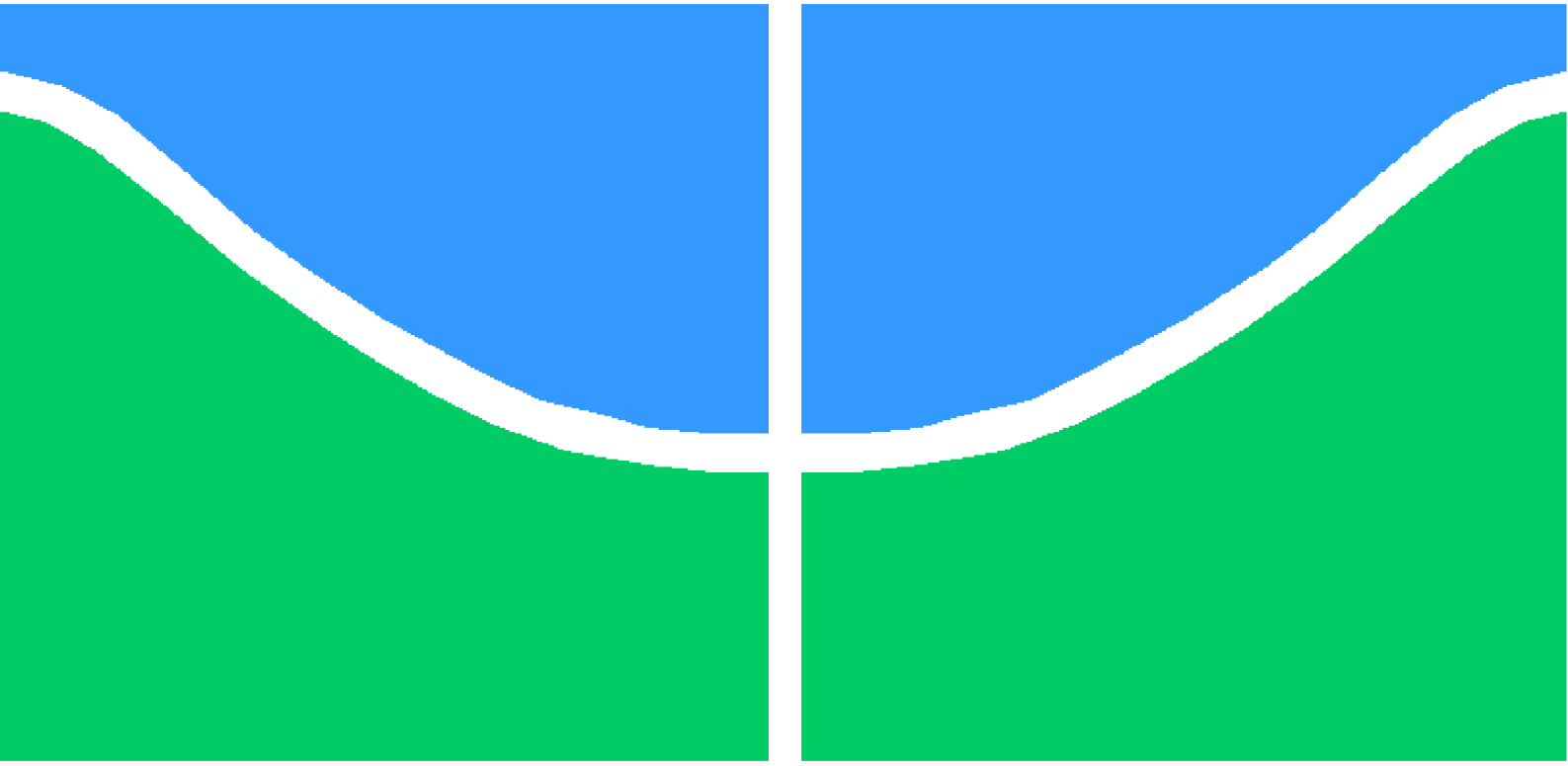

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Desenvolvimento de Software no Contexto Big Data

Autor: Guilherme de Lima Bernardes
Orientador: Prof. Dr. Fernando William Cruz

Brasília, DF
2015

Guilherme de Lima Bernardes

Desenvolvimento de Software no Contexto Big Data

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Fernando William Cruz

Brasília, DF

2015

Guilherme de Lima Bernardes

Desenvolvimento de Software no Contexto Big Data / Guilherme de Lima Bernardes. – Brasília, DF, 2015-

95 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Fernando William Cruz

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. Big Data. 2. Hadoop. I. Prof. Dr. Fernando William Cruz. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Desenvolvimento de Software no Contexto Big Data

CDU 02:141:005.6

Guilherme de Lima Bernardes

Desenvolvimento de Software no Contexto Big Data

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, :

Prof. Dr. Fernando William Cruz
Orientador

Prof. Dr. Paulo Roberto Miranda Meirelles
Convidado 1

Prof. Dr. Nilton Correia da Silva
Convidado 2

Brasília, DF
2015

Resumo

O armazenamento de informações em formato digital tem crescido consideravelmente nos últimos anos. Não apenas pessoas são responsáveis por produzir dados, equipamentos eletrônicos também se tornaram grandes geradores de registros, como servidores, aparelhos de GPS, microcomputadores espalhados nos mais variados contextos, entre uma infinidade de aplicações. O termo Big Data se refere a toda esta quantidade de dados que se encontra na ordem de petabytes e não pode ser analisada pelos métodos tradicionais. Neste trabalho, apresenta-se um estudo sobre uma das mais conhecidas arquiteturas para solucionar esses problemas, o software Hadoop. O desenvolvimento para o paradigma *MapReduce* é abordado, assim como os projetos que são construídos no topo do sistema Hadoop, provendo serviços em um nível de abstração maior. Além da etapa de pesquisa, uma arquitetura de aplicação Big Data voltada para um estudo de caso real é definida e implementada, a qual envolve a extração e análise de publicações de redes sociais com foco na política brasileira.

Palavras-chaves: Big Data. Hadoop. Computação Distribuída.

Abstract

The storage of information in digital format has grown considerably in recent years. Not only people are responsible for producing data, electronic equipment have also become major generators of records, such as servers, GPS devices, computers scattered in various contexts, among a multitude of applications. The term Big Data refers to all this amount of data that is on the order of petabytes and can not be analyzed by traditional methods. In this paper, we present a study of one of the most known architectures to solve these problems, Hadoop software. The development MapReduce paradigm is discussed, as well as designs that are built on top of Hadoop system providing service at a higher abstraction level. Beyond the research stage, an Big Data application architecture facing a real case study is defined and implemented, which involves the extraction and analysis of social networking publications focusing on Brazilian politics.

Key-words: Big Data. Hadoop. Distributed Computing.

Lista de ilustrações

Lista de tabelas

Lista de quadros

Lista de abreviaturas e siglas

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
BI	Business Intelligence
CPU	Central Processing Unit
CSS	Cascading Style Sheets
GFS	Google File System
HDFS	Hadoop Distributed File System
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDC	International Data Corporation
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OSI	Open Source Iniciative
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RPC	Remote Procedure Call
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	Structured Query Language
TCC	Trabalho de Conclusão de Curso
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
XML	Extensible Markup Language

Sumário

1 Introdução

A tecnologia nunca foi tão presente na sociedade como nos dias atuais. Não apenas pessoas são responsáveis por produzir informações, equipamentos eletrônicos também tornaram-se grandes criadores de dados, como registros de *logs* de servidores, sensores que são instalados nos mais variados contextos, entre uma infinidade aplicações. Mensurar o volume de todos estes registros eletrônicos não é uma tarefa fácil. White (2012) apresenta alguns exemplos de como grandes empresas geram quantidades extremamente grandes de dados. O Facebook[1] chega a armazenar 10 bilhões de fotos, totalizando 1 petabyte, já a organização Internet Archive[2] contém cerca de 2 petabytes de dados, com uma taxa de crescimento de 20 terabytes por mês.

Segundo levantamento feito pela IDC (GANTZ, 2011), a quantidade de informações capturadas, criadas, ou replicadas no universo digital ultrapassou a barreira de 1 zettabyte, equivalente a 1 trilhão de gigabytes. Entre o período de 2006 a 2011 o volume total de registros se multiplicava por nove a cada ano. O número de bits de todos estes dados pode ser comparado ao número de estrelas contidas no universo físico.

O grande problema consiste em tornar processável toda esta gama de informações, que podem estar persistidas de forma estruturada, semi estruturada, ou sem nenhuma organização. De acordo com Zikopoulos e Eaton (2011), o termo Big Data se aplica a todo este potencial de dados que não são passíveis de análise ou processamento através dos métodos e ferramentas tradicionais. Por muito tempo, várias empresas tinham a liberdade de ignorar o uso de grande parte deste volume de informações, pois não havia como armazenar estes dados a um custo benefício aceitável. Todavia, com o avanço das tecnologias relacionadas a Big Data, percebeu-se que a análise da maioria destas informações pode trazer benefícios consideráveis, agregando novas percepções jamais imaginadas.

A análise realizada sobre petabytes de informações pode ser um fator determinante para tomadas de decisões em vários contextos. A utilização desta tecnologia associada a algoritmos de aprendizagem de máquinas, por exemplo, pode revelar tendências de usuários, necessidades em determinadas áreas que não se imaginaria sem uso desta abordagem. Com isso, fica claro que o termo Big Data não está relacionado a apenas o armazenamento em grande escala, mas também ao processamento destes dados para agregar valor ao contexto em que for aplicado.

As tecnologias de Big Data descrevem uma nova geração de arquiteturas, projetadas para economicamente extrair valor de um grande volume, sobre uma grande variedade

[1] Umas das redes sociais de maior sucesso no mundo.
[2] Organização sem fins lucrativos responsável por armazenar recursos multimídia.

de dados, permitindo alta velocidade de captura, e/ou análise (GANTZ, 2011). Nesta definição, é possível identificar os três pilares deste novo movimento: velocidade, volume e variedade. De acordo com Zikopoulos e Eaton (2011), isto pode ser caracterizado como os três Vs presentes nas tecnologias Big Data.

O projeto Apache Hadoop[3] é uma das soluções mais conhecidas para Big Data atualmente. Sua finalidade é oferecer uma infraestrutura para armazenamento e processamento de grande volume de dados, provendo escalabilidade linear e tolerância a falhas. Segundo White (2012), em abril de 2008 o Hadoop quebrou o recorde mundial e se tornou o sistema mais rápido a ordenar 1 terabyte, utilizando 910 máquinas essa marca foi atingida em 209 segundos. Em 2009, este valor foi reduzido para 62 segundos.

Neste trabalho, será apresentando, inicialmente, um estudo sobre a arquitetura e o novo paradigma que caracteriza o projeto Hadoop, onde também serão mostradas as principais tecnologias que foram construídas em cima desta abordagem. Após essa etapa de pesquisa, estes conceitos serão aplicados em um estudo de caso real, onde será proposta e implementada uma arquitetura modelo responsável por coletar e processar publicações de redes sociais.

1.1 Objetivos

1.1.1 Objetivos Gerais

Para este trabalho de conclusão de curso, os principais objetivos são compreender as tecnologias que envolvem o projeto Hadoop, uma das principais soluções existentes para análise de dados dentro do contexto Big Data, e também aplicar estes conceitos em um estudo de caso voltado para análise de mensagens publicadas em redes sociais, com conteúdo relacionado a política brasileira.

1.1.2 Específicos

Os objetivos específicos desse trabalho são apresentados a seguir:

1. Analisar a arquitetura e funcionamento do núcleo do Hadoop, composto pelo sistema de arquivos distribuídos HDFS e pelo modelo de processamento distribuído *MapReduce*.

2. Descrever os passos necessários para o desenvolvimento de aplicações utilizando o paradigma *MapReduce*.

[3] <http://hadoop.apache.org/>

3. Analisar as ferramentas que fazem parte do ecossistema Hadoop e oferecem serviços executados no topo desta arquitetura.

4. Construção de senso crítico para análise dos cenários Big Data e as soluções possíveis para estes diferentes contextos.

5. Aplicar os conceitos absorvidos na etapa de pesquisa em um estudo de caso real.

6. Projetar e implementar uma arquitetura para análise de publicações de redes sociais, construindo um modelo capaz de prover escalabilidade linear.

