Apache Hadoop em conjunto com sua biblioteca de Streaming com Python 3. As aplicações foram desenvolvidas no modelo de programação MapReduce, fazendo uso de uma biblioteca, nomeada MapReduceLib, elaborada no decorrer deste trabalho. Os resultados obtidos demonstram que o uso do cluster representou um melhor desempenho, indicando a implementação de sistemas distribuídos como uma alternativa de baixo custo para processamento de grande volume de dados em larga escala.

Palavras chave: Sistemas distribuídos, Hadoop, Linguística de corpus, cluster de computadores

ABSTRACT

The present study details the making of a computer cluster and its aplication in the context of the processing of corpus linguistics analisys.The cluster was built using commodity hardware, using the Apache Hadoop framework along with its Streaming Library for Python 3. The applications were developed following the MapReduce development model, and a specific library, named MapReduceLib, that was built for this research. The results demonstrate that the use of the cluster improved the processing performance, suggesting that the implementation of distibuted systems is a fine low cost option for large scale data processing.

Keywords: Distibuted systems, Hadoop, Corpus Linguistics, computer clusters.

Lista de Figuras

Figura 1 - Dividir e conquistar....................5
Figura 2 - Decomposição de dados....................6
Figura 3 - Pipeline....................7
Figura 4 - Map em paradigma funcional....................13
Figura 5 - Reduce em paradigma funcional....................13
Figura 6 - Algoritmo MapReduce....................15
Figura 7 - Visão geral do Google File System....................17
Figura 8 - YARN....................21
Figura 9 - Map para Hadoop Streaming....................25
Figura 10 - Pipelines e Hadoop Streaming....................26
Figura 11 - Exemplo de uso do Hadoop Streaming....................27
Figura 12 - Exemplo de uso da MapReduceLib....................32
Figura 13 - Contagem de palavras....................36
Figura 14 - Concordância....................37
Figura 15 - Estatística lexicográfica em dígrafos....................37
Figura 16 - exemplo de uso de sistes de busca....................38
Figura 17 - Cluster....................40
Figura 18 - esquema de uso das ferramentas....................42

Lista de tabelas

Tabela 1- Comandos básicos do HDFS 23
Tabela 2 - Parâmetros principais do Hadoop Streaming 26
Tabela 3 – Parâmetros de definições do Streaming..28
Tabela 4 - Comandos principais da MapReduceLib 33
Tabela 5 - Configuração dos computadores...39
Tabela 6 – Programas utilizados 41
Tabela 7 - Arquivos de análise e tempo de processamento..44

LISTA DE ABREVIATURAS

ASF – Apache Software Foundation

B – Bytes

DN – DataNode

EXT4 – Extended Filesystem 4

FAT32 – File Allocation Table 32

FS – File System

GB – GigaBytes

GFS – Google File System

GNU – Gnu's Not Unix

HD – Hard Drive

HDFS – Hadoop Distributed File System

HOYA – Hbase On YARN

HTTP – HyperText Transfer Protocol

IEEE – Institute of Eletrical and Eletronics Engineers

ISO – International Organization for Standardlization

JVM – Java Virtual Machine

MB – MegaBytes

MR – MapReduce

NFS – Network File System

NLTK – Natural Language ToolKit

NN – NameNode

NOSQL – Not Only SQL

POSIX – Portable Operating System Interface

SSH – Secure Shell

STDIN – Standard Input

STDOUT –Standard Output

TB - TeraByte

TCP – Transmission Control Protocol

YARN – Yet Another Resource Negotiator

SUMÁRIO

INTRODUÇÃO **1**

ESTRUTURA DO TRABALHO **2**

1. REVISÃO DA LITERATURA **4**

1.1 PROCESSAMENTO PARALELO 4

1.1.1 *DIVIDE AND CONQUER* (DIVIDIR E CONQUISTAR) 5

1.1.2 *DATA DECOMPOSITION* (DECOMPOSIÇÃO DE DADOS) 6

1.1.3 PIPELINE 7

1.2 SISTEMAS DISTRIBUÍDOS 8

1.2.1 SISTEMAS DE ARQUIVOS DISTRIBUÍDOS 9

1.3 LINGUÍSTICA DE CORPUS 11

2. MAPREDUCE E GOOGLE FILE SYSTEM **12**

2.1 MAPREDUCE 12

2.1.1 ALGORITMO DE MAPREDUCE 14

2.2 GOOGLE FILE SYSTEM 16

2.2.1 ARQUITETURA 16

2.2.2 INTERAÇÃO 17

2.2.3 NÓ MASTER 18

3. APACHE HADOOP **20**

3.1 ARQUITETURA 20

3.1.1 HADOOP COMMON 20

3.1.2 APACHE YARN 21

3.1.3 HADOOP DISTRIBUTED FILE SYSTEM 22

3.1.4 MAPREDUCE 23

3.2 SUBPROJETOS OU ECOSSISTEMA HADOOP 24

3.3 HADOOP STREAMING...24

3.4 MODOS DE OPERAÇÃO...29

3.5 CONSIDERAÇÕES FINAIS SOBRE O HADOOP...30

3.6 MAPREDUCELIB 31

4. ESTUDO DE CASO 34

4.1 APLICAÇÕES..35

4.1.1 CONTAGEM DE OCORRÊNCIAS (WORDLIST)...35

4.1.2 CONCORDÂNCIA..37

4.1.3 DENSIDADE ESTATÍSTICA LEXICOGRÁFICA 37

4.2 DESCRIÇÃO DA INFRAESTRUTURA UTILIZADA 39

4.3 TESTES E RESULTADOS..41

4.3.1 TESTES 42

4.3.2 COMPARATIVO..43

CONSIDERAÇÕES FINAIS 46

REFERÊNCIAS...48

APÊNDICE A 50

APÊNDICE B 57

INTRODUÇÃO

O presente trabalho detalha a construção de um cluster de computadores e sua aplicação para o processamento de análises de linguística de corpus. O cluster foi construído utilizando hardware comum e fazendo uso do framework Apache Hadoop em conjunto com sua biblioteca de Streaming com Python 3. As aplicações foram desenvolvidas no modelo de programação MapReduce, fazendo uso de uma biblioteca, nomeada MapReduceLib, elaborada no decorrer deste trabalho. Os resultados obtidos demonstram que o uso do cluster representou um melhor desempenho, indicando a implementação de sistemas distribuídos como uma alternativa de baixo custo para processamento de grande volume de dados em larga escala.

Motivação

O crescimento desenfreado da tecnologia trouxe consigo um grande número de computadores e serviços ativos na internet e isso implica diretamente na quantidade de dados que trafegam pelos meios de comunicação. Essa quantidade de dispositivos interconectados geram grandes massas de arquivos não estruturados nunca vistos antes a serem tratados e analisados. Nesse momento os meios convencionais de armazenamento deixam a desejar, principalmente quando se pensa no quesito ‘desempenho’.

Com isso, em meio a este cenário caótico, os laboratórios da Google desenvolveram um modelo de programação denominado MapReduce que faz uso de um sistema de arquivos distribuído chamado Google File System. Embora esta solução seja proprietária, a comunidade do *software* livre desenvolveu diversas alternativas para processamento de massas de arquivos não estruturados, como o Apache Hadoop, Disco, extensões para bancos de dados não relacionais, entre outros.

Considerando esse contexto, um cluster que faz uso de computadores comuns e que podem estar sendo operados em conjunto ao processamento distribuído se dá como uma boa alternativa, pois dispensa a aquisição de novos computadores e o modelo de programação MapReduce é capaz de paralelizar de maneira simples processos

complexos sem a preocupação de construção de ferramentas que se responsabilizem pelo gerenciamento do cluster.

Justificativa

Este trabalho nasceu em uma proposta de iniciação científica apresentada pela FATEC Americana e foi incrementada nos laboratórios do Garoa Hacker Clube. Caso a justificativa seja pensada pela contribuição que esse trabalho traz a iniciantes, a proposta já se auto justifica, porém pode-se considerar utilização de *software* livre e baixo custo computacional como outra alternativa plena. Assim sendo, esse trabalho tem a finalidade de resolver problemas de processamento distribuído, que podem aparecer em todas as áreas onde a computação se aplica, e também quebrar paradigmas sobre a utilização de *software* livre em determinados ambientes.

O **Objetivo** geral desse trabalho é entender o funcionamento do Apache Hadoop, sua arquitetura e funcionalidades. E os **objetivos específicos** são listados a seguir:

- Mostrar a viabilidade de adoção do Apache Hadoop no tratamento de massas de arquivos, para processamento de linguagem natural, fazendo uso de técnicas de Linguística de Corpus;
- Auxiliar, se possível, e mesmo que de forma indireta, todos os interessados pelo assunto ou leigos à procura de informações sobre o funcionamento do Apache Hadoop.

Estrutura do trabalho

Este trabalho se divide em quatro capítulos distintos e que explicam os temas relacionados a este projeto. A lista a seguir mostra uma breve descrição do que é abordado em cada um deles:

- Capítulo 1: Revisão da literatura:

Este Capítulo se destina a definição e revisão bibliográfica de processamento paralelo e sistemas distribuídos. Aborda algoritmos que servem como base para a compreensão de outros capítulos e características de sistemas distribuídos para a

possibilidade de processamento paralelo.

- Capítulo 2: MapReduce e Google File System

No Capítulo 2 são abordados tópicos sobre MapReduce e Google File System e suas implementações originais nos laboratórios da Google.

- Capítulo 3: Apache Hadoop

O Capítulo 3 é destinado à arquitetura do Apache Hadoop, sua API de Streaming capaz de rodar diferentes linguagens de programação e as principais diferenças entre os sistemas desenvolvidos pela Google, assim como também tem em vista explanar as diferentes versões do Apache Hadoop.

- Capítulo 4: Estudo de caso

O presente estudo de caso explorado neste Capítulo tem em vista a construção de um cluster Hadoop para a resolução de problemas de linguística de corpus e também discute sua viabilidade, isto é, em que momento precisamos de um cluster e até onde o processamento de texto e linguagem natural pode requerer de processamento no estilo MapReduce.

- Considerações finais

As considerações finais resumem as dificuldades e observações apontadas, de maneira não dogmática, seguidas de uma sugestão para estudos futuros.

- Apêndice A: Construção de um cluster Hadoop

Este apêndice tem como objetivo expor toda a configuração usada para a construção do cluster Hadoop.

- Apêndice B: Códigos usados no desenvolvimento deste trabalho

Este apêndice se destina única e exclusivamente a expor todo código usado para os resultados apontados no estudo de caso.

1. REVISÃO DA LITERATURA

Esse Capítulo discute a ideia errônea, trazida do senso comum, de que todos os programas de computador são executados e processados na nossa máquina local (STRACK, 1998). Felizmente, em muitos casos, a realidade não se apresenta desta maneira. Ao efetuar uma simples conexão com a internet pode-se pensar na complexidade contida nesse processo altamente distribuído, como roteadores espalhados e interconectados pelo mundo, servidores que processam paralelamente a variedade de conexões que recebem simultaneamente (COULOURIS; DOLLIMORE; KINDBERG, 2007) ou até mesmo todo o processamento, paralelo e distribuído, que ocorre quando se executa uma simples pesquisa por palavras-chave em sites de busca. As próximas seções abordam os conceitos relacionados a processamento paralelo, processamento concorrente e sistemas distribuídos.

1.1 Processamento paralelo

Nomeiam-se paralelos todo os modelos de programação compatíveis com ambientes preparados para executar instruções simultaneamente (PALACH, 2014). No passado os computadores pessoais eram dotados de um único processador, que executava uma única instrução por vez e existia uma noção de pseudo paralelismo, ou concorrência, onde o processador se ocupava de intercalar os processos rapidamente, dando a impressão de várias tarefas sendo executadas simultaneamente. Porém, atualmente, com a evolução dos computadores, existem múltiplos núcleos acoplados a um único processador, e com isso, a execução realmente simultânea de programas (RAMALHO, 2015).

Desta forma, notam-se grandes vantagens em utilizar técnicas de computação distribuída, dentre elas pode-se destacar a alta velocidade de processamento (AMORIM; BARBOSA; FERNANDES, 1988), em relação a um único computador, mesmo trabalhando em paralelo.Um exemplo claro pode ser visto na resolução de problemas matemáticos, como um determinante de uma matriz, em que um processo pode ser responsável pelas operações da diagonal principal e, consequentemente o outro, pela diagonal secundária e ainda um terceiro por executar a junção dos dados

processados pelos dois processos anteriores.

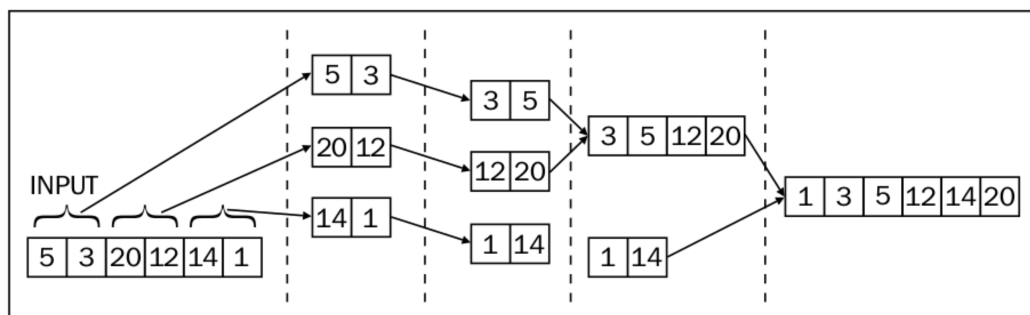
Algoritmos

Quando se pensa em desenvolver sistemas paralelos, vários aspectos devem ser levados em consideração antes mesmo de começar a desenvolver o código. Uma simples questão é a de como dividir um algoritmo sequencial e único em tarefas menores e paralelas. A existência de muitas técnicas pode prejudicar a escolha, mas existem algumas técnicas de paralelização, abordadas nas próximas subseções, que podem dar uma introdução e que podem ser consideradas a base para os algoritmos mais complexos (PALACH, 2014), como o MapReduce, abordado no Capítulo 2.

