

UNIVERSIDADE FEDERAL DO PARANÁ
SETOR DE CIÊNCIAS EXATAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MARIA TERESA KRAVETZ ANDRIOLI

OTIMIZAÇÃO DO HADOOP MAPREDUCE ATRAVÉS DO TUNING DOS PARÂMETROS
DE CONFIGURAÇÃO

CURITIBA

2022

MARIA TERESA KRAVETZ ANDRIOLI

OTIMIZAÇÃO DO HADOOP MAPREDUCE ATRAVÉS DO TUNING DOS PARÂMETROS
DE CONFIGURAÇÃO

Trabalho apresentado como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação no curso de Ciência da Computação, Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientador: Prof. Dr. Luiz Carlos P. Albini

CURITIBA

2022

TERMO DE APROVAÇÃO

MARIA TERESA KRAVETZ ANDRIOLI

OTIMIZAÇÃO DO HADOOP MAPREDUCE ATRAVÉS DO TUNING DOS PARÂMETROS DE CONFIGURAÇÃO

Trabalho apresentado como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação no curso de Ciência da Computação, Setor de Ciências Exatas da Universidade Federal do Paraná, pela seguinte banca examinadora:

Prof. Dr. Luiz Carlos P. Albini
Orientador

Professora
UFPR

Professora

Professora

Curitiba, Maio de 2022.

*Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.*

AGRADECIMENTOS

Os agradecimentos principais são direcionados à Gerald Weber, Miguel Frasson, Leslie H. Watter, Bruno Parente Lima, Flávio de Vasconcellos Corrêa, Otavio Real Salvador, Renato Machnievscz¹ e todos aqueles que contribuíram para que a produção de trabalhos acadêmicos conforme as normas ABNT com \LaTeX fosse possível.

Agradecimentos especiais são direcionados ao Centro de Pesquisa em Arquitetura da Informação² da Universidade de Brasília (CPAI), ao grupo de usuários *latex-br*³ e aos novos voluntários do grupo *abnT_EX2*⁴ que contribuíram e que ainda contribuirão para a evolução do *abnT_EX2*.

Os agradecimentos principais são direcionados à Gerald Weber, Miguel Frasson, Leslie H. Watter, Bruno Parente Lima, Flávio de Vasconcellos Corrêa, Otavio Real Salvador, Renato Machnievscz⁵ e todos aqueles que contribuíram para que a produção de trabalhos acadêmicos conforme as normas ABNT com \LaTeX fosse possível.

¹ Os nomes dos integrantes do primeiro projeto *abnT_EX* foram extraídos de <http://codigolivre.org.br/projects/abntex/>

² <http://www.cpai.unb.br/>

³ <http://groups.google.com/group/latex-br>

⁴ <http://groups.google.com/group/abntex2> e <http://abntex2.googlecode.com/>

⁵ Os nomes dos integrantes do primeiro projeto *abnT_EX* foram extraídos de <http://codigolivre.org.br/projects/abntex/>

*“Tudo que posso fazer é aceitar o pandemônio.
Encontrar felicidade na insanidade única de estar aqui agora.”
(Amram e Statsky (2019) - Pandemonium, The Good Place)*

RESUMO

O resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (...) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto. Ter no máximo 500 palavras!!! As palavras chave são separadas por ponto e vírgula.

Palavras-chaves: latex; abntex; editoração de texto.

ABSTRACT

This is the english abstract.

Key-words: latex. abntex. text editoration.

LISTA DE CÓDIGOS

2.1	Exemplo de função Map em pseudocódigo adaptado de (DEAN; GHEMAWAT, 2008)	15
2.2	Exemplo de função Reduce em pseudocódigo adaptado de (DEAN; GHEMAWAT, 2008)	15
3.1	Exemplo de execução do <i>TeraGen</i> adaptado de (WHITE, 2015)	26
3.2	Exemplo de execução do <i>TeraSort</i> adaptado de (WHITE, 2015)	26
3.3	Exemplo de execução do <i>TeraValidate</i> adaptado de (WHITE, 2015)	26

LISTA DE ILUSTRAÇÕES

FIGURA 1 – EXECUÇÃO GENÉRICA DO MAPREDUCE	16
FIGURA 2 – EXEMPLO DE EXECUÇÃO DO MAPREDUCE	17
FIGURA 3 – NOVA ARQUITETURA DO HADOOP 2.0	20
FIGURA 4 – ARQUITETURA DOCKER	20
FIGURA 5 – EXECUÇÃO DO <i>TERASORT</i> COM PARÂMETROS DE <i>TUNING</i> . . .	27

LISTA DE QUADROS

QUADRO 1 – FATORES PARA <i>TUNING</i> DO <i>JOB MAPREDUCE</i>	22
QUADRO 2 – PARÂMETROS DE AJUSTE DA QUANTIDADE DE TAREFAS <i>MAP</i> .	24
QUADRO 3 – PARÂMETROS DE AJUSTE DA QUANTIDADE DE TAREFAS <i>MAP</i> .	25
QUADRO 4 – PARÂMETROS ADICIONAIS DE CONFIGURAÇÃO	25

SUMÁRIO

1	INTRODUÇÃO	12
1.1	CONTEXTO	12
1.2	OBJETIVO	13
1.3	ESTRUTURA DO TRABALHO	13
2	REFERENCIAL TEÓRICO	14
2.1	CLUSTERS	14
2.2	MAPREDUCE	14
2.2.1	Modelo de programação	15
2.2.2	Execução do MapReduce	16
2.3	HADOOP	17
2.3.1	Hadoop Common	18
2.3.2	Hadoop HDFS	18
2.3.3	Hadoop MapReduce	19
2.3.4	Hadoop YARN	19
2.4	VIRTUALIZAÇÃO	20
3	OTIMIZAÇÃO DO MAPREDUCE	22
3.1	PARÂMETROS DO <i>MAPREDUCE</i>	22
3.1.1	Configuração de parâmetros	22
3.1.2	Aplicação do processo de <i>tuning</i>	25
3.2	TERASORT	25
3.3	AMBIENTE EXPERIMENTAL	27
3.4	RESULTADOS	27
4	CONCLUSÃO	28
	REFERÊNCIAS	29
	GLOSSÁRIO	31

1 INTRODUÇÃO

1.1 CONTEXTO

O uso, armazenamento e controle de dados é um tema muito discutido na área de computação desde seus primórdios até os dias de hoje. Dessa forma, muitos métodos e algoritmos e termos surgiram ao longo do tempo com o objetivo de gerenciar de forma eficiente esses dados. O surgimento de tais ferramentas computacionais e métodos de armazenamento é de grande importância para a evolução da área.