1.2 Organização do Trabalho

O primeiro passo para a construção deste trabalho ocorreu através de uma revisão bibliográfica sobre as tecnologias que se encaixam nesta nova abordagem Big Data. Com as pesquisas realizadas, o Hadoop foi escolhido como objeto de estudo, pois é uma das ferramentas mais utilizadas e consolidadas para este contexto. Para compreender esta tecnologia selecionada foi realizada uma pesquisa para detalhar sua arquitetura e funcionamento. Em seguida, foi realizada uma descrição dos passos necessários para o desenvolvimento para aplicações *MapReduce* e também dos projetos que compõe o ecossistema Hadoop.

Os capítulos subsequentes estão estruturados de acordo com a seguinte organização. O capítulo 2 apresenta uma descrição detalhada sobre o Hadoop, onde o foco está no sistema de arquivos distribuídos HDFS e também no *framework* para computação paralela *MapReduce*.

No capítulo 3, são discutidos os passos necessários para o desenvolvimento de software utilizando o *framework MapReduce*, onde são abordados aspectos técnicos e práticos que envolvem esta abordagem.

O capítulo 4 apresenta os projetos que fazem parte do ecossistema Hadoop. O foco desta seção está nas ferramentes construídas no topo da arquitetura Hadoop, em específico o banco de dados NoSQL HBase e também o *data warehouse* distribuído Hive.

Após a etapa inicial de pesquisa, o esforço foi concentrado na especificação e implementação da arquitetura proposta para o estudo de caso. Além dessas atividades relacionadas ao desenvolvimento, também realizou-se uma investigação sobre o funcionamento das plataformas de redes sociais e sobre novas tecnologias ligadas ao ecossistema Hadoop.

No capítulo 5, é apresentado o estudo de caso proposto para esse trabalho. Nesta seção, a arquitetura modelo para o problema é definida e especificada.

No capítulo 6, são apresentados os resultados obtidos após a etapa de implementação da arquitetura definida no estudo de caso.

No capítulo 7, são apresentadas as conclusões, considerações finais e sugestões para trabalhos futuros.

2 Hadoop

Uma das soluções mais conhecidas para análise de dados em larga escala é o projeto Apache Hadoop, concebido por Doug Cutting[1], o mesmo criador da biblioteca de busca textual Apache Lucene[2] (WHITE, 2012). Os componentes principais desta ferramenta consistem em um sistema de arquivos distribuídos para ser executado em *clusters*[3] compostos por máquinas de baixo custo, e também pelo *framework* de programação paralela *MapReduce*. Ambos foram inspirados nos trabalhos publicados por Ghemawat, Gobioff e Leung (2003) e Dean e Ghemawat (2008), e são objetos de estudo deste capítulo. O manual de instalação pode ser encontrado no apêndice A.

2.1 Hadoop Distributed File System

Quando uma base de dados atinge a capacidade máxima de espaço provida por uma única máquina física, torna-se necessário distribuir esta responsabilidade com um determinado número de computadores. Sistemas que gerenciam o armazenamento de arquivos em uma ou mais máquinas interligadas em rede são denominados sistemas de arquivos distribuídos (WHITE, 2012).

Para COULOURIS, DOLLIMORE e KINDBERG (2007), um sistema de arquivos distribuído permite aos programas armazenarem e acessarem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem arquivos a partir de qualquer computador em uma rede. Questões como desempenho e segurança no acesso aos arquivos armazenados remotamente devem ser comparáveis aos arquivos registrados em discos locais.

De acordo com White (2012), seu funcionamento depende dos protocolos de rede que serão utilizados. Portanto, todos os problemas relacionados a própria rede tornam-se inerentes a este tipo de abordagem, adicionando uma complexidade muito maior aos sistemas de arquivos distribuídos, por exemplo, em relação aos sistemas de arquivos convencionais.

No contexto Big Data, que engloba um volume muito grande de dados, a utilização de um mecanismo para armazenar informações ao longo de várias máquinas é indispensável. Neste sentido, a camada mais baixa do Hadoop é composta por um sistema de

[1] <http://en.wikipedia.org/wiki/Doug_Cutting>
[2] <http://lucene.apache.org/core/>
[3] Um *cluster* pode ser classificado como um aglomerado de computadores que simulam o comportamento de uma única máquina. Segundo Tanenbaum (2003), são CPUs fortemente acopladas que não compartilham memória.

arquivos distribuídos chamado Hadoop Distributed File System, ou HDFS.

O HDFS foi projetado para executar *MapReduce jobs*[4] que, por sua vez, realizam leitura, processamento e escrita de arquivos extremamente grandes (VENNER, 2009). Apesar de apresentar semelhanças com os sistemas de arquivos distribuídos existentes, seu grande diferencial está na alta capacidade de tolerância a falhas, no baixo custo de hardware que é requerido e também na escalabilidade linear.

2.1.1 Características Principais

O HDFS foi projetado para armazenar arquivos muito grandes a uma taxa de transmissão de dados constante, sendo executado em *clusters* com hardware de baixo custo (WHITE, 2012). Suas principais características de *design* podem ser descritas a seguir:

1. **Arquivos muito grandes:** Este sistema de arquivos distribuídos é capaz de armazenar arquivos que chegam a ocupar terabytes de espaço, portanto, é utilizado por aplicações que necessitam de uma base de dados extremamente volumosa. Segundo Shvachko et al. (2010), os *clusters* do Hadoop utilizados pela Yahoo[5] chegam a armazenar 25 petabytes de dados.

2. ***Streaming data access:*** Operações de leitura podem ser realizadas nos arquivos quantas vezes forem necessárias, porém um arquivo pode ser escrito apenas uma única vez. Esta abordagem pode ser descrita como *write-once*, *read-many-times* (WHITE, 2012). Este sistema de arquivos distribuídos foi construído partindo da ideia que este modelo é o mais eficiente para processar os dados. O HDFS também realiza a leitura de qualquer arquivo a uma taxa constante, ou seja, a prioridade é garantir vazão na leitura e não minimizar a latência ou prover interatividade com aplicações em tempo real, como é realizado em sistemas de arquivos com propósito geral.

3. **Hardware de baixo custo:** O HDFS foi projetado para ser executado em hardwares que não precisam ter necessariamente alta capacidade computacional e também alta confiabilidade. O sistema foi desenvolvido partindo do pressuposto que a chance dos nós ao longo do *cluster* falharem é extremamente alta, portanto, mecanismos de tolerância a falha foram desenvolvidos para contornar este problema, permitindo uma maior flexibilidade quanto ao uso de diferentes tipos de hardwares. A tabela 1 apresenta as configurações das máquinas utilizadas pelo *cluster* Hadoop presente na empresa Yahoo.

[4] Um *job* pode ser interpretado como uma unidade de trabalho a ser executada pelo sistema, ou seja, o próprio programa *MapReduce*.

[5] <http://pt.wikipedia.org/wiki/Yahoo!>

Processador (CPU)	Memória RAM	Espaço em disco	Sistema Operacional
2 quad-core de 2 a 2,5 GHz	16 a 24 GB	4 discos SATA de 1 Terabyte	Red Hat Enterprise Linux Server

Tabela 1 – Configurações do cluster Yahoo
Fonte: Adaptado de (WHITE, 2012) e (SHVACHKO et al., 2010)

2.1.2 Divisão em Blocos

Um disco é dividido em vários blocos com tamanho fixo, os quais compõe a menor estrutura de dados, onde são realizadas escritas e leituras. Em sistemas de arquivos convencionais, cada bloco não ocupa mais que alguns KB de espaço, geralmente 512 bytes, esta informação é totalmente transparente para o usuário do sistema (WHITE, 2012). Ele deve apenas realizar operações sobre arquivos de diferentes tamanhos.

O HDFS também possui o conceito de blocos, porém estes possuem tamanho bem maior, por padrão ocupam 64MB de espaço. Cada arquivo criado no sistema é quebrado em blocos com estas características, no qual cada um é salvo como uma unidade independente, podendo estar localizado em qualquer nó do *cluster*. O HDFS foi projetado para armazenar arquivos que ocupam grande quantidade de espaço em disco, portanto, com blocos de tamanho elevado o tempo de busca no disco é reduzido significativamente.

A figura 1 ilustra como ocorre o armazenamento no *cluster* HDFS. Quando um arquivo é escrito no sistema é realizada sua divisão em blocos, em seguida eles são espalhados pelos *datanodes*, os quais são responsáveis por armazenar fisicamente estes blocos. A máquina central *namenode* possui apenas o registro da localização exata de cada bloco no *cluster*. Os conceitos sobre *namenode* e *datanode* são apresentados na seção 2.1.3.

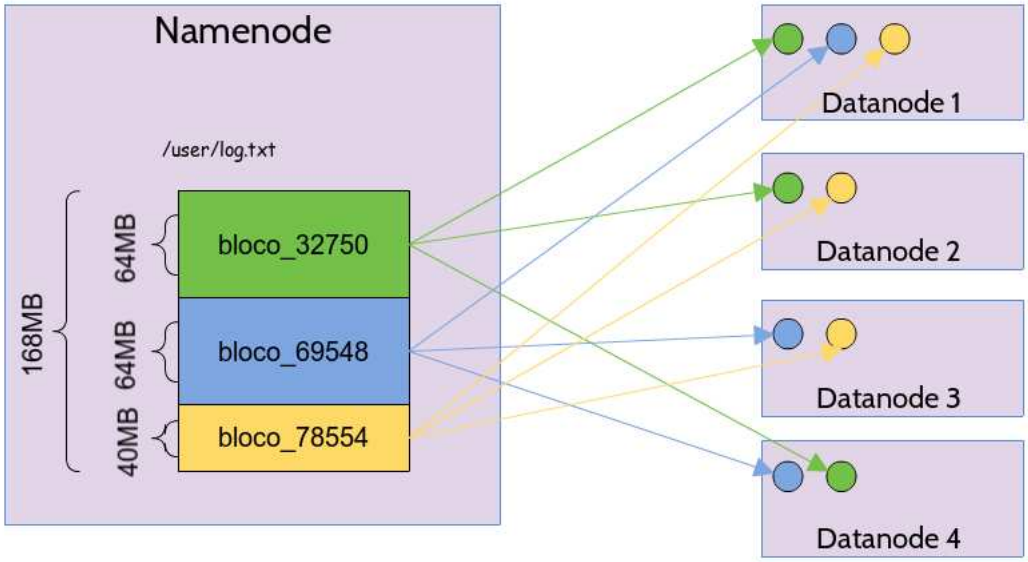

Figura 1 – Divisão de Arquivos em Blocos no HDFS
Fonte: Autor

A utilização de blocos permite simplificar o processo de gerenciamento do armazenamento dos dados. Uma vez que possuem tamanho fixo, a tarefa de calcular a quantidade de blocos necessária para todo disco torna-se mais simples. Segundo White (2012), esta abordagem também permite que blocos sejam replicados pelo sistema de arquivos, provendo tolerância a falhas e uma maior disponibilidade dos dados. No exemplo abordado pela figura 1, é possível perceber que o fator de replicação aplicado ao HDFS possui valor três. Desta maneira, cada bloco é copiado para três máquinas diferentes ao longo do *cluster*.

2.1.3 Arquitetura

Para compreender a arquitetura do HDFS é necessário uma introdução ao GFS, o sistema de arquivos distribuídos proposto pela Google[6], no qual foi projetado para atender a alta e crescente demanda de processamento de dados em larga escada encontrada na empresa. O GFS possui os mesmos objetivos de um sistema de arquivos distribuídos convencional: performance, escalabilidade, confiabilidade e disponibilidade (GHEMAWAT; GOBIOFF; LEUNG, 2003). Porém, as soluções dos sistemas existentes foram revistas e muitas questões de *design* foram alteradas radicalmente. Desta forma, foi possível garantir escalabilidade linear, utilização de hardwares de baixo custo e também escrita e leitura de arquivos na ordem de multi gigabytes através do padrão *write-once*, *read-many-times*.

Um *cluster* GFS é composto por um único *master* e múltiplos *chunkservers*, os quais são acessados por múltiplos clientes (GHEMAWAT; GOBIOFF; LEUNG, 2003).

[6] http://pt.wikipedia.org/wiki/Google

Esta abordagem é baseada no tipo de arquitetura mestre/escravo. Ambos são executados como aplicações em nível de usuário em máquinas com sistema operacional GNU/Linux. A figura 2 ilustra a arquitetura do GFS para operações de leitura.

Neste modelo, as aplicações que desejam utilizar o sistema de arquivos interagem primeiramente com o *master*, responsável por armazenar o *namespace* do GFS e também a localização dos blocos de arquivos ao longo do *cluster*. É retornado para o cliente uma lista de *chunkservers* que contém o arquivo solicitado. Posteriormente, a aplicação acessa diretamente os *chunkservers* para obter os blocos de arquivos.