1.1.1 *Divide and conquer* (Dividir e conquistar)

A técnica de dividir e conquistar se baseia em dividir os dados de entrada, de maneira recursiva, até que existam partes indivisíveis da entrada, como nota-se na Figura 1, em um algoritmo de ordenação de vetor.

Figura 1 – Dividir e conquistar



Fonte: Adaptado de PALACH, (2014, p. 20).

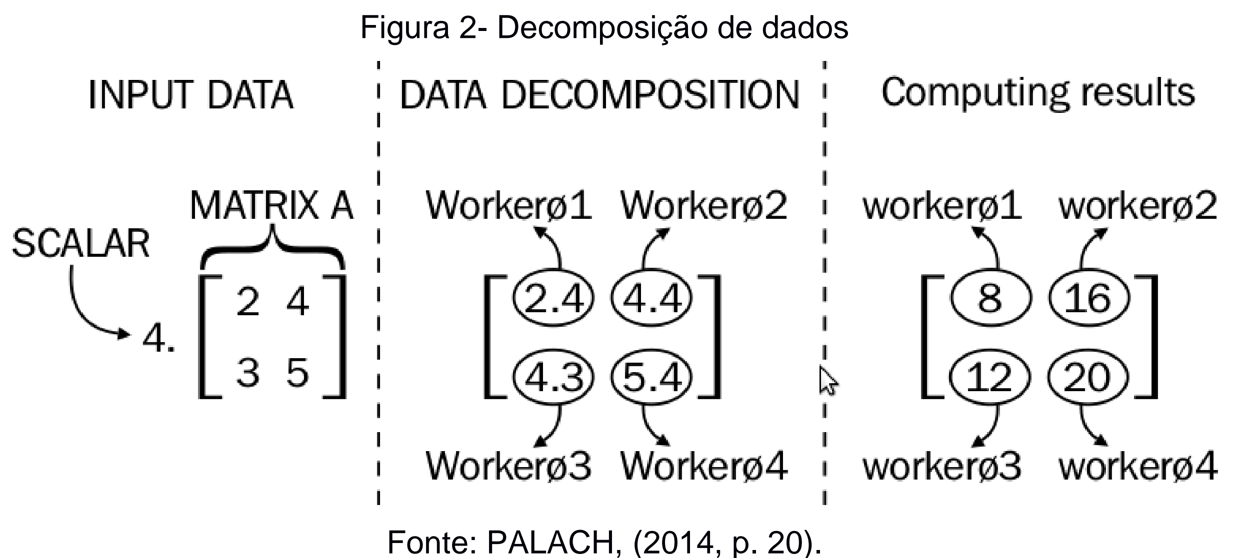
Neste exemplo pode-se notar, de maneira simples, a ordenação de um vetor em ordem crescente. Cada uma das duplas de números, na entrada (*input*), foi mapeada e ordenada separadamente. Como por exemplo, os números 3 e 5, isolados no primeiro quadro, ordenados em crescente no segundo quadro, e em seguida foram alocados em um novo vetor com os números 12 e 20 no terceiro quadro. Por fim, todos

os valores foram unidos em um novo vetor de saída.

1.1.2 *Data decomposition* (Decomposição de dados)

A proposta da "decomposição de dados" é suprir a necessidade de uma operação recursiva, como se nota na Figura 2. Imagine que se deve processar uma grande lista de dados, mas todos os dados receberão a mesma instrução, como uma multiplicação de um número por uma matriz.

Figura 2- Decomposição de dados



Fonte: PALACH, (2014, p. 20).

Quando se observa a parte *INPUT DATA* localizada no primeiro quadro da Figura 2, é possível notar o número escalar 4 que deve ser multiplicado por todos os elementos da matriz A. Sendo o objetivo da decomposição de dados paralelizar o processo, é possível notar que existem diferentes *workers*, ou processos, quatro neste exemplo, que se fazem responsáveis pela tarefa de multiplicar um número individual (2,4,3,5) e retornar a matriz dos valores processados. Em relação aos *workers* no segundo quadro (*DATA DECOMPOSITION),* pode-se imaginar nós de um cluster de computadores ou processos executando paralelamente em um único computador.

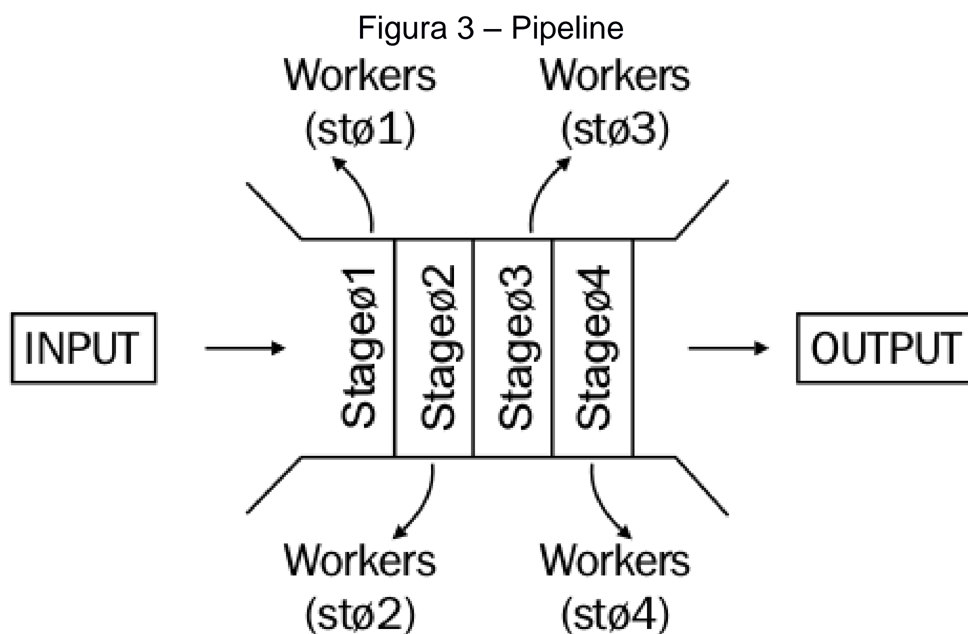
Também nesse exemplo, é possível concluir, que, mesmo de forma menos intensa, o algoritmo de *dividir e conquistar,* Figura 1, também faz uso da decomposição de

dados.

1.1.3 Pipeline

O *Pipeline,* em geral, é usado quando tenta-se resolver problemas de maneira colaborativa. Pode-se pensar um exemplo como tarifas bancárias. No fechamento do mês, o saldo anterior ao processamento fez a subtração de diversas tarifas e somas de rendimento da poupança, por exemplo. Mas a operação não pode ser feita em uma única conta e sim em uma base de dados como pode-se ver na Figura 3.

Figura 3 – Pipeline



Fonte: PALACH, (2014, p. 21).

Supõem-se que o primeiro *worker* é responsável pela captura de uma tupla no banco de dados. O segundo por executar as subtrações de impostos. O terceiro por somar o rendimento mensal da poupança e por consequência o último a devolver os novos valores processados ao banco de dados.

Abordadas técnicas e fundamentos de processamento paralelo, a Seção 1.2 se dedica exclusivamente a sistemas distribuídos e suas integrações para processamento paralelo.

1.2 Sistemas distribuídos

Sistemas distribuídos são caracterizados por um aglomerado de computadores que se comunicam a partir de mensagens em um mesmo canal de comunicação (AMORIM; BARBOSA; FERNANDES, 1988) e que se apresentam ao usuário como um sistema único e coerente (STEEN; TANENBAUM, 2008). Essa definição simples é capaz de abranger todas as maneiras pelas quais hardwares ou softwares individuais pertencentes a uma rede se comunicam, independentemente da distância (COULOURIS; DOLLIMORE; KINDBERG, 2007).

A motivação para a construção de um sistema distribuído é simplesmente a vontade de compartilhar recursos computacionais (COULOURIS; DOLLIMORE; KINDBERG, 2007). Como por exemplo:

- Cotas de disco, em um sistema de arquivos distribuído;
- Banco de dados para que seja possível compartilhar o acesso;
- Memória e processador, para um processamento mais eficaz.

Há características comuns entre todos sistemas distribuídos, Steen e Tanenbaum (2008) as definiram como metas que devem ser cumpridas para o desenvolvimento de um sistema distribuído funcional, tais como: acesso a recursos, transparência, abertura e escalabilidade, tratados a seguir:

- Acesso a recursos: esta é a meta mais importante para Steen e Tanenbaum (2008), pois trata-se de facilitar aos usuários, mesmo que sejam aplicações, acesso total a todos os recursos dos sistemas distribuídos.
- Transparência: É definida por Coulouris, Dollimore e Kindberg (2007, p. 35) como a “característica de percepção, para o usuário, de todo o sistema distribuído como um único sistema”.
- Abertura: Um sistema aberto se caracteriza, especialmente, pelo fato de suas especificações e documentações estarem disponíveis para consulta dos desenvolvedores de interfaces para o sistema distribuído em questão (COULOURIS; DOLLIMORE; KINDBERG, 2007) e que seguem regras que

descrevem a semântica e a sintaxe desses serviços para que seja possível se pensar em aspectos como: interoperabilidade, portabilidade, extensibilidade e adaptabilidade (KANGASHARJU, 2008)

- Escalabilidade: Essa meta pode ser dividida em três partes: Escalabilidade em relação ao tamanho e facilidade em crescimento do sistema; Escalabilidade geográfica, separando componentes e recursos de um sistema a uma distância considerável; Escalabilidade em nível administrativo, o que, em tese, é a característica que torna um sistema fácil de se gerenciar (STEEN; TANENBAUM, 2008).

Perante as características comuns entre todos os sistemas distribuídos, enxerga-se a predominância de alguns deles como a arquitetura cliente-servidor, que se responsabiliza por acessos de clientes a recursos de provedores externos; sistemas de múltiplos processadores, comuns em clusters de computadores que visam maior velocidade de processamento; e arquiteturas *peer-to-peer,* onde não há distinção entre servidores e clientes. Há uma diversidade considerável quando se fala de tipos de sistemas distribuídos, mas o escopo deste trabalho se limita unicamente a dois deles, Sistemas de arquivos distribuídos e sistemas de processamento paralelo. A escolha desse escopo se deu devido à sua relevância no estudo em que se pretende apresentar.

1.2.1 Sistemas de arquivos distribuídos

Segundo Kerrisk (2010, p. 254, tradução nossa), “um sistema de arquivos (*File System, FS)* é uma coleção de arquivos e diretórios que seguem uma organização”. Onde, a estrutura organizacional é definida pelas premissas e objetivos de cada FS.

Existe uma grande variedade de sistemas de arquivos, entre eles, locais e distribuídos.

Os mais comuns entre os sistemas locais são:

- No Linux: EXT4 e suas variações;
- No Windows: FAT32 e NTFS;
- O padrão ISO 9600 para CD-ROM.

Entre os sistemas distribuídos, os mais populares de acordo com Colouris, Dollimore e Kindberg (2007), Steen e Tanenbaum (2008) são:

- NFS[1], *Network File System* que utiliza a arquitetura de cliente-servidor simétrica; onde cada servidor mantém seus próprios arquivos;
- Andrew File System que também utiliza a arquitetura de cliente-servidor, mas faz uso de um cache de arquivos nos clientes para diminuir o tráfego na rede;
- GoogleFS, abordado com mais detalhes no Capítulo 2, que gera uma base realmente distribuída, dividindo blocos de todos os arquivos entre os nós que fazem parte do sistema distribuído;

Sistemas de arquivos distribuídos (*Distributed File System*, DFS), diferentemente dos sistemas locais, preocupam-se com o compartilhamento de dados em diferentes computadores espalhados em uma rede, presando sempre pela transparência de localização dos arquivos, além de todas as métricas anteriormente citadas para a existência de um sistema distribuído.

Como abordado anteriormente, os processadores atuais possuem recursos para que seja possível paralelizar programas em único computador. Mas nem sempre é possível resolver todos os problemas de forma eficiente. Em um processamento de uma grande massa de arquivos, segundo Ian (1995) e White (2014), pode ser necessário a distribuição de tarefas em diversos computadores ou clusters. Assim sendo, os algoritmos de processamento distribuído, como é o caso do MapReduce (descrito no Capítulo 2), dividem as tarefas de processamento paralelo em diferentes

[1] Os sistemas de arquivos como NFS e Andrew FS também são conhecidos como sistemas de arquivos de rede.

computadores, que se comunicam por mensagens em uma rede.

1.3 Linguística de Corpus

A linguística de Corpus se encarrega da coleta e análise de corpus, ou seja, é responsável pela coleta e processamento de dados linguísticos em linguagem natural, falados e escritos, que sejam legíveis por computador. Porém, como aponta sardinha (2004, p. 16) “Nem todo conjunto de dados é considerado um corpus”, a análise dos dados necessita de conjuntos de dados específicos para caracterizar uma variedade linguística (SARDINHA, 2004).