Atualmente, os métodos mais comuns são bancos de dados relacionais e *data warehouses* usando computação em nuvem (KUO; KUSIAK, 2019). Além disso, pesquisas nos campos de mineração de dados e aprendizagem de máquina cresceram bastante recentemente, de modo a prover técnicas que permitissem analisar dados complexos e variados entre si (BELCASTRO et al., 2022). Um grande desafio é o fato de algoritmos sequenciais não serem otimizados o suficiente para lidar com dados em grande quantidade. Assim, computadores de alta performance, com múltiplos *cores*, sistemas na nuvem e algoritmos paralelos e distribuídos são usados para lidar com esses empecilhos de *Big Data* (BELCASTRO et al., 2022).

Big Data refere-se a grandes conglomerados de dados complexos sobre os quais não é possível aplicar ferramentas tradicionais de processamento, armazenamento ou análise (KHALEEL; AL-RAWESHIDY, 2018). Estima-se que em 2025 os dados atuais criados, capturados ou replicados atinjam 175 Zettabytes, ou seja 175.000.000.000 Gigabytes (RYDNING, 2018).

A fim de lidar com essa enorme quantidade de dados, foi desenvolvido pelo Google o *MapReduce*, um modelo de programação com uma implementação associada feito para processar e gerar grandes conglomerados de dados. Esse modelo é inspirado nos conceitos de mapear e reduzir, ou seja, aplicar uma operação que conecta cada item da base de dados a um determinado par de chaves e valores, e então aplicar uma operação de reduzir, que une os valores que compartilham chaves (DEAN; GHEMAWAT, 2008). Com essas operações é possível paralelizar dados em grandes quantidades e utilizar mecanismos de reutilização para facilitar a busca e a manipulação destes.

Um dos *frameworks* mais populares que utiliza o *MapReduce* é o *Hadoop*, desenvolvido pela Apache em 2006 e capaz de armazenar e processar de Gigabytes a Petabytes de dados eficientemente, optando por usar múltiplos computadores (*clusters*) em paralelo (WHITE, 2015).

1.2 OBJETIVO

O *Hadoop MapReduce* é um *framework* extremamente personalizável e adaptável. Dessa forma, frequentemente utiliza-se o processo de *tuning*, que consiste em modificar os mais de 190 parâmetros desse *framework* de modo a maximizar a eficiência de um *cluster Hadoop*. Esses parâmetros podem ser alterados em diversas combinações e podem ter efeitos tanto no *cluster* quanto nas tarefas (*jobs*) do processo.

Esse trabalho tem como objetivo avaliar o comportamento do *Hadoop MapReduce* antes e depois do *tuning* de alguns parâmetros de configuração, observando através de métricas de *benchmark* se houve melhora na performance, considerando medidas como tempo e uso de memória.

1.3 ESTRUTURA DO TRABALHO

[TODO: ESTRUTURA DO TRABALHO]

2 REFERENCIAL TEÓRICO

Este capítulo tem como objetivo apresentar detalhadamente os conceitos técnicos que serão utilizados ao longo do trabalho. A seção 2.1 introduz o conceito de *clusters*. A seção 2.2 apresenta o *MapReduce*, o modelo de manipulação de dados feito pelo Google e a seção 2.3 trata do *Hadoop*, o *framework* desenvolvido pela Apache. Por fim, a seção 2.4 explica virtualização, contêineres e a ferramenta Docker.

2.1 CLUSTERS

Um *cluster* é um conjunto de computadores que trabalham juntos paralelamente em uma determinada aplicação. Cada computador desse conjunto é usualmente chamado de nodo. Além disso, existem diversas categorias de *clusters*, dependendo do problema que eles buscam computar (GOLDMAN et al., 2012).

Algumas aplicações comuns de *clusters* são modelagem de clima, simulação de acidentes automotivos, mineração de dados e aplicações da área de astrofísica. Além disso, é comumente visto em aplicações comerciais como bancos e serviços de email (SADASHIV; KUMAR, 2011).

Uma das maiores vantagens desse tipo de instalação é a tolerância de falhas, pois os sistemas conseguem continuar suas tarefas caso um nodo pare de funcionar. Além disso, é altamente escalável com a adição de novos nodos, não precisa de manutenção frequente e tem um gerenciamento centralizado. Por fim, uma das suas maiores possíveis vantagens é o balanceamento de carga, que busca atingir o equilíbrio entre as tarefas de cada nodo de modo a otimizar os recursos (SADASHIV; KUMAR, 2011).

2.2 MAPREDUCE

MapReduce é um modelo de programação associado a uma implementação que tem como objetivo processar, manipular e gerar grandes *datasets* de modo eficiente, escalável e com aplicações no mundo real. As computações acontecem de acordo com funções de mapeamento e redução e o sistema do *MapReduce* paraleliza essas computações entre grandes *clusters*, lidando com possíveis falhas, escalonamentos e uso eficiente de rede e discos (DEAN; GHEMAWAT, 2008).

As operações de mapeamento e redução são baseadas em conceitos presentes em linguagens funcionais e fazem com que seja possível fazer diversas reutilizações, assim lidando com tolerância de falhas (DEAN; GHEMAWAT, 2008).

2.2.1 Modelo de programação

A computação recebe um conjunto de pares (CHAVE, VALOR) e produz um conjunto de pares de (CHAVE, VALOR). O usuário cria as funções *Map* e *Reduce* de acordo com seu uso. *Map* recebe um único par (CHAVE, VALOR) e produz um conjunto intermediário de pares. Em seguida, a biblioteca *MapReduce* agrupa os valores com a mesma chave, os quais servirão de entrada para a função *Reduce*. Nesse momento a função *Reduce* unifica os valores com a mesma chave de modo a criar um conjunto menor, sendo possível dessa forma lidar com listas muito grandes para a memória disponível (DEAN; GHEMAWAT, 2008).

Entre o momento que são executadas as funções *Map* e *Reduce*, existe a fase *Shuffle*, que é criada automaticamente em tempo de execução e executa operações de ordenação (*sort*) e junção (*merge*) (VENNER, 2009). Para White (2015), a operação *Shuffle* é um dos fatores mais influentes no bom desempenho de aplicações *MapReduce*, uma vez que operações de ordenação e junções podem prejudicar ou melhorar muito um algoritmo conforme sua implementação.