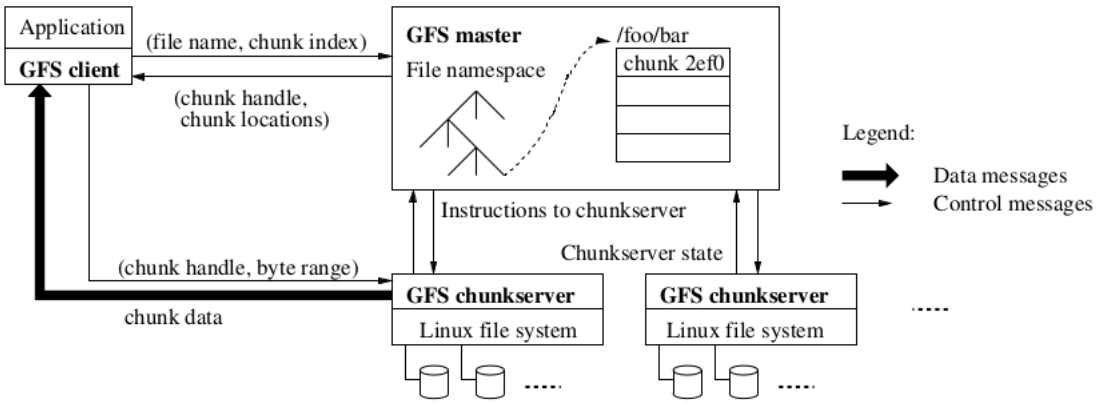

Figura 2 – Arquitetura do GFS
Fonte: (GHEMAWAT; GOBIOFF; LEUNG, 2003).

O HDFS foi construído a partir da arquitura apresentada por Ghemawat, Gobioff e Leung (2003) para desenvolver o GFS. Todas as características abordadas nas seções anteriores, referentes ao HDFS, foram definidas com base neste trabalho. O HDFS apresenta uma alternativa *open-source*[7] implementada em Java para o GFS.

Na solução apresentada pelo HDFS, o nó mestre é identificado como *namenode*. Seu papel consiste em gerenciar o *namespace* do sistema de arquivos e também em controlar as requisições de clientes para acesso aos arquivos armazenados. Este componente mantém a árvore de diretórios e todos os metadados relacionados. Como descrito anteriormente, o HDFS quebra um arquivo em vários blocos e os espalha pelos diversos *datanodes* do *cluster*, o *namenode* possui a tarefa de manter a localização de cada um destes blocos. Todas estas informações são armazenadas no disco local do servidor em dois arquivos: a imagem do sistema de arquivos e o registro de log (WHITE, 2012). A imagem do sistema também é mantida em memória enquanto o *cluster* está ativo, sendo constantemente atualizada.

[7] *Open-source* são projetos de software que seguem o padrão para código aberto determindo pela OSI.

Os *datanodes* representam os nós escravos do HDFS. Eles são responsáveis por armazenar fisicamente os blocos de arquivos e recuperá-los quando solicitado. Estes blocos são escritos no sistema de arquivos da própria máquina onde o *datanode* está localizado. Cada *datanode* se comunica com o *namenode* do *cluster* através da camada de transporte TCP/IP, na qual é utilizada uma abstração do protocolo RPC (HDFS. . . , 2013). Periodicamente os *datanodes* informam ao *namenode* quais blocos cada um deles está armazenando, desta forma, o *namenode* gerencia todos os *datanodes* presentes na rede. A figura 3 ilustra a arquitetura do HDFS.

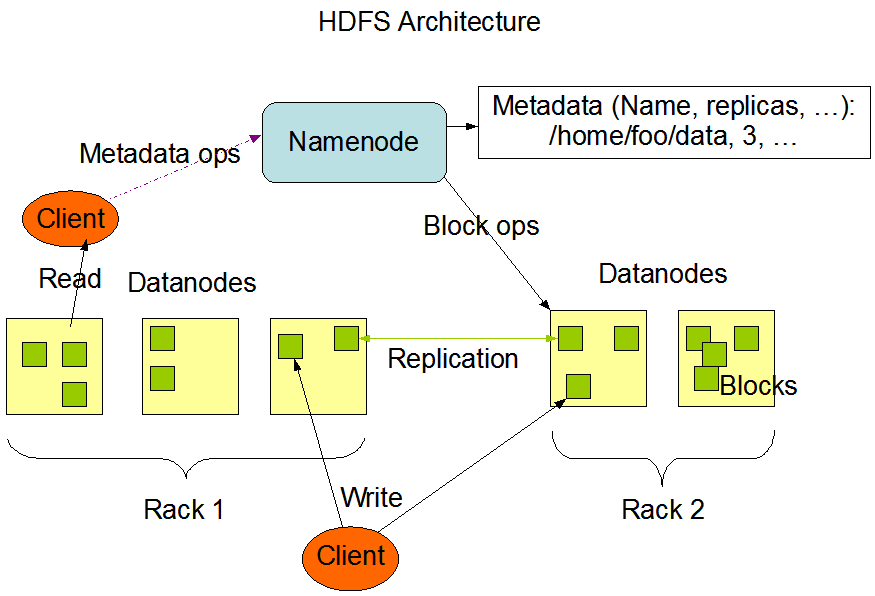

Figura 3 – Arquitetura do HDFS
Fonte: (HDFS. . . , 2013)

Na figura 3 é possível perceber a semelhança do HDFS com a arquitetura do GFS. A ilustração apresenta um *namenode* de um *cluster* Hadoop, que mantém o *namespace* do sistema de arquivos e recebe solicitações de aplicações clientes para escrita e leitura de arquivos. Posteriormente, estas aplicações acessam os *datanodes* diretamente para realizar estas operações sobre os blocos de arquivos, que estão persistidos e replicados ao longo do *cluster*.

Assim como o GFS, o HDFS foi projetado para ser executado em distribuições GNU/Linux, portanto, para que uma máquina seja um namenode ou datanode é necessário apenas que possua uma JVM disponível juntamente com o sistema operacional adequado (HDFS. . . , 2013).

2.1.4 Leitura e escrita

As aplicações de usuários acessam o sistema de arquivos utilizando o HDFS *Client*, uma biblioteca que disponibiliza uma interface de acesso aos arquivos do HDFS

(SHVACHKO et al., 2010). Assim como em sistemas de arquivos convencionais, são permitidas operações de leitura, escrita e remoção de arquivos e diretórios. Segundo White (2012), o HDFS também utiliza uma interface POSIX para sistemas de arquivos, portanto as funções exercidas pelo *namenode* e *datanodes* tornam-se transparentes para o usuário final, pois apenas utiliza-se o *namespace* do HDFS.

Para realizar operações de escrita o cliente deverá solicitar ao namenode a criação do arquivo desejado. Segundo White (2012), uma série de testes são realizadas para garantir que o cliente possui a permissão necessária e também para checar se o arquivo já existe no sistema. Caso esta etapa seja concluída com êxito, o namenode retorna uma lista com o endereço de N datanodes disponíveis para receber o primeiro bloco do arquivo, onde N é o fator de replicação do HDFS, ou seja, o número de cópias que um bloco possui ao longo dos datanodes do *cluster*.

O cliente realiza a escrita nos *datanodes* através de um *pipeline*. O bloco é copiado para o primeiro *datanode* e em seguida repassado para o próximo, assim sucessivamente até os dados serem armazenados no último *datanode* do *pipeline*. A figura 4 apresenta o fluxo do processo de escrita no HDFS.

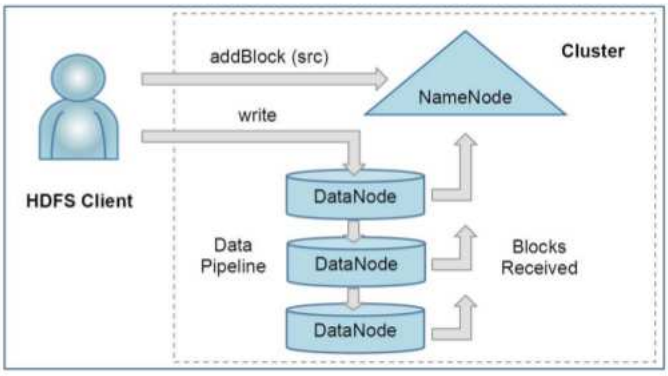

Figura 4 – Operação de escrita no HDFS
Fonte: (SHVACHKO et al., 2010)

Para cada bloco do arquivo, é realizado esse processo de *pipeline*. Na figura 4, é possível perceber que o fator de replicação do exemplo apresentado é igual a três, ou seja, cada bloco será repassado para três *datanodes* diferentes. De acordo com Shvachko et al. (2010), para realizar o processo de leitura o cliente solicita ao *namenode* a localização dos blocos que compõe o arquivo desejado, em seguida contacta diretamente os respectivos *datanodes* para recuperar estes dados.

2.2 *MapReduce*

MapReduce pode ser definido como um paradigma de programação voltado para processamento em *batch*[8] de grande volume de dados ao longo de várias máquinas, obtendo resultados em tempo razoável (WHITE, 2012). Utiliza-se o conceito de programação distribuída para resolver problemas, adotando a estratégia de dividi-los em problemas menores e independentes.

De acordo com Dean e Ghemawat (2008), o usuário deste modelo especifica uma função *map*, que deverá processar um par *{chave, valor}* gerando como produto conjuntos intermediários de pares *{chave, valor}*, e também define uma função *reduce*, responsável por unir todos os valores intermediários associados a uma mesma chave.

Este modelo é baseado nas primitivas *map* e *reduce* presentes na linguagem *Lisp* e também em muitas outras linguagens de programação funcional. Este paradigma foi adotado, pois percebeu-se que vários problemas consistiam em realizar o agrupamento das entradas de acordo com uma chave identificadora, para então processar cada um destes conjuntos (DEAN; GHEMAWAT, 2008).

Um programa *MapReduce* separa arquivos de entrada em diversas partes independentes que servem de entrada para as funções *map*. As saídas destas funções são ordenadas por *{chave, valor}* e, posteriormente, transformadas em entradas do tipo *{chave, lista(valores)}* para a função *reduce*, onde o resultado final será salvo em um arquivo de saída. A quadro 1 apresenta de maneira genérica as entradas e saídas das funções *map* e *reduce*.

```
map      (k1,v1)          -> list(k2,v2)
reduce   (k2,list(v2))    -> list(v2)
```

Quadro 1 – Entradas e saídas - *MapReduce*
Fonte: (DEAN; GHEMAWAT, 2008)

A grande contribuição desta abordagem está em disponibilizar uma interface simples e poderosa que permite paralelizar e distribuir computação em larga escala de forma automática (DEAN; GHEMAWAT, 2008). O desenvolvedor precisa apenas se preocupar com as funções *map* e *reduce*, todo esforço para dividir o trabalho computacional ao longo das máquinas, entre outras questões operacionais, são de responsabilidade do próprio *framework*.

A figura 5 ilustra um fluxo simplificado da execução de um programa *MapReduce*. As entradas são divididas em partes iguais denominadas *input splits*, cada uma destas partes dão origem a uma *map task*, responsável por gerar pares intermediários *{chave,*

[8] Processamento em *batch* (lotes) ocorre quando as entradas são lidas para posterior precessamento sequencial, sem interação com o usuário.

valor}. Cada *map task* realizada uma chamada a função *map* definida pelo usuário.