Contudo, esse trabalho se limita apenas ao processamento de corpora fazendo uso do paradigma informacional baseado em concordâncias e aplicadas a modelos estatísticos, fazendo uma simulação de comportamento do software desenvolvido por Mike Scott, WordSmith Tools, e processamento da densidade estatística lexicográfica proposto por Biber, Conrad e Reppen (1998). As técnicas utilizadas nesse são detalhadas no capítulo 4.

O próximo Capítulo explora o Google File System e o algoritmo de MapReduce, seus fundamentos e arquitetura. Também é uma revisão básica sobre o paradigma funcional, para um melhor entendimento do algoritmo.

2. MAPREDUCE E GOOGLE FILE SYSTEM

Existem diversas maneiras de se paralelizar ou distribuir processos, como abordado no Capítulo 1. Nesse Capítulo são abordados o algoritmo de MapReduce, para distribuição de processos, e o Google File System para que seja possível que todos os nós compartilhem os dados a serem processados também de maneira distribuída.

2.1 MapReduce

Map e *Reduce* são funções oriundas de linguagens funcionais, como Lisp e Scheme, e normalmente usadas para facilitar a concorrência entre processos, fazendo uso de funções matemáticas que comportam outras funções, ou seja funções de primeira classe[2]. O algoritmo de MapReduce, apresentado nesse trabalho, é resultado das pesquisas de Dean e Ghemawat, nos laboratórios da Google em 2008, para processamento de grandes massas de arquivos em um sistema paralelo.

Paradigma funcional

Para entender as das funções de Map e de Reduce utilizadas no algoritmo, deve-se entender as suas origens, presentes em linguagens de programação funcional (MERTZ, 2015). Para estes exemplos é utilizado Python, que é a linguagem abordada em todo esse trabalho e que carrega algumas características de programação funcional (RAMALHO,2015).

Função Map:

A função de Map recebe dois argumentos, uma função e uma lista de elementos a ser processada e obtém como retorno uma nova lista com todos os valores processados.

[2] Segundo Ramalho (2015, p. 175) Funções de primeira classe: Podem ser “criadas em tempo de execução; atribuídas a uma variável (...); passada como argumento de uma função; devolvida como resultado de uma função”

As listas podem conter qualquer tipo de dados aceito pela linguagem de programação e que possam ser processados pela função criada.

Figura 4 - Map em paradigma funcional

```
1  Lista = [1,2,3,4,5]
2
3  quadrado = lambda x: x**2
4
5  lista_map = map(quadrado, lista)
6  #[1,4,9,14,25]
```

Fonte: Desenvolvido pelo autor

Como se pode ver na Figura 4, uma lista com cinco valores é criada na linha 1, em seguida na linha 3 uma função anônima[3] é a atribuída uma variável chamada q*uadrado,* que eleva um elemento ao quadrado. Na linha 5 foram atribuídos os valores processados pela função Map a *lista_map.* A linha 6 exibe o valor contido na variável *lista_map*, *[1,4,9,14,25]*. Ou seja, exatamente os números da primeira lista elevados ao quadrado, *[1,2,3,4,5]*.

Função Reduce:

A função Reduce é encarregada de concatenar todos os dados de uma lista, utilizando uma função matemática e retornar um único valor.

Figura 5 - Reduce em paradigma funcional

```
1  Lista = [1,2,3,4,5]
2
3  adicao = lambda x,y: x+y
4
5  var_reduce = reduce(adicao, lista)
6  #((((1+2)+3)+4)+5)
7  #15
```

Fonte: Desenvolvido pelo autor

[3] Funções anônimas são funções de primeira classe, comuns em linguagens funcionais, que podem ser atribuídas a variáveis e podem receber funções como argumento ou retorna uma função como resultado (RAMALHO, 2015).

Como pode-se ver na Figura 5, é definida uma lista com cinco valores na linha 1, em seguida na, linha 3, foi declarada uma função chamada *adicao* e na linha 5 foi atribuído o valor processado pela função Reduce à variável *var_reduce*. Na linha 7, é possível notar o resultado final da soma e na linha 6 a operação executada pela função Reduce.

Pode-se concluir que as funções de Map e Reduce apresentadas nas linguagens funcionais são grandes auxiliadoras para trabalhar com uma quantidade de dados infinita. Pois, independentemente da quantidade de valores em uma lista a ser processada, a lista sempre receberá suas operações, independentemente de seu tamanho e espaço em memória.

Na seção 2.1.1 é abordado um aprofundamento nas funções de Map e Reduce e tenta explicar como elas se comportam em um sistema paralelo ou distribuído.

2.1.1 Algoritmo de MapReduce

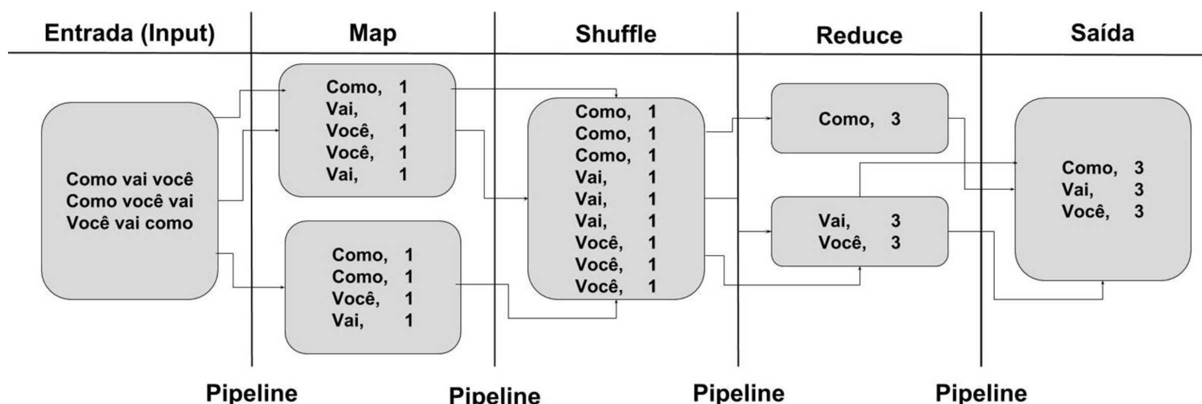
O algoritmo MapReduce (WHITE, 2015) se baseia em um sistema de *pipeline,* abordado no Capítulo 1, dividido em quatro partes. Como se pode notar na Figura 6, esse algoritmo também usa os conceitos de dividir e conquistar, abordado na Figura 1, e decomposição de dados, apresentado no Capítulo 1, divididos em diferentes nós, ou seja, operando de maneira distribuída.

Também é possível notar que existem diferentes nós trabalhando nessa operação, onde os primeiros se encarregam do *input* dos dados expostos no sistema de arquivos, seguidos pela função de Map que envia seus resultados para uma função de Shuffle, que por sua vez, compartilha seus resultados com a função Reduce, que se encarrega de concatenar a saída dos dados processados no sistema de arquivos.

Como é possível notar na Figura 6, foi usado um exemplo convencional de contagem de palavras (MINER; SHOOK, 2013), se baseando no algoritmo de dividir e conquistar, abordado na Figura 1, que é usado para o entendimento individual de cada função

contida no algoritmo.

Figura 6 - Algoritmo MapReduce



Fonte: Baseado em Bruno Reckziel Filho, 2013

Baseando-se na Figura 6, pode-se descrever todos os processos contidos no algoritmo:

- Input: a função se encarrega de extrair linha a linha as informações contidas nos arquivos do sistema e as encaminha para entrada padrão, STDIN, da função Map.
- Map: a função se encarrega de dividir as entradas em tuplas de <Chave, Valor> separando as palavras da entrada, uma a uma e as encaminhando para entrada padrão da função de Shuffle.
- Shuffle: como as entradas do Shuffle ocorrem em tuplas, esta função é encarregada de organizar em ordem alfabética, ou numérica, as chaves da entrada e sua saída padrão, STDOUT, é direcionada a função de Reduce.
- Reduce: a função executa a soma dos valores da tupla onde as chaves tenham o mesmo conteúdo e posteriormente a isso, encaminha sua saída para o sistema de arquivos.

É possível perceber também na Figura 6 que as ligações entre as funções são representadas por diversas saídas, onde múltiplos *workers*, conceito abordado na seção 1.1.2, leem as entradas de forma recursiva e paralela, pois diferentes *workers* se encarregam de um único *input* e fazem uso das entradas e saídas padrões de cada

worker. Com o uso do *pipeline* todos os *workers* recebem entradas e distribuem saídas para outros *workers* disponíveis.

Também vale ressaltar que quando se fala em MapReduce com processamento paralelo, existe um único nó na estrutura que se faz responsável pela distribuição de processos e *status* dos nós integrantes do cluster (ocioso, em andamento ou tarefa concluída), como pode-se ver na Figura 7, nomeado ‘Master’. Ele é o intermediário entre os clientes, nós ou aplicações, e designa quais nós serão responsáveis pelas tarefas vistas anteriormente e também responsável pelas falhas, caso um nó não execute a tarefa com sucesso o nó Master atribuirá a tarefa a outro nó ocioso.

2.2 Google File System

O Google File System, GFS, foi projetado para trabalhar com *hardware commodity*[4] que são altamente suscetíveis a falhas. Por esse motivo, existe a necessidade de monitoramento constante para uma rápida recuperação do sistema. O GFS também é projetado para otimizar a performance com arquivos grandes, que obtém a performance desejada por meio do particionando em partes menores (blocos de 64MB). Jeffrey Dean e Sanjay Ghemawat escreveram seu algoritmo de MapReduce em 2004, fazendo uso de uma tecnologia desenvolvida um ano antes no Google, o Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2013). O GFS é um sistema de arquivos, como visto no Capítulo 1, que se baseia em divisões dos blocos de arquivos em um cluster.

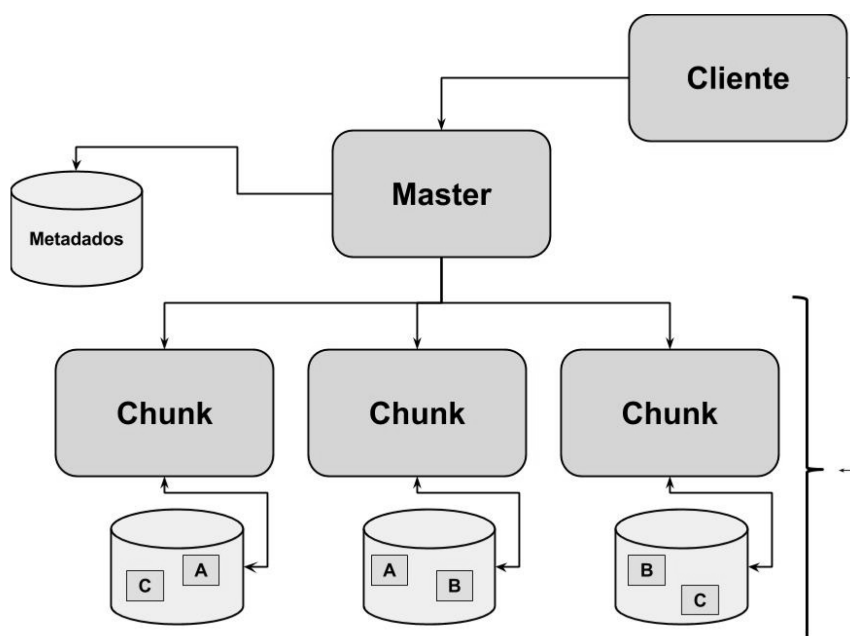
2.2.1 Arquitetura

O sistema de arquivos GFS se comporta de maneira similar, no cluster, ao algoritmo de MapReduce, tendo um nó ‘Master’, que é responsável pela organização dos metadados e pela localização de cada partição de arquivos contidos nos nós do sistema de arquivos (GHEMAWAT; GOBIOFF; LEUNG, 2013).

[4] Peças de baixo custo, computadores comuns

Os outros nós integrantes do cluster são nomeados de ‘ChunkServers’. Eles são os responsáveis pelo armazenamento real dos fragmentos dos arquivos e suas replicações, como se pode notar na Figura 7. Todos os arquivos existentes no GFS, após serem fragmentados em blocos caso tenham um volume superior a 64MBs, são replicados em três nós diferentes na configuração padrão, o que pode facilitar o acesso aos arquivos caso haja uma falha com o nó que armazena esse fragmento (DÍAZ-ZORITA, 2011).

Figura 7- Visão geral do Google File System



Fonte: Adaptado de Ghemawat; Gobioff; Leung (2013)

2.2.2 Interação

Após a interação com o nó Master, que dirá a localização dos arquivos, os clientes podem se comunicar diretamente com os ChunkServers, por meio de uma comunicação TCP, que armazenam os arquivos necessários, evitando assim um gargalo na interação dos nós de arquivos com o Master que pode atender requisições de múltiplos clientes simultaneamente. Para evitar “gargalos de rede”, os clientes mantêm um cache local dos fragmentos a serem compartilhados.

2.2.3 Nó Master

Diferentemente dos sistemas de arquivos tradicionais, o GFS não lista os dados organizados por diretórios, nem tem suporte a links, como os FS baseados no padrão POSIX[5]. A sua tabela de metadados é formada com o caminho absoluto dos arquivos, assim sendo facilmente transferida para a memória RAM, se comprimida.