Como exemplo, considere-se o problema de contar quantas vezes determinada palavra aparece em um documento. Nesse problema, as funções *Map* e *Reduce* seriam similares aos seguintes pseudocódigos (DEAN; GHEMAWAT, 2008):

```

1 map(String chave, String valor):
2   // chave: nome do documento
3   // valor: conteudo do documento
4
5   para cada palavra W em valor:
6     criaIntermediario(W, 1);

```

CÓDIGO 2.1 – Exemplo de função Map em pseudocódigo adaptado de (DEAN; GHEMAWAT, 2008)

```

1 reduce(String chave, Iterador valores):
2   // chave: uma palavra
3   // valores: lista de contagens
4
5   int resultado = 0;
6   para cada V em valores:
7     resultado = resultado + 1;
8   cria(resultado);

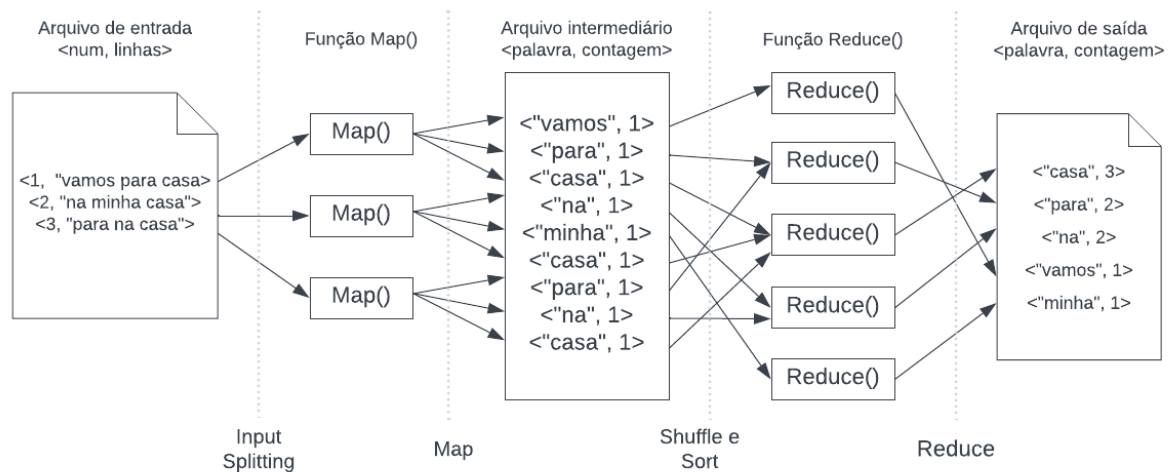
```

CÓDIGO 2.2 – Exemplo de função Reduce em pseudocódigo adaptado de (DEAN; GHEMAWAT, 2008)

A função *Map* gera um objeto intermediário de cada palavra associada a uma lista do seu número de ocorrências no texto e a função *Reduce* soma os valores até que essas ocorrências por palavras sejam totalizadas. Além disso, o usuário cria uma configuração de *MapReduce* com os parâmetros de entrada e saída e eventuais parâmetros de *tuning*.

Para exemplificar ainda mais, considere um arquivo de texto com três linhas nas quais estão as seguintes frases, respectivamente, uma em cada linha: "vamos para casa", "na minha casa", "para na casa". Nesse exemplo, a função *Map* é chamada três vezes, uma para cada linha, gerando os pares (CHAVE, VALOR) intermediários, um para cada palavra encontrada no texto, como é mostrado na FIGURA 1. Para cada palavra distinta ("vamos", "para", "casa", "na", "minha"), é executada a função *Reduce*, que soma quantas vezes cada uma dessas palavras apareceu no texto e gera um arquivo de saída.

FIGURA 1 – EXECUÇÃO GENÉRICA DO MAPREDUCE



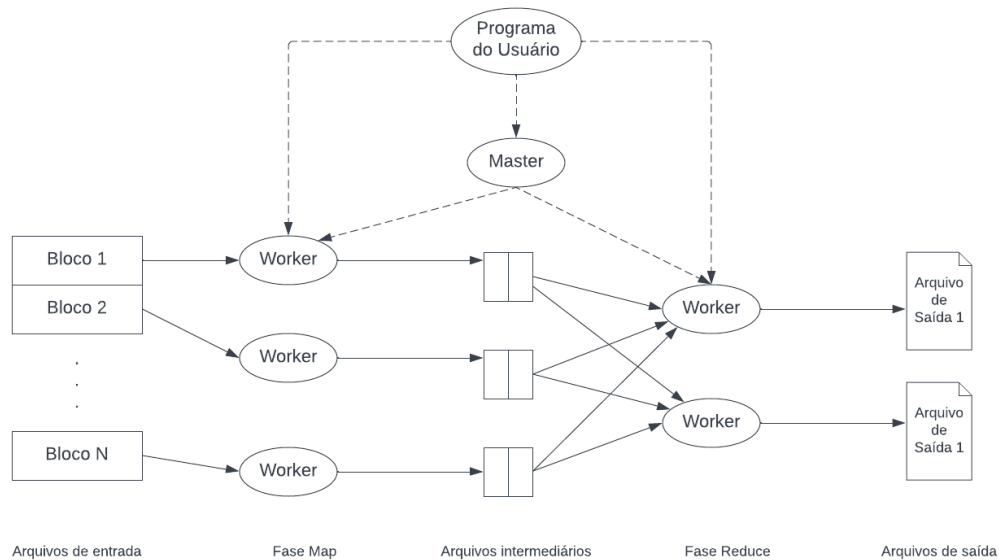
FONTE: A autora (2022)

2.2.2 Execução do MapReduce

O *MapReduce* funciona usando uma estrutura Cliente/Servidor sobre um *cluster* que, segundo Dean e Ghemawat (2008), consiste em primeiramente particionar os dados de entrada em blocos de tamanho já definidos e depois distribuir cópias do programa *MapReduce* entre cada um desses blocos. Existe uma cópia Master, que é responsável por repartir as tarefas (*tasks*), enquanto as demais cópias - denominadas *Workers* - recebem da Master as tarefas junto com os arquivos de entrada. Ao finalizar a execução de uma tarefa do tipo *Map*, a cópia *Worker* responsável repassa a Master os arquivos de saída e esta repassa a um *Worker* esse arquivo com a tarefa de *Reduce*. Por fim, esse worker executa a redução, lendo os pares intermediários que passaram pela fase *Shuffle* e agrupando as instâncias de mesma chave. Quando todas as tarefas *Map* e *Reduce* forem executadas, o programa é finalizado.