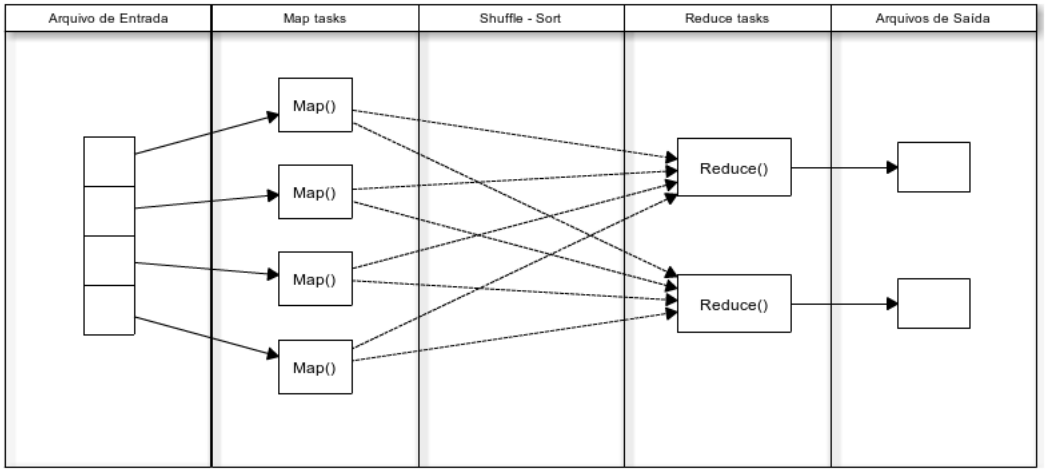

Figura 5 – Fluxo de um programa *MapReduce*
Fonte: Autor

Nas fases *shuffle* e *sort*, os pares intermediários são agrupados e ordenados de acordo com sua chave. Este processo é realizado pelo *framework* e será detalhado nas seções seguintes, sendo uma das etapas mais complexas de todo fluxo. Por fim, as *reduce tasks* recebem como entrada os valores resultantes das etapas *shuffle* e *sort*. Para cada entrada *{chave, lista(valores)}* existente, uma *reduce task* executa uma chamada a função *reduce* especificada pelo desenvolvedor.

2.2.1 Contador de Palavras

Um exemplo simples da aplicabilidade do *MapReduce* pode ser observado em um problema definido como contador de palavras. Suponha que exista um arquivo de texto com várias palavras inseridas, onde o objetivo seja contar a quantidade de ocorrências de cada uma destas palavras ao longo de todo texto. A princípio parece ser uma atividade trivial, entretanto se o tamanho do arquivo estiver na ordem de gigabytes e aumentarmos a quantidade de arquivos a serem processados o tempo de execução aumentará consideravelmente, tornando-se inviável realizar esta análise.

Uma alternativa para contornar este problema seria o uso da programação paralela, analisando os arquivos em diferentes processos, utilizando quantas *threads* fossem necessárias. Todavia esta solução não é a mais eficiente, já que os arquivos podem apresentar diferentes tamanhos, ou seja, alguns processos seriam finalizados em um intervalo de tempo menor, impossibilitando maximizar a capacidade de processamento.

Uma abordagem mais eficiente seria separar todos os arquivos em blocos pré definidos e então dividi-los em processos distintos. Esta solução requer um mecanismo de sincronização complexo e de difícil implementação. Seria necessário agrupar todas as pa-

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

Quadro 2 – Pseudo código *MapReduce*
Fonte: (DEAN; GHEMAWAT, 2008)

lavras e suas respectivas ocorrências em cada uma das *threads* nos diferentes processos em execução. E mesmo assim a capacidade de processamento estaria limitada a apenas uma máquina.

Este problema que se mostrou complexo possui uma solução simples e de fácil construção quando utiliza-se a abordagem apresentada pelo paradigma *MapReduce*. Nesta situação, torna-se necessária apenas a definição de uma função *map* para realizar a contagem de cada palavra presente nos arquivos de entrada, e também de uma função *reduce* para agrupar cada uma destas palavras e realizar a contagem final dos registros de ocorrências. Um pseudo código com as funções *map* e *reduce* para este problema são apresentadas no quadro 2.

No quadro 2, o programa *MapReduce* divide todos os arquivos de entrada em *input splits*, na qual a chave do par *{chave, valor}* é composta pelo número do respectivo *input split*, enquanto o valor é o próprio conteúdo de sua parte no texto. Para cada par de entrada uma função *map* será chamada e realizará quebra da linha em palavras. Cada um destes *tokens* é associado ao valor 1, gerando pares de saída no formato *{palavra, 1}*.

O *framework MapReduce* realiza as etapas *shuffle* e *sort* para agrupar e ordenar os resultados das *map tasks*. Em seguida, para cada chave existente será realizada uma chamada a uma função *reduce*, na qual é responsável por percorrer uma lista efetuando a soma dos valores encontrados. O resultado obtido é registrado em um arquivo de saída. A figura 6 apresenta o fluxo completo da execução de um programa *MapReduce* para o problema da contagem de palavras.

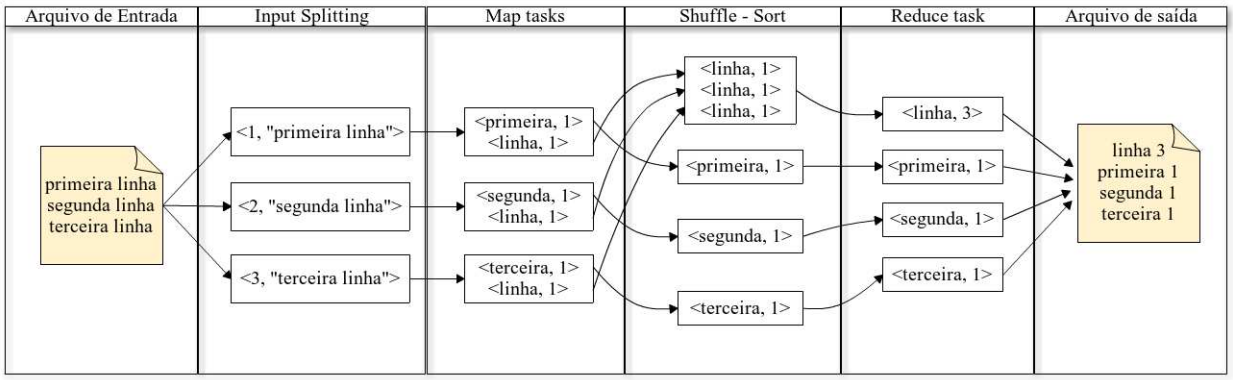

Figura 6 – Fluxo de atividades para o contador de palavras
Fonte: Autor

2.2.2 Arquitetura

O modelo *MapReduce* foi criado pela Google para que os programas escritos neste estilo funcional fossem automaticamente paralelizados e executados em um *cluster* composto por hardwares de baixo custo (DEAN; GHEMAWAT, 2008). As responsabilidades do *framework* consistem em particionar as entradas em *input splits*, gerenciar a execução e comunicação entre os processos do programa ao longo das máquinas e também tratar as falhas que podem ocorrer em cada nó da rede. De acordo com Dean e Ghemawat (2008), com este cenário, um desenvolvedor sem nenhuma experiência em computação paralela seria capaz de desenvolver soluções aplicadas para este contexto.

Este modelo pode ser aplicado em diversas plataformas, mas foi projetado, principalmente, para ser executado sobre o sistema de arquivos distribuídos GFS. Na figura 7, é apresentada a arquitetura do *MapReduce* proposta pela Google. Através dela, é possível visualizar que um *cluster* é composto por dois tipos de nós: um *master* e diversos *workers*.

Inicialmente, os arquivos de entrada são divididos em *input splits* de aproximadamente 64MB, assim como o tamanho dos blocos de arquivos no GFS e também no HDFS. Segundo Dean e Ghemawat (2008), as *map tasks* criadas para cada uma destas entradas são espalhadas pelo *cluster* e executadas em paralelo nos nós do tipo *worker*. As *reduce tasks* criadas para computar os valores intermediários também são processadas em *workers*.

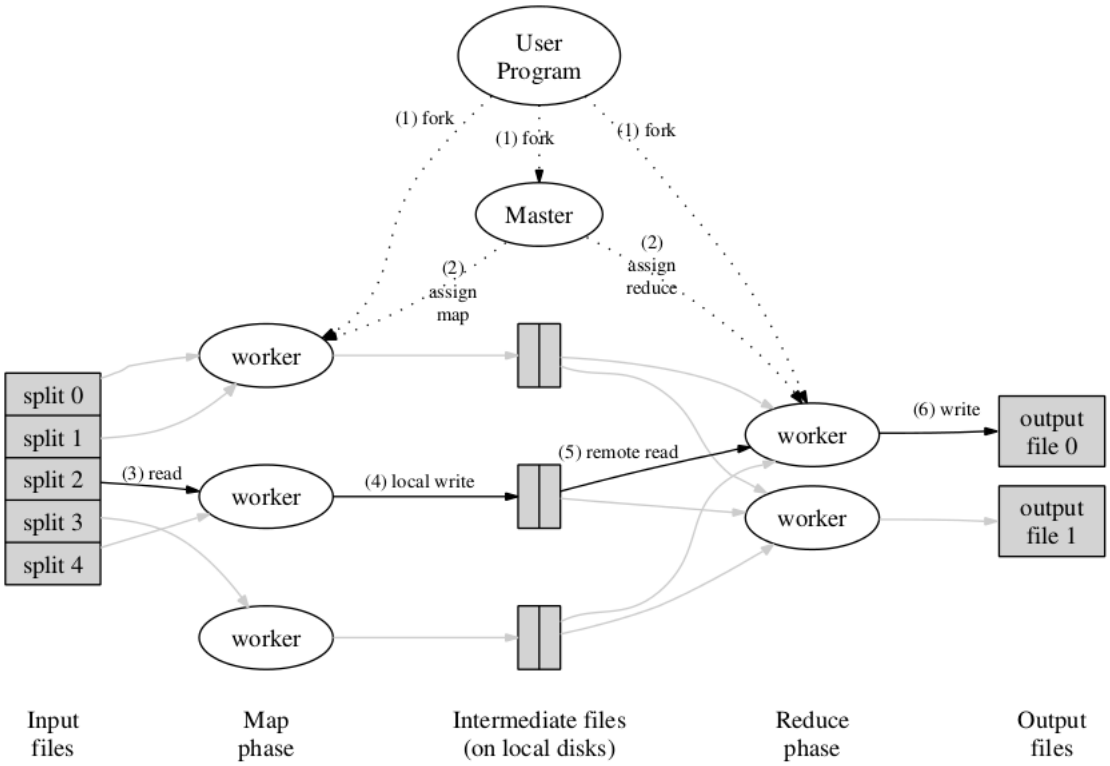

Figura 7 – Arquitetura *MapReduce*
Fonte: (DEAN; GHEMAWAT, 2008)

O *MapReduce job* submetido pelo usuário faz o cálculo de quantas *map tasks* serão necessárias para realizar o processamento dos *input splits*. O nó *master* então possui a tarefa de distribuir as *map tasks* e também as *reduce tasks* para os *workers* disponíveis na rede. De acordo com Dean e Ghemawat (2008), o nó *master* também é responsável por manter o estado de todas as funções *map* e *reduce* que são executadas no *cluster*. Estes dados informam se uma determinada tarefa está inativa, em progresso ou finalizada.

A máquina *master* também monitora os *workers* através de *pings* que são realizados periodicamente. Caso um *worker* não responda, será declarado como indisponível, todas as tarefas deste nó serão reagendadas para serem executadas por outro *worker*. Esta preocupação em monitorar os nós ao longo do *cluster* ocorre porque, segundo Dean e Ghemawat (2008), o *MapReduce* é uma biblioteca projetada para auxiliar no processamento de dados em larga escala ao longo de milhares de máquinas, portanto, precisa prover um mecanismo eficiente para tolerar as falhas que ocorrem em computadores da rede.

A largura de banda da rede pode ser considerada um recurso crítico mediante ao contexto apresentado até o momento, a utilização da rede para transportar dados deve ser otimizada ao máximo para que este fator não prejudique o desempenho de um programa *MapReduce*. Em virtude disso, o *MapReduce* procura tirar vantagem do fato de que os arquivos de entrada são persistidos no disco local dos *workers*. Portanto, o *master*

obtém a localização de cada *input split* e procura executar as *map tasks* exatamente nestas máquinas (DEAN; GHEMAWAT, 2008). Desta forma, a execução da fase de mapeamento é realizada localmente, sem consumir os recursos da rede. Segundo White (2012), este processo pode ser definido como *data locality optimization*.

Por sua vez, as *reduce tasks* não possuem a vantagem de serem executadas localmente, pois sua entrada pode estar relacionada às saídas de um conjunto de diferentes *map tasks*. Portanto, os valores intermediários gerados pelas funções *map* devem ser transportados pela rede até a máquina onde a função *reduce* está sendo processada.

O Hadoop *MapReduce* apresenta uma implementação *open-source* consolidada para o modelo *MapReduce*, na qual é construída a partir da linguagem de programação Java (MapReduce. . . , 2013), diferentemente do *framework* original desenvolvido em C++ pela Google. A arquitetura do *MapReduce* pode ser facilmente associada ao paradigma mestre/escravo. A máquina *master* representa o mestre do sistema, enquanto os *workers* simbolizam os escravos. No contexto proposto pelo Hadoop, estes elementos são identificados como *jobtracker* e os *tasktrackers*, respectivamente. A figura 8 ilustra a arquitetura do Hadoop *MapReduce*.