As políticas de replicação do GFS existem com o propósito de aumentar a disponibilidade, maximizar a confiabilidade e, principalmente, evitar problemas com a rede. Caso haja algum problema com um ChunkServer, ou um aglomerado deles, a proposta é que sempre exista o bloco esperado em algum nó e também para aproveitar toda a largura de banda existente, caso haja diversos *switches* para interligar os nós.

Quando existe uma grande demanda por um bloco específico de um arquivo, o servidor Master tenta transferir cópias do bloco para nós espalhados em diferentes servidores utilizando como critério sua distância por tempo de rede e periodicamente executa um balanceamento de carga, com base nos três fatores já citados (GHEMAWAT; GOBIOFF; LEUNG, 2013).

Para o processo de balanceamento, das réplicas e dos ChunkServers, o GFS considera 3 fatores:

1. Alocação de novas réplicas em ChunkServers abaixo da média de utilização de espaço em disco;
2. Limitação do número de criações recentes, pois após o processo de balanceamento alguns nós podem ficar sem espaço o suficiente para alocar blocos de arquivos e o número de réplicas especificadas pelo usuário pode ser grande e impactar no desempenho;
3. Alocação de blocos em nós que não fazem parte da mesma rede IP. Para minimizar o impacto das falhas em aplicações em execução.

[5] POSIX, *Portable Operating System Interface*. É uma família de normas para manutenção e compatibilidade entre sistemas operacionais definidas pela IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos)

Como é possível notar, a junção de um sistema de arquivos distribuídos, como o GFS, e o algoritmo de MapReduce é considerada uma boa proposta para desempenho de processamento distribuído, tolerância a falhas e escalabilidade. O próximo Capítulo aborda o Apache Hadoop, um *framework* de código aberto, que faz uso do algoritmo de MapReduce em um sistema de arquivos distribuído.

3. APACHE HADOOP

O Apache Hadoop é um *framework* de MapReduce, baseado nas implementações da Google descritas no Capítulo 2, administrado pela Apache Software Foundation (ASF) e construído na linguagem Java. O Apache Hadoop tem como premissa fazer computação distribuída de alta escalabilidade, grande confiabilidade e tolerante a falhas (DÍAZ-ZORITA, 2011), fazendo uso de um cluster de computadores construído com *hardware commodity* (DEROOS *et al*, 2014).

Contudo, existem diversas implementações de MapReduce, como Disco, um *framework* construído em Python para resolver problemas de paralelização de processos (DISCO, 2015); O MongoDB, um banco de dados NoSQL[6], que também implementa as funções de Map e Reduce para expor resultados de consultas efetuadas no banco (MONGODB, 2015). Entre todas as implementações optou-se pelo Apache Hadoop, por ser um *framework* bem documentado e, como abordado na seção 3.4, permite trabalhar com uma infinidade de linguagens de programação de maneira simples.

3.1 Arquitetura

Pode-se dividir a arquitetura do Apache Hadoop em quatro pilares fundamentais. O Hadoop Common, que são as bases para que os outros subsistemas se comuniquem; O YARN (*Yet Another Resource Negotiator*) que é encarregado das funções de gerenciamento do cluster e intermediário entre frameworks de processamento. Um sistema arquivos distribuídos, nomeado *Hadoop Distributed File System* (HDFS); E um motor de processamento distribuído chamado de MapReduce (WHITE, 2014). As seções a seguir exploram cada um dos quatro pilares.

3.1.1 Hadoop Common

Hadoop common é uma coleção de bibliotecas e utilitários que oferecem suporte a todo o ecossistema Hadoop, fazendo dele a base de funcionamento para todos os

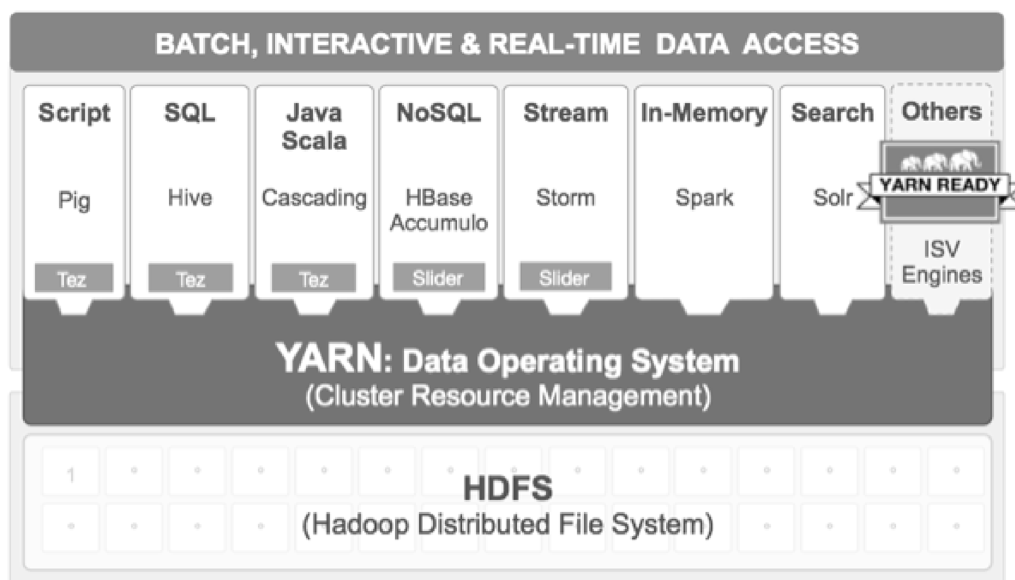
[6]Banco de dados não relacionais, são bancos onde não existem esquemas de tabelas fixas e geralmente não suportam interação com bancos relacionais (SQL)

sistemas que interagem, de alguma maneira, com o Hadoop, ou interações internas de todo o *framework*, como, por exemplo, a criação das páginas HTTP, para que seja possível interagir com o cluster sem linhas de comando. Também é responsável pela interface de interação com a linguagem Java e a comunicação SSH[7] entre os nós.

3.1.2 Apache YARN

Segundo Sulivan, 2014 (Tradução do nossa) "YARN é um gerenciador de recursos que foi criado para separar as capacidades do motor de processamento e gestão de recursos de MapReduce (...) implementado no Hadoop 1". É o responsável pela intermediação entre as aplicações de computação distribuída, como o MapReduce, Spark, Storm, Hoya, Malhar e Samza[8] e por esse motivo, muitas vezes o YARN é chamado de Sistema Operacional do Hadoop. Como é possível notar na Figura 8, as aplicações enviam requisições para o YARN, que se encarrega de encontrar nós ociosos ou disponíveis para o processamento, independente do modelo, e também interage com os administradores de dados do HDFS para saber se o nó disponível contém os blocos de arquivos a serem processados, por meio de conexões TCP.

Figura 8 - YARN



Fonte: HortonWorks, 2015a.

[7] Secure Shell
[8] Frameworks de processamento compatíveis com YARN

A partir da versão 2.0 do Apache Hadoop, junto à inserção do YARN, foram adicionados outros modelos de processamento, além do MapReduce, como algoritmos para aprendizado de máquina e processamento interativo (APACHE, 2014). Também podem ser usados outros *frameworks* como Spark[9] e scripts construídos em Pig latin[10].

Como visto no Capítulo 2, o algoritmo de MapReduce implementado pela Google faz uso de um sistema de arquivos distribuído, o GFS. Com o Apache Hadoop não é diferente, ele faz uso do Hadoop Distributed File System e a Seção 3.2.3 se dedica inteiramente a ele e sua integração no ecossistema Hadoop.

3.1.3 Hadoop Distributed File System

O HDFS é uma implementação do GFS, com algumas modificações, que se baseia no padrão POSIX e, como o YARN, também nomeia os nós integrantes do cluster. O nó Master recebe o nome de NameNode, NN, e os demais integrantes do HDFS foram nomeados DataNodes, DN. Embora o HDFS seja baseado no GFS existem diferenças entre o GFS e o HDFS, algumas delas são listadas a seguir (DAPHALAPURKAR; SHIMPI; NEWALKAR, 2014):

- Implementação: O HDFS é multiplataforma, escrito em Java e segue uma licença livre, Apache 2.0. Já o GFS é escrito especialmente para sistemas Linux, escrito em C/C++ e têm uma licença proprietária;
- Arquitetura: O HDFS mantém somente os metadados em seu servidor de arquivos. Diferente disso, o GFS armazena arquivos em seu servidor e cria políticas de armazenamento;
- Integridade de dados: O GFS executa uma comparação de réplicas, enquanto o HDFS deixa a reponsabilidade do *checksum* para o último nó a receber os arquivos;

[9] O Apache Spark é abordado na seção 3.6
[10] Linguagem de alto nível criada pela Yahoo para facilitar o processamento com Hadoop

- Tamanho dos blocos: O GFS tem como padrão blocos de 64MB, já mencionado na seção 2.2. Diferente disso o HDFS armazena blocos de 128MB.

Como dito anteriormente, pode-se notar na Tabela 1, que os comandos e a estrutura do HDFS são similares aos sistemas de arquivos do mundo UNIX, que usam o padrão POSIX.

Tabela 1- Comandos básicos do HDFS

cat	Escreve o arquivo em nossa saída padrão
get	Insere um arquivo do HDFS em nosso FS
ls	Lista o diretório corrente
mkdir	Cria uma diretório no diretório corrente
put	Insere um arquivo local no HDFS
rm	Remove o arquivo do HDFS
rmr	Remove um diretório

Fonte: Lan (2011, p. 298).

Embora as interações com o HDFS sejam similares ao padrão POSIX (como *ls*,*cat*,*mkdir*, entre outros), existem comandos específicos do HDFS que não fazem parte do padrão, como o *CopyFromLocal* e *CopyToLocal*, mas que têm extrema importância para a interação do HDFS com o FS local. Por exemplo: o comando *CopyToLocal*, ou seu atalho *get*, faz a cópia de um arquivo existente do HDFS e que está fragmentado em blocos para um arquivo único no sistema local. Semelhante a isso, embora de forma invertida, o comando *CopyFromLocal*, ou seu atalho *put*, faz uma cópia de um arquivo local e o fragmenta em blocos de 128MB e o distribui para o HDFS em 3 réplicas (LAN, 2011).

3.1.4 MapReduce

Anteriormente à inserção do YARN como um subprojeto do Hadoop, o MapReduce, baseado nas implementações da Google, era responsável por todas as interações de nós e também a única arquitetura de processamento disponível no Projeto Hadoop. Com a chegada do YARN o MapReduce se tornou um dos modelos de programação

distribuída aceitos pelo Hadoop, mas se mantém como parte fundamental da arquitetura, pois a grande maioria dos problemas resolvidos usando Hadoop ainda são efetuados usando MapReduce (HORTONWORKS, 2015b).

Embora o MapReduce seja utilizado interagindo com YARN, ele também nomeia os nós integrantes do cluster com uma nomenclatura própria. O nó que designa as funções e gerencia as operações recebeu o nome de JobTracker. Os nós restantes foram chamados de TaskTrackres e designa as funções de maneira similar ao MapReduce da Google como as funções de *Input*, Map, Shuffle e Reduce (WHITE, 2014). Como abordado na Figura 6.

3.2 Subprojetos ou Ecossistema Hadoop

Existem diversos subprojetos integrados ao Apache Hadoop, formando um ecossistema, e grande parte deles são mantidos pela ASF. Os principais subprojetos de acordo com Apache (2015a) são:

- Ambari: Um gerenciador de clusters Hadoop que fornece uma interface amigável para inclusão e exclusão de nós;
- Avro: Sistema de serialização de dados, para compactação de arquivos em forma binária;
- Cassandra: Um banco de dados distribuído e tolerante a falhas;
- Chuka: Um *framework* escalável para análises de logs do Hadoop;
- Hbase: Um banco de dados distribuído, que pode estar contido no HDFS;
- Hive: Uma estrutura de *Datawarehouse* com um compactador de arquivos e consultas *ad hoc*.

3.3 Hadoop Streaming

Como mencionado anteriormente, o Hadoop Streaming é uma API do Hadoop

Commom que permite trabalhar com outras linguagens de programação, além do Java, para escrita de aplicações de MapReduce. Ele permite escrever qualquer aplicação que seja capaz de ler e escrever na entrada e saída padrão (STDIN e STDOUT) (LAN, 2011). É possível visualizar na Figura 9, o *script* de Map escrito em Python compatível com a API de Streaming oferecida pelo Apache Hadoop.

Figura 9 - Map para Hadoop Streaming

```
1  import sys
2
3  #Eduardo escrevendo TCC
4  for entrada in sys.stdin:
5      lista_de_palavras = entrada.split()
6      #['Eduardo', 'escrevendo', 'TCC']
7
8      for palavra in lista_de_palavras
9          print(("%s\t1")%(palavra))
10         #Eduardo    1
11         #escrevendo 1
12         #TCC        1
```

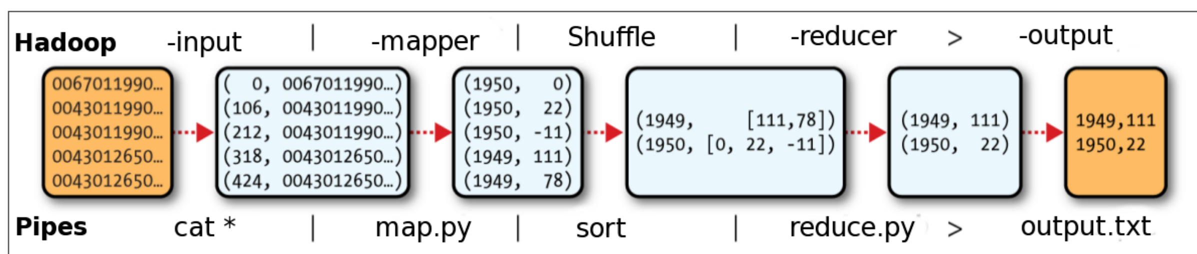
Fonte: Desenvolvido pelo autor

Na linha 1 a biblioteca sys é importada, para que seja possível ler a entrada padrão. Com isso, na linha 4 é feita uma iteração em cada linha obtida na entrada padrão e são atribuídas à variável entrada. Na linha 5, a *string* contida na entrada, que é um objeto do tipo string, executa o método *split*, que retornará a *lista_de_palavras*, a nossa *string* em formato de lista. Como por exemplo a entrada da linha 3, é retornada no formato apresentado na linha 6. Em seguida, cada valor iterado na lista é encaminhado a saída padrão seguido de um valor 1, como se pode ver nas linhas 10, 11 e 12.

