UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ELDER LUCAS KUSS

OTIMIZAÇÃO DE DESEMPENHO DO HADOOP MAPREDUCE: UM CASO PRÁTICO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2017

ELDER LUCAS KUSS

OTIMIZAÇÃO DE DESEMPENHO DO HADOOP MAPREDUCE: UM CASO PRÁTICO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Tarcizio Alexandre Bini

PONTA GROSSA

2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

OTIMIZAÇÃO DE DESEMPENHO DO HADOOP MAPREDUCE:
UM CASO PRÁTICO

por

ELDER LUCAS KUSS

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 08 de junho de 2017 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. Tarcizio Alexandre Bini
Orientador

_____
Simone de Almeida
Membro titular

_____
Richard Duarte Ribeiro
Membro titular

_____
Prof. Dr. Ionildo José Sanches
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. Dr. Erikson Freitas de Morais
Coordenador do curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso –

RESUMO

KUSS, Elder Lucas. **Otimização de desempenho do Hadoop MapReduce: Um caso prático**. 2017. 64 páginas. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2017.

Com a popularização da Internet, quantidades massivas de dados têm sido geradas diariamente, principalmente nas mídias sociais. A crescente demanda pelo gerenciamento de grandes volumes de dados fez com que novas soluções fossem desenvolvidas. Atualmente o Hadoop é uma das soluções mais empregadas. Algumas configurações podem ser aplicadas no Hadoop para extrair um melhor desempenho. Este trabalho realiza um estudo sobre a influência dos parâmetros de configuração na performance do Hadoop *MapReduce*, utilizando para isso um *cluster* virtualizado no ambiente Docker para o desenvolvimento de testes. Os resultados obtidos nesse trabalho demonstram que é possível alcançar melhorias de desempenho no Hadoop por meio do *tuning* dos valores de seus parâmetros de configuração.

Palavras-Chave: Hadoop. MapReduce. Big Data. Otimização. Parâmetros de configuração.

ABSTRACT

KUSS, Elder Lucas. **Optimizing Performance Hadoop MapReduce: A case study**. 2017. 64 pages. Work of Conclusion Course (Graduation in Computer Science) – Federal University of Technology - Paraná. Ponta Grossa, 2017.

With the popularization of the Internet, massive amounts of data have been generated on a daily basis, especially in the social media. The growing demand for managing large volumes of data meant that new solutions were developed. Currently Hadoop is one of the solutions used. Settings can be applied in Hadoop to extract better performance. This paper carries out a study about the influence of configuration parameters on the performance of Hadoop MapReduce, and for reach that goal, uses a virtualized cluster Docker environment for testing development. The results obtained in this paper demonstrate that it is possible to achieve performance improvements in Hadoop by tuning the values of its configuration parameters.

Keywords: Hadoop. MapReduce. Big Data. Optimization. Configuration parameters.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CODEC	Codificador/Decodificador
IC	Intervalo de confiança
RPM	Rotações por Minuto
SO	Sistemas Operacionais

LISTA DE SIGLAS

API	*Application Programming Interface*
CPU	*Central Processing Unit*
HDFS	Hadoop *Distributed File System*
HMR	Hadoop *MapReduce*
JVM	*Java Virtual Machine*
MMVs	Monitor de Máquinas Virtuais
MVs	Máquinas Virtuais
SQL	*Structured Query Language*
YARN	Monitor de Máquinas Virtuais

LISTA DE ACRÔNIMOS

LXC	Linux *Container*
NOSQL	*Not Only SQL*

SUMÁRIO

1 INTRODUÇÃO

Dados são gerados diariamente por diversas fontes, incluindo transações comerciais, redes sociais e sensores. Neles existem grande potencial, podendo conter informações que auxiliem em processos decisórios (LOHR, 2012).

A *International Data Company*, corporação que realiza previsões acerca de tecnologias, sugeriu o termo “Universo Digital” para se referenciar a todos os dados criados, replicados e consumidos em um único ano. Foi estimado um aumento de 300 vezes no tamanho desse Universo, entre os anos de 2005 à 2020 passando dos 130 exabytes[1] para 40.000 exabytes, ou 40 trilhões de gigabytes (GANTZ e REINSEL, 2012).

Coletar, armazenar e processar esses dados é um desafio, visto que seu tamanho cresce de forma alarmante, se tornando cada vez mais complexo. Neste contexto, o termo *Big Data* surgiu para descrever cenários onde as quantidades de dados excedem a capacidade do modelo tradicional de gerenciamento e processamento (MADDEN, 2012).

Foram desenvolvidas diversas soluções inovadoras para o processamento e armazenamento do *Big Data*, sendo em sua maioria soluções pertencentes a classe de banco de dados não-relacionais (NoSQL) (GOLDMAN, 2012). É uma tendência o uso dessa classe de banco de dados em arquiteturas de computadores quando há a necessidade de escalabilidade, ou seja, possuir a capacidade de ampliar o poder de processamento na medida em que há aumento na demanda (MONIRUZZAMAN e HOSSAIN, 2013).

O principal paradigma de processamento de dados utilizado pelos produtos NoSQL é o *MapReduce*. Esse é um paradigma que utiliza sistema distribuído em sua arquitetura e aplica processamento paralelo sobre os dados (JEFFREY e GHEMAWAT, 2004). O Hadoop *MapReduce* é um *framework* inspirado no conceito de *MapReduce* (VENNER, 2009). É considerado uma grande referência para a manipulação do *Big Data*, utilizada por grandes companhias como IBM[2], Twitter[3], Yahoo![4], Oracle[5] e Facebook[6] para as mais diversas finalidades.

---

[1] http://pcworld.com.br/reportagens/2008/03/11/voce-sabe-o-que-e-um-hexabyte/

[2] http://www.ibm.com/analytics/us/en/technology/hadoop/

[3] https://blog.twitter.com/2010/hadoop-at-twitter

[4] https://developer.yahoo.com/hadoop/

Na computação há sempre uma busca incessante por melhorias no desempenho, e no caso de um banco de dados é possível alcançar essas melhorias fazendo refinamentos após a sua implantação. Este processo de ajuste é conhecido como *tuning* (TRAMONTINA, 2008). O Hadoop possui parâmetros de configurações nos quais seus ajustes podem prover o *tuning* do *framework*. Porém, o correto ajuste desses parâmetros não é uma tarefa simples. Exige conhecimento sobre o funcionamento interno do *framework*, saber quais parâmetros, entre centenas, são os mais sensíveis e quais são os valores que realmente irão prover um aprimoramento no cenário que está sendo utilizado (PICOLI, 2014).

1.1 OBJETIVOS

O propósito deste trabalho é realizar o *tuning* do Hadoop *MapReduce* por meio de ajustes em alguns de seus parâmetros de configurações.

1.1.1 Objetivos Específicos

Os objetivos específicos são:

- Criar um cenário para implementar o paradigma *MapReduce* por meio do Hadoop;
- Realizar o *tuning* de alguns dos parâmetros de configuração do Hadoop *MapReduce*;
- Executar a suíte de ferramenta TeraSort (O'MALLEY, 2008), embarcada no Apache Hadoop, que permite fazer *benchmark* para a análise de desempenho;
- Por fim, comparar o desempenho do sistema com diferentes valores nos parâmetros de configurações escolhidos para o *tuning*, verificando se houve redução no tempo de respostas na execução do *benchmark*.

---

[5] http://www.oracle.com/br/bigdata/overview/index.html

[6] https://www.facebook.com/notes/paul-yang/moving-an-elephant-large-scale-hadoop-data-migration-at-facebook/10150246275318920

1.2 JUSTIFICATIVA

As companhias perceberam o benefício da análise do *Big Data* para os negócios, trazendo competitividade, produtividade e inovação. Mas, a maioria das organizações não tem sido capazes de realmente gerenciar quantidades massivas de dados, sendo assim não estão aptas para gerar informação em vantagem para tomar melhores decisões (LOHR, 2012).

Para se manter à frente dos concorrentes é necessário “garimpar” informações nos dados que a companhia possui, e para isso e necessário um sistema de armazenamento e processamento eficiente, sendo que o tempo de resposta é algo imprescindível. Essa crescente necessidade de gerenciamento do *Big Data* e as limitações das tradicionais ferramentas de banco de dados fez com que as empresas investissem em inovação e assim novas soluções surgiram (MADDEN, 2012).

O Hadoop é uma das soluções utilizadas para o processamento e armazenamento do *Big Data* (DITTRICH e QUIANÉ-RUIZ, 2012). Só a implantação do *framework* Hadoop por vezes não atinge o desempenho esperado, e para contornar essa situação é necessário fazer o *tuning* dos valores nos parâmetros de configurações do *framework* que mais se adequem ao cenário no qual está sendo utilizado.

O propósito deste trabalho é apresentar os parâmetros e seu respectivos valores que são os mais sensíveis a alterações, e por fim efetuar testes para comprovar a eficácia dessas alterações. A comprovação do *tuning* é feita basicamente com a comparação do tempo de resposta da execução do *benchmark* TeraSort com diferentes valores de parâmetros.

1.3 ESTRUTURA DO TRABALHO

O restante do trabalho está estruturado da seguinte forma: O Capítulo 2 descreve a revisão bibliográfica necessária acerca do paradigma *MapReduce*, do *framework* Hadoop e da ferramenta de virtualização Docker (VITALINO e CASTRO, 2016). O Capítulo 3 apresenta a metodologia, presente na literatura, para a realização do *tuning* dos parâmetros de configurações e aborda o tema *benchmark*.

O Capítulo 4 apresenta o ambiente experimental para análise juntamente com os resultados das alterações nos valores dos parâmetros de configuração. No Capítulo 5 são apresentadas as conclusões e propostos os trabalhos futuros.

2 FUNDAMENTACÃO TEÓRICA

Este capítulo é dedicado a apresentação de conceitos que são de fundamental importância para a compreensão de assuntos que serão recorrentes ao longo do trabalho. A Seção 2.1 descreve conceitos relacionados ao paradigma de programação *MapReduce*. A Seção 2.2 descreve o framework Hadoop. A Seção 2.3 descreve conceitos fundamentais de virtualização e apresenta a ferramenta Docker. Finalmente, a Seção 2.5 descreve as considerações finais do capítulo.

2.1 MAPREDUCE

O paradigma *MapReduce* foi desenvolvido pela empresa Google Inc.[7] inspirado nas primitivas *Map* e *Reduce* presentes na linguagem LISP e em diversas outras linguagens funcionais (JEFFREY e GHEMAWAT, 2004). Na linguagem LISP a função *Map* recebe uma lista de elementos (podendo ser: letras ou números) como entrada, aplica uma função (definida pelo programador) sobre a mesma e gera uma nova lista como saída. Na função *Reduce*, novamente é recebida uma lista (podendo ser do resultado de uma função *Map*) e sobre ela é aplicada uma função (definida pelo programador) para que essa seja reduzida a um único valor (GOLDMAN, 2012).

O Quadro 1 ilustra o funcionamento das funções *Map* e *Reduce* na linguagem LISP. Para melhor exemplificar o Quadro 1 serão demonstrados os passos de execução: Primeiramente uma função *Map* denominada **metade**, como o nome sugere, realiza a divisão por 2 sobre os elementos da lista (1, 2, 3, 4). O que resulta em uma nova lista. A nova lista contém os elementos (0.5, 1, 1.5, 2), e sobre essa lista é aplicada a função *Reduce* intitulada **máximo**, a qual reduz a lista a um único elemento. Esse elemento é o de maior valor da lista, que nesse caso é **2**.

Quadro 1 - Funções Map e Reduce na linguagem LISP

Entrada	Saída
map ({1,2,3,4}, metade)	{0.5, 1, 1.5, 2}
reduce ({0.5, 1, 1.5, 2}, máximo)	2

Fonte: Autoria própria

[7] https://www.google.com.br/intl/pt-BR/about/

O paradigma demonstrou-se adequado para o processamento de grandes volumes de dados, quando estes podem ser particionados em instâncias menores (GROLINGER, 2014). Isso é devido à característica do modelo, no qual mais de uma instância das funções *Map* e *Reduce* pode ser aplicada em paralelo sobre parcelas distintas dos dados de entrada (GOLDMAN, 2012).

O *MapReduce* utiliza o modelo chave/valor para abstrair os dados de entrada para a aplicação das funções *Map* e *Reduce* (JEFFREY E GHEMAWAT, 2004). Esse modelo é composto por um conjunto de chaves, aos quais são associados um único valor. As chaves podem representar campos como nome, ano, palavra, entre outros. Enquanto que, o valor associado a essa chave representa a instância do campo.

Para o usuário abstrair os dados do seu problema e submetê-lo ao paradigma *MapReduce*, é necessário que ele especifique a função *Map* que irá processar os pares chave/valor e gerar os novos pares intermediário de chave/valor. Também é preciso especificar uma função *Reduce* para associar e unificar todas as saídas das funções *Map* onde os valores do conjunto possuem a mesma chave (GOLDMAN, 2012).

A Tabela 1 representa um modelo no qual foram abstraídos os dados no modelo chave/valor de um arquivo de texto contendo vários dados climatológicos de uma região. Para esse modelo foi escolhido a chave representando o ano, e o valor representado a temperatura máxima atingida naquele ano.

Tabela 1 - Exemplo do modelo Chave/Valor

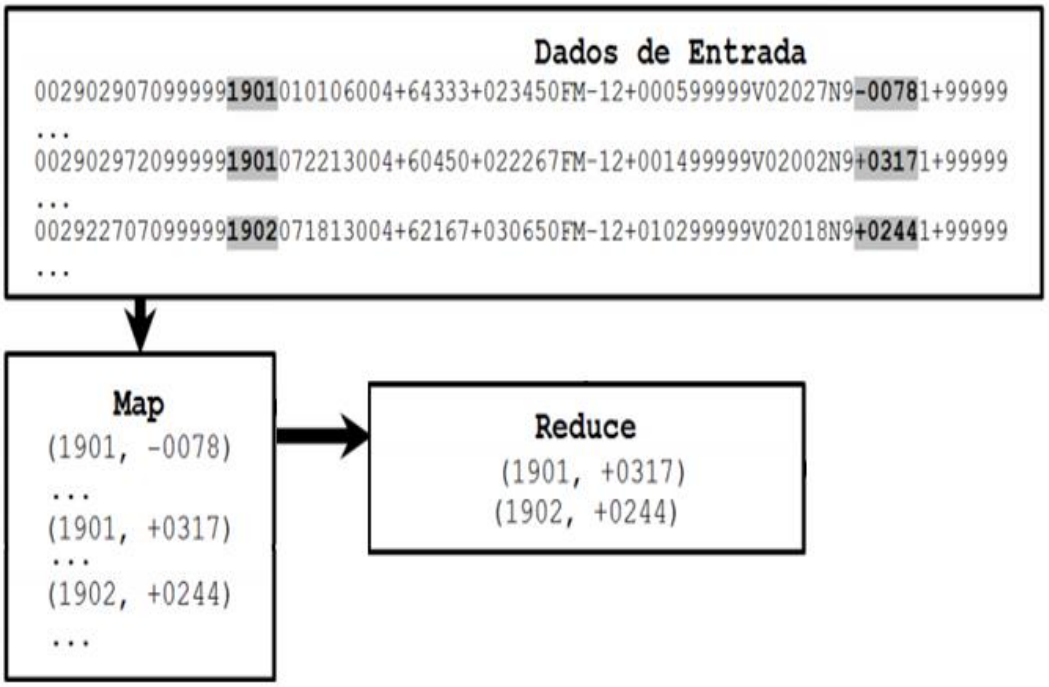
Chave	**Valor**
Ano	**Temperatura Máxima**
1991	31,7
1992	24,4
1993	26,1
1994	32,6

Fonte: Autoria própria

A Figura 1 (pág. 18) ilustra o processo para a geração da Tabela 1 por meio do paradigma *MapReduce.* Primeiramente uma função *Map* processou os dados climatológicos hipotéticos e gerou um conjunto de pares intermediários chave/valor contendo todas as temperaturas registradas nos diferentes anos. Uma função

Reduce abastecida pela saída da função *Map*, processou os dados e como resultado a cada ano foi exibida a temperatura máxima.