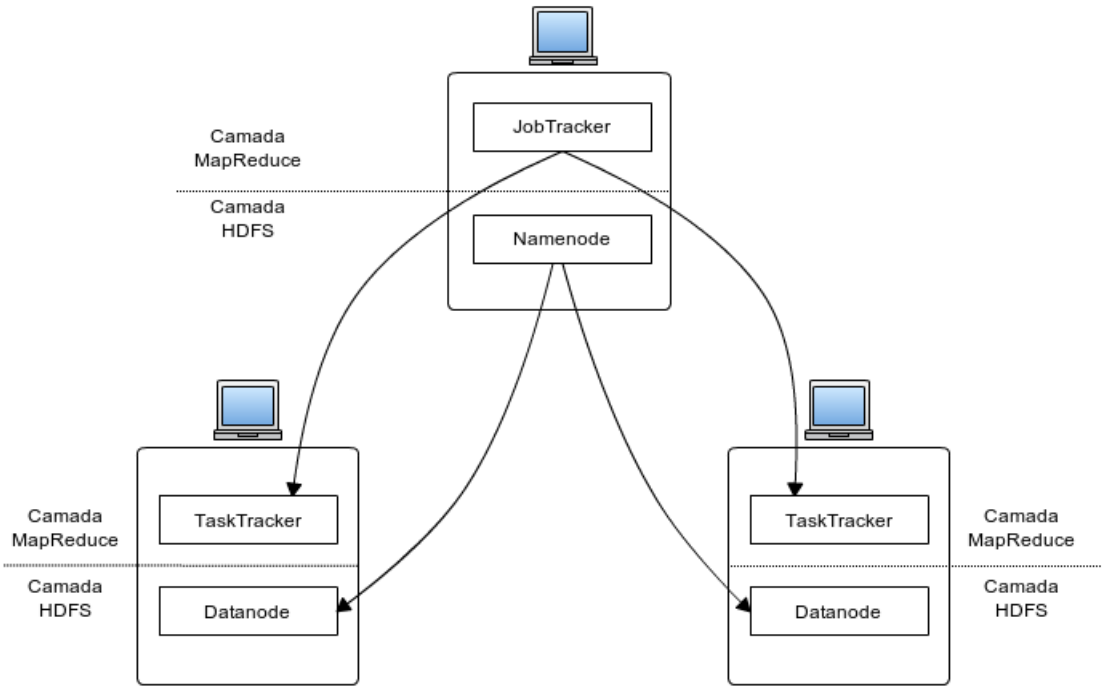

Figura 8 – Arquitetura de um *cluster* Hadoop
Fonte: Autor

Assim como no ambiente proposto pela Google, o Hadoop *MapReduce* também é executado sobre um sistema de arquivos distribuídos em larga escala, no caso o HDFS. Segundo Venner (2009), é comum o *jobtracker* e o *namenode* estarem localizados na mesma máquina do *cluster*, especialmente em instalações reduzidas.

2.2.3 *Shuffle e Sort*

Uma das responsabilidades do *framework MapReduce* é garantir que os pares intermediários *{chave, valor}* resultantes das funções *maps* sejam ordenados, agrupados e passados como parâmetro para as funções de redução. Esta fase é classificada como *shuffle* e *sort*. De acordo com White (2012), esta é a área do código base do Hadoop que está em contínua evolução, podendo ser classificada como o núcleo do *MapReduce*. A figura 9 ilustra os processos de *shuffle* e *sort* que ocorrem entre a execução das funções *map* e *reduce*.

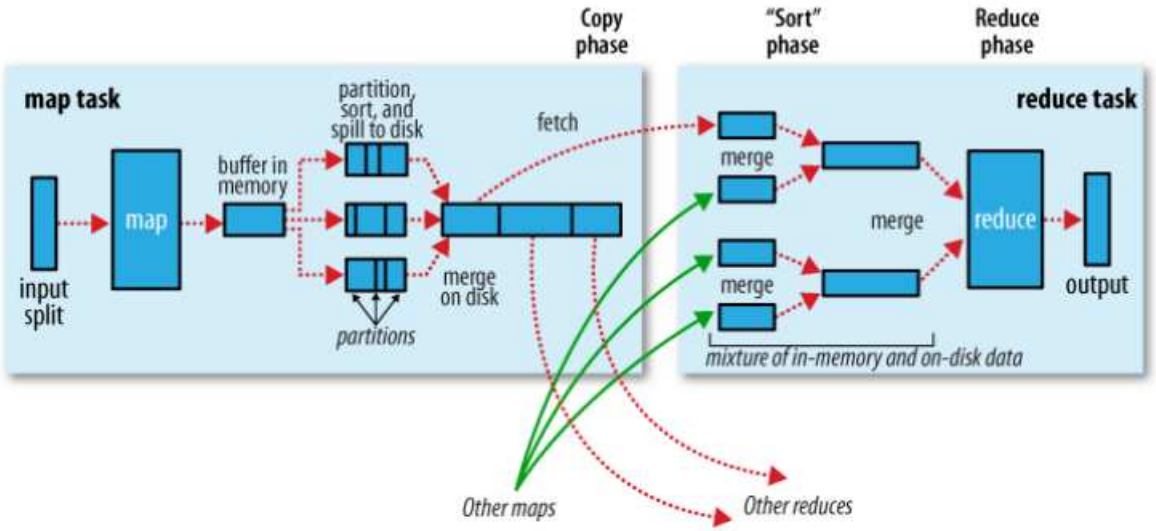

Figura 9 – Fluxograma das etapas *Shuffle* e *Sort*
Fonte: (WHITE, 2012)

Durante a execução de uma função *map*, os pares *{chave, valor}* resultantes são escritos em um *buffer* na memória na medida em que são gerados. Os registros são divididos em R partições, na qual R representa a quantidade de funções *reduce* que seriam necessárias para processar os resultados. Em seguida, as partições são ordenadas de acordo com as chaves e escritas no disco local. Segundo White (2012), Os resultados das *map tasks* são escritos no próprio disco da máquina porque se tratam de arquivos intermediários, não há necessidade de armazená-los no sistema de arquivos distribuídos. Esta etapa de particionamento e ordenação é denominada *shuffle*.

Após o término de uma *map task*, ocorre a fase de cópia. Nesta etapa, as máquinas onde serão executadas as *reduce tasks* são informadas pelo *master* sobre a localização das partições a elas destinadas. As partições geradas pelas *map tasks* são ordenadas localmente, justamente para auxiliar neste procedimento. Ao término da cópia de todas as partições, ocorre a fase *sort*, onde os resultados são agrupados por chave, mantendo-se a ordenação pela mesma. Assim, as entradas para as funções de redução estão prontas para serem computadas.

3 Desenvolvimento *MapReduce*

Neste capítulo, discute-se sobre a construção de aplicações baseadas no modelo *MapReduce*. Anteriormente, foi apresentada uma introdução a este novo paradigma, porém o foco desta seção é deixar claro quais são os passos necessários para o desenvolvimento neste ambiente. Será utilizada uma abordagem técnica para descrever os recursos disponibilizados por este *framework*, originalmente criado para linguagem Java. O manual de instalação do *plugin* Hadoop para a ferramenta Eclipse[1] pode ser encontrado no apêndice B.

Hadoop provê uma API para *MapReduce* que permite a construção das funções *map* e *reduce* em outras linguagens de programação (WHITE, 2012). É possível realizar o desenvolvimento de aplicações *MapReduce* em linguagens de *script*, por exemplo, Python e Ruby, usando o utilitário *Hadoop Stream*. Para a implementação em C++, utiliza-se uma interface denominada *Hadoop Pipe*.

De acordo com a pesquisa realizada por Ding et al. (2011), a implementação em outras linguagens pode ocasionar uma perda de performance, pois a chamada do programa ocasiona *overhead*. Porém quando um job necessita de alto trabalho computacional o uso de linguagens mais eficientes que Java pode trazer benefícios no tempo de execução. O foco deste capítulo será na implementação do *MapReduce* para a linguagem Java.

Para demonstrar sua implementação, será utilizado o exemplo do contador de palavras, também abordado na seção 2.2.1. Este problema consiste em realizar a contagem de ocorrências das diferentes palavras que aparecem ao longo de um arquivo de texto. Primeiramente será adotada uma solução convencional para sua resolução, posteriormente esta abordagem será adaptada para o modelo *MapReduce*.

A implementação para este contador de palavras pode ser realizada de diversas maneiras. Um algoritmo que pode ser usado para solucionar esta questão é mostrado no quadro 3.

A construção deste algoritmo consiste na utilização de um *HashMap*[2] para relacionar cada String encontrada a um valor inteiro, incrementado a cada ocorrência no texto, representando sua frequência. A utilização desta estrutura de dados permite uma simples implementação, entretanto para análise de arquivos em larga escala esta estratégia convencional pode se tornar inviável.

A classe *HashMap* compõe uma das implementações da interface *Map*, que por sua vez faz parte dos tipos de coleções disponibilizados pela linguagem Java. O aumento

[1] <http://eclipse.org/>
[2] <http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>

```
1  HashMap<String, Integer> wordsMap;
2  InputStream isFile;
3  BufferedReader reader;
4  String line, token;
5  int tempCount;
6
7  isFile = new FileInputStream(PATH + FILE);
8  reader = new BufferedReader(new InputStreamReader(isFile));
9  wordsMap = new HashMap<String, Integer>();
10
11 while( (line=reader.readLine())!=null ){
12     StringTokenizer tokens = new StringTokenizer(line);
13
14     while(tokens.hasMoreTokens()){
15         token = tokens.nextToken();
16
17         if( !wordsMap.containsKey(token) ){
18             wordsMap.put(token, 1);
19         }else{
20             tempCount = wordsMap.get(token);
21             wordsMap.put(token, tempCount+1);
22         }
23     }
24 }
25
26 for(Entry<String, Integer> entry : wordsMap.entrySet()){
27     System.out.println(entry.getKey() + " " + entry.getValue());
28 }
29
30 reader.close();
31 isFile.close();
```

Quadro 3 – Algoritmo convencional para contador de palavras
Fonte: Autor

excessivo da quantidade de elementos desta estrutura pode elevar significativamente o uso de memória e também ocasionar uma brusca queda de performance (OAKS, 2014).

A solução adequada para o contexto Big Data, na qual são analisados arquivos extremamente volumosos, consiste na construção de um algoritmo baseado em computação distribuída. A grande vantagem de utilizar o modelo *MapReduce* é que o desenvolvedor precisa apenas adaptar o problema para ser resolvido com as funções *map* e *reduce*. Toda a complexidade envolvida em paralelizar o processamento é realizada pelo *framework*.

O *MapReduce* é um paradigma que permite uma implementação flexível e simples para esta situação, para tal é necessário apenas três coisas: Uma função *map*, uma função *reduce* e um pedaço de código para execução do *job* (WHITE, 2012). Na tabela 2 estão as atividades que ocorrem em um *MapReduce job* e quem é o responsável por cada uma delas.

Atividade	Responsável
Configurar *Job*	Desenvolvedor
Dividir arquivos de entrada em *input splits*	Hadoop *Framework*
Iniciar *map tasks* com seus respectivos *input splits*	Hadoop *Framework*
Definir função *map* utilizada pelas *map tasks*	Desenvolvedor
Shuffle, onde as saídas de cada *map task* são divididas e ordenadas	Hadoop *Framework*
Sort, onde os valores de cada uma das chaves geradas pelas *map tasks* são agrupados	Hadoop *Framework*
Iniciar *reduce tasks* com suas respectivas entradas	Hadoop *Framework*
Definir função *reduce*, na qual é chamada uma vez para cada chave existente	Desenvolvedor
Escrever os resultados das *reduce tasks* em N partes no diretório de saída definido nas configurações do job, onde N é o número de *reduce tasks*	Hadoop *Framework*

Tabela 2 – Atividades de um *MapReduce job*
Fonte: Adaptado de (VENNER, 2009)

No quadro 3, a leitura dos arquivos de entrada é realizada pelo próprio programador, na qual utiliza-se um *BufferedReader* para efetuar o processamento da *stream* dos dados. Para o desenvolvimento no paradigma *MapReduce*, as entradas são obtidas pelo próprio *framework*. O programador deve informar a localização dos arquivos e especificar uma classe que implemente a interface *InputFormat*.