O Hadoop streaming também dispõe de seus próprios parâmetros para um processamento distribuído mais eficiente e personalizado. O que resulta em uma grande gama de opções, nas funções de *Map*, *Reduce* e Shuffle, similar aos parâmetros usados nos pipelines do Linux. A Figura 10, baseada em Tom White (2014), mostra as similaridades entre os dois sistemas (Hadoop Streaming e Linux Pipelines). O Capítulo 4 apresenta um comparativo de tempo de contagem de palavras

usando *pipes* e a API do Hadoop streaming.

Figura 10- Pipelines e Hadoop Streaming



Baseado em: White, 2015

Todas as aplicações de Map e Reduce podem ser escritas de maneira simples e podem ser comparadas a escrita de aplicações que usam o Pipeline no mundo UNIX, onde cada entrada é processada em sua ordem de chegada (WHITE, 2014).

Dito isso, a documentação oficial do Hadoop Streaming (APACHE, 2015b), apresenta uma tabela com todos os parâmetros aceitos pela API e, mesmo com a existência e completude da documentação, a Tabela 2 expõe os principais parâmetros para o entendimento do Capítulo 4.

Tabela 2 - Parâmetros principais do Hadoop Streaming

-input <arquivo ou diretório>	Indica quais serão os arquivos do HDFS a serem processados pelo hadoop
-output <diretório>	Indica o diretório do HDFS onde serão dispostos os arquivos após o processamento
-mapper <arquivo>	Indica qual será o script de *Map*
-reducer <arquivo>	Indica qual será o script de *Reduce*
-D <propriedade>	Define propriedades para o Streaming, como é observado na Tabela 3
-combiner[11] <arquivo>	Cria um pequeno Reduce de cache na memória RAM

Fonte: Apache 2015b.

[11] O parâmetro *combiner* recebe o mesmo script usado no Reduce, para criação de um *cache* e cria um *pipe* de redução antes mesmo do script de redução, evitando o acumulo de pacotes trafegados pela rede.

Um exemplo de execução do script de Streaming pode ser visto na Figura 11, onde são utilizados diversos parâmetros de configuração da API de Streaming. Vale ressaltar que todos os scripts de streaming são baseados em Shell Script e rodam com o auxílio do BASH[12].

Figura 11- Exemplo de uso do Hadoop Streaming

```
1   #!/bin/bash
2   streaming='/usr/local/hadoop/share/hadoop/tools/lib'
3
4   hadoop jar $streaming/hadoop-streaming-2.7.1.jar \
5   -D dfs.data.dir=/tmp \
6   -D mapreduce.job.reduces=2 \
7   -D mapreduce.job.name=teste-1 \
8   -D stream.num.map.output.key.fields=1 \
9   -mapper `pwd`/map.py \
10  -reducer `pwd`/reduce.py \
11  -input /data/* \
12  -output /saida
```

Fonte: Desenvolvido pelo autor

É possível, com uso das tabelas 2 e 3, observar todas as nuances existentes nesse script, embora nas linhas 1,2,4 e de 9 a 11 estejam presentes peculiaridades dos sistemas GNU/Linux. Pode-se observar que a linha 1 do nosso script se encarrega de indicar o caminho executável do nosso interpretador de comandos, o bash. Na segunda linha uma variável nomeada de *streaming* recebe o caminho de nosso sistema de arquivos onde a biblioteca da API do Hadoop Streaming está localizada. O comando *pwd* presente nas linhas 9 e 10 indicam ao Streaming que nossos *scripts* de Map e Reduce estão disponíveis no diretório corrente. Também é possível notar na linha 11 um caractere '*', para o interpretador de comandos o caractere '*' é um coringa que significa qualquer coisa. Nesse exemplo, dentro da pasta /data existem diversos arquivos. Então, com o uso do coringa, a entrada do Streaming será todo e

[12] BASH ou Born Again SHell, interpretador de comandos criado pelo projeto GNU.

qualquer arquivo existente no diretório /data.

Além dos parâmetros abordados na Tabela 2, para um melhor entendimento do estudo de caso, deve-se atribuir algumas propriedades ao Streaming com o parâmetro –D. Elas podem variar entre o nome da seção de Streaming e até modificar propriedades presentes no mapred-site.xml (abordado no apêndice A).

Tabela 3 – Parâmetros de definições do Streaming

dfs.data.dir	Altera o diretório temporário de armazenamento de arquivos em fase de processamento
mapreduce.job.map	Define quantos *Jobs* serão utilizados para executar o script de *map*
mapreduce.job.name	Define qual será o nome da rodada de Streaming para facilitar a leitura de *logs*
mapreduce.job.reduce	Define quantos *Jobs* serão utilizados para executar o script de *reduce*
stream.num.map.output.key.fields	Define quantas chaves serão utilizadas para a função pela função de Shuffle em sua execução
stream.output.field.separator	Define qual será o delimitador na função de Shuffle

Fonte: Apache 2015b.

Como dito anteriormente, existem diversos comandos também de definição de *Jobs* que podem ser usados no Hadoop streaming, proporcionando uma grande gama de operações diferentes. Por exemplo, pode-se definir qual será o delimitador da função de *Shuffle*, em processamento de textos, como é visto no Capítulo 4, é uma função de extrema importância, pois não seria possível fazer uma grande ordenação alfabética de todas as saídas da função de map.

Também vale ressaltar que para a atribuição de propriedades deve-se usar o caractere ‘=’, como pode-se ver na Figura 11, linha 8 e replicado a seguir:

-D stream.num.map.output.key.fields=1

Como abordado na Tabela 3, esta linha define a quantidade de campos de chaves que devem ser processados pelo Shuffle, no caso 1. Caso a saída seja apresentada como:

Eduardo fazendo TCC   1

Somente a primeira *string, Eduardo,* será usada para ordenação. O que pode trazer uma maior velocidade de tempo de processamento.

Entretanto, o Hadoop Streaming não é a única maneira de interagir com diferentes linguagens a partir da versão 2 do Hadoop. Mas para usuários de sistemas baseados em UNIX ela se mostra extremamente simples e com baixa curva de aprendizado, pois não é necessário o aprendizado de um novo framework de interação, como o Spark.

Assim, na próxima seção, 3.4, são abordados os diferentes modos de operação do Hadoop, o que faz dele um *framework* ainda mais versátil, pois além de poder ser usado com diferentes linguagens de programação existem diversas alternativas de instalação.

3.4 Modos de operação

Existem três modos de operação quando se usa o Apache Hadoop, a instalação de ambos os métodos é apresentada no apêndice A:

1. *Standalone*: Seria uma versão para testes, sem *daemons*, e que não roda de maneira distribuída.
2. *Pseudodistributed*: O modo Pseudo distribuído usa os *daemons* do Hadoop e

trabalha com processamento paralelo e um único computador. Onde diferentes *threads* trabalham individualmente.

3. *Fully Distributed*: Sendo o modo de processamento mais poderoso do Hadoop, trabalha com distribuição em clusters, como é abordado em todo esse trabalho.

3.5 Considerações finais sobre o Hadoop

Embora sejam abordados os aspectos e diferenças entre a versão 1 e a versão 2 do Hadoop no decorrer do Capitulo 3, esta curta seção se dedica a explicar as principais diferenças entre as versões e a gama de novos problemas que podem ser resolvidos com a inserção do YARN, ou o uso de outros gerentes de computação como o Weave ou Cloudera SDK. Contudo, também é importante abordar a importância dos subprojetos do Hadoop para uso em problemas específicos.

Como explicado por Murphy *et. al*, 2014, “O aspecto mais interessante do Hadoop 2 é essa habilidade de trabalhar com múltiplos modelos de programação e *frameworks* de aplicação” (Traduzido pelo autor, p. 241). Existem diversas vantagens em usar o YARN e existem diferentes *frameworks* aplicáveis a esse novo conceito, como são explicados na lista a seguir:

- Distributed-Shell: é uma maneira de executar comandos em todos os nós pertencentes ao cluster em um único terminal, ou seja, todos os comandos atribuídos ao Distributed-Shell são executados em tempo real em todos os computadores;
- Hadoop MapReduce: Como é abordado na seção 3.2.4, o Hadoop MapReduce deixou de ser aceito como o único modelo de programação pertencente ao ecossistema Hadoop, mas, contudo, ainda continua sendo mantido pelo projeto. Com isso, temos total compatibilidade entre programas escritos para o Hadoop 1, ou para API’s como o Hadoop Streaming;
- Apache Tez: É um automatizador de tarefas de MapReduce. Com o Hadoop 1 só era possível rodar uma instancia de MapReduce por vez, ou seja, devia se esperar uma tarefa ser executada, o que pode levar horas, em algumas ocasiões. Com o Apache Tez, é possível criar uma fila de sessões de MapReduce a serem executadas e nós ociosos podem executar as tarefas

presentes na fila, o que pode diminuir o tempo de espera para tarefas serem concluídas;

- Apache Spark: um *framework* construído para trabalhar com computação distribuída com uso de *pipelines*, abordado na seção 1.1.3, que oferece um terminal interativo, como o Distributed-Shell, e permite, com o uso do YARN, processamento paralelo em tempo de execução. Por exemplo, pode-se fazer uso do Python terminal ou Scala[13] terminal e existem funções adicionais interpretadas pelo Spark que se encarrega de as paralelizar (SANKAR; KARAU, 2015);
- Apache Storm: é um *framework* que faz o YARN lidar com dados em tempo real, com uma interface amigável e oferece suporte ao HDFS (STORM, 2015), também pode executar funções de MapReduce sem pausa. Com isso, o cluster Hadoop pode executar as entradas sem que exista uma sessão de execução (Murphy *et al*, 2014);
- Apache HOYA (Hbase on YARN): disponibiliza uma interface dinâmica do HBase que interage com o YARN e salva os dados de sessão de aplicação no HDFS.

3.6 MapReduceLib

Durante o processo de desenvolvimento deste trabalho foi construída uma biblioteca para interação entre o Python e o Hadoop, Nomeada MapReduceLib. A MapReduceLib, como é vista na Figura 12, pode ser utilizada para diversas finalidades do Hadoop e também pode ser importada no terminal interativo do Python, para um comportamento similar ao Distributed-Shell ou a interface Python para o Spark (PySpark).

Embora o nome da biblioteca seja MapReduceLib, ela também incorpora algumas das principais funções do HDFS, descritas na Tabela 1, executa funções administrativas do Hadoop, como a formatação de DataNodes e NameNodes, e também interage com os scripts localizados no diretório /sbin do Hadoop que iniciam os serviços do HDFS

[13] Scala: Uma linguagem de programação funcional que faz uso da Java Virtual Machine (JVM).

e do YARN individualmente e também em conjunto, deixando o cluster pronto para sessões de MapReduce executadas pelo Streaming.

Dito isso, a MapReduceLib também pode ser importada no terminal interativo do Spark, provendo a ele funções de manipulação de dados no HDFS não disponíveis em sua configuração *default*.

Figura 12- Exemplo de uso da MapReduceLib

```
1   from mapreducelib import *
2
3   hadoop("start")
4
5   def run(put,map,reduce,entrada,saida,nome):
6       hdfs_rm_dir(entrada)
7       hdfs_mkdir(entrada)
8       hdfs_put(put, entrada)
9       hdfs_rm_dir(saida)
10      run_map_reduce(map,reduce,entrada,saida)
11      hdfs_get("saida",nome)
12
13  run("moby/*","map.py","reduce.py","data","saida","saida.dat")
14
15  hadoop("stop")
```

Fonte: Desenvolvido pelo Autor

Pode-se notar na Figura 12, que a primeira linha do script faz a importação de todos os componentes da biblioteca MapReduceLib, com o uso do caractere coringa '*', abordado na sessão 3.6. Já a linha 3 inicia todos o daemons do Hadoop, o que o deixa pronto para uso. Em seguida, na linha 5 é criada uma função chamada *run* que recebe como parâmetros *put, map, reduce, entrada, saída* e *nome*. Os quais são utilizados da linha 6 a 9 e também na linha 10 para interação com o HDFS. Assim sendo, a linha 10 se destina a executar uma sessão de MapReduce, usando o Hadoop Streaming e a linha 13 executa a função run. Por fim, a linha 15 finaliza todos os daemons do Hadoop**.** As linhas de 6 à 11 fazem uso das funções definidas na biblioteca MapReduceLib.