Figura 1 - Processo MapReduce



Fonte: Adaptado de GOLDMAN (2012)

O *MapReduce* é normalmente executado em *clusters* ou *grids*, que por meio de um conjunto de computadores comuns, conseguem agregar um alto poder de processamento a um custo relativamente baixo (GOLDMAN, 2012). Este é um modelo de alta escalabilidade, já que é possível aumentar o poder de processamento e armazenamento adicionando máquinas ao conjunto existente.

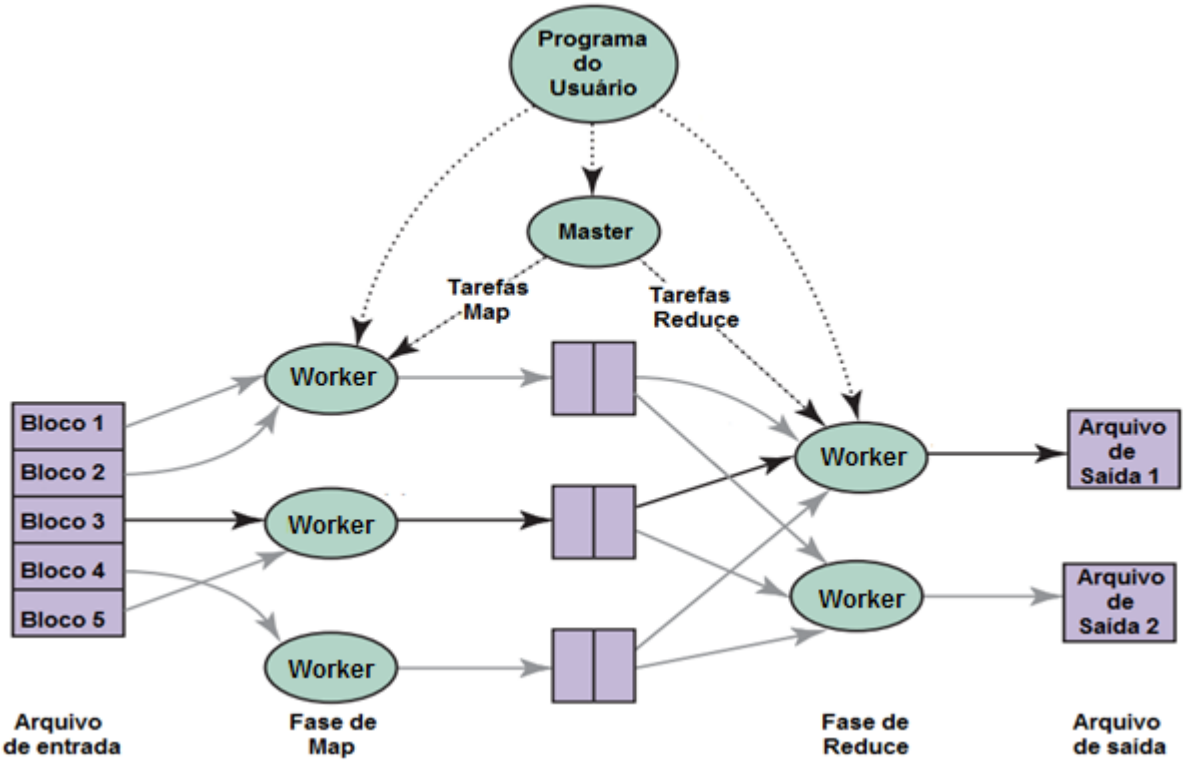
Segundo Jeffrey e Ghemawat (2004) a visão geral da execução do *MapReduce* sobre um *cluster*, demonstrada na Figura 2 (pág. 19), consiste nos seguintes passos:

- Primeiramente os dados de entrada são particionados em blocos de tamanho pré-definidos e várias cópias do programa *MapReduce* são feitas e distribuídas entre as máquinas do *cluster*. Uma dessas cópias é denominada *Master* e as demais são denominadas *Workers* ou *Slaves*. As máquinas do *cluster* são comumente referenciadas pelo nome da cópia do programa *MapReduce* que recebeu, ou seja, *Master* ou *Worker*.
- A máquina que recebe a cópia *Master* tem a responsabilidade de distribuir *Tasks Map* ou *Reduce* para as múltiplas máquinas ociosas

que receberam cópias *Workers*. Quando um *Worker* recebe uma *Tasks Map*, juntamente ele recebe um bloco de dados da entrada.

- Quando um *Worker* é notificado sobre uma *Tasks Map* prontamente ele interpreta os dados do bloco no modelo chave/valor e passa para a função *Map* definida pelo usuário, que processa as chaves/valores gerando um conjunto de chave/valor intermediário. Quando o processo da *Tasks Map* chega ao fim, o *Master* é informado da localização e da saída para repassar aos *Workers* que forem realizar as *Tasks Reduce*.
- Quando um *Worker* recebe uma *Tasks Reduce*, recebe também a informação da localização dos dados com os conjuntos de chaves/valores intermediários, e então efetua a leitura dos pares intermediários. Em seguida ordena, e agrupa todas as ocorrências que possuem as mesmas chaves. A execução do programa se encerra quando todas as *Tasks Map* e *Reduce* forem finalizadas.

Figura 2 - Modelo MapReduce



Fonte: Adaptado de JEFFREY e GHEMAWAT (2004)

Existem algumas implementações derivadas do paradigma *MapReduce*, entre elas ganha grande destaque o Hadoop *MapReduce* (KATAL *et al*, 2013). O Hadoop é uma plataforma de *software* de código fonte aberto, atualmente mantido pela comunidade de desenvolvedores da fundação Apache (WHITE, 2015).

2.2 HADOOP

O Hadoop é um *framework* desenvolvido na linguagem Java, que permite utilizar o modelo de programação *MapReduce*. Ele disponibiliza APIs (*Application Programming Interface*) para que os usuários possam escrever as funções *Map* e *Reduce*, nas linguagens Java, C++, Python, Ruby, entre outras, conforme a necessidade do problema (WHITE, 2015).

O Hadoop possui a mesma arquitetura *Master/Worker* utilizada pelo paradigma *MapReduce*. Permite também que programadores sem experiência com sistemas paralelos e distribuídos o utilizem sem grandes dificuldades, pois oculta toda a complexidade do sistema. O particionamento dos dados de entrada, agendamento das execuções no conjunto de máquinas, manipulação das falhas e das comunicações entre as máquinas são tratados pelo próprio sistema (GOLDMAN, 2012).

2.2.1 Arquitetura Geral do Hadoop

A estrutura principal do Hadoop é subdividida em quatro principais módulos, sendo: o Hadoop *Common*, o Hadoop *Distributed FileSystem* (HDFS), o Hadoop *MapReduce* (HMR) e o Hadoop *Yet Another Resource Negotiator* (YARN). A partir da versão 2.0, o Hadoop tornou-se mais flexível permitindo a execução de aplicações não HMR, como Spark, Storm, entre outros (VAVILAPALLI, 2013).

As máquinas que operam sobre sistemas distribuídos são comumente chamadas de nós. No Hadoop o nó *Master* é o responsável pela coordenação dos nós *Workers.* É ele que mantém os metadados de referência para todos os dados distribuídos no *cluster*, recupera o sistema em caso de falhas e faz o gerenciamento dos recursos computacionais. Nós *Workers* tem como propósito o armazenamento e

processamentos dos dados, além de frequentemente informar ao nó *Master* sobre seu estado atual (processando ou ocioso) (WHITE, 2015).

Nesta seção serão abordados os módulos do Hadoop *MapReduce*. A apresentação das funções desses módulos faz-se necessária pois permite com mais clareza compreender a qual módulo pertence cada configuração de parâmetro tratada no Capítulo 3.

2.2.1.1 Hadoop *common*

É no Hadoop *Common* que se encontram todas as bibliotecas e arquivos comuns e necessários para os outros módulos do Hadoop (HMR, HDFS e YARN). Bibliotecas para serialização de dados, chamada remota de métodos, manipulação de arquivo, entre outras, ficam neste subprojeto (KOCAKULAK E TEMIZEL, 2011).

2.2.1.2 Hadoop d*istributed file system*

Quando o conjunto dos dados supera a capacidade de armazenamento de uma única máquina, se torna necessário particionar esses dados ao longo de um conjunto de máquinas. O sistema que gerencia o armazenamento distribuído é denominado d*istributed file system* (COULOURIS, 2005).

O Hadoop utiliza um sistema de arquivos global distribuído chamado Hadoop *Distributed File System* ou HDFS. O HDFS é otimizado para atuar em dados estruturados e não estruturados de forma distribuída, e foi baseado no Google *File System* (GHEMAWAT et al, 2003). O HDFS possui um conjunto de funcionalidades que merecem destaque como: armazenamento, organização, nomeação, recuperação, compartilhamento, proteção e permissão de acesso aos arquivos (WHITE, 2015). Os arquivos no HDFS são divididos em blocos de tamanho fixo padrão (128 Megabytes), valor este que pode ser alterado.

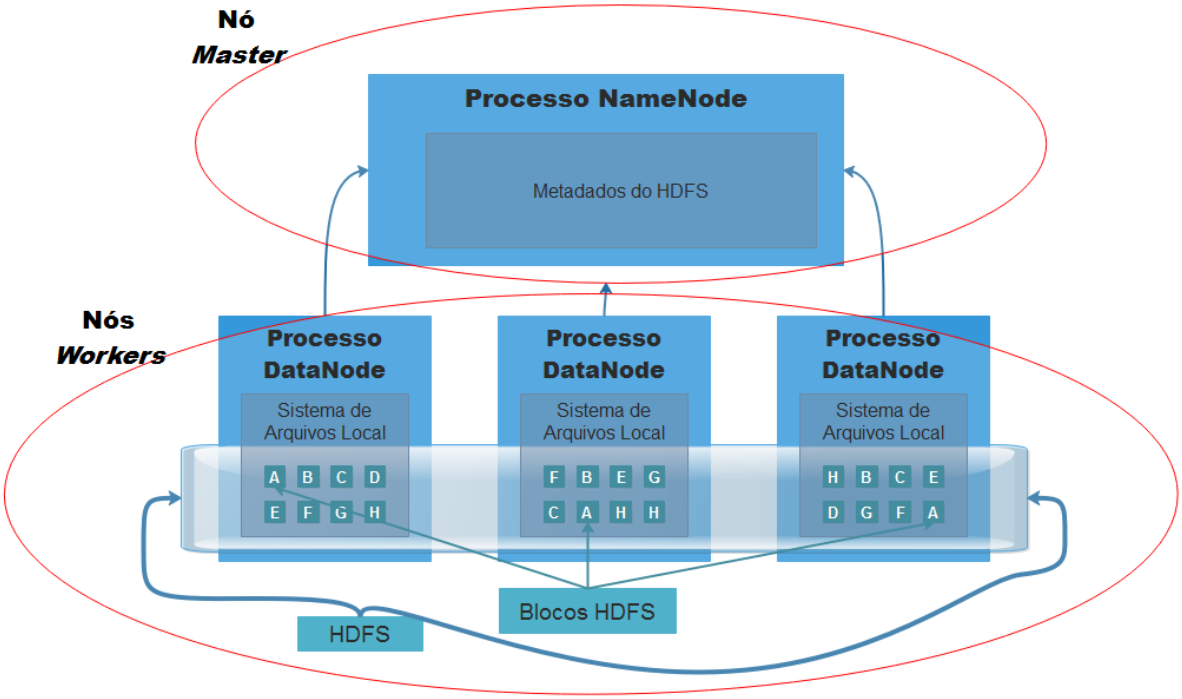
O HDFS é o responsável pela persistência e consistência dos dados, possui também transparência à falhas de rede e replicação de dados. Para assegurar que os dados não estejam corrompidos e nenhuma falha de máquina e disco interrompa o processamento, cada bloco do HDFS é replicado em um pequeno número de máquinas físicas (tipicamente três). Caso algum bloco não esteja acessível devido a

existência de algum problema, uma cópia pode ser lida de outra localização de modo totalmente transparente ao usuário (WHITE, 2015).

O nó *Master* do Hadoop precisa saber quais dados estão em cada nó *Worker*, para assim atribuir *Tasks* aos nós de maneira a evitar ao máximo a ocorrência de tráfego de rede. O HDFS precisa de dois tipos de processos para o seu funcionamento, sendo: um *Namenode* contido no nó *Master* e para os nós *Workers* o *Datanode* (GOLDMAN, 2012).

Os *Datanodes* armazenam e recuperam blocos de dados, e enviam a lista de blocos que estão armazenados para o *Namenode*, atualizando a informação periodicamente. O *Namenode* mantém *metadados* para todos os arquivos e diretórios do sistema, sabendo assim a localização de um determinado arquivo em meio a todos os *Datanodes* (GOLDMAN, 2012). A Figura 3 mostra em alto nível uma visão geral da estrutura do HDFS.

Figura 3 - Arquitetura do HDFS



Fonte: Autoria própria

O HDFS possui um ponto crítico de falha, no caso do nó *Master* que execute o processo NameNode falhar. Porém existem mecanismos de recuperação caso isso ocorra (WHITE, 2015). Um desses mecanismo é a utilização de um *NameNode* que fica em *stand-by,* que entra em funcionamento em caso de falha no *NomeNode*

ativo. Ainda havendo um erro no *NameNode* em *stand-by* é possível utilizar o *Namenode* secundário, que entra em funcionamento através de uma cópia dos metadados do *Namenode* principal.