Na FIGURA 1 foi possível ver como o *MapReduce* funcionaria em pequena escala. Uma das maiores vantagens do *MapReduce* é, no entanto, sua escalabilidade, visto que ele permite uma execução distribuída entre uma grande quantidade de nodos. A FIGURA 2 representa uma execução genérica do *MapReduce*, descrita no parágrafo acima.

FIGURA 2 – EXEMPLO DE EXECUÇÃO DO MAPREDUCE



FONTE: Adaptado de (DEAN; GHEMAWAT, 2008)

2.3 HADOOP

Hadoop é um *framework* desenvolvido na linguagem Java pela Apache Software Foundation com os seguintes princípios arquiteturais, segundo Navarro Belmonte (2018):

- A possibilidade de escalar o sistema ao adicionar nodos no *cluster*.
- Possibilidade de funcionar bem em *hardware* que não necessite ser caro e de luxo.
- Tolerância a falhas, com implementações que as identificam e permitem que o sistema funcione independente delas acontecerem.
- Fornecimento de serviços para que o usuário foque no problema que deseja resolver.

Esse *framework* disponibiliza ferramentas para que o usuário possa escrever as funções necessárias em diversas linguagens de programação, conforme a necessidade do programador. O *framework* funciona na mesma estrutura de Cliente/Servidor apresentada anteriormente, utilizada pelo *MapReduce*. Além disso, oferece ao programador um sistema paralelo e distribuído (*Hadoop HDFS*), com os recursos ocultos ao usuário, mas capaz de lidar com a comunicação entre as máquinas e quaisquer falhas que possam vir a ocorrer e o escalonamento das tarefas.

Além do *Hadoop Map Reduce* e do *Hadoop HDFS*, existem outros subprojetos do Hadoop que compõem sua estrutura principal: o *Hadoop Common*, que fornece ferramentas comuns aos outros subprojetos o *Hadoop YARN*, um *framework* para escalonamento de tarefas e gerenciamento de recursos em *clusters*.

2.3.1 Hadoop Common

Esse subprojeto contém os utilitários e bibliotecas comuns aos outros subprojetos. Por exemplo, funções de manipulação de arquivos, funções auxiliares de serialização de dados, etc (GOLDMAN et al., 2012).

2.3.2 Hadoop HDFS

Segundo Borthakur (2020) o *Hadoop HDFS* é um sistema de arquivos distribuídos criado para funcionar em *hardware* facilmente obtido e relativamente barato. Suas características principais são a alta capacidade de lidar com falhas e a possibilidade de ser usado com aplicações que possuem grande quantidades de dados como entrada.

Uma instância HDFS é composta de centenas ou milhares de máquinas, cada uma responsável por armazenar uma parte dos dados do sistema. Dessa forma, a rápida detecção e recuperação de falhas é essencial para sua estrutura. Seu *design* foi pensado em aplicações de processamento de dados em blocos e o tamanho de seus arquivos pode variar entre Giga e Terabytes. Além disso, é adaptado para funcionar em diferentes plataformas e prover interfaces que possibilitam mover a aplicação para perto dos dados, permitindo que qualquer operação computacional aplicada seja muito mais eficiente (BORTHAKUR, 2020).

O *Hadoop HDFS* também possui uma estrutura Cliente/Servidor, em que o *Namenode* - responsável por gerenciar o sistema e regular o acessos aos arquivos - é o nodo Master e os *Datanodes* - responsáveis por gerenciar o armazenamento dos nodos aos quais eles estão conectados - são os nodos Worker. O sistema é implementado usando uma estrutura comum de diretórios na qual é possível criar, mover, renomear e remover arquivos, mas ainda não implementa funções como quota de usuários, permissões de acesso ou *links* simbólicos (BORTHAKUR, 2020).

Uma das características essenciais desse sistema é sua capacidade de lidar com grandes quantidades de dados. Segundo Borthakur (2020), isso é feito através do armazenamento dos arquivos como uma sequência de blocos, cujo tamanho e fator de replicação são configuráveis pelo usuário, mas possuem um valor padrão de 64MB. Periodicamente, o *Namenode* recebe dos *Datanodes* um sinal indicando se o funcionamento está correto. O posicionamento e a quantidade de réplicas de um bloco é crítica na análise da boa performance do HDFS, e é um dos fatores que o diferencia de outros sistemas de arquivos distribuídos. Quando executada de forma otimizada, pode aumentar confiabilidade, disponibilidade e uso de redes do sistema.

2.3.3 Hadoop MapReduce

O *Hadoop MapReduce* é um *framework* que implementa o modelo de programação *MapReduce* para facilitar a criação de aplicações que são capazes de processar grandes quantidades de dados em paralelo em *clusters* de uma forma confiável e com tolerância a falhas.

Esse *framework* é constituído de um *Job* responsável por dividir os arquivos de entrada em blocos independentes que serão processados pelas tarefas (*tasks*) *Map*, ordenados na fase *Shuffle* e inseridos nas tarefas *Reduce* de forma paralela. Os arquivos de entrada e saída são armazenados no sistema de arquivos e o sistema é responsável pelo escalonamento, agendamento e reexecução de tarefas que tenham falhado (MAPREDUCE. . . , 2022).

A estrutura Cliente/Servidor tem como nodo Master o *JobTracker* e como nodos Worker os *TaskTrackers*. Como apresentado anteriormente, o nodo Master designa as tarefas e os nodos Worker as executam. A aplicação do programador fornece o local dos arquivos de entrada e saída, a implementação das funções de *Map* e *Reduce* e outros parâmetros de configuração do *Job*. Então, o cliente *Hadoop* envia o *Job* e o arquivo de configuração para o *JobTracker* que distribui as tarefas e controla o funcionamento desse *Job*.