Tabela 4 - Comandos principais da MapReduceLib

Comandos	**Função**
hdfs_ls(<diretório>)	Lista o diretório no HDFS
hdfs_rm(<arquivo>)	Remove o arquivo presente no HDFS
hdfs_rm_dir(<diretório>)	Remove um diretório presente no HDFS
hdfs_cat(<arquivo>)	Exibe o arquivo do HDFS na STDOUT
hdfs_put(<arquivo>, <diretório>)	Insere um arquivo em um diretório do HDFS
hdfs_get((<arquivo ou diretório>))	Insere um arquivo um do HDFS no diretório local
run_map_reduce(map,Reduce,input, output)	Executa uma sessão de MapReduce usando Hadoop Streaming
hadoop(<estado>)	Executa uma função administrativa no Hadoop

Fonte: Desenvolvido pelo Autor

É possível visualizar em conjunto com a Figura 12 e a Tabela 4 alguns dos principais comandos disponíveis na biblioteca MapReduceLibe, que se encontra ainda em fase de desenvolvimento no link: github.com/z4r4tu5tr4/mapreducelib.

Com o fato de a biblioteca estar disponível no GitHub, qualquer modificação pode ser executada no código, tanto quanto a inserção de funções triviais para o usuário, quanto para a modificação de parâmetros usados por esse trabalho.
Explanada a revisão bibliográfica do Apache Hadoop, sua arquitetura e conceitos e também suas diferenças entre versões, o próximo Capítulo se destina à exploração do Apache Hadoop e sua API de streaming.

4. ESTUDO DE CASO

Este estudo de caso se destina a criar e executar aplicações paralelas para processamento de linguagem natural, mantendo o foco em Linguística de Corpus. Linguística de corpus, segundo Berber Sardinha (2004, p.3) "Ocupa-se da coleta e da exploração de corpora, conjuntos de dados linguísticos textuais coletados criteriosamente, com o propósito de servirem para a pesquisa de uma língua ou variedade linguística". Em seu Livro, Sardinha (2004) apresenta duas ferramentas que trabalham com processamento de corpus:

- WordSmith Tools: *software* prioritário desenvolvido por Mike Scott que se divide em três partes:
    - WordList: Um contador de palavras, como é abordado na Figura 6, exibe dados estatísticos e oferece resultados ordenados por frequência ou em ordem alfabética;
    - KeyWords: Faz a comparação da contagem de palavras entre dois corpora[14];
    - Concord: Onde se faz uma busca recursiva por uma palavra chave e exibe sua colocação em todas as frases onde a palavra chave existe.
- MicroConcord: *Software* gratuito e não livre, desenvolvido por Mike Scott e Tim Jhons que exibe listas de concordâncias, como a função *concord* do WordSmith Tools, mas se limita a cerca de 1.600 ocorrências de concordâncias por execução.

Todavia, os principais motivos para a construção de uma aplicação distribuída para processamento linguagem pode ser encontrada em dois fatores, listado a seguir:

- A não existência de um projeto de código aberto que trabalhe com linguística de corpus de maneira veloz; e
- A execução de testes de desempenho em processamento distribuído, para verificar a viabilidade de clusters para processamento de linguagem natural.

[14] Corpora: plural de corpus

4.1 Aplicações

Uma das aplicações demostradas nessa seção executa a contagem de ocorrências em coleções de textos e arquivos[15] formados por sistemas de gramática gerativa de linguagem. A outra aplicação busca concordâncias em coleções de textos. Por fim, a terceira aplicação simula o comportamento de aplicações de medidas estatísticas lexicográficas sugeridas por Biber, Conrad, Reppen (1998) em *Corpus linguistics*.

4.1.1 Contagem de ocorrências (WordList)

A aplicação de contagem se encarrega de criar uma lista, em ordem alfabética ou de frequência, de contagem de quantas palavras existem no texto e sua ocorrência (SARDINHA, 2004) fazendo uso de itemização[16]. As listas em ordem alfabética, por exemplo, podem ser apresentadas da seguinte forma:

Palavra	*Caractere de tabulação*	*Ocorrência*
GNU	*\t*	*3*
Linux	*\t*	*3*

O algoritmo exemplificado na Seção 2.1.1 e na Figura 6 realiza a contagem de palavras, permitindo uma simplificação do script usado para a função map de contagem de palavras, como demonstrado na Figura 9.

[15] Na linguística de corpus, segundo Sardinha (2004), arquivos são depósitos de textos sem organização própria.
[16] Processo de separar caracteres não alfabéticos associados as palavras, como ‘,’, ‘.’, ‘!’, entre outros.

Assim sendo, a STDOUT da função de *map* é direcionada a função *shuffle* que faz uso do parâmetro indicado na linha 8 da Figura 11, -D *stream.num.map.output.key.fields=1*, para a ordenação do primeiro campo da saída (No exemplo: GNU e Linux). Após a ordenação do primeiro campo, as saídas do *Shuffle,* são direcionadas a função *Reduce* que executa a verdadeira contagem de palavras e retorna um arquivo de texto similar a Figura 13.

Figura 13 - Contagem de palavras

```
1  Palavra             |Frequência
2  --------------------+--------
3  a                   |174
4  à                   | 2
5  abertura            | 4
6  ação                | 2
7  acarreta            | 4
8  acima               | 3
9  acompanhamento      | 4
10 acumuladas          | 4
11 administrativas     | 5
12 adoção              | 5
13 adotados            | 3
14 afeta               | 4
15 agilidade           | 4
16 agrega              | 2
17 ainda               | 2
```

Fonte: Desenvolvido pelo Autor

Como pode-se visualizar na Figura 13, estão dispostas as 15 primeiras palavras, em ordem alfabética, coletadas de um arquivo de texto. Contudo, a função de streaming também faz uso de um cache de combinação, por meio do parâmetro *–combiner,* abordado na Tabela 2.

Embora a aplicação de contagem de palavras pareça extremamente simples, é altamente paralelizável, além de utilizar a recursividade; ela lê todas as palavras presentes de um arquivo de texto ou uma sequência deles.

4.1.2 Concordância

A aplicação de concordância é destinada a encontrar a palavra-chave dentro de corpus linguístico e exibir suas concordâncias, similar ao comando grep do UNIX, porém de maneira distribuída usando MapReduce. Como é possível notar na Figura 14.

Figura 14 - Concordância

```
 1 odos os recursos funcionais envolvidos . Desta maneira , a determinação clara d
 2 ão dos modos de operação convencionais . As experiências acumuladas demonstram
 3 dos procedimentos normalmente adotados . O que temos que ter sempre em mente é
 4 as condições inegavelmente apropriadas . A nível organizacional , o julgamento
 5  utilizados na avaliação de resultados . Desta maneira , o início da atividade
 6  de alternativas às soluções ortodoxas . A certificação de metodologias que nos
 7 financeiras e administrativas exigidas . Por conseguinte , a estrutura atual da
 8 rizes de desenvolvimento para o futuro . As experiências acumuladas demonstram
 9 financeiras e administrativas exigidas . É claro que o aumento do diálogo entre
10 as consequências das novas proposições . Por conseguinte , o julgamento imparci
```

Fonte: Desenvolvido pelo Autor

Pode-se visualizar na Figura 14 a busca pelo caractere '.', que indica o final de uma frase e em alguns casos o início de outras. Desta maneira é possível se obter todas as ocorrências de uma palavra, como no exemplo da contagem, só que acompanhada de seu contexto

Contudo, existem diversas ferramentas para buscar concordância em palavras, como o *Concord* usado no WordSmith Tool, mas também para uso não paralelo, existe uma biblioteca para Python chamada NLTK[17], a qual serviu de inspiração de inspiração para a construção dos scripts de Map e Reduce utilizados nessa ferramenta.

4.1.3 Densidade estatística lexicográfica

A estatística pode auxiliar na tomada de decisões para produções de texto, como ensinar um falante não nativo de uma língua, por exemplo. A partir de uma palavra 'x', é possível encontrar a frequência das palavras 'y' relacionadas a ela, como demonstrado na Figura 15.

[17] Natural Language ToolKit

Figura 15 - Estatística lexicográfica em dígrafos

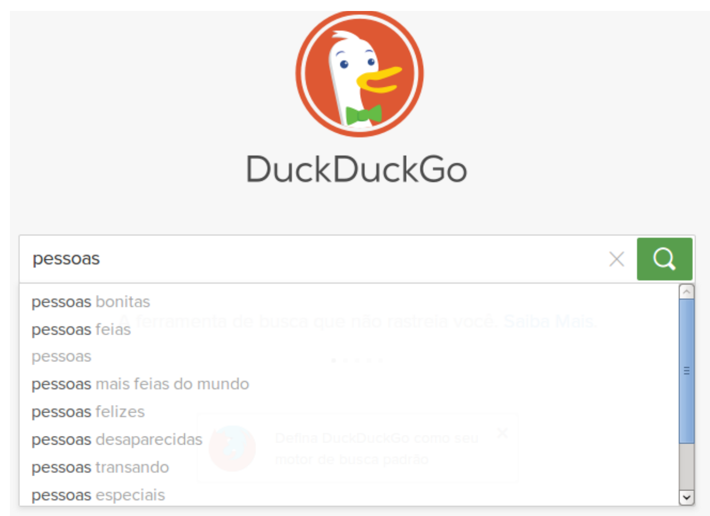
```
1  Nº  |Frase                          |Prob    |Cont
2  ---+-----------------------------+-------+-----
3  0   |     as equipes                |    28% |    35
4  1   |     as experiências           |    29% |    36
5  2   |     as hierarquias            |    24% |    30
6  3   |     as possibilidades         |    18% |    23
```

Fonte: Desenvolvido pelo Autor

Diferente da aplicação da concordância, a aplicação da estatística seleciona uma determinada palavra e mostra a ocorrência dela associada a outras palavras, bem como sua porcentagem de representatividade no corpus. No exemplo da Figura 15, a palavra escolhida 'as' aparece junto das palavras 'equipe', 'ocorrências', 'hierarquias' e 'possibilidades'.

Um exemplo de uso de estatística lexicográfica pode ser encontrado em páginas de busca, como Google, DuckDuckGo, Yahoo, entre outras. Como é visto na figura 16.

Figura 16 - exemplo de uso de sistes de busca



Fonte: *Printscreen* do site ddg.gg

É possível notar na Figura 16 que o DuckDuckGo usa associação de frequências de buscas anteriores e a partir de palavras que buscamos em tempo real.

4.2 Descrição da infraestrutura utilizada

Após a descrição das funcionalidades dos sistemas criados para este estudo de caso, esta seção é dedicada à infraestrutura usada para a criação do cluster Hadoop e suas nuances.

Como abordado no Capítulo 3, um cluster Hadoop faz uso de *hardware commodity* e a configuração do cluster usado neste estudo de caso está descrita na Tabela 5.

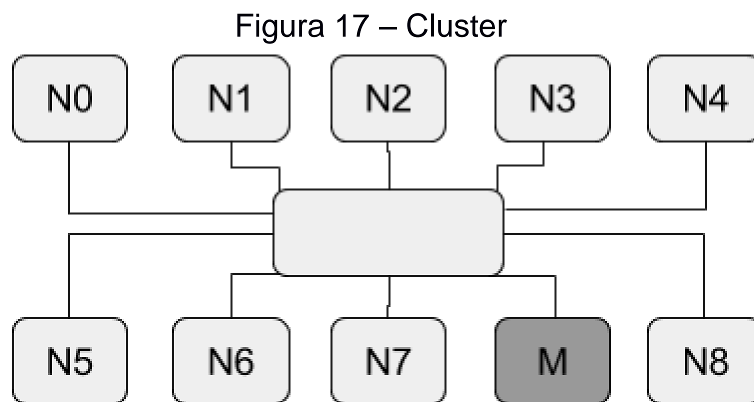
Tabela 5 - Configuração dos computadores

Processador	Core2duo E7500
Memória	4GB
Sistema Operacional	Slackware 14.1 64bits
Sistema de arquivos	EXT4

Fonte: Desenvolvido pelo autor

Pode-se notar que o cluster não é construído de *hardware* de última geração, pois a proposta deste estudo é analisar a viabilidade da construção de um cluster formado por 10 computadores comuns. É possível notar, na Figura 17, a topologia usada no cluster utilizado neste estudo de caso.

Figura 17 – Cluster



Fonte: Desenvolvido pelo Autor

Com isso, a Figura 14, deixa visível que existem diversos computadores, de N0 e N8, que como abordado no Capítulo 3, são os nós escravos do cluster e o nó ‘M’ que faz uso das funções de Master. A lista a seguir tem como objetivo expor os nomes e funções dos nós em cada camada abordada na Figura 8:

- Para o HDFS, as máquinas com o nome iniciado em N, são usadas como DataNode e a Máquina ‘M’ é o servidor de nomes, NameNode.
- Já para as funções de MapReduce os nós iniciados em N, são usados como TaskTrackers e o nó ‘M’ como JobTracker.
- Para o YARN, os nós de nome ‘N’ executam funções de NodeManager. Já o nó ‘M’ executa funções de ResourceManager.

A importância de saber qual nó é responsável por determinada função é considerada importante na leitura de logs. Pois cada nó pode apresentar problemas de configuração em uma camada diferente. Por exemplo, caso haja falta de espaço em um determinado DataNode, o mesmo pode se encontrar indisponível para a função de TaskTracker, pois não há espaço em disco para fazer a modificação temporária dos arquivos em uso durante uma determinada sessão de processamento de *Jobs*.

A versão utilizada do Hadoop para a construção do cluster é a 2.6.0, visível na Tabela

6, versão estável quando foram iniciados os testes disponíveis neste estudo de caso.