2.2.1.3 Hadoop *MapReduce*

O Hadoop *MapReduce* ou HMR é responsável pelo processamento dos dados armazenados no HDFS ou em outro sistema de armazenamento. A execução completa de um programa HMR é chamado de *Job* e a divisão de um *Job* entre os nós *Workers* é denominado *Task.*

O gerenciador de recursos YARN (que será explicado adiante) no Hadoop utiliza a definição de *container* para a alocação de recursos. Um *container* é uma parcela de recursos (CPU, memória, etc) de uma máquina do *cluster* que pode ser reservada para a execução de uma aplicação, sendo esta aplicação HMR ou não.

Tom White (2015) cita que em alto nível, para a execução de um *Job* no HMR existem 5 entidades independentes:

- **O cliente:** submete o *Job MapReduce* a ser executado;
- **O YARN *Resource Manager*:** coordena a alocação dos recursos computacionais no *cluster*;
- **O YARN *Node Manager*:** carrega e monitora os *containers* nas máquinas do *cluster*;
- **O *MapReduce Aplication Master*:** coordena as *Tasks* que executam os *Jobs MapReduce*;
- **O HDFS:** compartilha os arquivos entre os nós do *cluster*.

Abaixo é feita uma explicação para melhor compreender os locais e processos das entidades YARN, *Aplication Master* e HDFS no cluster (WHITE, 2015).

O Nó Master possui dois componentes sendo eles:

- ***Resource Manager Yarn:*** Um por *cluster*, e é dividido em dois componentes:

  Schedular: Responsável por fornecer os recursos requisitados pelo *Applications Master*, e acompanhar a execução dessa aplicação;

Application Manager: Responsável por manter uma coleção de submissões de aplicações. Após a submissão de uma aplicação primeiro valida-se suas especificações e rejeita qualquer aplicação que solicita recursos insatisfatível (isto é, não há nenhum nó no *cluster* que tenha recursos suficientes para executar o *Application Master*). Em seguida, ele garante que nenhuma outra aplicação já tenha sido apresentada com a mesma identificação.

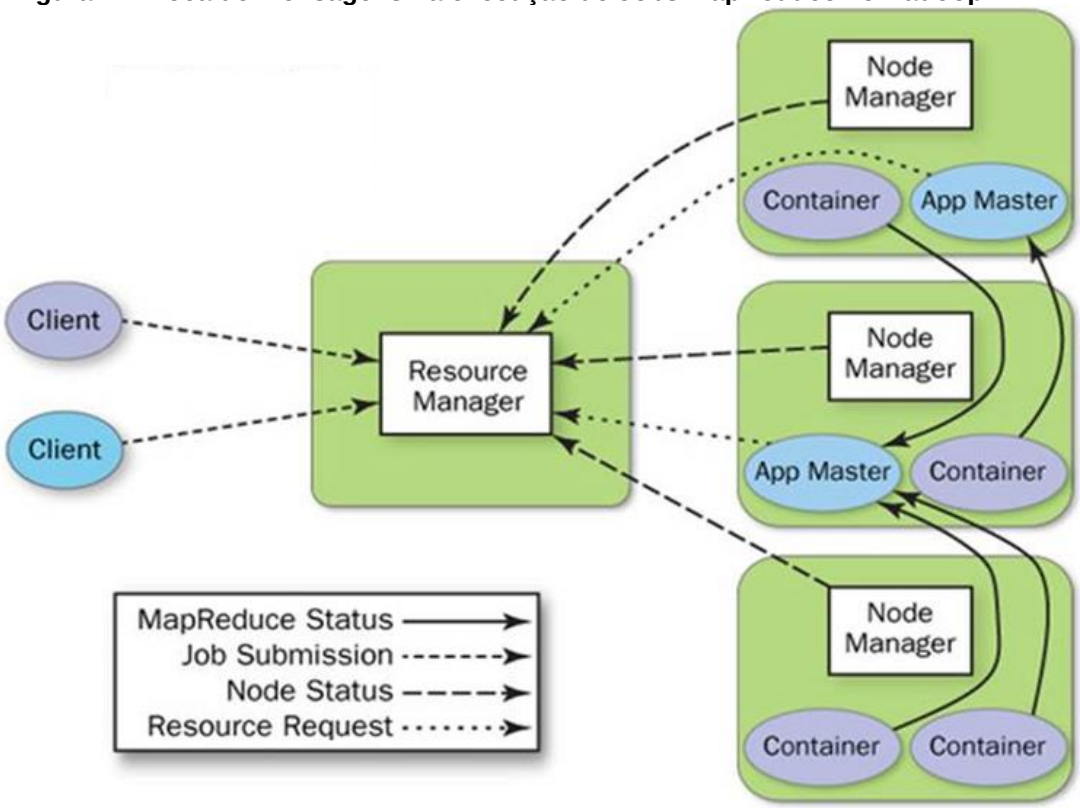
- ***NameNode:*** Um por *cluster*. Contêm metadados de todos os arquivos e diretórios do sistema.

A partir da versão 2.0 do Hadoop a coordenação de um *Job* não ocorre no nó *Master*, mas sim em um *container* instanciado em uma máquina *Worker*. Uma máquina *Worker* possui geralmente dois componentes, podendo possuir três no caso da instância de gerenciamento da aplicação. Esses componentes são:

- ***Node Manager***: Um por *Worker*. Responsável por gerenciar o ciclo de vida de um *container* e monitorar a utilização de cada recurso do *container*;
- ***Application Master:*** Um por aplicação. Responsável pelo gerenciamento do ciclo de vida da aplicação, pelas requisições de recursos ao *schedular*, e pela execução e acompanhamento das *Tasks* através da comunicação com o *Node Manager*;
- ***Data Node:*** Um por *Worker*. Responsável por armazenar e recuperar blocos de dados armazenados localmente.

A Figura 4 (pág. 25) exemplifica a troca de mensagens entre os módulos e nós do *cluster* ao executar dois *Jobs*. Os *containers* que executam as *Tasks* reportam seu *status* ao *Application Master* que por sua vez reporta ao *Resource Manager*. O *Application Master* faz as requisições de *container* para a execução das *Tasks* ao *Resource Manager.* O *NameNode* por sua vez recebe periodicamente os blocos armazenados em cada *DataNode* (Kobylinska e Martins, 2014).

Figura 4 - Troca de mensagens na execução de Jobs MapReduce no Hadoop



Fonte: Kobylinska e Martins (2014)

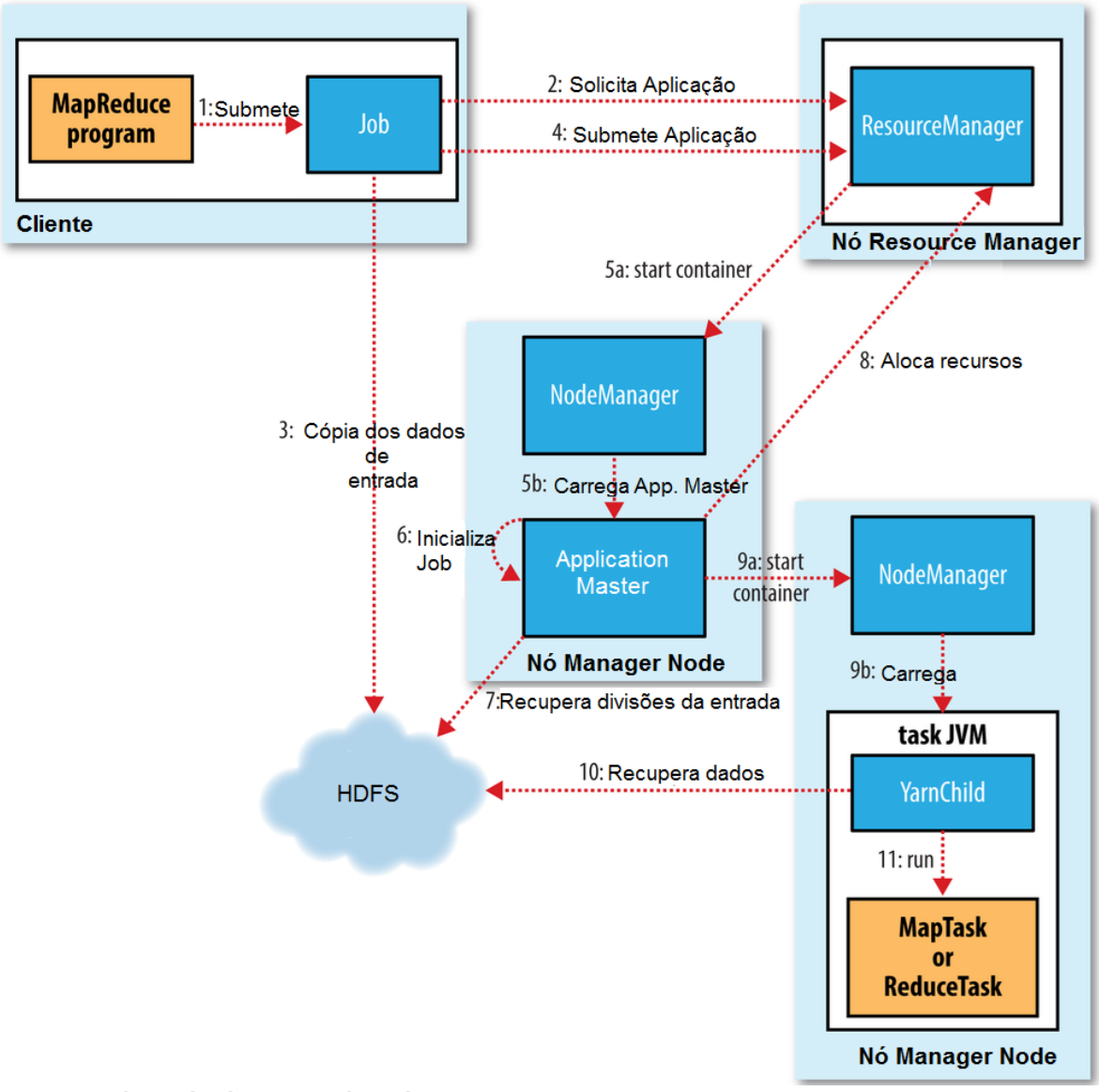
A Figura 5 (pág. 26) ilustra em alto nível de abstração o processo de execução de um *Job MapReduce* no Hadoop. Segundo White (2015) as etapas são:

- O cliente submete um *Job (**passo 1**);*
- É solicitado ao *Resource Manager* um identificador para essa novo Job (**passo 2**).
- Na sequência os dados de entrada desse novo *Job* são particionados em blocos, e copiados para o HDFS (**passo 3**).
- Após essa etapa, o *Job MapReduce* entra em execução (**passo 4**).
- *O schedular* YARN aloca um *container* com a *Application Master*, e então o *Resource Manager* executa o processamento da aplicação *Master* nesse container (**passo 5**).
- *Application Master MapReduce* é que inicializa o processamento do *Job* (**passo 6**).
- Em seguida são recuperados os blocos da entrada do HDFS (**passo 7**).
- Para cada bloco da entrada é criado uma *Task Map*, e o número de *Tasks Reduce* é determinado pela propriedade **mapreduce.Job.reduces.** *Application Master* é quem decide como executar o *Job*. Caso o *Job* seja

pequeno ele pode ser executado localmente, do contrário o *Application Master* requisita *containers* para todas as *Task Map* e *Reduce* ao *Resource Manager* (**passo 8**).

- Uma vez que o *Resource Manager* tenha atribuído um container *Task* que o *Application Master* solicitou, o *Applicantion Master* inicializa o container se comunicando com o *NodeManager* (**passo 9**).
- Antes de executar a *Task*, são localizados os recursos que ela necessita (**passo 10**).
- Por fim, a *Task* é executada (**passo 11**).

Figura 5 - Detalhes da execução de um *Job MapReduce* no Hadoop



Fonte: Adaptado de WHITE (2015)

Para processar uma *Task* é criada uma nova instância da Máquina Virtual Java (JVM) denominada *YarnChild*. O *YarnChild* é uma JVM dedicada a *Task*, evitando assim que qualquer *bug* oriundo de funções *Map* e *Reduce* mal definidas afete o *NodeManager* (WHITE, 2015).

Quando a *Application Master* recebe a notificação que a última *Task* do *Job* está completa, seu *status* é alterado para "*successful*". Então, o *JobClient* exibe a mensagem para o usuário informando que o processo foi finalizado com sucesso.

2.2.1.4 Hadoop YARN

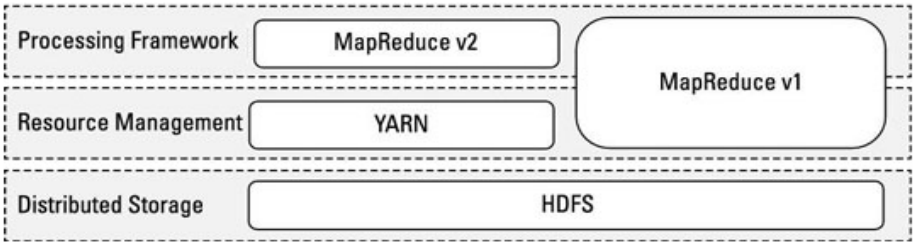
Em versões anteriores à adoção do YARN, uma entidade denominada *JobTracker* contida no nó *Master* era responsável pela reserva e monitoramento de *slots* para a execução das *Tasks*. Era também responsabilidade do *JobTracker* "limpar" os recursos temporários e tornar os *slots* reservados disponíveis para outras *Tasks*. O fato de existir somente uma instância *Jobtracker* no *cluster* limitava a escalabilidade e levava ao problema da execução *MapReduce* falhar, caso houvesse interrupção do *JobTracker*.

A ideia de pré-definir os *slots Map* e *Reduce* causava problemas de recursos, como no caso de todos os *slots Map* estarem ocupados enquanto ainda existiam *slots Reduce* disponíveis e vice-versa. Enfim, não era possível usar a infraestrutura do Hadoop para outro tipo de processamento.

Com a adoção do YARN no Hadoop como gerenciador de recursos, não mais utiliza-se a definição de *slot* para gerenciar recursos. Ao invés disso, tem-se agora recursos computacionais que podem ser alocados pela aplicação. Dessa maneira aplicações *MapReduce* podem ser executadas juntamente com aplicações não *MapReduces* (WHITE, 2015).

Na Figura 6 (pág. 28) é possível visualizar que anteriormente no Hadoop versão 1.0 toda a estrutura de gerenciamento de recursos era acoplada ao módulo Hadoop *MapReduce* o que restringia a plataforma a um único modelo de programação. Mas, com a adoção do YARN outros *frameworks* de processamento podem fazer uso da infraestrutura do Hadoop já que são executados em uma camada superior e independente (KOBYLINSKA e MARTINS, 2014).