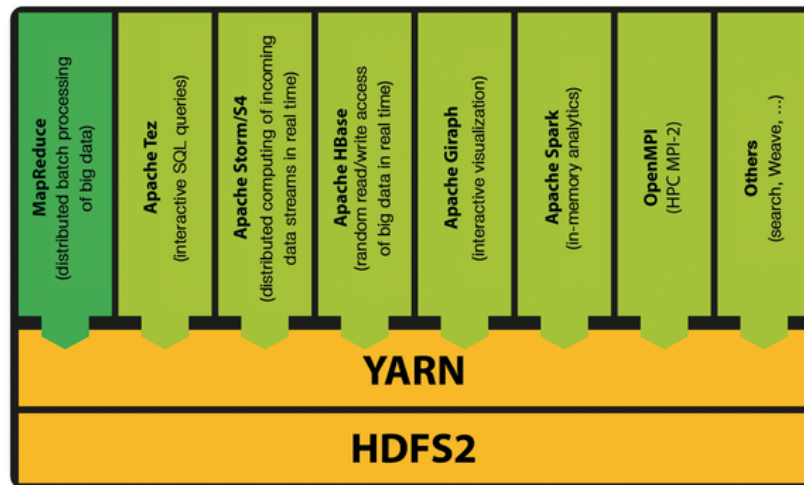
2.3.4 Hadoop YARN

O *Hadoop YARN* é um subprojeto que tem como objetivo dividir as funcionalidades de gerenciamento de recursos de escalonamento de tarefas em módulos diferentes, tendo então um gerenciador global de recursos e um gerenciador local por aplicação (GOLDMAN et al., 2012).

O gerenciador global trabalha em conjunto com um gerenciador de nodos responsável pelos contêineres e pelo monitoramento de uso de recursos, como CPU, memória, uso de disco e uso de redes, assim como o repasse dessas informações para o gerenciador global (APACHE. . . , 2022).

O YARN foi adicionado ao *Hadoop* versão 2.0 permitindo a separação das camadas de gerenciamento de recursos que possam ser alocados pela aplicação. Com essa camada independente, ilustrada na FIGURA 3, as aplicações *MapReduce* podem ser utilizadas em conjunto com aplicações não *MapReduce*. Além disso, esse formato de implementação possibilita economizar custos com o melhor aproveitamento dos nodos (KOBYLINSKA; MARTINS, 2014).

FIGURA 3 – NOVA ARQUITETURA DO HADOOP 2.0



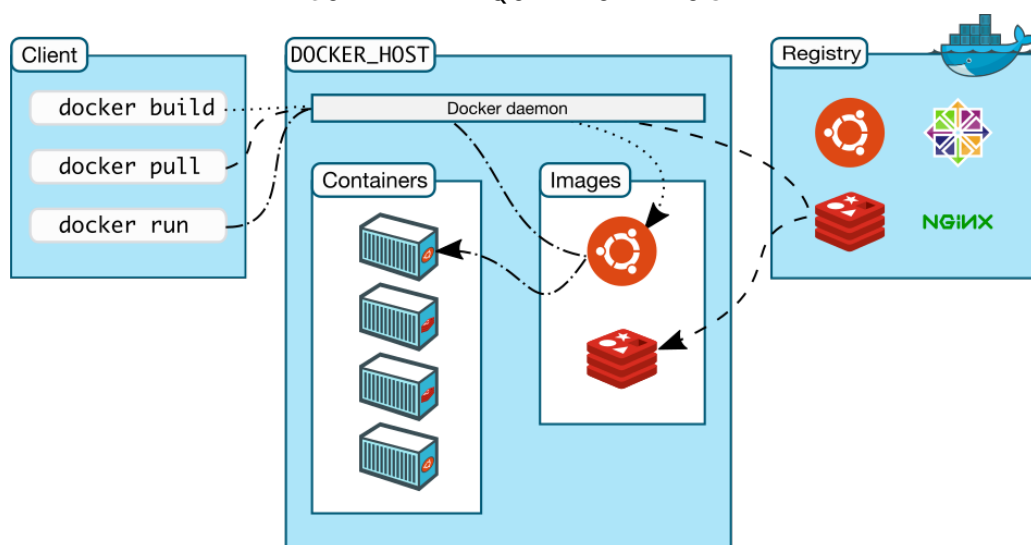
FONTE: (KOBYLINSKA; MARTINS, 2014)

2.4 VIRTUALIZAÇÃO

Virtualização é o processo de criar um ambiente ou uma versão virtual de algum componente computacional, tal como *hardwares*, dispositivos de armazenamento e recursos de rede. A virtualização permite que haja economia nos custos de *hardware*, melhoria na recuperação em caso de falhas e redução da necessidade de espaço físico para *datacenters* (PORTNOY, 2012).

Uma das técnicas da virtualização é a utilização de contêiners. Contêiners, uma virtualização a nível de sistema, permitem que existam múltiplos espaços do usuário por cima de um determinada kernel de sistema.

FIGURA 4 – ARQUITETURA DOCKER



FONTE: (DOCKER..., 2022)

Docker é uma ferramenta que tem como objetivo automatizar a implantação de aplicações em contêiners, cuja estrutura geral está ilustrada na FIGURA 4. Essa ferramenta empilha uma implantação de uma aplicação em cima de um ambiente de execução em um contêiner, ou seja, simula um ambiente virtual de modo que o programador possa trabalhar com sua aplicação em produção de forma extremamente configurável para as suas necessidades. Para isso, o Docker utiliza um recurso de imagem, que se refere aos arquivos de sistemas que determinada aplicação necessita para ser executada. Então, esses arquivos são empilhados entre si e servem como uma receita para construção de um ou de múltiplos contêiners (TURNBULL, 2014).

3 OTIMIZAÇÃO DO MAPREDUCE

Como visto anteriormente, o *MapReduce* é um processo com diversas etapas, e portanto, há muitas permutações possíveis das suas configurações que permitem a melhora da performance que, através da modificação dos parâmetros de configuração, é chamada de *tuning*. Para White (2015), os fatores do *Job* mais relevantes a serem considerados com objetivo de obter aumento da performance são exibidos no QUADRO 1.

QUADRO 1 – FATORES PARA *TUNING* DO *JOB MAPREDUCE*

ÁREA A SER OTIMIZADA	COMO OTIMIZAR
Quantidade de mapeadores	Verificar se é possível diminuir a quantidade de execuções da função <i>Map</i> de modo que cada uma seja executada por mais tempo. O tempo médio recomendado na literatura é de 1 minuto.
Quantidade de redutores	Verificar se mais de um redutor está sendo utilizado. O recomendado é que cada tarefa seja executada em média durante 5 minutos e produza 1 bloco de dados.
Uso de combinadores	Verificar se é possível utilizar algum combinador de dados de modo que a quantidade de data passada à função <i>Shuffle</i> seja menor.
Compressão	Usualmente, o tempo de execução de um <i>Job</i> é diminuído ao usar compressão de dados.
Ajustes na parte <i>Shuffle</i>	A parte <i>Shuffle</i> do processo possui vários parâmetros de <i>tuning</i> de memória que podem ser utilizados para a melhora da performance do <i>Job</i> .