Desta forma, todas as entradas serão convertidas em *input splits* com pares *{chave, valor}* especificados pelo *InputFormat* escolhido. O desenvolvedor pode criar sua própria implementação, porém o Hadoop disponibiliza alguns formatos pré-definidos. Algumas destas classes são apresentados na tabela 3. Segundo White (2012), por padrão o Hadoop adota a classe *TextInputFormat*, a mesma utilizada pelo exemplo que será apresentado na próxima seção.

InputFormat	Chave / Valor
KeyValueTextInputFormat	Os pares chave/valor são identificados a cada linha, separados por um caractere de tabulação horizontal
TextInputFormat	A chave representa o número da linha, enquanto o valor é o respectivo texto
NLineInputFormat	Similar ao *TextInputFormat*, porém uma chave pode estar relacionada a N linhas

Tabela 3 – *InputFormat* disponibilizados pelo Hadoop
Fonte: Adaptado de (VENNER, 2009)

3.1 Mapper

No quadro 2, foi apresentado um algoritmo para resolver o problema da contagem de palavras de acordo com o paradigma proposto pelo modelo *MapReduce*. O primeiro passo para converter este pseudo código para um programa real consiste na criação da função *map*. De acordo com White (2012), esta função é representada pela classe *Mapper*, na qual deve ser implementado o método abstrato *map*(). No quadro 4, é apresentada uma função *map* para o problema proposto.

```
1  public static class TokenizerMapper extends Mapper<LongWritable, Text,
       Text, IntWritable>{
2      private final static IntWritable one = new IntWritable(1);
3      private Text word = new Text();
4
5      public void map(LongWritable key, Text value, Context context)
           throws IOException, InterruptedException {
6          StringTokenizer itr = new StringTokenizer(value.toString());
7          while (itr.hasMoreTokens()) {
8              word.set(itr.nextToken());
9              context.write(word, one);
10         }
11     }
12 }
```

Quadro 4 – Classe *Mapper*
Fonte: Autor

Os parâmetros presentes na assinatura da classe representam os tipos *{chave, valor}* de entrada e saída referentes a função *map*. No quadro 4, é possível identificar que as entradas são compostas por uma chave do tipo *LongWritable* e por valores da classe

Text. Isso significa que cada um dos *input splits* são identificados com um inteiro longo e passados em forma de texto para as funções *map*. As saídas de cada função *map* possuem chaves do tipo *Text* que representam as palavras encontradas em cada *input split*, na qual estão associadas a um valor inteiro *IntWritable* indicando uma ocorrência no texto.

O Hadoop utiliza suas próprias variáveis primitivas, elas foram criadas para maximizar a serialização[3] dos objetos transmitidos pela rede (WHITE, 2012). Os tipos *LongWritable*, *IntWritable* e *Text* representam respectivamente os tipos básicos *Long*, *Integer* e *String*. Segundo White (2012), o mecanismo de serialização disponibilizado pela linguagem Java não é utilizado, pois não atende os seguintes critérios: compacto, rápido, extensível e interoperável.

O Hadoop permite que o desenvolvedor implemente seu próprio tipo primitivo e, para isto, basta criar um objeto que implemente a interface *Writable*. Na maioria das situações isto não é necessário, pois são disponibilizados objetos para *arrays*, mapas, e também para todos os tipos primitivos da linguagem Java (com exceção ao *char*). A figura 10 ilustra todas as implementações do Hadoop para a interface *Writable*.

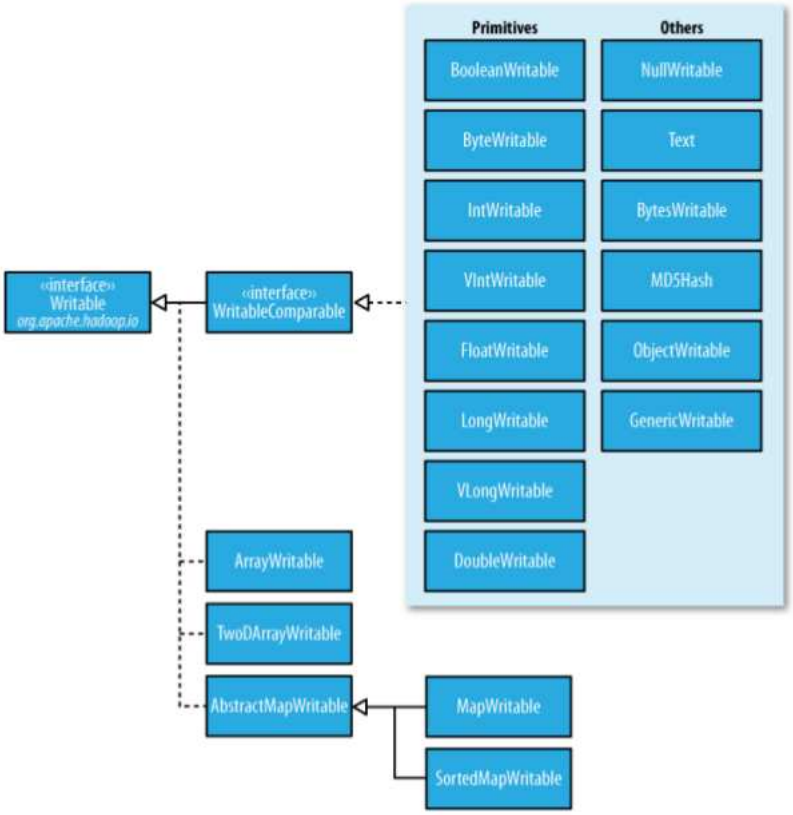

Figura 10 – Hierarquia *Writable*
Fonte: (WHITE, 2012)

[3] Serialização é a habilidade de converter objetos em memória para uma forma externa em que possa ser transmitida (byte a byte) e recuperada posteriormente (DARWIN, 2014).

A função *map*, definida no quadro 4, é executada para cada *input split* existente. Seus parâmetros representam os pares *{chave, valor}* de entrada, e um objeto da classe *Context*, onde os resultados intermediários são armazenados. Dentro da função é possível converter os tipos de dados do Hadoop para os equivalentes em Java, utilizando os recursos disponibilizados pela linguagem normalmente. De acordo com Venner (2009), os objetos da classe *Writable* usados para a escrita dos resultados na classe *Context* devem ser reutilizados, evitando a criação de novas instâncias desnecessárias.

3.2 Reducer

A função *reduce* é implementada através da classe *Reducer*, pela qual define-se um método abstrato *reduce*() que deve ser sobrescrito. Assim como a classe *Mapper*, a assinatura de um *Reducer* possui quatros parâmetros indicando os pares *{chave, valor}* de entrada e saída. As entradas para a função de redução consistem nos tipos de dados de saída especificados pela função *map*, e também em um objeto da classe *Context*. Assim como na classe *Mapper*, ele será responsável por receber os resultados gerados por esta etapa.

O primeiro parâmetro da função reduce indica a chave do tipo *Text*, que representa uma palavra encontrada no texto pela função *map*. A próxima entrada é uma lista de valores do tipo *IntWritable* resultante do agrupamento realizado pelas fases *shuffle* e *sort*. A responsabilidade deste método é apenas somar os valores da lista e escrever os resultados da contagem da chave no objeto *Context*. O quadro 5 apresenta a implementação da classe *Reducer*.

```
1  public static class IntSumReducer extends Reducer<Text,IntWritable,Text,
       IntWritable> {
2      private IntWritable result = new IntWritable();
3
4      public void reduce(Text key, Iterable<IntWritable> values, Context
           context) throws IOException,
                                  InterruptedException {
5          int sum = 0;
6          for (IntWritable val : values) {
7              sum += val.get();
8          }
9          result.set(sum);
10         context.write(key, result);
11     }
12 }
```

Quadro 5 – Classe *Reducer*

Fonte: Autor

Após a leitura do capítulo 2, fica claro que o *MapReduce* procura otimizar a quantidade de dados que são transmitidos pela rede, desta forma, o ideal é realizar o máximo de operações locais possíveis. As *map tasks* podem gerar uma grande quantidade de resultados intermediários, que posteriormente serão copiados para as máquinas onde são executadas as *reduce tasks*. Muitos destes valores podem ser simplificados antes de serem enviados para a rede. O Hadoop permite que o desenvolvedor especifique uma função chamada *combine* a fim de reduzir as saídas geradas pelas funções *map* (WHITE, 2012).

Esta função pode ser interpretada como uma pequena fase *reduce*, que ocorre logo após uma *map task* produzir seus resultados. De acordo com Dean e Ghemawat (2008), em muitas situações o código utilizado para a função *combine* é o mesmo para a função *reduce*. Considere os seguintes resultados gerados por uma *map task* para o exemplo de contagem de palavras:

```
(the, 1)
(the, 1)
(Hadoop, 1)
(Hadoop, 1)
(Hadoop, 1)
(Hadoop, 1)
(framework, 1)
```

Ao aplicar a função combine, os resultados gerados são simplificados para:

```
(the, 2)
(Hadoop, 4)
(framework, 1)
```

Desta forma, os resultados transmitidos pela rede podem ser reduzidos significativamente. É importante ressaltar que existem situações em que esta abordagem não pode ser aplicada. Para um programa que realize a média aritmética, por exemplo, usar uma função *combine* pode gerar resultados incorretos, pois simplificar valores intermediários resultará em um cálculo impreciso da média de todos os valores. Portanto é necessário analisar o contexto para a aplicação de uma função *combine*.

3.3 Configuração do Programa

O último passo para finalizar a construção do programa *MapReduce* consiste em definir as configurações do *job* e utilizar uma classe principal para executá-lo. A classe *Job* representa o programa *MapReduce* em questão. Através dos métodos *setMapperClass*(), *setCombinerClass*() e *setReducerClass*() é possível especificar as classes utilizadas para as funções *map*, *combine* e *reduce*, respectivamente. A classe *WordCount* apresentada

no quadro 6 é responsável pela execução do *job*, assim como especificado pelo método *setJarByClass*().

A localização dos arquivos de entrada, que serão utilizados pelo programa, são indicados pelo método *addInputPath*(). No quadro 3, a escrita dos resultados do programa é realizada pelo próprio programador percorrendo os valores do *HashMap*, já no modelo *MapReduce* o desenvolvedor apenas especifica o formato de saída do arquivo através dos métodos *setOutputKeyClass*() e *setOutputValueClass*().

```
1  public class WordCount {
2      public static void main(String[] args) throws Exception {
3
4          Configuration conf = new Configuration();
5          String[] otherArgs = new GenericOptionsParser(conf, args).
               getRemainingArgs();
6
7          if (otherArgs.length != 2) {
8              System.err.println("Usage: wordcount <in> <out>");
9              System.exit(2);
10         }
11
12         Job job = Job.getInstance(conf);
13         job.setJobName("word count");
14
15         job.setJarByClass(WordCount.class);
16
17         job.setMapperClass(TokenizerMapper.class);
18         job.setCombinerClass(IntSumReducer.class);
19         job.setReducerClass(IntSumReducer.class);
20
21         job.setOutputKeyClass(Text.class);
22         job.setOutputValueClass(IntWritable.class);
23
24         FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
25         FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
26
27         System.exit(job.waitForCompletion(true) ? 0 : 1);
28     }
29 }
```

Quadro 6 – Classe para executar *MapReduce job*
Fonte: Autor

Após a execução das reduce tasks, o *framework* será responsável por escrever os resultados registrados no objeto da classe *Context* em um diretório especificado pelo método *setOutputPath*(). Este ficheiro não deve existir no HDFS, segundo White (2012), isto

evita que informações sejam sobrescritas acidentalmente. O método *waitForCompletion*() submete o *job* para ser executado e aguarda o término do processo. O parâmetro booleano indica a escrita das informações do programa no *console* durante a execução.

4 Ecossistema Hadoop

Os capítulos anteriores abordaram os principais componentes do Hadoop: o sistema de arquivos distribuídos HDFS e o *framework MapReduce*. Estes podem ser considerados o núcleo de todo o sistema, porém o Hadoop também é composto por um conglomerado de projetos que fornecem serviços relacionados a computação distribuída em larga escala, formando o ecossistema Hadoop. A tabela 4 apresenta alguns projetos que estão envolvidos neste contexto.