Figura 6 - Hadoop versões 1.0 e 2.0



Fonte: Adaptado de Kobylinska e Martins (2014)

2.2.2 Exemplificando as Funções Map e Reduce

A demonstração mais clássica do funcionamento do paradigma *MapReduce* é o programa que realiza a contagem de palavras (*WordCount*), o qual é distribuído como parte do pacote de exemplos no Hadoop. O *WordCount* possui como entrada um conjunto de arquivos de texto a partir dos quais a frequência de registros da palavra é contada, e para a saída é gerado um arquivo de texto contendo a quantidade de vezes que cada palavra foi encontrada nos arquivos.

No exemplo demonstrado no Quadro 2, a função *Map* recebe como chave o nome do documento e um valor contendo o conteúdo do documento. Para cada palavra presente no documento emite-se um par intermediário, onde a chave representa a palavra e o valor representa a frequência de aparições dela. A função *Reduce* recebe como parâmetro as saídas das funções *Map*, sendo a chave (palavra) e o valor (frequência). Na sequência, a variável ***result*** recebe o somatório de valores da lista ***values*** e então emite o valor total de ocorrência da palavra recebida.

Quadro 2 – Funções Map e Reduce do programa WordCount

map(String key, String value):
// key: nome do documento
// value: conteúdo do documento
for each word w in value:
  EmitIntermediate(w, 1);
	reduce(String key, String value):
	// key: palavra - value: lista de quantidade
	int result = 0;
	for each v in value:
	  result += ParseInt(v);
	Emit(AsString(result));

Fonte: Picoli (2014)

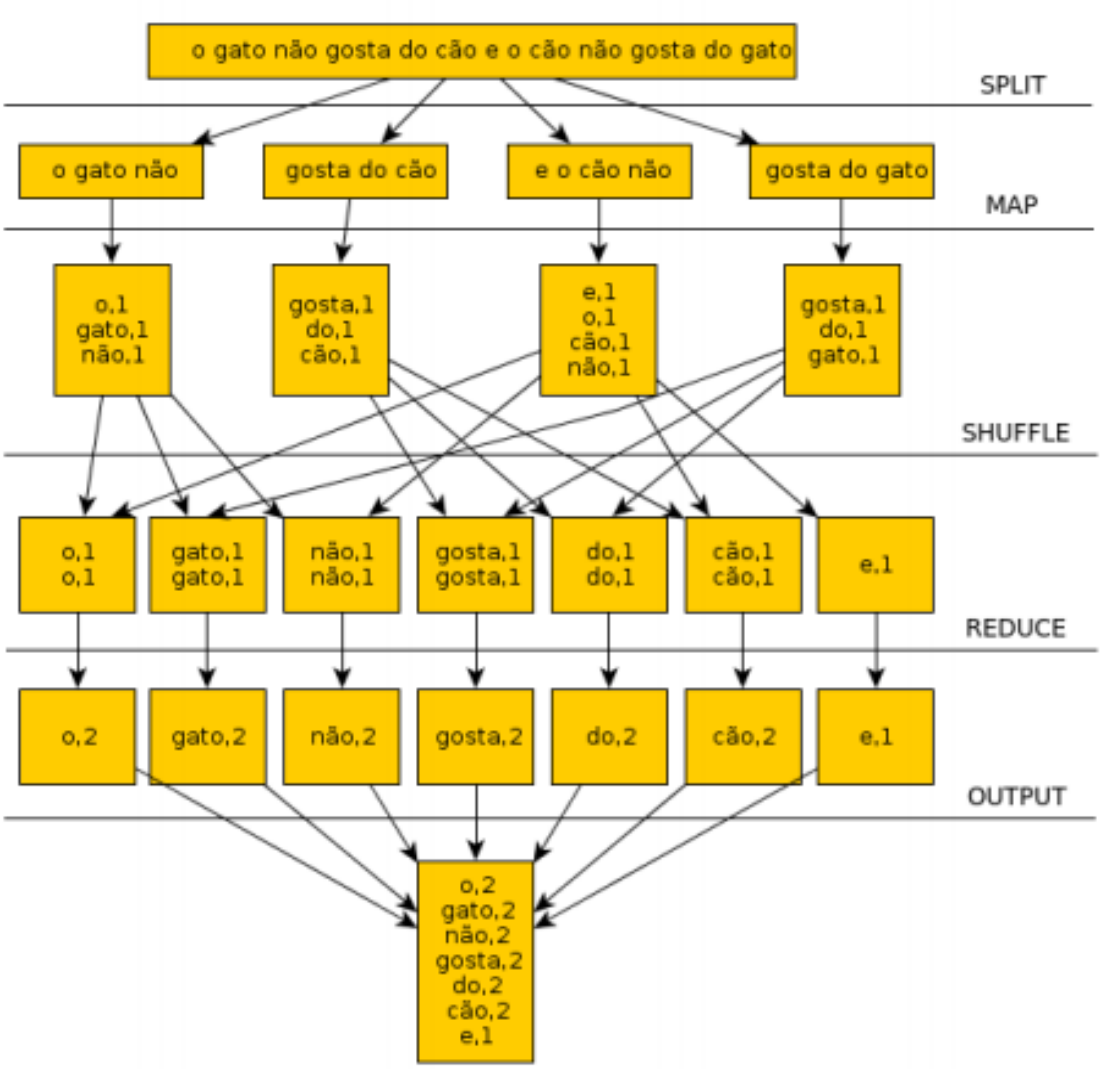
Para essa demonstração, utiliza-se o arquivo de texto contendo a frase:

"O gato não gosta do cão e o cão não gosta do gato"

Segundo PICOLI (2014) o processo de execução do *Job WordCount*, demonstrado na Figura 7 (pág. 30), é composto pelas seguintes fases:

- ***Split*:** Primeiramente seleciona-se os arquivos necessário no HDFS e divide-os em tamanho fixos.
- ***Map*:** O nó contendo o *Aplication Master* é incumbido de criar os *containers* para que executem as *Tasks Map.* Esses *containers* processam as *Tasks e* geram pares intermediários de chave/valor, sendo nesse caso palavra/frequência.
- ***Shuffle*:** Antes de repassar as saídas *Map* para que a *Application Master* são criados os *containers* que irão executar as funções *Reduces*, é feito um agrupamento pela chave. Esse agrupamento tem como propósito facilitar o trabalho na execução da função *Reduce*.
- ***Reduce*:** Os *containers* responsáveis pela execução das funções *Reduce* buscam por chaves iguais nos pares intermediários, ou seja, a ocorrência das mesmas palavras, e quando ela ocorre é efetuado a soma dos valores armazenados.
- ***Output*:** Após a execução de todas as *Tasks,* o *framework* salva no HDFS um arquivo contendo a frequência das palavras.

Figura 7 – Exemplo de um fluxo de Informações de um programa MapReduce



Fonte: Picoli (2014)

2.2.3 Verificação do Andamento dos *Jobs* e *Tasks*

O andamento e histórico das execuções dos *Jobs* podem ser consultados no *framework* Hadoop. Os mecanismos de consultas são: O terminal de execução do Hadoop e a *interface Web* do *framework* (WHITE, 2015).

São disponibilizadas algumas *interfaces Web* para a consulta do andamento da execução de um *Job*, sendo elas:

- A *interface Resource Manager* que contém informações referentes aos *Jobs* em execução, completos e os que falharam;
- A *interface Namenode* que contém informações referentes ao HDFS;

- E a *interface MapReduce JobHistory Server* onde podem ser observadas informações sobre os *Jobs* já finalizados.

Por meio de um terminal também é possível visualizar os passos de execução de um *Job MapReduce*. O andamento das *Tasks Map* e *Reduce*, a ocorrência de erros, os resultados intermediários gerados e outras informações podem ser analisadas pelo terminal. Todas as informações geradas durante a execução de um *Job* são gravadas em arquivos de registros, que podem ser acessados posteriormente. O Quadro 3 apresenta os endereços das *interfaces web* do Hadoop (WHITE, 2015).

Quadro 3 – *Web UI* Hadoop.

Daemon	**Saída**
NameNode	http://host:50070
ResourceManager	http://host:8088
MapReduce JobHistory Server	http://host:19888

Fonte: Autoria própria

2.3 VIRTUALIZAÇÃO

Em meados de 1960 percebeu-se que para otimizar as tarefas computacionais era necessário executar processos em paralelo. Foi essa ideia de compartilhamento que culminou no advento da virtualização (BOSING e KAUFMANN, 2012). Com esta solução os equipamentos computacionais, que eram de alto custo, ficaram mais acessíveis e trouxeram um melhor retorno de investimento ao usuário, que poderia compartilhar um computador e mesmo assim utilizar a sua totalidade de recursos (*apud* OLIVEIRA, 2015).

A virtualização permitiu a um computador desempenhar o papel de muitos. Cada um desses pseudo-computadores são chamados de máquinas virtuais (MVs). Uma MV abstrai as características de *hardware* e *software* da máquina física hospedeira (*host system*) que está sendo executada, tornando-se assim um ambiente operacional autossuficiente (CARISSIMI, 2008).

O Monitor de Máquinas Virtuais (MMVs), é um componente de *software,* que permite o suporte e gestão das MVs no hospedeiro. Entre suas principais tarefas estão: orquestração de acessos aos recursos da máquina hospedeira, criação,

escalonamento, isolamento e gestão das Máquinas Virtuais criadas (OLIVEIRA, 2015).

2.3.1 Arquiteturas de Virtualização

Existem algumas abordagens de virtualização que se diferenciam principalmente no nível de abstração e nos métodos utilizados para implementação (SMITH e NAIR, 2005). Dentre as principais arquiteturas estão:

Virtualização Total: Provê uma réplica virtual do sistema subjacente, e por conta disso algumas vezes é interpretado como sendo uma emulação de *hardware* (OLIVEIRA, 2015). Como toda a estrutura de *hardware* é virtualizada, o processo pode levar a perdas significantes de desempenho (*apud* BINI, 2014).

Paravirtualização: Contornando a inconveniência da virtualização total essa arquitetura permite, com restrições administradas pelo MMVs, acessar diretamente os recursos de *hardware* da máquina hospedeira (BINI, 2014).

Virtualização Assistida por *Hardware*: Utiliza suporte de *hardware* para virtualização diretamente no processador, por meio das tecnologias de virtualização da Intel (VT-x) e AMD (AMD-v) (FISHER-OGDEN, 2006). Remove a necessidade de tradução binária ou paravirtualização realizando acesso diretamente ao *hardware* sem a emulação de um processador.

Virtualização em nível de Sistema Operacional: De forma a isolar múltiplas áreas de usuário, intituladas *containers*, o sistema operacional (SO) *host* é modificado (HELSLEY, 2009). Para isso, é necessária uma camada de *software* de virtualização instalada no sistema operacional anfitrião, que permita o compartilhamento do *kernel* (núcleo do SO). Ao contrário da virtualização total e da paravirtualização essa arquitetura não depende de uma MMVs. A principal desvantagem dessa abordagem é não permitir o uso nas MVs de SOs com *kernels* diferente do instalado na máquina hospedeira.

2.3.1.1 Linux *container* e Docker

O Linux *Container* (LXC) é um dos mais expressivos representantes da virtualização em nível de Sistema Operacional. Os *containers* no LXC são

gerenciados pela ferramenta *lxc-tools (*HELSLEY, 2009*).* Essa ferramenta permite criar, congelar ou destruir *containers*.

Containers compartilham o *kernel* do SO da máquina *host,* o que proporciona melhor desempenho se comparado a MVs, devido ao gerenciamento único dos recursos (BOETTIGER, 2015). Máquinas virtuais emulam um novo sistema operacional e todo o seu *hardware*, desse modo o processo precisa de mais recursos da máquina *host* (MERKEL, 2014). Com a utilização de *containers* na substituição de máquinas virtuais é possível executar mais ambientes virtuais com a mesma quantidade de recursos.

Docker é um projeto para automatizar a implantação de aplicativos dentro de *containers*. O Docker torna as aplicações portáteis e isoladas por meio do empacotamento em *containers* baseados na tecnologia LXC (MERKEL, 2014). *Container* não é uma tecnologia nova, pelo contrário possui mais de uma década de vida, mas o Docker trouxe maior destaque nos últimos anos para essa tecnologia (VITALINO e CASTRO, 2016).

2.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Foram apresentados neste capítulo todos os conceitos considerados fundamentais para a compreensão deste trabalho. Foram introduzidos o conceito de programação do paradigma *MapReduce*, os módulos e entidades do *framework* Hadoop e os tipos de virtualização. Por fim, foi apresentado a ferramenta Docker que utiliza virtualização em nível de SO, e é empregada neste trabalho. No Capítulo 3 a seguir é abordado os parâmetros de configuração do Hadoop para a realização do *tuning*.

3 OTIMIZAÇÃO E ANÁLISE DE DESEMPENHO DO HADOOP

O Apache Hadoop é executado em pequenos, médios e grandes *clusters,* para as mais diversas finalidades. O Hadoop é aplicado na: otimização de busca (LEWIS, 2012), aprendizado de máquinas (CHU, 2007) e mineração de dados (MENEZES *et al*., 2016). Sua utilização já ocorre em áreas como: meteorologia (XUE e PAN, 2012), biologia (RECKZIEGEL, 2013) e finanças (ZHANG *et al*., 2011).

Existe uma grande gama de informações disponíveis nos mais diversos meios de comunicação abordando assuntos relacionados ao Hadoop. É possível encontrar vários tutoriais explicando o funcionamento e o processo de instalação do *framework* (WHITE, 2015). O mesmo não ocorre quando o assunto é otimização de seu desempenho. Poucas informações são encontradas na literatura, principalmente em português.

O Capítulo 3 apresenta os parâmetros de configurações considerados mais relevantes segundo a literatura (WHITE, 2015), e fornece sugestões para ajustá-los. Também apresenta o *benchmark* utilizado no escopo desse trabalho.

3.1 PARÂMETROS DE CONFIGURAÇÃO DO HADOOP

O Hadoop possui mais de 200 parâmetros de configurações (*apud* PICOLLI, 2014). As alterações nos valores desses parâmetros influenciam no desempenho do *framework*.

As seções subsequentes apresentam os parâmetros de configurações mais relevantes para o escopo desse trabalho, e fornece sugestões para ajustá-los de maneira a prover ganhos de desempenho. Também são realizadas explanações sobre os processos internos do *framework* para se compreender o motivo das configurações serem feitas de tal maneira.

3.1.1 O Processo *Shuffle*

O processo no qual o sistema ordena e transfere as saídas *Map* para as entradas *Reduce* é conhecido como *shuffle* (embaralhamento). Entender como o

processo *shuffle* funciona pode ajudar a otimizar programas *MapReduce*, pois esse processo é considerado o coração do HMR (WHITE, 2015).