FONTE: Adaptado de (WHITE, 2015)

3.1 PARÂMETROS DO *MAPREDUCE*

O *tuning* pode ser realizado através da avaliação e mudança dos parâmetros de configuração do *MapReduce*. Cada parâmetro tem um objetivo específico e pode melhorar uma característica do processo. Algumas variáveis mudam configurações no *Job* e algumas afetam o *cluster* diretamente.

Hadoop foi criado para processar grandes arquivos de entradas e é otimizado para *clusters* em máquinas heterogêneas, ou seja, sistemas que usam mais de um tipo de processador com o objetivo de melhorar a performance. Cada *Job* segue a seguinte sequência de passos: configuração, fase *shuffle/sort* e fase *reduce*. O *Hadoop* é responsável por configurar e gerenciar cada um desses passos (VENNER, 2009).

White (2015) explica o processo de *tuning* e explicar os parâmetros específicos para otimização da cada passo, detalhado nas seções a seguir.

3.1.1 Configuração de parâmetros

O objetivo principal a ser atingido durante o *tuning* é possibilitar que a fase *Shuffle* tenha a maior quantidade de memória disponível, ao mesmo tempo que as fases *Map* e *Reduce* te-

tenham memória suficiente para funcionar propriamente. A quantidade de memória disponibilizada a cada *Java Virtual Machine* é determinada pelo parâmetro *mapred.child.java.opts* e também pode ser definida separadamente para cada fase com os parâmetros *mapreduce.reduce.java.opts* e *mapreduce.map.java.opts* (WHITE, 2015).

A fase *Map* recebe de entrada um arquivo e o parâmetro *dfs.block.size* é responsável por determinar o tamanho do bloco em *bytes* sobre o qual este arquivo será dividido. Ainda nessa fase, os pares (CHAVE, VALOR) são particionados, e essas partições são ordenadas na fase *Shuffle*. O arquivo criado para cada partição é chamado de *spill*. Para cada tarefa *Reduce* existe um *spill*, que passará por uma ordenação do tipo *Merge Sort*, na segunda etapa da fase *Shuffle*, chamada de *Sort* (VENNER, 2009).

Segundo White (2015), a melhor performance da fase *Map* pode ser obtida através da minimização da quantidade de *spills*, cujos parâmetros de controles são *mapreduce.task.io.sort.factor* - número máximo de entradas para a função *Merge Sort* - e *mapreduce.task.io.sort.mb* - tamanho do *buffer* de memória para a saída da função *Map*. O último é especialmente importante e deve ser aumentado sempre que possível. Quando o *buffer* atinge a capacidade percentual determinada pelo parâmetro (*mapreduce.map.sort.spill.percent*) ocorre um vazamento de memória e os conteúdos restantes do arquivo de saída são colocados no disco (no arquivo chamado de *spill*).

Caso exista mais de uma determinada quantidade de arquivos *spill* (quantidade determinada pela propriedade *mapreduce.map.combine.minspills*), a função combinadora é executada novamente antes de ser criado o arquivo de saída. Uma outra otimização possível é o uso de compressores de dados nos arquivos de saída da fase *Map*, processo facilmente habilitado através dos parâmetros *mapreduce.map.output.compress* - valor booleano que habilita a compressão - e *mapreduce.map.output.compress.codec* - classe que vai realizar a compressão (WHITE, 2015).

A fase *Reduce* é otimizada quando os dados intermediários são armazenados na memória, o que não acontece por padrão, visto que sem a alteração dos parâmetros toda a memória é alocada para a fase *Reduce* em si. As propriedades que podem ser alteradas para atingir esse objetivo são *mapreduce.reduce.merge.inmem.threshold*, *mapreduce.reduce.input.buffer.percent* e *mapreduce.reduce.shuffle.merge.percent*, cujos valores ótimos são 0 e 1.0, respectivamente (WHITE, 2015).

Um dos parâmetros que pode ser otimizado na fase *Reduce* é o *mapreduce.reduce.shuffle.parallelcopies*, que determina a quantidade de tarefas que serão executadas em paralelo pelos *reducers* para copiar os arquivos de saída das tarefas *Map* quando estas são finalizadas e seu valor ótimo depende da quantidade de dados que já passou pela fase *Shuffle* (LI et al., 2014).

Na parte do processo onde as cópias dos arquivos de saída da fase *Map* são feitas, o

tamanho do *buffer* é controlado pela propriedade *mapreduce.reduce.shuffle.input.buffer.percent*, que especifica a proporção do *heap* que será usada para a finalidade mencionada. Quando esse *buffer* atinge um número determinado por *mapreduce.reduce.shuffle.merge.percent* ou *mapreduce.reduce.merge.inmem.threshold*, o restante dos arquivos é alocado no disco (WHITE, 2015).

Para cada gerenciador de nodo, o número de partições do arquivo de saída disponibilizados para a fase *Reduce* é determinado pela propriedade *mapreduce.shuffle.max.threads*. O valor padrão de 0 determina que o número máximo de tarefas é o dobro do número de processadores da máquina (VENNER, 2009).

Os quadros a seguir resumem os parâmetros mencionados previamente, assim como outros parâmetros relevantes e os valores padrões de cada um:

QUADRO 2 – PARÂMETROS DE AJUSTE DA QUANTIDADE DE TAREFAS MAP

PARÂMETRO	DESCRIÇÃO	VALOR PADRÃO
mapreduce.task.io.sort.mb	Tamanho em <i>bytes</i> usado no <i>buffer</i> de memória na ordenação na saída da função <i>Map</i> .	100
mapreduce.task.io.sort.factor	Número máximo de arquivos para juntar simultaneamente durante a ordenação.	10
mapreduce.map.sort.spill.percent	Limite de uso do <i>buffer</i> de memória que pode ser usado antes que os dados sejam colocados em disco.	0.80
mapreduce.map.combine.minspills	Número mínimo de arquivos de vazamento necessário para o combinador funcionar (se um combinador for especificado).	3
mapreduce.map.output.compress	Define se a saída da função <i>Map</i> será comprimida.	false
mapreduce.map.output.compress	Codificador usado na compressão.	DefaultCodec (classe)
mapreduce.shuffle.max.threads	Número de tarefas <i>Worker</i> por gerenciador de nodos. Esse parâmetro não funciona por <i>Job</i> e sim por <i>cluster</i> .	0