Projeto	Descrição
Common	Conjunto de componentes e interfaces para sistemas de arquivos distribuídos e operações de Entrada/Saída.
HDFS	Sistema de arquivos distribuídos executado sobre clusters com máquinas de baixo custo
MapReduce	*Framework* para processamento distribuído de dados, aplicado em *clusters* com máquinas de baixo custo.
Pig[1]	Linguagem de procedimentos de alto nível para grandes bases de dados. Executada em *clusters* HDFS e *MapReduce*.
Hive[2]	Um *data warehouse* distribuído. Gerencia arquivos no HDFS e provê linguagem de consulta baseada em SQL.
HBase[3]	Banco de dados distribuído orientado a colunas. Utiliza o HDFS para armazenamento dos dados.
ZooKeeper[4]	Coordenador de serviços distribuídos.
Flume[5]	Ferramenta para transferência de dados entre diferentes fontes.
Sqoop[6]	Ferramenta para mover dados entre banco relacionais e o HDFS.

Tabela 4 – Ecossistema Hadoop
Fonte: Adaptado de (WHITE, 2012) e (SHVACHKO et al., 2010)

1 <http://pig.apache.org/>
2 <http://hive.apache.org/>
3 <http://hbase.apache.org/>
4 <http://zookeeper.apache.org/>
5 <https://flume.apache.org/>
6 <http://sqoop.apache.org/>

Apesar do Hadoop apresentar uma boa alternativa para processamento em larga escala, ainda existem algumas limitações em seu uso. Como discutido anteriormente, o HDFS foi projetado de acordo com o padrão *write-once*, *read-many-times*, desta forma, não há acesso randômico para operações de leitura e escrita. Outro aspecto negativo se dá pelo baixo nível requerido para o desenvolvimento neste tipo de ambiente. Segundo Thusoo et al. (2009), a utilização do *framework MapReduce* faz com que programadores implementem aplicações difíceis de realizar manutenção e reuso de código,

Alguns dos projetos do ecossistema citado na tabela 4 foram criados justamente para resolver estes problemas. Desta maneira, utilizam-se do Hadoop para prover serviços com um nível de abstração maior para o usuário. Neste capítulo, discute-se sobre a ferramenta de *data warehouse* distribuído Hive e também sobre o banco de dados orientado a colunas HBase. No capítulo 5, durante a especificação do estudo de caso, serão abordadas as ferramentas Flume e Sqoop, utilizadas, principalmente, para transferência de dados entre o *cluster* Hadoop e outros ambientes.

4.1 Hive

Antes de iniciar a discussão proposta por esta seção, será apresentado o conceito de *data warehouse*. Inmon (2005) define *data warehouse* como uma coleção de dados integrados, orientados por assuntos, não voláteis e variáveis com o tempo, na qual oferece suporte ao processo de tomada de decisões. Uma arquitetura deste tipo armazena, de forma centralizada, os dados granulares de uma determinada empresa e permite uma análise elaborada destas informações que são coletadas de diferentes fontes.

De acordo com Kimball e Ross (2013), um dos principais objetivos de um *data warehouse* é facilitar o acesso à informação contida nos dados armazenados, de forma que não apenas os desenvolvedores sejam capazes de interpretar, mas também usuários com uma visão voltada para o negócio e não para aspectos técnicos de implementação. Este tipo de solução estrutura os dados para auxiliar a realização de consultas e análises.

O Apache Hive é uma ferramenta *open-source* para *data warehousing* construída no topo da arquitetura Hadoop (THUSOO et al., 2009). Este projeto foi desenvolvido pela equipe do Facebook para atender as necessidades de análise do grande volume de informações geradas diariamente pelos usuários desta rede social. A motivação encontrada para a criação deste projeto se deu pelo crescimento exponencial da quantidade de dados processados pelas aplicações de BI, tornando as soluções tradicionais para *data warehouse* inviáveis, tanto no aspecto financeiro, como computacional.

Além das aplicações de BI utilizadas internamente pelo Facebook, muitas funcionalidades providas pela empresa utilizam processos de análise de dados. Até o ano de 2009, a arquitetura para este requisito era composta por um *data warehouse* que utilizava

uma versão comercial de um banco de dados relacional. Entre os anos de 2007 e 2009 a quantidade de dados armazenados cresceu de forma absurda, passando de 17 terabytes para 700 terabytes. Alguns processos de análise chegavam a demorar dias, quando executados nestas condições.

Segundo Thusoo et al. (2009), para resolver este grave problema a equipe do Facebook optou por adotar o Hadoop como solução. A escalabilidade linear, capacidade de processamento distribuído e habilidade de ser executado em *clusters* compostos por hardwares de baixo custo motivaram a migração de toda antiga infraestrutura para esta plataforma. O tempo de execução dos processos de análise que antes podiam levar dias foi reduzido para apenas algumas horas.

Apesar desta solução encontrada, o uso do Hadoop exigia que os usuário desenvolvessem programas *MapReduce* para realizar qualquer tipo de análise, até mesmo pequenas tarefas, por exemplo, contagem de linhas e cálculos de médias aritméticas. Esta situação prejudicava a produtividade da equipe, pois nem todos eram familiarizados com esse paradigma, e em muitos casos era necessário apenas realizar análises que seriam facilmente resolvidas com o uso de uma linguagem de consulta. De acordo com Thusoo et al. (2009), o Hive foi desenvolvido para facilitar este processo introduzindo os conceitos de tabelas, colunas e um subconjunto da linguagem SQL ao universo Hadoop, mantendo todas as vantagens oferecidas por esta arquitetura.

4.1.1 Características

Como dito anteriormente, o Hive utiliza o conceito de tabelas para registrar as informações em sua base de dados. Cada coluna está associada a um tipo específico que pode ser primitivo ou complexo. Segundo Thusoo et al. (2009), quando um registro é inserido os dados não precisam ser convertidos para um formato customizado, como ocorre em bancos de dados convencionais, apenas utiliza-se a serialização padrão do Hive, desta forma, economiza-se tempo em um contexto que envolve um grande volume de dados. White (2012) afirma que esta abordagem pode ser denominada como *schema on read*, pois não há verificação do esquema definido com os dados que estão sendo armazenados, apenas ocorrem operações de cópia ou deslocamento, diferentemente de como acontece em bancos de dados relacionais. Os tipos primitivos são apresentados na tabela 5.

Categoria	Tipo de dado	Descrição
Primitivo	Inteiro	Permite a declaração de inteiros de 1 byte até 8 bytes
	Ponto flutuante	Permite variáveis do tipo float ou de dupla precisão (double)
	Booleano	Verdadeiro ou falso
	String	*Array* de caracteres
	Binário	*Array* de caracteres
	Timestamp	Indica uma marcação temporal com precisão de nanosegundos
Complexo	*Array*	Lista ordenada de elementos do mesmo tipo
	Map	Estrutura associativa que associa uma chave a um tipo especifico de valores
	Struct	Estrutura que é composta por um conjunto de campos associados a um tipo específico

Tabela 5 – Tipos de dados - Hive
Fonte: Adaptado de (WHITE, 2012)

O Hive disponibiliza uma linguagem para consultas denominada HiveQL, na qual é composta por um subconjunto do padrão SQL com adaptações para grandes bases de dados. Segundo White (2012), esta abordagem foi fortemente influenciada pela linguagem de consulta presente no banco de dados relacional MySQL[7]. Utilizando este recurso, usuários podem realizar análises em grandes volumes de informações sem a necessidade de desenvolver aplicações *MapReduce*, apenas com o conhecimento em SQL. O quadro 7 apresenta um exemplo de comando escrito em HiveQL para criar uma tabela, composta por colunas de tipos primitivos e complexos.

```
CREATE TABLE tabela_hive(c1 string, c2 float,
      c3 list<map<string, struct<v1:int, v2:int>>);
```

Quadro 7 – Comando HiveQL
Fonte: Autor

[7] <http://www.mysql.com/>

4.1.2 Arquitetura

Na seção anterior, foi apresentado o modelo de dados utilizado pelo Hive, onde a organização se dá pelo uso de tabelas compostas por colunas, onde os registros são incluídos em linhas. Estes itens são apenas uma representação lógica, o armazenamento físico dos dados é realizado no HDFS. De acordo com Thusoo et al. (2009), o mapeamento entre estas abordagens pode ser identificado a seguir:

- **Tabelas:** Uma tabela representa um diretório no HDFS, todas as informações são registradas dentro deste ficheiro.
- **Partições:** O usuário pode particionar uma tabela de acordo com uma ou mais colunas. Assim, cada partição é registrada em um subdiretório do ficheiro onde está localizado a tabela.
- ***Buckets:*** Um *bucket* é o arquivo onde os registros da tabela de fato são armazenados. Estas estruturas representam o último nível desta árvore de diretórios de uma tabela no HDFS.