O processo *shuffle* é mais complexo do que aparenta ser. Para compreender melhor seu papel no Hadoop, as Seções 3.1.2 e 3.1.3, embasadas no livro de White (2015), apresentam uma visão desde o início da produção das saídas *Map* até o ponto em que a funções *Reduce* são alimentadas.

3.1.2 O Processo Map

Quando as saídas das funções *Map* começam a ser geradas, elas não são simplesmente gravadas no disco. Há um processo mais complexo, que utiliza a escrita em *buffers* de memória e a aplicação de ordenações para melhorar a eficiência.

Cada *Task Map* possui um *buffer* na memória em formato circular o qual é utilizado para as escritas de sua saída. Esse *buffer,* que por padrão possui 100 MB, é uma parte da memória da JVM criada quando uma *Task* é submetida a um *NodeManager*.

Seu tamanho é gerenciado pelo valor do parâmetro de configuração **mapreduce.task.io.sort.mb**, e uma pequena parte desse tamanho é reservado para armazenar metadados referentes aos registros de despejo em disco.

Ao alcançar o valor configurado em **mapreduce.map.sort.spill.percent**, uma *thread* de segundo plano começa a despejar o conteúdo no disco. Enquanto o conteúdo é despejado no disco a escrita no *buffer* continua. Caso ocorra *buffer overflow* durante esse tempo, a *Task Map* é interrompida até que o processo de despejo no disco esteja completo.

Antes de escrever diretamente no disco, a *thread* divide os dados de entrada em partições do tamanho correspondente aos enviados para as últimas *Tasks Reduce*. Para cada uma dessas partições, uma *thread* de segundo plano executa em memória uma classificação por chave, e se houver uma função de combinação, ela é executada sobre a saída da classificação.

A função de combinação é executada quando o número de despejos alcança o valor configurado em **mapreduce.map.combine.minspills**. A ideia da função de

combinação é tornar os dados de saída da função *Map* mais compactos, assim menos dados são escritos em disco e transferidos para as funções *Reduce*.

Cada vez que o *buffer* de memória ativa a *thread* de despejo, um novo arquivo de despejo é criado. Antes da finalização da *Task*, existem vários arquivos que são unificados e ordenados. É possível controlar o número máximo de uniões por etapa, para isso é necessário alterar a propriedade de configuração **mapreduce.task.io.sort.factor**.

Despejar saídas *Map* em disco múltiplas vezes pode levar a um *overhead* adicional para leitura e união dos registros despejados. Uma maneira fácil de detectar se na fase *Map* estão ocorrendo despejos adicionais, é verificar os registros de saída *Map* (*Map output records*) e o registro de despejos (*Spilled records*) imediatamente depois que a fase *Map* esteja completa. Se o número de registro de despejo forem maiores que o de saídas *Map,* então está havendo despejos adicionais.

Deve-se tentar minimizar o número de despejos ajustando os valores dos parâmetros **mapreduce.task.io.sort.mb** e **mapreduce.map.sort.spill.percent**. Nesse caso para eliminar despejos desnecessários em disco a propriedade **mapreduce.task.io.sort.mb** deve ser configurada para acomodar ambos os dados e registros no *buffer* e o valor de **mapreduce.map.sort.spill.percent** deve ser ajustado para um valor próximo de 99%, deixando que o *buffer* seja preenchido perto de sua capacidade máxima. Desse modo, será feito um único despejo em disco quando a *Task Map* finalizar.

3.1.3 O Processo Reduce

A fase *Reduce* pode ser crucial e influenciar o tempo total de execução de um *Job MapReduce* no Hadoop. Todas as saídas geradas pelas *Tasks Map* precisam ser copiadas, agregadas, processadas e possivelmente todos os dados deverão ser reescritos no HDFS. Na Figura 8 (pág. 37) é ilustrado desde o processo de alimentação da função *Map* até a saída da função *Reduce*.

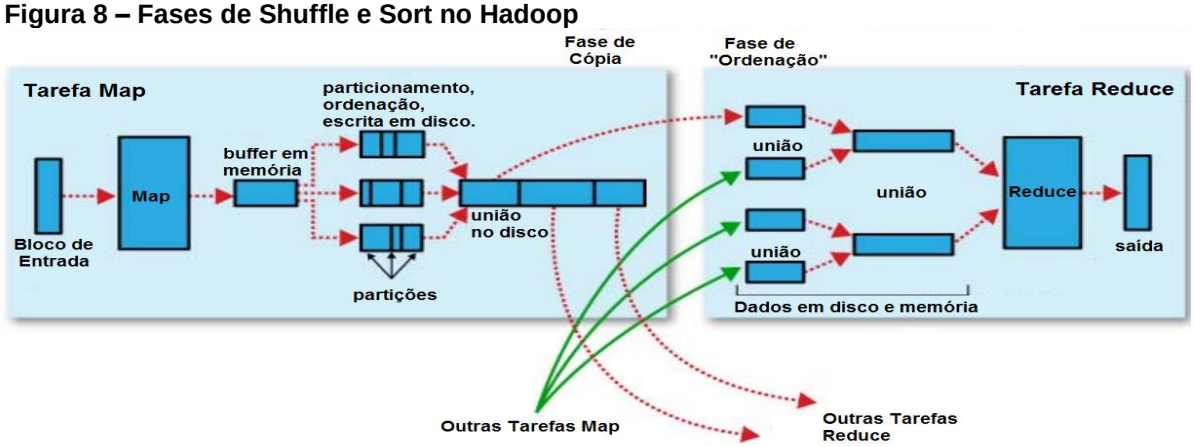
Os arquivos de saída das funções *Map* estão armazenados no disco local do nó *Worker* que processou a tarefa, e são requisitados pelos nós que irão executar a função *Reduce*. Além disso, as *Tasks Reduces* precisam das saídas de várias *Tasks*

Map que executaram ao longo do *cluster*. As *Tasks Map* finalizam em diferentes tempos, e as *Tasks Reduce* começam a copiar a sua saída logo que estejam completas. Esse processo é conhecido como fase de cópia (*copy phase*). As *Tasks Reduce* possuem um pequeno número de *threads* que efetuam essas cópias e que podem executar em paralelo.

As saídas das funções *Map* são copiadas, dependendo do seu tamanho para o *buffer* de memória ou disco, para os *nós* que executam as funções *Reduce*. O tamanho do *buffer* é configurado a partir da propriedade **mapreduce.reduce.shuffle.input.buffer.percent**, que se for grande o suficiente, irá comportar toda a saída *Map*. Quando os dados no *buffer* de memória alcançam a faixa estabelecida na propriedade **mapreduce.reduce.shuffle.merge.percent** ou o número de saídas *Map* alcança o limiar estabelecido na propriedade **mapreduce.reduce.merge.inmem.threshold**, ele é unido e despejado no disco. Configurar grandes valores de *buffer* para **mapreduce.reduce.shuffle.input.buffer.percent** e **mapreduce.reduce.input.buffer.percent** pode ajudar a evitar operações adicionais de leitura e escrita causadas pelo despejo da fase *Reduce*.

A fase de ordenação (*Sort Phase*) ocorre após todas as saídas *Map* serem copiadas. Ela então unifica todas as saídas *Map* mantendo sua ordem de classificação. Este processo é feito em etapas. Por exemplo, se há 60 saídas *Map*, e o fator de união configurado em **mapreduce.task.io.sort.factor** está com valor 10, então significa que serão necessárias 6 etapas. Cada etapa unifica 10 arquivos, a saída final das etapas são 6 arquivos.

Figura 8 – Fases de Shuffle e Sort no Hadoop



Fonte: Adaptado de WHITE (2015)

Ao invés de haver uma etapa final que unifique os 6 arquivos em um único, a fase de unificação economiza a leitura no disco direcionando sua saída para a função *Reduce* que é a última fase (*Reduce Phase*). Na fase final da execução do *Reduce* a sua saída é gravada no HDFS. A primeira réplica é armazenada no disco local que processou o *Reduce*.

3.1.4 BLOCO HDFS

O parâmetro de configuração **dfs.blocksize** é responsável pelo tamanho do bloco de dados armazenados no HDFS. Já os parâmetros **mapred.min.split.size e mapred.max.split.size** decidem o tamanho lógico da divisões de entrada. O número de *Tasks Map* é determinado pelos valores desses parâmetros (SHRINIVAS, 2012).

Suponha que há um arquivo de 100MB e o **dfs.blocksize** está com valor padrão de 64MB, então ele será dividido em 2 blocos. Um programa *Map/Reduce* para processar esses blocos, como não foi especificado qualquer valor de **mapred.min.split.size** ou **mapred.max.split.size,** irá atribuír 2 *Task Map* uma para cada bloco. Agora, suponha que antes de executar esse mesmo programa *Map/Reduce* fosse atribuído um valor de 100MB para **mapred.max.split.size** então seria criado apenas 1 *Task Map*. Se ao invés de atribuido **mapred.max.split.size** fosse atribuído **25MB** para **mapred.min.split.size** então seriam criados 4 *Task Map*.

Reduzir o número de *Tasks Maps* diminui o tempo de criação e destruição de JVMs e também reduz o custo de unificar saídas *Map* durante a fase *Reduce*. Se um *Job* no Hadoop gera um grande número de *Tasks Map* é melhor utilizar blocos maiores, mesmo aumentando o tempo de execução de cada *Task Map* (WHITE, 2015). Os parâmetros de tamanho do bloco do HDFS devem seguir o modelo:

mapred.min.split.size <= dfs.blocksize <= mapred.max.split.size

3.1.5 Resumo dos Parâmetros de Configuração

Segundo Tom White (2015) as configurações mais relevantes que podem melhorar o desempenho de um *Job MapReduce* no Hadoop são mostrados nos Quadros 4 e 5, junto com o seus propósitos gerais e valores padrões (valores definidos pelo desenvolvedor da ferramenta).

Quadro 4 - Propriedades de *tuning Map*

Nome da Propriedade	Tipo	Valor Padrão	Descrição
mapreduce.task.mapreduce.task.io.sort.mb	int	100	O Tamanho, em Megabytes, do *buffer* de memória para ser utilizado durante a ordenação da saída *Map*.
mapreduce.map.sort.spill.percent	float	0.80	Valor limiar do uso do buffer configurado em mapreduce.task.io.sort.mb para iniciar o processo de despejo em disco.
mapreduce.task.io.sort.factor	int	10	Número máximo de união das saídas *Map* por etapa.
mapreduce.map.combine.minspill	int	3	Número mínimo de despejos em disco para iniciar o processo de combinação.
mapreduce.map.output.compress	boolean	False	Compressão das saídas *Map*.
mapreduce.map.output.compression.codec	Nome da Classe	Default Codec	O tipo de compressão para ser usado na saída *Map*. Valor padrão: org.apache.hadoop.io compress.DefaultCodec.
mapreduce.shuffle.max.threads	int	0	O número máximo de threads *worker* que serão usadas para fornecer saídas Map para os *Reducers*.

Fonte: White (2015)

Quadro 5 - Propriedades de *tuning Reduce*

(continua)

Nome da Propriedade	Tipo	Valor Padrão	Descrição
mapreduce.reduce.shuffle.parallelcopies	int	5	O número de *threads* utilizadas para realizar as cópias das saídas *Map* para as entradas *Reduce*.
mapreduce.reduce.shuffle.maxfetchfailures	int	10	O número de vezes que um *Reduce* tenta trazer uma saída *Map* antes de reportar erro.
mapreduce.task.io.sort.factor	int	10	Número máximo de união das saídas *Map* por etapa.
mapreduce.reduce.shuffle.input.buffer.percent	float	0.70	Proporção do total da memória da JVM alocada para o buffer de saídas *Map* durante a fase de cópia no *shuffle*.
mapreduce.reduce.shuffle.merge.percent	float	0.66	Valor limiar utilizado para o buffer de saída *Map* (definido por mapreduce.reduce.shuffle.input.buffer.percent) para começar o processo de união das saídas e despejo no disco

Quadro 5 - Propriedades de *tuning Reduce*

(conclusão)

Nome da Propriedade	Tipo	Valor Padrão	Descrição
mapreduce.reduce.merge.inmem.threshold	int	1000	O número limitante de saídas *Map* para começar o processo de união da saídas e despejo no disco. Um valor de 0 ou menos significa que não há limite, e o comportamento de despejo é controlado somente pela propriedade **mapreduce.reduce.shuffle.merge.percent.**
mapreduce.reduce.input.buffer.percent	float	0.0	A proporção da memória da JVM para ser usada para reter saídas *Map* na memória durante a fase *Reduce*. Para a fase de *Reduce* começar, o tamanho das saídas *Map* não pode ser maior que os configurados aqui. Se o *Reduce* precisa de pouca memória, este valor pode ser incrementado para minimizar o número de consultas ao disco

Fonte: White (2015)

3.2 OUTROS PARÂMETROS E ASPECTOS RELEVANTES DO HADOOP

O Hadoop foi desenvolvido com características que tem por objetivo a otimização de desempenho do *framework*. A Seção 3.2.1 traz a característica do armazenamento local e as Seções 3.2.2 e 3.2.3 trazem os parâmetros de configurações responsáveis, respectivamente, pela compressão de dados e pelo reuso da JVM.

3.2.1 Armazenamento Local

As *Tasks Map* são atribuídas preferencialmente aos nós *Workers* que já contenham em seu disco local os dados que serão utilizados para o processamento, esse processo evita gasto desnecessário de banda de rede. Caso não seja possível, a prioridade para a escolha segue a sequência (WHITE, 2015):

1. Nós do mesmo processo;

2. Nós no mesmo *rack* do *cluster*;
3. Nós em diferentes *racks,* porém no mesmo *data center*;
4. Nós em diferentes *data centers*.

As *Tasks Reduce* geralmente são atribuídas às máquinas que possam utilizar dados locais produzidos pelas funções *Map*, porém isso depende da disponibilidade de recursos. Caso não seja possível atribuir a execução no mesmo nó a prioridade segue o padrão utilizado para as funções *Map*.

3.2.2 Compressão dos Dados

O Hadoop suporta compressão em três diferentes níveis: dados de entrada, saídas intermediárias *Map* e saída *Reduce*. Ele também suporta múltiplos *codecs* que podem ser utilizados na compressão e descompressão dos dados (WHITE, 2015). Alguns desses *codecs* tem um fator de compressão maior, porém levam mais tempo para a descompressão.

Ativar compressão de dados reduz o espaço em disco utilizado e o tráfego de rede, contudo utiliza mais ciclos de CPU para compressão e descompressão (SHRINIVAS, 2012). Por padrão os dados não são comprimidos, para habilitar essa opção basta alterar a configuração **mapreduce.map.output.compress** para *true* e escolher o codec de compressão. A escolha do codec é por meio da configuração do parâmetro **mapreduce.map.output.compression.codec**. Porém a ativação de compressão de dados exige uma maior quantidade de memória e CPU, pois é necessário descompactar os dados para o processamento, então é preciso verificar se ao habilitar o parâmetro de configuração o efeito alcançado reduz o tempo de execução do *Job*.