FONTE: Adaptado de (WHITE, 2015)

QUADRO 3 – PARÂMETROS DE AJUSTE DA QUANTIDADE DE TAREFAS MAP

PARÂMETRO	DESCRIÇÃO	VALOR PADRÃO
mapreduce.task.io.sort.factor	Número máximo de arquivos para juntar simultaneamente durante a ordenação.	10
mapreduce.reduce.shuffle.parallelcopies	Número de tarefas usadas para copiar saída de funções <i>Map</i> para funções <i>Reduce</i> .	5
mapreduce.reduce.shuffle.maxfetchfailures	Número de vezes que uma tarefa <i>Reduce</i> tenta obter arquivo de entrada.	10
mapreduce.reduce.shuffle.input.buffer.percent	Porcentagem de tamanho do <i>heap</i> a ser alocada para a saída da fase <i>Map</i> .	0.70
mapreduce.reduce.shuffle.merge.percent	Limite porcentual da saída da fase <i>Map</i> para iniciar o processo de juntar saídas.	0.66
mapreduce.reduce.merge.inmem.threshold	Quantidade de saídas da função <i>Map</i> para a saída da fase <i>Map</i> . Se for igual ou menor a zero, esse fator é definido apenas pelo <i>mapreduce.reduce.shuffle.merge.percent</i> .	1000
mapreduce.reduce.input.buffer.percent	Percentual que determinar o tamanho do <i>heap</i> que será utilizado para armazenar saídas da função <i>Map</i> durante a fase <i>Reduce</i> .	0.0

FONTE: Adaptado de (WHITE, 2015)

QUADRO 4 – PARÂMETROS ADICIONAIS DE CONFIGURAÇÃO

PARÂMETRO	DESCRIÇÃO	VALOR PADRÃO
dfs.block.size	Tamanho do bloco em <i>bytes</i> sobre o qual arquivo de entrada do <i>Map</i> será.	67108864 bytes
mapred.child.java.opts	Memória alocada para a <i>Java Virtual Machine</i> que executará o processo de <i>MapReduce</i> .	200MB
mapreduce.map.java.opts	Memória alocada para a fase de <i>Map</i> .	200MB
mapreduce.reduce.java.opts	Memória alocada para a fase de <i>Reduce</i> .	200MB

FONTE: A autora (2022)

3.1.2 Aplicação do processo de *tuning*

3.2 TERASORT

A técnica de *benchmarking* consiste na execução, por diversas vezes, de um programa (no caso do *MapReduce*, de um *Job*), a fim de testar se os resultados obtidos são os esperados. Esse processo é eficiente porque é possível comparar os diversos resultados obtidos e obter avaliações de performance (WHITE, 2015).

O *Hadoop* possui várias métricas de *benchmarks* embutidas que podem ser utilizadas

para melhorar o funcionamento do *Job*, cada uma delas com o objetivo de monitorar um fator diferente, como por exemplo, TestDFSIO - responsável por testar a performance dos dispositivos de entrada e saída - e MRBench/NNBench - que testam em conjunto vários *Jobs* pequenos múltiplas vezes (WHITE, 2015).

Neste trabalho será utilizado a ferramenta *TeraSort*. Para White (2015), ela é extremamente eficaz no *benchmarking* do *HDFS* e *MapReduce* em conjunto, já que executa e avalia todos os passos do paradigma *MapReduce*. Essa ferramenta funciona em três etapas:

- *TeraGen* executa um *Job* só de funções *Map* que cria um conjunto de dados binários aleatórios. A execução desse comando é exemplificada no CÓDIGO 3.1.

```
1  hadoop jar hadoop-mapreduce-examples-*.jar \
2  teragen <numero de linhas de 100 bytes cada> <diretorio de saida>
```

CÓDIGO 3.1 – Exemplo de execução do *TeraGen* adaptado de (WHITE, 2015)

- *TeraSort* executa a ordenação dos dados. É nesse passo que é avaliada a performance do *MapReduce*, pois é aqui que as operações do paradigma são executadas. A execução desse comando é exemplificada no CÓDIGO 3.2.

```
1  hadoop jar hadoop-mapreduce-examples-*.jar \
2  terasort <diretorio de entrada> <diretorio de saida>
```

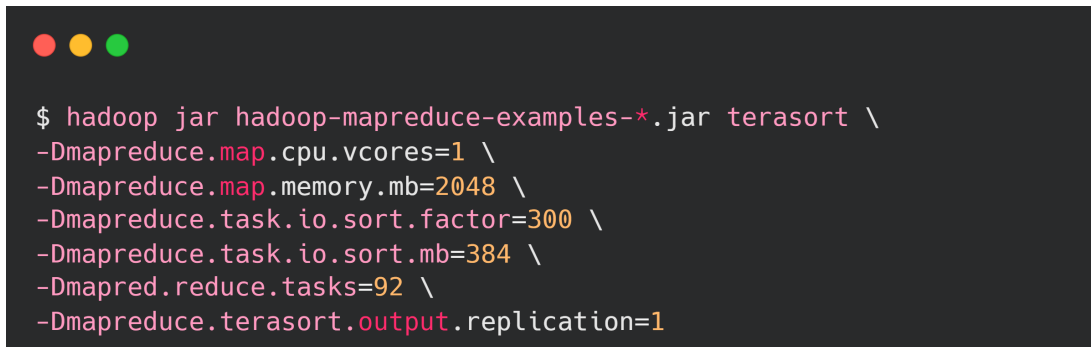
CÓDIGO 3.2 – Exemplo de execução do *TeraSort* adaptado de (WHITE, 2015)

- *TeraValidate* executa checagens nos dados ordenados resultantes da fase anterior para verificar se a ordenação foi feita corretamente. A execução desse comando é exemplificada no CÓDIGO 3.3.

```
1  hadoop jar hadoop-mapreduce-examples-*.jar \
2  teravalidate <arquivo de entrada (diretorio de saida do
    terasort)> <diretorio de saida>
```

CÓDIGO 3.3 – Exemplo de execução do *TeraValidate* adaptado de (WHITE, 2015)

Os parâmetros de configuração do *tuning* podem ser usados no comando que executa o *TeraSort*, processo exemplificado na FIGURA 5.