Tabela 6 – Programas utilizados

Software	Versão	Observação
Hadoop	2.6.0	Modo Fully Distributed
Python	3.5.0	Interpretador Cpython
Bash	4.3.42(1)	UTF-8 encode

Fonte: Desenvolvido pelo autor

Tendo em vista a Tabela 6, são exploradas as versões dos principais *softwares* instalados no cluster e que podem alterar os resultados apresentados nesse trabalho. Um exemplo de modificação seria o uso de um terminal com um encode diferente dos utilizados pelos arquivos de entrada.

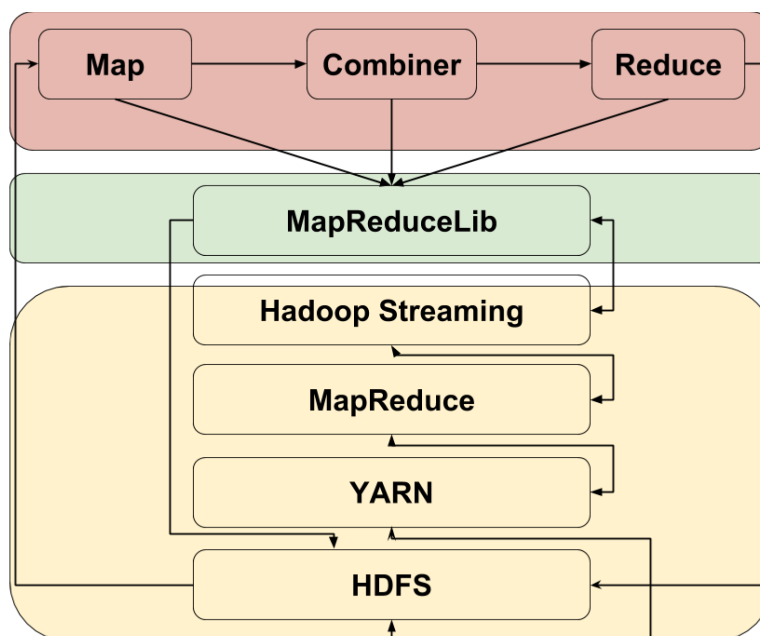
4.3 Testes e resultados

Embora a sessão 4.2 aborde toda a infraestrutura utilizada, é nesta sessão que são apresentados os testes e resultados obtidos com os dados processados usando a combinação de um cluster usando Apache Hadoop, Hadoop Streaming e a biblioteca MapReduceLib, como é possível observar na Figura 18.

Todos os testes que foram executados com Hadoop Streaming fazem uso do paramêtro –*combiner*, abordado na Tabela 2, embora na versão final do aplicativo de concordância não tenha obtido uma diferença significativa. Na Figura 18, é disposto a relação e comunicação de todos utilitários usados neste trabalho. O que pode resultar em um melhor entendimento e talvez justifique a necessidade da criação da biblioteca

MapReduceLib.

Figura 18 - esquema de uso das ferramentas



Fonte: Desenvolvido pelo Autor

Embora as relações entre os projetos aceitos pelo Hadoop estejam disponíveis na Figura 8, a Figura 18 tenta explanar o uso da MapReduceLib em relação a arquitetura do Hadoop. É possível notar que os scripts de *map, combiner* e *reduce*, fazem interação direta com a biblioteca que também oferece um suporte de comunicação com o HDFS sem o uso do YARN.

4.3.1 Testes

Foram executados testes de comparação de desempenho de 1 nó fazendo uso dos Linux Pipes e também usando os sistemas de *ThreadPool*[18] disponíveis na biblioteca

[18] ThreadPool é um utilitário pertencente a biblioteca multiprocessing.pool que cria threads assíncronas de quantidades prédefinidas a partir de uma função.

padrão do Python 3, como é possível observar no comando a seguir:

cat * | python3 <map> | sort –k1,1 –t . | python3 <reduce>

Porém, mesmo com o uso de processamento paralelo nas funções de *map* e *reduce* não houve diferenças significativas de desempenho. Conforme o tamanho do arquivo a resposta da aplicação de ordenação, *sort*, se mostrou ineficiente. Foi implementada uma solução alternativa usando o *quicksort* e as aplicações se mostram mais eficientes, porém não é possível atribuir outro algoritmo de ordenação na aplicação do cluster.

No processamento em cluster foram usadas funções do Hadoop Streaming similares à Figura 11. Com a adição do parâmetro *combiner*, fazendo uso do mesmo script de r*educe*. A função de *combiner* gera um cache em memória da junção das funções de s*huffle* e *reduce* sendo executadas pelo próprio nó onde a função de *map* é executada, o que pode gerar um menor fluxo de rede e uma pré-redução, gerando mais desempenho nos nós responsáveis pela redução e consequentemente um menor tempo de execução.

A próxima sessão se dedica ao comparativo de tempo de execução em diferentes arquivos, tanto no uso dos *pipes* em um único fazendo uso do *sort* implementado pelo projeto GNU, quanto no uso do Apache Hadoop em um cluster de 10 computadores com a MapReduceLib.

4.3.2 Comparativo

Como dito anteriormente, um dos objetivos deste trabalho é analisar a viabilidade do uso de um cluster para processamento de linguagem. Contudo, só será apresentado o comparativo da aplicação de contagem de palavras, descrita na sessão 4.1.1, pois é a única aplicação totalmente recursiva e que deve ser executada usando todos os *tokens* gerados pelo processo de itemização. A Tabela 7 apresenta um comparativo de execução da aplicação de *wordcount*.

Tabela 7 - Arquivos de análise e tempo de processamento

Tamanho	Tempo em 1 nó	Tempo Cluster
138 B	0.032s	4.274s
1.6 KB	0.034s	4.182s
15.2 KB	0.063s	5.165s
154.3 KB	0.243s	5.201s
1.5 MB	1.716s	7.259s
15.3 MB	16.581s	27.372s
153.4 MB	2m 44.602s	3m 53.429s
1.5 GB	33m 28.294s	26m 43.265s
15.3 GB	355m 1.318s	97m 56.008s

Fonte: Desenvolvido pelo autor

É possível visualizar 3 campos na Tabela 7: *Tamanho*, que aponta o tamanho do arquivo de texto usado para comparação; *Tempo em 1 nó*, onde é possível visualizar o tempo de execução em 1 nó fazendo uso de *pipelines*, e por fim o campo *Tempo* cluster que disponibiliza o tempo de processamento com 10 nós fazendo uso do Apache Hadoop e a biblioteca MapReduceLib.

Como apontado na seção 3.2.3, os blocos da configuração padrão do HDFS têm o volume de 128MB e três replicações. Como é visível na Tabela 7, o primeiro arquivo que contém mais de 1 bloco, 153MB, faz uso de 6 nós executando as funções de *map* e *reduce,* o que, mesmo assim, ainda não se fez mais eficiente que o uso da ordenação *sort*. Porém em arquivos maiores, que geram mais *cache* de combinação e fazem uso recursivo de múltiplos nós a execução com Apache Hadoop se faz mais eficiente.

Portanto, é possível concluir que para pequenas aplicações o uso de um cluster é desnecessário, entretanto conforme a massa de arquivos cresce o uso de um cluster acaba se tornando indispensável por 3 fatores, como listados a seguir:

1. O tempo de espera da conclusão da tarefa cresce exponencialmente quando se usa a função *sort* em *pipelines*; e
2. O computador pode ser encontrado inoperável durante a execução das tarefas;
3. O uso do cluster executa de forma mais rápida e eficaz o processamento paralelo.

Contudo, o uso de um cluster para processamento de linguagem deve ser pensado somente quando existe uma massa de arquivos a serem processados. Caso contrário o uso de um único computador pessoal pode ser usado com a garantia do processamento estar sendo executado de forma satisfatória. Uma recomendação a ser adotada é utilização de outro algoritmo de ordenação.

O próximo capítulo aborda tópicos referentes à conclusão deste trabalho e também à sugestão de estudos futuros.

CONSIDERAÇÕES FINAIS

Como demonstrado pelos resultados obtidos no estudo de caso, a utilização do cluster para processamento de linguagem mostrou-se mais eficiente mesmo com um volume baixo de dados, cerca de 1GB. O que sugere que a utilização do cluster é altamente recomendada para estudo acadêmicos e semelhantes.

Com isso, a adoção do Apache Hadoop se dá de maneira simples, como abordado no Capítulo 4, e como uma solução de baixo custo, pois faz uso de computadores comuns, independentemente de seu sistema operacional. Outra grande vantagem de se aderir ao Hadoop se dá pela comunidade ativa e grandes empresas envolvidas em seu desenvolvimento, como a Yahoo.

Considera-se que os objetivos tanto gerais, quanto específicos, foram considerar plenamente atingidos. Pois, esse trabalho mostrou a viabilidade técnica e teórica de tratamento de massas de arquivos usando o Hadoop, o que pode auxiliar interessados no assunto, servindo como base para assuntos mais complexos, caso a leitura seja efetuada por um leigo. E, finalmente, foram adquiridos conhecimentos sobre as estruturas e funcionamentos do *framework* Apache Hadoop.

Os interessados em dar continuidade a esse trabalho podem ainda seguir quatro caminhos:

- Eficácia: Os interessados em melhorar ainda mais os resultados apontados neste trabalho, podem explorar outros modelos de programação compatíveis com Hadoop. Também podem ser atribuídos ao Python os novos recursos da linguagem como Asyncio, incluída na versão 3.5 e verificar se o uso de geradores, no lugar de iteradores apontariam diferentes resultados, além da inclusão de expressões regulares.
- Linguística: Para os interessados em análise linguística, podem ser construídas novas ferramentas com o uso do modelo MapReduce.
- Metalinguagem: A análise de metalinguagem, como logs, ou linguagem de programação, pode ajudar em sistemas de segurança como respostas a incidentes detectadas com o uso do *Storm*, por exemplo. Também na área de

segurança o processamento de dados pode ajudar ao interessado em tarefas triviais que envolvem perícia forense computacional.

- *Machine Learning*: Para os interessados em processamento de linguagem e tratamento de massas de arquivos, esse trabalho pode servir como base para criação de um *bot* gerador de estruturas gramaticais. Como um gerador de sentenças, por exemplo.

REFERÊNCIAS

AMORIM, Claudio. **Uma introdução a computação paralela e distribuída**. UNICAMP: Campinas. 1988.

APACHE, YARN. Disponível em: <hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/YARN.html>. Acesso em: 19/11/2015

APACHE, Streaming. Disponível em: <hadoop.apache.org/docs/r2.6.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/HadoopStreaming.html>. Acesso em: 19/11/2015a

APACHE, Hadoop. Disponível em: <hadoop.apache.org> Acesso em: 19/11/2015b

BIBER, Douglas; CONRAD, Susan; REPPEN, Randi. **Corpus linguitics**: Investigatin Language Structure and use. Cambrige: Cambrige press, 1998.

COULORIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Sistemas distribuídos**: conceitos e projetos. Bookman: Porto alegre, 2013.

DEAN, Jeffrey GHEMAWAT, Sanjay. MapReduce: Simplified Data Processing On Large Clusters. Disponível em: <static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/mapreduce-osdi04.pdf>. Acesso em: 19/11/2015

DEROOS *et al*. **Hadoop for dumies**. Hoboken: New Jersey, 2014

DÍAZ-ZORITA, CARMEM P. Evaluación de la herramienta de código libre Apache Hadoop. Novembro 2011. 135. Dissertação. Universidad Carlos III de Madrid.

DISCO. Disponível em: <discoproject.org>. Acesso em: 19/11/2015

GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak. The Google File System. Disponível em: <static.googleusercontent.com/media/research.google.com/pt-BR//archive/gfs-sosp2003.pdf>. Acesso em: 19/11/2015

HORTONWORKS. YARN, The Architetural Center of Enterprise Hadoop, 2015. Disponível em: hortonworks.com/hadoop/yarn. Acesso em 15/11/2015.

KANGASHARJU, Jussi. Distributed System. Disponível em:

<www.cs.helsinki.fi/u/jakangas/Teaching/DistSys/DistSys-08f-1.pdf>. Acesso em 15/11/2015.

KERRISK, MICHAEL. **The Linux programming interface**: A Linux and UNIX System Programming Handbook. No starch press: San Fancisco, 2010, 1506p.

MERTZ, David; Functional Programming in Python. O'Reilly Media, Inc: Sebastopol. 2015.

MINER, Donald; SHOOK, Adam. **MapReduce Design Patterns.** O'Reilly Media, Inc: Sebastopol. 2013

MONGODB. Disponível em: <mongodb.org>. Acesso em: 19/11/2015

PALACH, Jan; **Parallel Programming with Python**. Packt Publishing Ltd: Birmingham. 2014.

RAMALHO, LUCIANO. **Python fluente**: Programação clara, concisa e eficaz. São Paulo: Novatec, 2015.

RECKEZIEL, Bruno. Disponível em: http://www.lume.ufrgs.br/bitstream/handle/10183/77306/000896370.pdf. Acesso em 15/12/2015

SARDINHA, TONY B. **Lingüística de corpus**. Manole: São Paulo, 2004, 410p.

Tanenbaum, Andrew; **Sistemas distribuídos**: Princípios e paradigmas. Pearson: São Paulo. 2007.

STORM. STORM on HDFS 2015. Disponível em http://storm.apache.org/documentation/storm-hdfs.html. Acesso em 13/11/2015

STRACK, Jair. **Sistemas de processamento distribuído**. LTC: Rio de Janeiro. 1984.