3.2.3 Reuso da JVM

O Hadoop suporta a configuração do parâmetro **mapreduce.job.jvm.numtasks** que controla se as JVMs geradas para a execução de *Tasks Map* e *Reduce* podem ser reutilizadas para executar outras *Tasks* (HEGER, 2013). Esta propriedade possui valor padrão “1” que significa não

reutilizar. O valor “-1” indica que um número ilimitado de *Tasks* pode ser escalado para aquela instância da JVM. Habilitar o reuso da JVM reduz o *overhead* da inicialização e da destruição da JVM. O reuso traz melhorias de desempenho geralmente em cenários onde existe um grande número de pequenas *Tasks* rodando.

3.3 BENCHMARK

Para Ciferri (1995) a técnica de *benchmark* consiste em um modelo de análise experimental onde um conjunto fixo de teste é executado sobre um sistema com o objetivo de avaliar o seu desempenho. Essa é uma técnica amplamente utilizada para mensurar o desempenho de sistemas computacionais.

Segundo Mclaren (1990), um *benchmark* é composto por um conjunto de teste funcional e um conjunto de teste de desempenho, que são executados por meio de um subconjunto de dados, simulando assim o ambiente de aplicação proposto pelo sistema. Essa técnica utiliza o próprio sistema para obtenção dos resultados de desempenho o que torna esses resultados altamente confiáveis.

A subseção 3.3.1 apresenta as ferramentas utilizadas nesse trabalho para efetuar a análise de desempenho. O *benchmark* será necessário para avaliar se as técnicas de *tuning* sobre os parâmetros de configurações do Hadoop apresentam melhorias em seu desempenho.

3.3.1 Ferramenta de *Benchmark*

O Hadoop conta com uma *suíte* de aplicativos para as mais diversas finalidades. Existe em especial um programa *MapReduce* que executa uma ordenação de dados. Ele é muito útil para fazer *benchmark* do sistema *MapReduce,* pois combina a utilização do HDFS e HMR. (WHITE, 2015). Para a sua completa execução são necessários o emprego de três programas (O’MALLEY, 2008), seguindo a ordem:

1. **TeraGen:** é um programa *Map* que gera os dados de entrada;
2. **TeraSort:** utiliza uma amostra de dados de entrada, gerado pelo TeraGen, e por meio de um programa *Map*/*Reduce* ordena esses dados;

3. **TeraValidate:** é um programa *Map/Reduce* que verifica se os dados de saída do TeraSort estão ordenados corretamente.

Por meio do TeraGen são gerados dados aleatórios. Este executa um *Job MapReduce* que reproduz dados binários aleatórios, com chaves e valores de diferentes tamanhos (O'MALLEY, 2008). O parâmetro de entrada para geração de dados do TeraGen é o número de linhas de 100 bytes. O formato dos dados gerados em cada linha de 100 bytes, consiste em: 10 bytes para a chave 10 bytes para identificação do dado gerado e 78bytes de caracteres (7 sequências de 10 caracteres de 'A' a 'Z') (WHITE, 2015).

O TeraSort utiliza os dados de entrada do TeraGen e efetua um processo de ordenação sobre eles. Para isso cria uma lista classificada de N – 1 chaves, sendo N o número de *Tasks Reduces*, que definem o intervalo de chave para cada *Task Reduce*. Assim, as chaves geradas pelo TeraGen que seguem lista[i-1]<=chave<lista[i] são enviadas para *Reduce* i (O'MALLEY, 2008). Em cada *Reduce* são ordenadas as chaves recebidas. Por fim, as saídas de cada *Reduce* i são reunidas seguindo a ordem i. A saída do *Reduce* tem replicação definida como 1, em vez do padrão 3, porque não é necessário que as saídas sejam replicadas em vários nós.

O tempo total de ordenação da entrada do TeraSort serve de métrica para avaliar o desempenho do sistema. É frequentemente utilizado para comparar a performance entre *clusters*, sendo útil também para determinar se as atribuições nos valores dos parâmetros de configuração do Hadoop são satisfatórias (EADLINE, 2015).

Na interface *web*[8], é possível ver o progresso do trabalho mais detalhadamente em cada fase, o que facilita a verificação do desempenho após a aplicação de *tuning* no sistema. Ao final é executada a validação por meio do TeraValidate para verificar se os dados de saída estão corretamente ordenados.

3.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Foram apresentados neste capítulo os parâmetros de configurações considerados mais relevante, segundo WHITE (2015), SHRINIVAS (2012) e HEGER

[8] http://host-master:8088

(2013), para o *tuning* do HRM e HDFS. Esses parâmetros estão descritos nos Quadros 4 e 5 com breves explanações sobre os seus papeis desempenhados no *framework*, há também sugestões de ajustes em seus valores. Por fim, na terceira seção são descritas as ferramentas de *benchmark* que foram utilizadas para avaliar as técnicas de *tuning* aplicadas ao *framework* Hadoop.

4 AMBIENTE EXPERIMENTAL E RESULTADOS

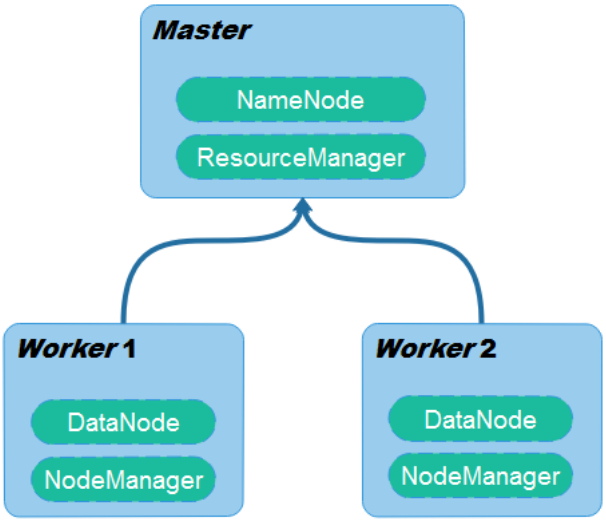
Este capítulo é dedicado as apresentações do ambiente experimental e dos resultados obtidos nas várias execuções da suíte de *benchmark* TeraSort nesse ambiente. É possível mensurar a importância da configuração de cada parâmetro no *framework* Hadoop realizando comparações de desempenho com variações de seus valores na execução do *benchmark*.

4.1 AMBIENTE EXPERIMENTAL

Para o presente trabalho utilizou-se na simulação do *cluster* uma máquina com 12GB de memória RAM, processador AMD Phenon II x4 955 com 6MB de memória *cache* L3 e disco rígido SATA de 7200 RMP com 1 TeraByte de espaço de armazenamento. O sistema operacional utilizado foi o Ubuntu 16.04 LTS 64-bits com kernel 4.4. Para a implementação do ambiente de teste virtualizado empregou-se a ferramenta Docker em sua versão 1.12 e o Apache Hadoop 2.7.2.

Para o ambiente de testes, foi configurado um Docker *Container* executando o processo *Namenode* e *Resource Manager*. Também foram configurados dois Docker *Containers* executando em cada um os processos *DataNode* e *Node Manager*. Estes foram nomeados respectivamente: *Master*, *Worker1* e *Worker2*. A Figura 9 exemplifica os processos executados em cada máquina do ambiente.

Figura 9 – Ambiente de Teste



Fonte: Autoria própria

Os containers dividem os recursos de *hardware* do sistema hospedeiro por igual. Como são três *containers* no ambiente experimental cada um ficou com uma parcela de aproximadamente 33% dos recursos disponíveis na máquina hospedeira.

4.2 AJUSTES DOS PARÂMETROS DE CONFIGURAÇÕES

O livro de White (2015) “Hadoop: *The Definitive Guide*” possui a seção “*Tuning a Job*” na qual são apresentadas as sequências de verificações que devem ser feitas em um *Job* para certificar-se que ele está otimizado para a execução. No Quadro 6 é apresentado essa sequência de verificação.

Quadro 6 – Lista de verificação tuning.

Parâmetro	Melhores práticas
Número de *Task Map*	Caso as *Tasks Map* estiverem concluindo suas execuções em poucos segundos, se for possível, é bom reduzir o seu número.
Número de *Task Reduce*	*Tasks Reduce* devem executar por 5 minutos ou mais, e produzir ao menos um bloco de dados.
Função de Combinação	Verificar se é possível a adoção de uma função de combinação.
Compressão dos dados	Verificar se a adoção de compressão de dados traz algum benefício.
Ajustes dos parâmetros do processo *Shuffle*	Dezenas de parâmetros que envolvem o processo *Shuffle* podem ser configurados. Ajustes em seus valores podem trazer melhorias de desempenho para o Hadoop.

Fonte: Adaptado de White (2015)

Escolher um valor adequado para a quantidade de Tasks executando no cluster pode mudar radicalmente o desempenho do Hadoop. Valores baixos reduzem a distribuição e valores muito elevados esgotam os recursos da máquina exigindo mais etapas de execução.

Entre os parâmetros relacionados ao processo *Shuffle* estão o número de *threads* que efetuarão a cópia da saída das *Tasks Map* para as *Tasks Reduce*, o tamanho do *buffer* de memória da JVM que processou a *Task* e o fator de união das saídas *Map*. O ajuste fino nos valores desses parâmetros permite um maior controle sobre a utilização dos recursos no *cluster* e consequentemente leva a um melhor desempenho no *framework* Hadoop.

As Seções 4.2.1, 4.2.2 e 4.2.3 apresentam os tempos de execução do *benchmark* com variações respectivamente, no número de *Tasks Map*, número de *Tasks Reduce* e nos valores dos parâmetros de configuração do Hadoop relacionados ao processo *shuffle.* Nesse estudo não foi empregada uma função de combinação, pois o *benchmark* utilizado a título de comparação não permite. Também não utilizou-se compressão de dados, pois não há trafego de rede no ambiente de teste. Sendo assim, a ativação deste parâmetro não surtirá o efeito desejado. Na Seção 4.2.4 é feita uma combinação composta pelos valores dos parâmetros de configuração do Hadoop (número de Tasks Map, número de Tasks Reduce e parâmetros de configuração do processo *shuffle*) que apresentaram o melhor desempenho no *benchmark*.

Para cada alteração nos valores do número de *Tasks Maps*, de *Tasks Reduce* e nos parâmetros de configuração do Hadoop submeteu-se a execução de um total de cinco vezes no *benchmark,* com o objetivo de verificar variações no tempo de execução. A partir do tempo de execução do *framework* Hadoop foi possível inferir conclusões.

Contudo antes de qualquer conclusão para obter informações adicionais faz-se necessário o cálculo da média aritmética, do desvio padrão e do intervalo de confiança (IC) de cada conjunto de 5 execuções. O desvio padrão permite verificar a dispersão em relação à média. A Equação 4.1 demonstra como foi obtido esse valor. O IC obtido pela Equação 4.2, com nível de confiança de 95%, é utilizado para indicar a confiabilidade de uma estimativa (GUIMARÃES, 2007).

$$\sigma i = \sqrt{\frac{\sum_{i=1}^{n}(xi-\bar{x}i)^2}{n-1}} \tag{4.1}$$

Onde:

$\boldsymbol{\sigma i}$ = Desvio padrão. $\boldsymbol{\bar{x}i}$ = Média aritmética. $\boldsymbol{xi}$ = Série de valores. $\boldsymbol{n}$ = N° de amostras

$$ICi = \bar{x}i \pm z * \frac{\sigma i}{\sqrt{n}} \tag{4.2}$$

Onde:

$\boldsymbol{\bar{x}i}$ = Média aritmética. $\boldsymbol{\sigma i}$ = Desvio padrão $\boldsymbol{z}$ = Valor crítico (1,96) $\boldsymbol{n}$ = N° de amostras

O cálculo dos limites superiores (Li) e inferiores (Ii), é representado pelas equações 4.3 e 4.4 na ordem. O limite inferior representa o valor mínimo de tempo de execução do *benchmark*, e o limite superior o valor máximo.

$$Ii = \overline{xi} - ICi \quad (4.3)$$

$$Li = \overline{xi} + ICi \quad (4.4)$$

Na qual:

$\overline{xi}$ = Média aritmética $\mathbf{IC}i$ = Intervalo de confiança

Por fim, na Seção 4.2.4 é apresentado a média aritmética, limites inferior e superior da execução do TeraSort com e sem o *tuning* nos parâmetros de configuração do Hadoop. O procedimento para efetuar as configurações dos parâmetros do Hadoop está descrito no Apêndice A. No Apêndice B encontra-se a demonstração de como é feita a execução da suíte de *benchmark TeraSort*.

4.2.1 Número de Tasks Map

O TeraGen é um programa *MapReduce* que executa somente *Tasks Map* para a produção de dados binários aleatórios, que posteriormente alimentam o programa TeraSort. A Tabela 2 apresenta o tempo transcorrido (em segundos) do TeraGen para a geração de 10 Gigabytes de dados, com variações no número de *Tasks Map*.

Tabela 2 – Tempo de execução do TeraGen (em segundos)

Nº *Tasks Map*	Média (x)	Des. Padrão (σ)	IC	Lim. Inferior (Ii)	Lim. Superior (Li)
1	199,42	9,60	4,29	195,13	203,71
2	212,01	2,37	1,06	210,95	213,07
5	252,39	4,65	2,08	250,31	256,47
10	330,08	4,82	2,15	327,93	332,23
20	321,39	28,26	12,64	308,75	334,03
40	326,87	15,09	6,75	320,12	333,62
80	410,90	87,30	39,04	371,86	449,94

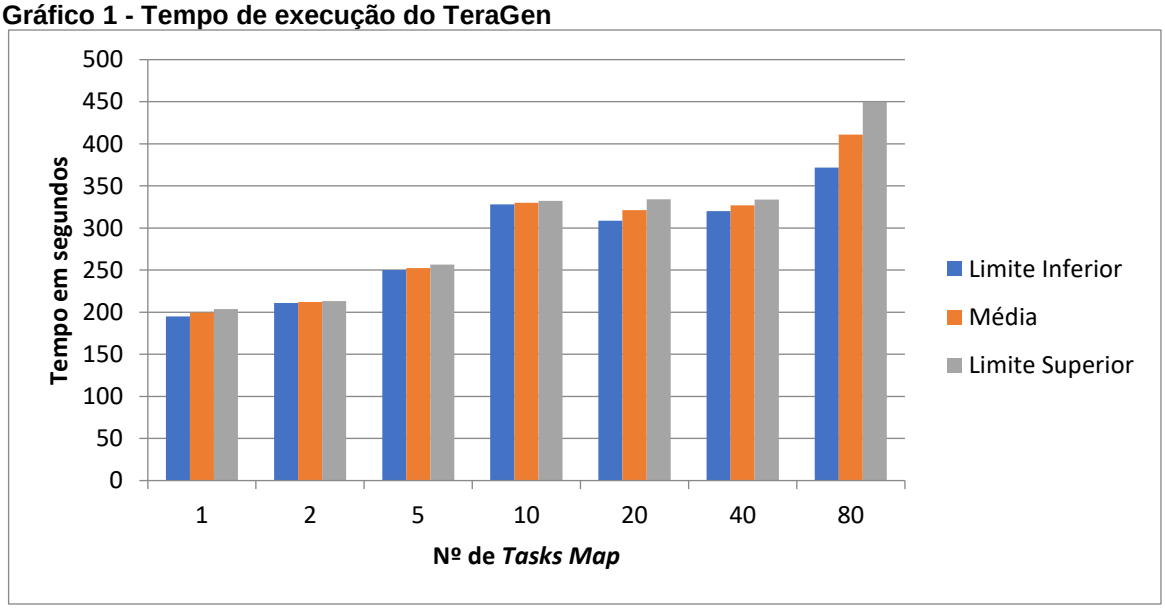
Fonte: Autoria própria

O número de *Tasks Map* no TeraSort é controlado pelo número de saídas do TeraGen. No caso da geração de 10GB de dados e configuração do tamanho dos blocos do HDFS variando de 128MB até 512MB serão produzidos de 20 a 80 *Tasks Maps* no TeraSort. Valor este obtido pela divisão da quantidade de dados produzido no TeraGen pelo tamanho do bloco configurado para o HDFS.