FIGURA 5 – EXECUÇÃO DO *TERASORT* COM PARÂMETROS DE *TUNING*


```
$ hadoop jar hadoop-mapreduce-examples-*.jar terasort \
-Dmapreduce.map.cpu.vcores=1 \
-Dmapreduce.map.memory.mb=2048 \
-Dmapreduce.task.io.sort.factor=300 \
-Dmapreduce.task.io.sort.mb=384 \
-Dmapred.reduce.tasks=92 \
-Dmapreduce.terasort.output.replication=1
```

FONTE: A autora (2022)

3.3 AMBIENTE EXPERIMENTAL

O ambiente no qual serão realizadas as execuções e testes tem as seguintes configurações: LENOVO Ideapad 310 com processador Intel(R) Core(TM) i5-6200U, 2.30GHz de velocidade de processamento, memória RAM de 8GB, armazenamento de disco de 1TB e SSD Kingston A400 de 480, com leitura de 500 MB/s e gravação de 450 MB/s.

No entanto, a imagem Docker é executada no Windows WSL2, Subsistema do Windows para Linux, que possibilita aos programadores executar um ambiente Linux mesmo usando um sistema Windows (WINDOWS..., 2022). Nesse ambiente virtual está sendo executado Linux na distribuição Ubuntu 20.04, Docker na versão 20.10.12 e Hadoop na versão 3.2.1.

Usando a imagem do *Hadoop* para Docker disponibilizada pelo Big Data Europe (BIGDATAEUROPE, 2020), foi configurado um *cluster Hadoop* em um contêiner Docker com 3 *datanodes (workers)*, um *namenode HDFS (master)*, um gerenciador de recursos YARN, um servidor com histórico de operações e um gerenciador de nodos.

3.4 RESULTADOS

benchmark tuning hardware datacenters buffer byte heap Merge Sort Java Virtual Machine

4 CONCLUSÃO

REFERÊNCIAS

AMRAM, M.; STATSKY, J. **Pandemonium**. v. 3. [S.l.: s.n.], jan. 2019. Citado 1 vez na página 5.

APACHE Hadoop YARN. [S.l.: s.n.], 2022. Disponível em: <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html>. Citado 1 vez na página 19.

BELCASTRO, L.; CANTINI, R.; MAROZZO, F.; ORSINO, A.; TALIA, D.; TRUNFIO, P. Programming big data analysis: principles and solutions. **Journal of Big Data**, SpringerOpen, v. 9, n. 1, p. 1–50, 2022. Citado 2 vez na página 12.

BIGDATAEUROPE. **docker-hadoop**. [S.l.]: GitHub, 2020. Disponível em: <https://github.com/big-data-europe/docker-hadoop>. Citado 1 vez na página 27.

BORTHAKUR, D. **HDFS architecture guide**. [S.l.: s.n.], out. 2020. Disponível em: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Citado 4 vez na página 18.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Communications of the ACM**, ACM New York, NY, USA, v. 51, n. 1, p. 107–113, 2008. Citado 12 vezes nas páginas 8, 12, 14–17.

DOCKER Overview. [S.l.]: Docker Inc., abr. 2022. Disponível em: <https://docs.docker.com/get-started/overview/>. Citado 0 vez na página 20.

GOLDMAN, A.; KON, F.; JUNIOR, F. P.; POLATO, I.; FÁTIMA PEREIRA, R. de. Apache Hadoop: conceitos teóricos e práticos, evolução e novas possibilidades. **XXXI Jornadas de atualizações em informática**, p. 88–136, 2012. Citado 3 vezes nas páginas 14, 18, 19.

KHALEEL, A.; AL-RAWESHIDY, H. Optimization of computing and networking resources of a Hadoop cluster based on software defined network. **IEEE Access**, IEEE, v. 6, p. 61351–61365, 2018. Citado 1 vez na página 12.

KOBYLINSKA, A.; MARTINS, F. **Big data tools for midcaps and others**. [S.l.: s.n.], 2014. Disponível em: <https://www.admin-magazine.com/Archive/2014/20/Big-data-tools-for-midcaps-and-others>. Citado 1 vezes nas páginas 19, 20.

KUO, Y.-H.; KUSIAK, A. From data to big data in production research: the past and future trends. **International Journal of Production Research**, Taylor & Francis, v. 57, n. 15-16, p. 4828–4853, 2019. DOI: [10.1080/00207543.2018.1443230](https://doi.org/10.1080/00207543.2018.1443230). eprint:

<https://doi.org/10.1080/00207543.2018.1443230>. Disponível em:

<https://doi.org/10.1080/00207543.2018.1443230>. Citado 1 vez na página 12.

LI, M.; ZENG, L.; MENG, S.; TAN, J.; ZHANG, L.; BUTT, A. R.; FULLER, N. Mronline: Mapreduce online performance tuning. In: PROCEEDINGS of the 23rd international symposium on High-performance parallel and distributed computing. [S.l.: s.n.], 2014. P. 165–176. Citado 1 vez na página 23.

MAPREDUCE Tutorial. [S.l.: s.n.], 2022. Disponível em:

https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html. Citado 1 vez na página 19.

NAVARRO BELMONTE, V. P. **Improving Real Time Tuning on YARN**. 2018. Tese (Doutorado) – Carleton University. Citado 1 vez na página 17.

PORTNOY, M. **Virtualization essentials**. [S.l.]: John Wiley & Sons, 2012. v. 19. Citado 1 vez na página 20.

RYDNING, D. R.-J. G.-J. The digitization of the world from edge to core. **Framingham: International Data Corporation**, p. 16, 2018. Citado 1 vez na página 12.

SADASHIV, N.; KUMAR, S. D. Cluster, grid and cloud computing: A detailed comparison. In: IEEE. 2011 6th international conference on computer science & education (ICCSE). [S.l.: s.n.], 2011. P. 477–482. Citado 2 vez na página 14.

TURNBULL, J. **The Docker Book: Containerization is the new virtualization**. [S.l.]: James Turnbull, 2014. Citado 1 vez na página 21.

VENNER, J. **Pro hadoop**. [S.l.]: Apress, 2009. Citado 4 vezes nas páginas 15, 22–24.

WHITE, T. **Hadoop: The definitive guide**. [S.l.]: "O'Reilly Media, Inc.", 2015. Citado 21 vezes nas páginas 8, 12, 15, 22–26.

WINDOWS Subsystem for Linux Documentation. [S.l.]: Microsoft, abr. 2022. Disponível em: <https://docs.microsoft.com/en-us/windows/wsl/>. Citado 1 vez na página 27.