SULIVAN, Dan. Disponível em: http://www.tomsitpro.com/articles/hadoop-2-vs-1,2-718.html. Acesso em 15/12/2015

WHITE, Tom. **Hadoop**. The definitive guide. 4ª ed.O'Reilly Media, Inc: Sebastopol. 2015.

Apêndice A

Esse apêndice é destinado para a instalação e configuração dos três modos de operação do Apache Hadoop descritas no Capítulo 3.

Standalone

Antes da instalação do Hadoop existe uma única dependência para que ele funcione, o Java 7 ou maior.

Para instalar o Hadoop devemos efetuar seu Download em: *hadoop.apache.org/releases* e descompactar seu pacote usando o comando:

tar xvvf hadoop-<versão_do_pacore>.tar

Em seguida deve-se mover o Hadoop para o diretório /usr/local, uma convenção criada pela Apache:

mv hadoop-<versão_do_pacore> /usr/local/Hadoop

Após ser movido, se deve configurar a variável global do Java em nosso sistema, para que a mesmo seja acessível ao Hadoop, na linha 18 do arquivo: */usr/local/Hadoop/etc/hadoop/Hadoop-env.sh*:

export JAVA_HOME = <caminho_de_instalação_do_seu_java>

Após configurada a variável global, pode-se configurar variáveis que auxiliam no

manuseio e criação de scripts usando o Hadoop:

HADOOP_INSTALL=/usr/local/Hadoop

HADOOP_INSTALL=/usr/local/Hadoop

HADOOP_SBIN=/usr/local/Hadoop/sbin

HADOOP=/usr/local/Hadoop/bin/Hadoop

E com isso tem-se o Hadoop, no modo *standalone, instalado e configurado.*

Pseudodistributed

Para configurar o modo pseudo-distribuído deve-se configurar o modo single como vimos na sessão anterior.

Por default todas as jobs do Hadoop são executadas pelo usuário 0 (root) e se você deseja que as operações sejam executadas em um user específico é possível de ser alterado na linha *HADOOP_USERNAME /etc/conf.d/hadoop* alterando a linha da seguinte maneira:

HADOOP_USERNAME = "[seu nome de usuário]"

E em seguida temos que configurar os XMLs do Hadoop e o SSH.

Os arquivos XML estão localizados em */usr/local/hadoop/etc/hadoop.*

Logo após, juntam-se aqui arquivos básicos de configuração para iniciar e finalizar os serviços do Hadoop.

1. Arquivos de configuração:

core-site.xml:

```
<configuration>
   <property>
      <name>fs.defaultFS</name>
      <value>hdfs://localhost:9000</value>
   </property>
</configuration>
```

hdfs-site.xml:

```
<configuration>
   <property>
         <name>dfs.replication</name>
            <value>1</value>
   </property>
      <property>
         <name>dfs.namenode.name.dir</name>
            <value>file:/usr/local/hadoop/hadoop_data/hdfs/namenode</value>
   </property>
      <property>
         <name>dfs.datanode.data.dir</name>
            <value>file:/usr/local/hadoop/hadoop_store/hdfs/datanode</value>
   </property>
</configuration>
```

mapred-site.xml:

```
<configuration>
   <property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
   </property>
</configuration>
```

yarn-site.xml:

```
<configuration>
   <property>
         <name>yarn.nodemanager.aux-services</name>
            <value>mapreduce_shuffle</value>
   </property>
   <property>
         <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
            <value> org.apache.hadoop.mapred.ShuffleHandler</value>
   </property>
</configuration>
```

Configuração SSH:

```
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
ssh-keyscan -H localhost, localhost >> ~/.ssh/known_hosts
ssh-keyscan -H localhost, 0.0.0.0 >> ~/.ssh/known_hosts
```

Criação do script para iniciar os serviços:

```
$HADOOP_INSTALL/hadoop-2.6.0/sbin/start-dfs.sh
$HADOOP_INSTALL/hadoop-2.6.0/sbin/start-yarn.sh
$HADOOP_INSTALL/hadoop-2.6.0/sbin/mr-jobhistory-daemon.sh start historyserver
```

Criação do script para finalizar os serviços:

```
$HADOOP_INSTALL/hadoop-2.6.0/sbin/mr-jobhistory-daemon.sh stop historyserver
$HADOOP_INSTALL/hadoop-2.6.0/sbin/stop-yarn.sh
$HADOOP_INSTALL/hadoop-2.6.0/sbin/stop-dfs.sh
```

Teste de funcionamento: acesse http://localhost:8088

Fully Distributed

Para configurar o modo totalmente distribuído você deve configurar o modo pseudo-distribuído como vimos na página anterior.

Existem poucas diferenças entre a configuração do pseudo-distribuído e o modo completamente distribuído. Na verdade, o Hadoop só precisa estar instalado em todos os nós do nosso cluster e podemos fazer isso de uma maneira muito simples:

rsync -avxP /usr/local/hadoop root@[ip_do_cliente]:/usr/local/hadoop

Mas para que isso aconteça de uma maneira transparente temos que configurar o SSH do server, para autenticação sem senha, em todos os outros nós que farão parte do nosso cluster como visto anteriormente.

Os arquivos XML tem algumas pequenas modificações:

core-site.xml - aqui definimos o uso do HDFS e o endereço do seu Master e a porta que será usada:

```
<configuration>
   <property>
      <name>fs.defaultFS</name>
      <value>hdfs://NOME_DO_SEU_MASTER:9000</value>
   </property>
</configuration>
```

hdfs-site.xml - aqui está sendo definido o número de replicações de cada arquivo do HDFS:

```
<configuration>
   <property>
         <name>dfs.replication</name>
         <value>3</value>
   </property>
</configuration>
```

mapred-site.xml - aqui passamos a bola do gerenciador de MapReduce para o Yarn e dizemos quem vai ser o Master para distribuir tarefas:

```
<configuration>
   <property>
         <name>mapred.job.tracker</name>
         <value>NOME_DO_SEU_MASTER:5431</value>
   </property>
   <property>
         <name>mapred.framework.name</name>
         <value>yarn</value>
   </property>
</configuration>
```

yarn-site.xml:

```
<configuration>
   <property>
         <name>yarn.resourcemanager.resource-tracker.address</name>
         <value>NOME_DO_SEU_MASTER:8025</value>
      </property>
      <property>
         <name>yarn.resourcemanager.scheduler.address</name>
         <value>NOME_DO_SEU_MASTER:8035</value>
   </property>
      <property>
         <name>yarn.resourcemanager.address</name>
         <value>NOME_DO_SEU_MASTER:8050</value>
      </property>
</configuration>
```

Todos os outros nós precisam estar nomeados no seu */etc/hosts*, como por exemplo:

hadoopmaster [IP]
hadoopslave1 [IP]
hadoopslave2 [IP]
hadoopslave3 [IP]
hadoopslave4 [IP]
hadoopslave5 [IP]
hadoopslave6 [IP]
hadoopslave7 [IP]
hadoopslave8 [IP]
hadoopslave9 [IP]

Agora, dois novos arquivos precisam ser criados dentro de */usr/local/hadoop/etc/hadoop. slaves* - que são todos os computadores que farão parte do nosso cluster como "clientes":

hadoopslave1
hadoopslave2
hadoopslave3
hadoopslave4
hadoopslave5
hadoopslave6
hadoopslave7
hadoopslave8
hadoopslave9

master - que é o computador responsável pelo gerenciamento dos nós:

hadoopmaster

Teste de funcionamento: acesse http://localhost:8088

Apêndice B

Run_map_Reduce.py

```
from mapreducelib import *

hadoop("start")

def run(put,map,reduce,entrada,saida,nome):
        hdfs_rm_dir(entrada)
        hdfs_mkdir(entrada)
        hdfs_put(put, entrada)
        hdfs_rm_dir(saida)
        run_map_reduce(map,reduce,entrada,saida)
        hdfs_get("saida",nome)

run(put,map,reduce,entrada,saida,nome)

hadoop("stop")
```

map_lexico.py

```
#!/usr/bin/env python3

import sys

for linha in sys.stdin:
   linha = linha.strip()
   palavras = linha.split()

   for x,y in enumerate(palavras):
      if y.lower() == "as":
         palavras[x+1] = palavras[x+1].lower().replace("!"," !")
         palavras[x+1] = palavras[x+1].replace("?"," ? ").replace(","," , ")
         palavras[x+1] = palavras[x+1].replace(";"," ; ").replace("..."," ... ")
         palavras[x+1] = palavras[x+1].replace(":"," : ").replace("("," ( ")
         palavras[x+1] = palavras[x+1].replace(")"," ) ").replace("."," . ").replace("\""," \"
")
         print (("%s\t%s\t1")%(palavras[x].lower(),palavras[x+1].split()[0]))
      else:
         pass
```

map_contador.py

```
#!/usr/bin/env python3

from sys import stdin

for line in stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        word = word.lower().replace("!"," !")
        word = word.replace("?"," ? ").replace(","," , ")
        word = word.replace(";"," ; ").replace("..."," ... ")
        word = word.replace(":"," : ").replace("("," ( ")
        word = word.replace(")"," ) ").replace("."," . ")
        word = word.replace("\""," \" ").replace("”", " ”")
        word = word.replace("“"," “").replace("-", " -")
        word = word.replace("‘", " ‘").replace("…"," …")

        word = word.split()
        if len(word) > 1:
          for x in word:
                print (("%s\t\t%s") % (x, 1))

        else:
          print (("%s\t\t%s") % (word[0], 1))
```

reduce_lexico.py

```
#!/usr/bin/env python3
from sys import stdin

##Print de apresentação e indicação
print (("Nº |%-20s%s|%-7s|%s") % ("Frase", " " * 8, "Prob", "Cont"))
print (("%s+%s+%s+%s") % ("-"*3, "-" * 28,"-" * 7 ,"-" * 5))

#Variáveis de contagem e organização
palavra_corrente = None
contador_corrente = 0
palavra = None
contador_total = 0
dicionario = {}

#Iteração para contagem
for linha in stdin:
    contador_total += 1
    linha = linha.strip()
    palavra_1, palavra_2, contador = linha.split() #Split das palavras vinda do shuffle
```

```
para contagem

    palavra = (("%s %s") % (palavra_1, palavra_2)) #Junção das palavras, separadas
de seu contador
    contador = int(contador)

    #validação para saber se ainda estamos na mesma palavra
    if palavra_corrente == palavra:
        contador_corrente += int(contador)

    #Caso não seja é atribuída a um dicionário a palavra e seu contador
    else:
        if palavra_corrente:
            dicionario[palavra_corrente] = contador_corrente
        contador_corrente = contador
        palavra_corrente = palavra

if palavra_corrente == palavra: #Print da ultima iteração
    dicionario[palavra_corrente] = contador_corrente

#print (('%-20s|%2s') %("Total",contador_total))

#Iteração sobre o dicionário para exibição em ordem alfabética, pode ser substituído
por ordem de porcentagem
for y,x in enumerate(sorted(dicionario)):
    print(('%s  |\t%-20s\t|%5s%%\t|%5s')%(y,x,
int(dicionario[x]/contador_total*100),dicionario[x]))
```

Reduce_contador.py

```
#!/usr/bin/env python3
from sys import stdin

##Print de apresentação e indicação
print (("Nº |%-20s%s|%-7s|%s") % ("Frase", " " * 8, "Prob", "Cont"))
print (("%s+%s+%s+%s") % ("-"*3, "-" * 28,"-" * 7 ,"-" * 5))

#Variáveis de contagem e organização
palavra_corrente = None
contador_corrente = 0
palavra = None
contador_total = 0
dicionario = {}

#Iteração para contagem
for linha in stdin:
    contador_total += 1
    linha = linha.strip()
    palavra_1, palavra_2, contador = linha.split() #Split das palavras vinda do shuffle
```

```
para contagem

    palavra = (("%s %s") % (palavra_1, palavra_2)) #Junção das palavras, separadas
de seu contador
    contador = int(contador)

    #validação para saber se ainda estamos na mesma palavra
    if palavra_corrente == palavra:
        contador_corrente += int(contador)

    #Caso não seja é atribuída a um dicionário a palavra e seu contador
    else:
        if palavra_corrente:
            dicionario[palavra_corrente] = contador_corrente
        contador_corrente = contador
        palavra_corrente = palavra

if palavra_corrente == palavra: #Print da ultima iteração
    dicionario[palavra_corrente] = contador_corrente

#print (('%-20s|%2s') %("Total",contador_total))

#Iteração sobre o dicionário para exibição em ordem alfabética, pode ser substituído
por ordem de porcentagem
for y,x in enumerate(sorted(dicionario)):
    print(('%s  |\t%-20s\t|%5s%%\t|%5s')%(y,x,
int(dicionario[x]/contador_total*100),dicionario[x]))
```

exemplo_pool_map_pipe.py

```
#!/usr/bin/env python3
from multiprocessing.pool import ThreadPool as Pool
import sys

def worker(line):
    line = line.strip()
    words = line.split()
    for word in words:
        word = word.lower().replace("!"," !")
        word = word.replace("?"," ? ").replace(","," , ")
        word = word.replace(";"," ; ").replace("..."," ... ")
        word = word.replace(":"," : ").replace("("," ( ")
        word = word.replace(")"," ) ").replace("."," . ").replace("\""," \" ")

        word = word.split()
        if len(word) > 1:
            for x in word:
                print (("%s\t\t%s") % (x, 1))
        else:
            print (("%s\t\t%s") % (word[0], 1))

pool_size = 4
pool = Pool(pool_size)

for item in sys.stdin:
    pool.apply_async(worker(item), item)

pool.close()
pool.join()
```