Segundo a literatura o nível aceitável de *Tasks Maps* executando em paralelo variam de 10 até 100 por nó do *cluster* (*apud* NGHIEM, 2016). Como o ambiente de teste possui 2 nós *Workers*, a geração de 10GB mostra-se apropriada, sendo que ao menos 10 *Tasks Maps* serão executadas por nó (no caso do bloco de dados possuir valor de 512MB).

Com o intuito de facilitar a análise das informações da Tabela 2, apresenta-se o Gráfico 1. Este gráfico distribui, em seu eixo x, a média das 5 execuções do TeraGen com variações no número de *Tasks Map* e no eixo y o tempo em segundos de cada execução. Neste gráfico também é possível visualizar os limites inferiores e superiores.

Gráfico 1 - Tempo de execução do TeraGen



Fonte: Autoria própria

4.2.2 Número de Tasks Reduce

O TeraSort é um programa *MapReduce* que executa *Tasks Map* e *Reduce* para a realização da ordenação dos dados gerados pelo programa TeraGen. A

Tabela 3 apresenta o tempo transcorrido (em segundos) do TeraSort para a ordenação de 10GB de dados, com variações no número de *Task Reduce*, para demonstrar a performance do Hadoop. Para fins de estudo, decidiu-se considerar apenas o tempo de execução das *Tasks Reduce* (início da execução da primeira *Task Reduce* até o encerramento da última *Task*).

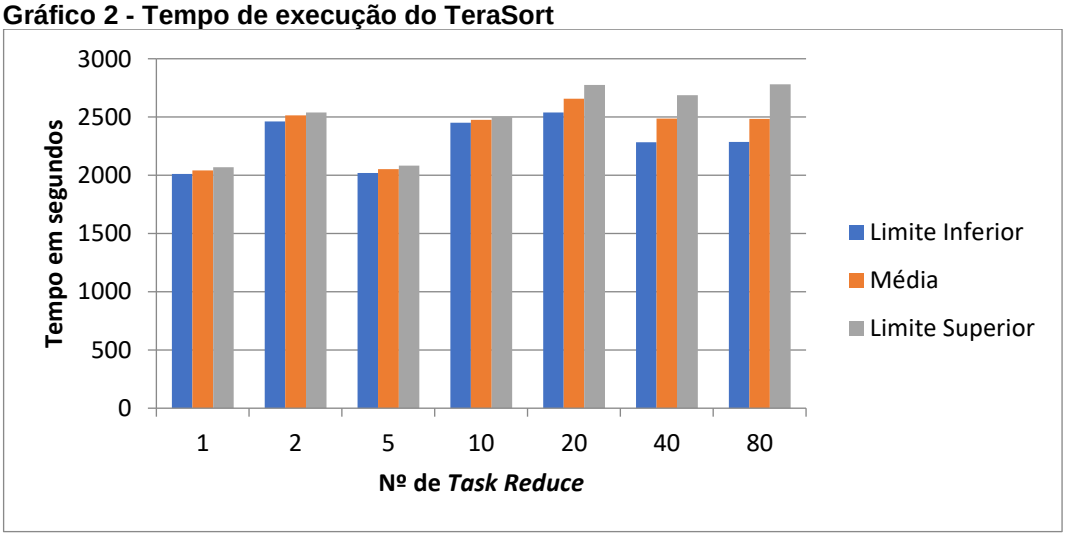
Tabela 3 – Tempo de execução do TeraSort (em segundos)

Nº *Tasks Reduce*	Média (x)	Des. Padrão (σ)	IC	Lim. Inferior (li)	Lim. Superior (Li)
1	2040,67	63,95	28,59	2011,08	2069,26
2	2514,13	53,25	23,81	2460,88	2537,94
5	2051,44	70,63	31,58	2019,86	2083,02
10	2475,32	53,74	24,03	2451,29	2499,35
20	2656,52	264,27	118,18	2538,34	2774,7
40	2485,56	675,13	301,92	2183,64	2787,48
80	1682,47	440,23	196,87	1485,60	1879,34

Fonte: Autoria própria

Com a apresentação do Gráfico 2 é possível verificar facilmente as alternâncias do tempo de execução. No eixo x desse gráfico apresentam-se as variações no número de *Tasks Reduce* na execução do TeraSort e no eixo y o tempo em segundos. O gráfico possui também o intervalo de confiança por meio da apresentação dos limites inferiores e superiores de cada execução.

Gráfico 2 - Tempo de execução do TeraSort



Fonte: Autoria própria

4.2.3 Ajustes dos Parâmetros do Processo *Shuffle*

O processo *shuffle* no TeraSort é o responsável pela transferência das saídas *Map* para as entradas *Reduce*. É ele que garante que todas as chaves enviadas ao *Reduce* i são menores que as enviadas ao *Reduce* i+1 (O'MALLEY, 2008).

O Quadro 7 exibe os parâmetros de configuração relacionados ao processo s*huffle* e seus valores padrões no Hadoop (valores *default* do framework). Parâmetros estes que foram citados no Capítulo 3 e que segundo Tom White (2015) são relevantes para o aprimoramento de performance do *framework*. Também para cada parâmetro é apresentado o arquivo responsável pela sua configuração.

Quadro 7 – Parâmetros e seus valores padrões no Hadoop.

Parâmetro de Configuração	Valor Padrão	Arquivo Responsável
dfs.blocksize	128Mb	hdfs-site.xml
mapreduce.task.io.sort.mb	100	mapred-site.xml
mapreduce.map.sort.spill.percent	0.8	mapred-site.xml
mapreduce.reduce.shuffle.input.buffer.percent	0.7	mapred-site.xml
mapreduce.task.io.sort.factor	10	mapred-site.xml
mapreduce.reduce.shuffle.merge.percent	0.66	mapred-site.xml
mapreduce.reduce.input.buffer.percent	0	mapred-site.xml
mapreduce.job.jvm.numtasks	1	mapred-site.xml
mapreduce.reduce.shuffle.parallelcopies	5	mapred-site.xml

Fonte: Autoria própria

Ao executar o TeraSort para ordenar 10GB com os parâmetros de configurações padrões foram feitas 80 divisões na entrada e cada uma dessas divisões foi processada por uma *Task Map*. Foi executada apenas uma *Task Reduce*. Quando verificada a interface *NodeManager* desse *Job* contatou-se que são possíveis apenas 14 execuções simultâneas de *Tasks* (devido à escassez de recursos computacionais). Nesse caso foi necessária a execução de 5 etapas com 14 *Tasks Maps* e 1 etapa com 10 *Tasks Map* e 1 *Task Reduce*.

Com a constatação de que o *cluster* de máquinas virtuais utilizado nessa simulação tem o limite de execução de 14 *Task*s simultaneamente, alterou-se a configuração do valor do parâmetros **dfs.blocksize** para 512MB na geração da carga de trabalho de 10GB. Com essa configuração o TeraSort passou a executar

20 divisões de entrada e assim apenas 2 etapas de execução. O Quadro 8 exibe os valores do conjunto de parâmetros relacionado ao processo *shuffle* que obtiveram menor tempo de resposta das várias execuções do TeraSort realizado no ambiente de teste.

Quadro 8 – Parâmetros e seus valores tunados no Hadoop

Parâmetro de Configuração	Valor Tunado
mapreduce.task.io.sort.mb	110
mapreduce.task.io.sort.factor	14
mapreduce.job.jvm.numtasks	-1
mapreduce.reduce.shuffle.parallelcopies	14

Fonte: Autoria própria

O valor máximo do parâmetro **mapreduce.task.io.sort.mb** que o ambiente de teste suportou foi 110MB devido a escassez de memória RAM da máquina hospedeira. Sendo assim, não é possível evitar escrita adicional em disco, pois esse tamanho é inferior ao tamanho do bloco de dados. Então, como haverá mais de um despejo em disco por *Task Map*, a configuração do parâmetro **mapreduce.map.sort.spill.percent** que controla a porcentagem de preenchimento da memória do *buffer* da JVM que executa a *Task* não possui muita utilidade. Sendo assim, seu valor foi mantido padrão.

Foram ajustados os parâmetros **mapreduce.reduce.shuffle.parallelcopies** e **mapreduce.task.io.sort.factor** para o equivalente ao número de *Tasks Map* executados no ambiente de teste. Foram feitos vários ajustes nos valores dos parâmetros **mapreduce.reduce.shuffle.input.buffer.percent** e **mapreduce.reduce.shuffle.merge.percent,** mas não houve melhora significativa em relação aos valores padrões. O reuso da JVM passou a ser utilizado, pois há mais de uma etapa de execução e assim evita o *overhead* na criação de novas JVM.

4.2.4 Configurações Padrões e com *Tuning* do *Framework* Hadoop

A Tabela 4 apresenta o tempo transcorrido (em segundos) do TeraSort na ordenação de 10 GB de dados, com e sem o *tuning* dos parâmetros de configuração do Hadoop. Os parâmetros com *tuning* são os demonstrados no Quadro 8, porém

utilizando também 14 *Tasks Reduce* (valor máximo de execuções em paralelo) e dados de entrada produzidos no TeraGen com o valor da propriedade **dfs.blocksize** igual 512MB**.**

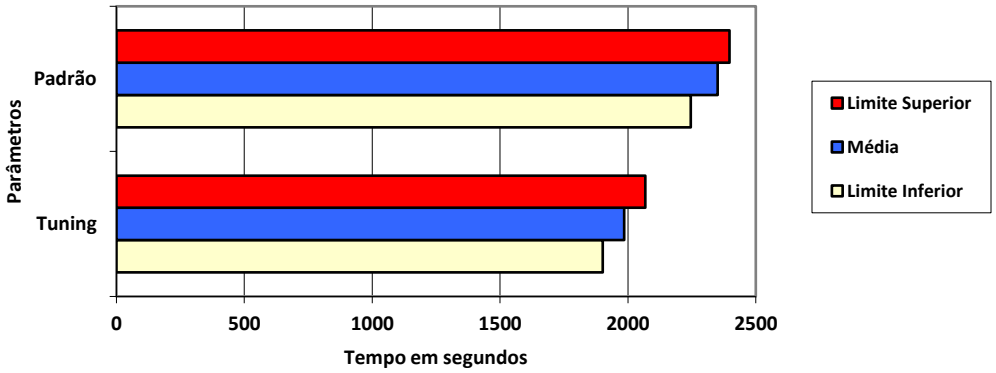
Tabela 4 – Tempo de execução do TeraSort (em segundos)

	Média (x)	**Des. Padrão (σ)**	**IC**	**Lim. Inferior (li)**	**Lim. Superior (Li)**
Padrão	2350,84	104,54	46,75	2246,30	2397,59
Tuning	1984,98	185,73	83,06	1901,92	2068,04

Fonte: Autoria própria

Para melhor ilustrar os dados da Tabela 4, o Gráfico 3 é apresentado. O eixo x desse gráfico destina-se ao tempo em segundos e no eixo y a execução do TeraSort com e sem os *tuning* dos parâmetros. Também são representados os limites inferiores e superiores de cada execução.

Gráfico 3 - Tempo de execução do TeraSort



Fonte: Autoria própria

4.3 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Foram apresentados neste capítulo os resultados do tempo de execução da suíte de *benchmark* TeraSort sob variações de valores nos parâmetros de configurações do Hadoop. A busca pela otimização do *framework* foi orientada com base no livro de White (2015).

Os resultados obtidos demonstraram que um número maior de *Tasks* executando o *Job* pode não significar uma execução mais rápida. Quanto mais

Tasks Map e *Reduce* existirem, um maior número de leitura e escrita em disco ocorre, e como o acesso ao disco no ambiente de teste é concorrente, mais tempo será despendido quanto maior o número de operações em disco ocorrer.

Outro ponto observado com os resultados foi que os parâmetros relacionados ao processo *shuffle* podem determinar a otimização do *framework*. Configurar altos valores para os *buffers* das JVMs que processam as *Tasks* evitam escritas adicionais em disco, criações de *threads*, e consequentemente interrupção no processamento, no caso de o *buffer* ser preenchido antes da escrita no disco ser efetivada.

O Capítulo 5 apresenta as conclusões acerca dos resultados do presente trabalho e as recomendações para a continuidade dos trabalhos nesta área de estudo.

5 CONCLUSÃO

Com a popularização da internet no final dos anos 90, quantidades massivas de dados têm sido geradas incessantemente. Segundo O'MALLEY (2008), *Big Data* são dados que podem ser estruturados e não estruturados, gerados por múltiplas fontes em altas velocidades e grandes quantidades. Nele há informações que auxiliam as tomadas de decisões, tornado as mais eficientes e inteligentes. Devido as características desses dados, ferramentas e técnicas especiais são necessárias para a sua manipulação. Neste contexto destaca-se o *framework* Hadoop.

O Hadoop é considerado uma grande referência para a manipulação do *Big Data*, e para esse propósito utiliza a implementação do paradigma de programação *MapReduce*. Esse paradigma permite o processamento paralelo de conjuntos de dados, sendo assim é adequado para a implantação em *clusters*. O Hadoop oculta toda a complexidade do sistema paralelo e distribuído, assim o desenvolvedor preocupa-se apenas com a programação das funções *Map* e *Reduce*. As funções *Map* e *Reduce* são responsáveis pela maneira com que os dados são processados.

Apenas a implantação do *framework* Hadoop por vezes não atende aos requisitos de desempenho esperados. Contudo, existem centenas de parâmetros de configurações que podem prover uma melhoria de desempenho do Hadoop, após a sua implantação, por meio do *tuning* de seus valores. Mas para atingir a melhoria de performance por intermédio do *tuning* dos valores dos parâmetros de configuração são demandados profundos conhecimentos a respeito da ferramenta, da plataforma operacional, dos recursos computacionais disponíveis e da carga de trabalho utilizada. Em um ambiente virtual essa complexidade é intensificada devido ao fator de acesso concorrente ao disco.

Para atingir o objetivo principal deste trabalho de realizar o *tuning* do Hadoop por meio de ajustes em seus parâmetros de configurações, foi necessária a execução do *framework* Hadoop em um *cluster* virtualizado, com o auxílio da ferramenta Docker. Por meio da suíte de ferramentas de *benchmark* TeraSort sob variações de valores nos parâmetros de configurações do Hadoop no ambiente de teste, foram extraídos os tempos de execução. A partir dos tempos obtidos, foi possível verificar o impacto das variações nos valores dos parâmetros configurações no desempenho do Hadoop.

Os resultados obtidos apontam que é possível obter melhorias de desempenho no Hadoop por meio do *tuning* dos valores de seus parâmetros de configurações. Para isso é necessário avaliar a execução do *Job* aplicado no Hadoop e segundo as orientações de verificações encontradas na literatura e apresentadas no Quadro 6 certificar-se de que a ferramenta está otimizada no cenário apresentado. No caso da execução do *Job* não estar otimizada é necessário aplicar ajustes nos valores dos parâmetros de configuração do Hadoop.

O *tuning* do *framework* depende inteiramente do *Job* que está sendo executado. Diferentes *Jobs* exigem diferentes valores nos parâmetros de configuração. Assim, *clusters* que utilizam o Hadoop para propósitos gerais tornam-se difíceis de otimizar, já que diferentes *Job* exigem diferentes configurações de parâmetros, e nesse caso os valores padrões dos parâmetros de configurações são satisfatórios.

A pesquisa realizada neste trabalho contribui para o aprimoramento de *clusters* executando o Apache Hadoop. O *tuning* evita o desperdício de recurso computacionais, tempo e consequentemente recursos financeiros.

5.1 TRABALHOS FUTUROS

Os seguintes temas destacam-se como potenciais trabalhos futuros:

1. Realizar um comparativo de performance do *framework* Spark com o Hadoop *MapReduce* para análise de *Big Data*.
2. Analisar o desempenho da ferramenta HBase na substituição do HDFS no *framework* Hadoop.
3. Utilizar os diferentes algoritmos de compressão de dados do Hadoop em um *cluster* de máquina física para a análise de desempenho.
4. Apresentar o ecossistema do Hadoop (HDFS, HMR, Yarn, Pig, Hive, Spark, HBase, Zookeper, Mahout, RedShift, Storm e outros). Para que se destina cada ferramenta. Apresentando exemplos e casos de utilização para resolução de problemas de *Big Data*.

REFERÊNCIAS

BINI, Tarcizio Alexandre. **Análise da aplicabilidade das regras de ouro ao tuning de sistemas gerenciadores de bancos de dados relacionais em ambientes de computação em nuvem**. 2014. 100f. Tese (Doutorado) – Programa de Pós-Graduação em Informática, Universidade Federal do Paraná. Curitiba, 2014.

BOETTIGER, Carl. An introduction to Docker for reproducible research. **ACM SIGOPS Operating Systems Review**, v. 49, n. 1, p. 71-79, 2015.

Bosing, A.; Kaufmann, E. R. Virtualização de servidores e desktops. **Unoesc & Ciência-ACET**, vol. 3–1, 2012, p. 47–64.

Carissimi, A. Virtualização: da teoria a soluções, **Minicursos do Simpósio Brasileiro de Redes de Computadores–SBRC**, vol. 2008, 2008, p. 173– 207.

CHU, Cheng et al. Map-reduce for machine learning on multicore. **Advances in neural information processing systems**, v. 19, p. 281, 2007.

CIFERRI, R. R. **Um benchmark voltado à análise de desempenho de sistemas de informações geográficas**. 1995. 164f. Dissertação (Mestrado), Departamento de Ciência da Computação, UNICAMP. Campinas, 1995.

COULOURIS, George F.; DOLLIMORE, Jean; KINDBERG, Tim. Distributed systems: concepts and design. **pearson education**, 2005.

DITTRICH, Jens; QUIANÉ-RUIZ, Jorge-Arnulfo. Efficient big data processing in Hadoop MapReduce. **Proceedings of the VLDB Endowment**, v. 5, n. 12, p. 2014-2015, 2012.

EADLINE, Douglas. **Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem**. Addison-Wesley Professional, 2015.

FISHER-OGDEN, John. **Hardware support for efficient virtualization**. University of California, San Diego, Tech. Rep, 2006.

GANTZ, John; REINSEL, David. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, ang Biggest Growth in the Far East. **IDC iView: IDC Analyze the future**, v. 2007, p. 1-16, Framingham, dec. 2012. Disponível em: <http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf>. Acesso em: 30 ago. 2015.

GHEMAWAT, S et al. The Google file system. In: ACM SIGOPS OPERATING SYSTEMS REVIEW. **Anais**. [S.1:s.n], 2003. V.37, n.5, p.29-43.

GOLDMAN, Alfredo et al. Apache Hadoop: conceitos teóricos e práticos, evolução e novas possibilidades. **XXXI Jornadas de atualizações em informática**, p. 88-136, 2012.

GUIMARÃES, Rui Campos; CABRAL, José A. Sarsfield. **Estatística**. McGraw-Hill, Interamericana de España, 2007.

GROLINGER, Katarina et al. Challenges for mapreduce in big data. **In: 2014 IEEE World Congress on Services**. IEEE, 2014. p. 182-189.

HEGER, Dominique. Hadoop performance tuning-a pragmatic & iterative approach. **CMG Journal**, v. 4, p. 97-113, 2013.

JEFFREY, Dean; GHEMAWAT, Sanjay. MapReduce: Simplified data processing on large clusters**. In: Symposium on Operating System Design and Implementation (OSDI)**, 6., 2004, São Francisco. Technical Program, USENIX Association Berkele, 2004. páginas 137–150.

KATAL, Avita; WAZID, Mohammad; GOUDAR, R. H. Big data: issues, challenges, tools and good practices. **In: Contemporary Computing (IC3), 2013 Sixth International Conference on.** IEEE, 2013. p. 404-409.

KOBYLINSKA, Anna; MARTINS, Filipe. **Big data tools for midcaps and others**. Disponível em: < http://www.admin-magazine.com/Archive/2014/20/Big-data-tools-for-midcaps-and-others>. Acesso em: 25 Ago. 2016.

KOCAKULAK, Hakan; TEMIZEL, Tugba Taskaya. A hadoop solution for ballistic image analysis and recognition. **In: High Performance Computing and Simulation (HPCS), 2011 International Conference on.** IEEE, 2011. p. 836-842.

LAI, Yang; ZHONGZHI, Shi. An efficient data mining framework on Hadoop using java persistence API. **In: Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on.** IEEE, 2010. p. 203-209.

LEWIS, Steven et al. Hydra: a scalable proteomic search engine which utilizes the Hadoop distributed computing framework. **BMC bioinformatics**, v. 13, n. 1, p. 324, 2012.

LOHR, Steve. **The age of big data**. New York Times, v. 11, 2012.

MADDEN, Sam. From databases to big data. **IEEE Internet Computing**, n. 3, p. 4-6, 2012

MCLAREN, R.A. **The Art of Benchmarking GJS Technology.** Know Edge Ltd., Edinburgh, 1990. (Technical Repor!).

MENEZES, Sandro Loiola; FREITAS, Rebeca Schroeder; PARPINELLI, Rafael Stubs. Mineração em Grandes Massas de Dados Utilizando Hadoop MapReduce e Algoritmos Bio-inspirados: Uma Revisão Sistemática. **Revista de Informática Teórica e Aplicada**, v. 23, n. 1, p. 69-101, 2016.

MERKEL, Dirk. Docker: lightweight linux containers for consistent development and deployment. **Linux Journal**, v. 2014, n. 239, p. 2, 2014.

MONIRUZZAMAN, A. B. M.; HOSSAIN, Syed Akhter. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. **arXiv preprint arXiv:1307.0191**, 2013.

NGHIEM, Peter P.; FIGUEIRA, Silvia M. Towards efficient resource provisioning in MapReduce. **Journal of Parallel and Distributed Computing**, v. 95, p. 29-41, 2016.

O'MALLEY, Owen. **Terabyte sort on apache hadoop**. Disponível em: <http://sortbenchmark. org/Yahoo-Hadoop>. Acesso em: 05 nov. 2015. p. 1-3, 2008.

OLIVEIRA, Israel Campos de et al. **Aprimorando a elasticidade de aplicações de banco de dados utilizando virtualização em nível de sistema operacional**. 2015. 95f. Dissertação (Mestrado), Pontifícia Universidade Católica do Rio Grande do Sul, PUC-RS. Porto Alegre, 2015.

PICOLI, Ivan. **Uma abordagem de classificação não supervisionada de cargas de trabalho de sistemas analíticos em apache Hadoop através de análise de log**. 2014. 69f. Dissertação (Mestrado) – Programa de Pós-Graduação em Informatica, Universidade Federal do Paraná. Curitiba, 2014.

RECKZIEGEL, Bruno. **Aplicação do MapReduce na análise de mutações genéticas de pacientes**. 2013. 41f. Trabalho de Graduação - Universidade Federal do Rio Grande do Sul, 2013.

SHRINIVAS, Joshi. Hadoop Performance Tuning Guide. **AMD White Paper**, 2012.

SMITH, J. E.; NAIR, R. The architecture of virtual machines. computer, **IEEE Computer Society Press**, vol. 38–5, May 2005, pp. 32–38.

TRAMONTINA, Gregório. B. **Database tuning: Configurando o Interbase e o PostgreSQL**. Campinas, 2008. Disponível em: <http://www.ic.unicamp.br/~geovane/mo410-091/Ch20-ConfigInterbasePosgres-art.pdf> Acesso em: 29 jul. 2015.

VAVILAPALLI, Vinod Kumar et al. Apache hadoop yarn: Yet another resource negotiator. **In: Proceedings of the 4th annual Symposium on Cloud Computing**. ACM, 2013. p. 5.

VENNER, Jason. **Pro Hadoop:** Building scalable, distributed application in the cloud. 1. ed. Apress, 2009.

VITALINO, Jeferson Fernado Notonha; Castro, Mascos andré nunes. **Descomplicando o Docker**. Rio de Janeiro, Brasport, 2016.

WHITE, Tom. **Hadoop The Definitive Guide.** 4. ed. O'Reilly Media, 2015.

XUE, S; PAN, W. Parallel PK-means Algorithm on Meteorological Data Using MapReduce. **Journal of Wuhan University of Technology**, [S.1.], v.34, n.12, p.139-142, 2012.

ZHANG, Y. et al. Parallel option pricing with BSDEs method on MapReduce. **In: COMPUTER RESEARSH AND DEVELOPMENT (ICCRD)**, 2011 3RD International conference on. Anais..[S.1.:s.n], 2011. v.1, p.289-293.

APÊNDICE A - Ajustes dos Parâmetros de Configurações

Ajustes dos Parâmetros de Configurações

É possível ajustar os parâmetros de configurações do Hadoop de duas maneiras, sendo:

1. Passando como atributo na execução de um *Job*. É mais utilizado para efetuar testes e verificar como o ambiente se comporta com os ajustes dos parâmetros. O exemplo a seguir mostra como é feita a definição do parâmetro de configuração que controla o tamanho dos blocos de dados na execução do TeraGen.
   - hadoop jar hadoop-mapreduce-examples-2.7.2.jar\ teragen **-Ddfs.blocksize=1024M** 10000000 nome-arquivo
2. Modicando os valores de parâmetros de configuração nos arquivos XMLs responsáveis. Esse método é geralmente utilizado para perpetuar as definições que repercutiram no melhoramento de desempenho do *framework* para todos os Jobs que serão executados no ambiente. O Hadoop possui vários arquivos XML responsáveis por definições do ambiente, entre eles estão:
   - hdfs-site.xml, responsável pelas definições do sistema HDFS;
   - mapred-site.xml, responsável pelas definições do HMR;
   - yarn-site.xml, responsável pelas definições do YARN;
   - core-site.xml, responsável pelas definições do *cluster* (memória, diretórios, etc.)

   Quando configurado um parâmetro no arquivo *-site.xml ele sobrescreve o configurado por padrão no arquivo *-default.xml. É necessário reinicializar o *framework* para que as configurações realizadas no XML entrem em vigor. Para realizar a configuração do parâmetro basta editar o arquivo *-site.xml responsável e entre as *tags* <property> e </property> adicionar os valores e *tags*, como segue o exemplo:

```
<property>
        <name> dfs.blocksize </name>
        <value>512Mb</value>
</property>
```

APÊNDICE B - Execução do *benchmark* TeraSort

Execução do *benchmark* TeraSort

Cada linha produzida de chave/valor do TeraGen possui 100 Bytes, então para a geração de 10GB de dados o valor passado para função em Bytes é 10000000 (EADLI, 2015). A Figura 10 apresenta a sintaxe para a execução do TeraSort:

Figura 10 – Exemplo de *script* de execução TeraGen

```
$ hadoop jar hadoop-*examples*.jar teragen

<número de linhas de 100-bytes> <diretório de saída TeraGen >
```

Fonte: Autoria própria

A Figura 11 apresenta a sintaxe para a execução do TeraSort:

Figura 11 – Exemplo de *script* de execução TeraSort

```
$ hadoop jar hadoop-*examples*.jar teragen

<diretório de saída TeraGen> <diretório de saída TeraSort>
```

Fonte: Autoria própria

A Figura 12 apresenta a sintaxe para a execução do TeraValidate:

Figura 12 – Exemplo de *script* de execução TeraValidate

```
$ hadoop jar hadoop-*examples*.jar teravalidate

< diretório de saida TeraSort> <diretório de saida TeraValidate>
```

Fonte: Autoria própria

As Figura 13 apresenta um modelo de scripts para a execução do programa TeraSort no Hadoop versão 2.7.2. Nesse script é possível verificar os ajustes de alguns parâmetros de configuração do Hadoop.

Figura 13 – Parâmetros TeraSort

```
#!/bin/bash

hadoop fs -rm -r user/hdfs/sorted-data

hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar terasort \
-D mapreduce.job.maps=20 \
-D mapreduce.job.reduces= 20   \
-D mapreduce.task.io.sort.factor=14 \
-D mapreduce.reduce.shuffles.parallelcopies=14 \
-D mapreduce.task.io.sort.mb=110 \
user/hdfs/teragen1-data user/hdfs/sorted-data
```

Fonte: Autoria própria