Como discutido no capítulo 3, o Hadoop permite armazenar aquivos de diversos formatos, que podem ser inclusive customizados através da interface *InputFormat*. De acordo com Thusoo et al. (2009), o Hive não impõe nenhum restrição quanto ao formato escolhido para gravar os *buckets* no HDFS, inclusive permite que no comando HiveQL o usuário especifique este parâmetro. Isto pode ser feito com o uso da cláusula *STORED AS*. O quadro 8 apresenta um comando para criação de uma tabela, na qual deve ser armazenada fisicamente como um arquivo binário, de acordo com as classes *SequenceFileInputFormat* e *SequenceFileOutputFormat*.

```
CREATE TABLE dest1(key INT, value STRING)
STORED AS
INPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```

Quadro 8 – Uso da cláusula STORED AS
Fonte: (THUSOO et al., 2009)

O uso da ferramenta Hive é disponibilizado através de três serviços que permitem usuários executarem comandos no formato HiveQL. Uma das opções é o uso de uma interface em linha de comando, similar a um terminal Unix, assim como ocorrem em bancos de dados relacionais, como por exemplo, MySQL e PostgreSQL[8]. Segundo Thusoo et al.

[8] <http://www.postgresql.org/>

(2009), as consultas também podem ser submetidas através do uso de um serviço *web* ou com a utilização de *drivers* de conexão JDBC/ODBC[9] para a interação com outras aplicações. Estes serviços estão presentes no topo pilha de tecnologias do Hive e podem ser observados na figura 11, onde é apresentada a arquitetura geral da plataforma.

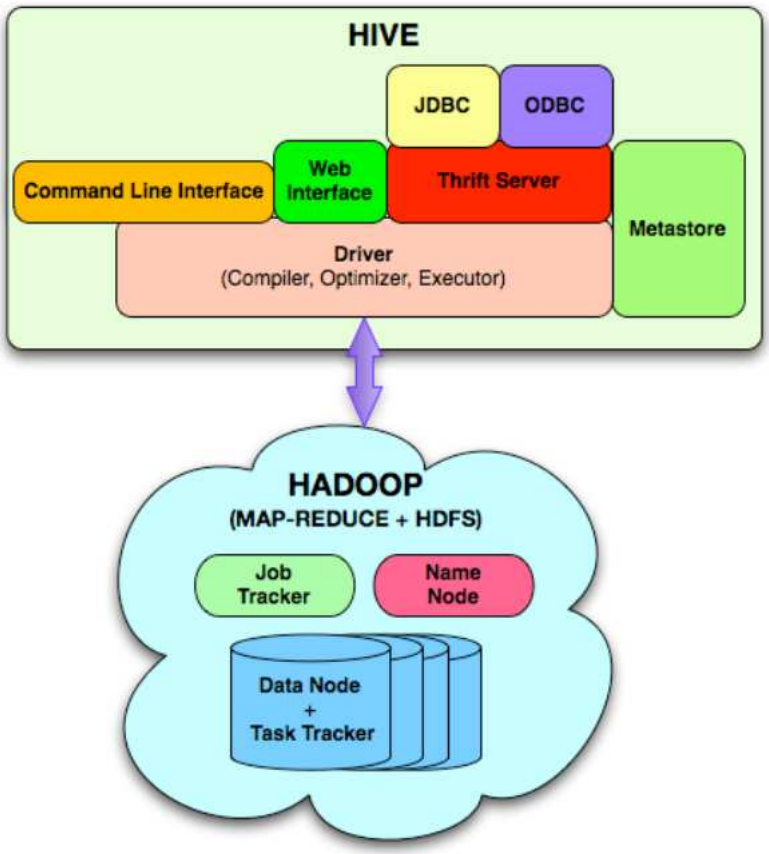

Figura 11 – Arquitetura Hive
Fonte: (THUSOO et al., 2009)

O grande diferencial está no componente denominado *driver*, o qual está presente no núcleo da arquitetura. Nesta etapa, o comando HiveQL é compilado e convertido para um grafo acíclico, onde cada nó é representado por um *MapReduce job*. São realizadas otimizações e a ordenação topológica, resultando em uma sequência de *jobs* que serão executados no *cluster* Hadoop. Um comando HiveQL pode gerar vários *MapReduce jobs*, que são intercalados para se obter o resultado final.

4.2 HBase

Os bancos de dados relacionais sempre desempenharam um papel importante no *design* e implementação dos negócios da maioria das empresas. As necessidades para este contexto sempre envolveram o registro de informações de usuários, produtos, entre

[9]

inúmeros exemplos. Este tipo de arquitetura oferecida pelos SGBDs foram construídas de acordo com o modelo de transações definido pelas propriedades ACID. Segundo George (2011), desta forma, é possível garantir que os dados sejam fortemente consistentes, o que parece ser um requisito bastante favorável. Esta abordagem funciona bem enquanto os dados armazenados são relativamente pequenos, porém o crescimento desta demanda pode ocasionar sérios problemas estruturais.

De acordo com George (2011), os bancos de dados relacionais não estão preparados para análise de grande volume de dados caracterizados pelo contexto Big Data. É possível encontrar soluções que se adaptem a esta necessidade, porém na maioria das vezes envolvem mudanças drásticas e complexas na arquitetura e também possuem um custo muito alto, já que em muitos casos a resposta está ligada diretamente ao uso de escalabilidade vertical, ou seja, ocorre com a compra de máquinas caras e computacionalmente poderosas. Contudo não há garantias de que com um aumento ainda maior da quantidade de dados todos os problemas inicias não voltem a acontecer, isso porque a relação entre o uso de transações e o volume de informações processadas não é linear.

4.2.1 NoSQL

Um novo movimento denominado NoSQL surgiu com propósito de solucionar os problemas descritos na seção anterior. Segundo Cattell (2011), não há um consenso sobre o significado deste termo, esta nomenclatura pode ser interpretada como *not only* SQL (do inglês, não apenas SQL), ou também como uma forma de explicitar o não uso da abordagem relacionada aos bancos de dados relacionais. De acordo com George (2011), as tecnologias NoSQL devem ser compreendidas como um complemento ao uso dos SGBDs tradicionais, ou seja, esta abordagem não é revolucionária e sim evolucionária.

Uma das principais características destes sistemas é descrita por Cattell (2011) como a habilidade em prover escalabilidade horizontal para operações e armazenamento de dados ao longo de vários servidores, ou seja, permitir de maneira eficiente a inclusão de novas máquinas no sistema para melhora de performance, ao invés do uso da escalabilidade vertical, na qual procura-se aumentar a capacidade de processamento através do compartilhamento de memória RAM entre os computadores ou com a compra de máquinas de alto custo financeiro.

De acordo com o trabalho realizado por Cattell (2011), os bancos de dados NoSQL abrem mão das restrições impostas pelas propriedades ACID para proporcionar ganhos em performance e escalabilidade. Entretanto as soluções existentes diferem-se quanto ao nível de desistência na utilização deste modelo de transações.

Ao contrário dos bancos de dados relacionais, os novos sistemas NoSQL diferem entre si quanto aos tipos de dados que são suportados, não havendo uma terminolo-

gia padrão. A pesquisa realizada por Cattell (2011) classifica estas novas tecnologias de acordo com o modelo de dados utilizados por cada uma delas. As principais categorias são definidas a seguir:

- Armazenamento chave/valor: este tipo de sistema NoSQL armazena valores e um índice para encontrá-los, no qual é baseado em uma chave definida pelo programador.
- Registro de Documentos: sistemas que armazenam documentos indexados, provendo uma linguagem simples para consulta. Documentos, ao contrário de tuplas, não são definidos por um esquema fixo, podendo ser associados a diferentes valores.
- Armazenamento orientado a colunas: sistemas que armazenam os dados em formato de tabelas, porém as colunas são agrupadas em famílias e espalhadas horizontalmente ao longo das máquinas da rede. Este é um modelo híbrido entre as tuplas utilizadas em banco de dados relacionais e a abordagem de documentos.
- Bancos de dados de grafos: sistemas que permitem uma eficiente distribuição e consulta dos dados armazenados em forma de grafos.

A tabela 6 apresenta alguns exemplos de bancos de dados NoSQL, de acordo com as categorias citadas acima. Na próxima seção, será discutido o banco de dados HBase.

Categoria	Exemplos
Chave/Valor	Voldemort[10], Riak[11]
Documentos	CouchDB[12], MongoDB[13]
Orientado a colunas	Google Bigtable, HBase
Grafos	Neo4J[14]

Tabela 6 – Exemplos de bancos de dados NoSQL
Fonte: Autor

4.2.2 Hadoop Database

O HBase é um banco de dados orientado a colunas que foi desenvolvido para oferecer aos usuários do Hadoop uma solução para aplicações em tempo real, na qual os requisitos consistem em acesso randômico para operações de leitura e escrita em larga

10 <http://www.project-voldemort.com/voldemort/>
11 <http://basho.com/riak/>
12 <http://couchdb.apache.org/>
13 <http://www.mongodb.org/>
14 <http://www.neo4j.org/>

escala (WHITE, 2012). O propósito desta ferramenta apresenta diferenças se comparado ao modelo *MapReduce*, e por consequência ao próprio Hive, onde o objetivo é realizar processamento em *batch* sem a preocupação com latência.

Este projeto é uma solução *open-source* para o Google BigTable, um sistema de armazenamento distribuído para dados estruturados publicado por Chang et al. (2006). Apesar de ser um banco de dados, o HBase não utiliza abordagem relacional e não tem suporte para linguagens de consulta SQL. Entretanto, de acordo com White (2012), oferece uma solução ao problema de espaço físico encontrado pelos SGBDs convencionais, estando apto a armazenar tabelas extremamente grandes, populadas ao longo de *clusters* compostos por máquinas de baixo custo.

As colunas compõe a menor unidade lógica representada pelo HBase. Uma ou mais colunas podem formar uma linha, que é endereçada por uma chave única. Uma tabela é constituída por um conjunto de linhas, onde cada célula está sob controle versão, sendo representada por várias instâncias ao longo do tempo. Segundo George (2011), o HBase sempre mantém as células ordenadas de acordo com seu tipo dado, o que pode ser comparado com os índices de chave primária utilizados pelos bancos de dados relacionais.

O HBase permite que as linhas de uma tabela sejam agrupadas por famílias de colunas. Este processo deve ser declarado pelo usuário durante a criação da tabela e é recomendável que esta informação não seja alterada posteriormente. Todas as colunas que fazem parte de uma mesma família devem ser declaradas com um prefixo comum, que deve obedecer o formato *família:qualificador*, onde o termo qualificador se refere ao tipo de dado que representa a coluna (WHITE, 2012). De acordo com George (2011), todas as colunas de uma mesma família são armazenadas fisicamente no HDFS em um arquivo chamado *HFile*. Utilizando esta abstração, o HBase é capaz de prover acesso randômico para escrita e leitura de arquivos no Hadoop. Utilizando estes conceitos, os dados podem ser expressos da seguinte forma:

```
(Table, RowKey, Family, Column, Timestamp) -> Value
```

Esta representação pode ser traduzida para o contexto de linguagens de programação, como mostrado a seguir:

```
SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>
```

De acordo com esta representação, uma tabela pode ser interpretada como um mapa ordenado, o qual é composto por uma lista de famílias de colunas associadas a uma chave única de uma linha específica. As famílias de colunas também são identificadas como um mapa ordenado, que representa as colunas e seus valores associados: o tipo de dado armazenado pela coluna e um *timestamp* para identificar sua versão temporal. A figura 12 ilustra um exemplo desta representação.

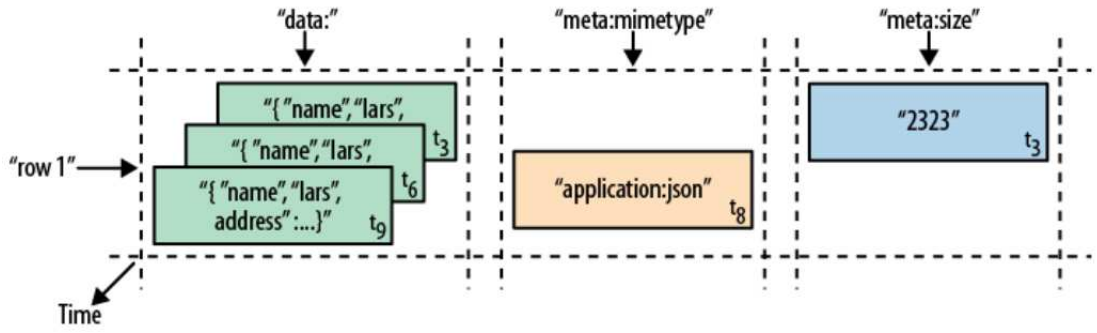

Figura 12 – Linha armazenada no HBase
Fonte: (GEORGE, 2011)

A figura 12 apresenta como uma linha está organizada em uma tabela composta por três colunas, duas delas agrupadas em uma família nomeada como *meta*. Cada célula possui um identificador *ti*, onde *i* significa o tempo em que a célula foi escrita na tabela. A figura 13 ilustra este mesmo exemplo, porém representado em uma tabela.

Row Key	Time Stamp	Column "data:"	Column "meta:" "mimetype"	Column "meta:" "size"	Column "counters:" "updates"
"row1"	t_3	"{ "name": "lars", "address": ...}"		"2323"	"1"
	t_6	"{ "name": "lars", "address": ...}"			"2"
	t_8		"application/json"		
	t_9	"{ "name": "lars", "address": ...}"			"3"

Figura 13 – Tabela HBase
Fonte: (GEORGE, 2011)

Uma das principais características do HBase pode ser identificada na sua capacidade para auto gerenciar sua base de dados. Segundo White (2012) e George (2011), O conteúdo de todas as tabelas é automaticamente particionado e espalhado pelo *cluster* em regiões distintas. Cada região é composta por um servidor que fica responsável por armazenar um intervalo de valores de uma determinada tabela. A figura 14 apresenta como uma tabela é dividida ao longo de regiões pelo HBase.

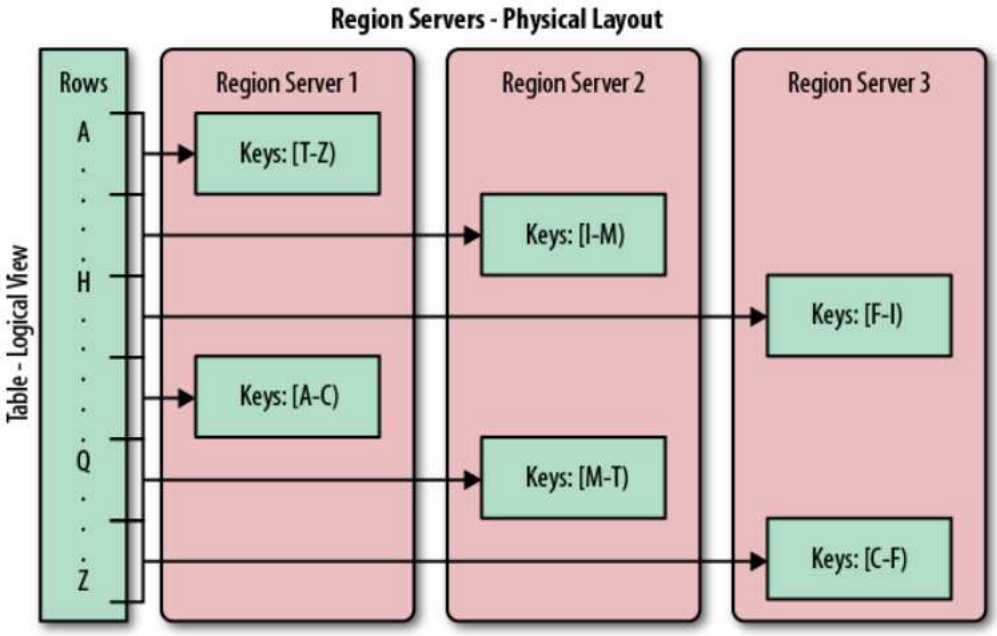

Figura 14 – Regiões HBase
Fonte: (GEORGE, 2011)

5 Estudo de Caso

Nos capítulos anteriores foram apresentas algumas tecnologias importantes para a implementação de aplicações que se encaixam no contexto Big Data. O propósito desta etapa inicial consistiu em levantar o conhecimento necessário para que seja possível analisar e projetar soluções baseadas em diferentes cenários que envolvem o processamento de grandes volumes de dados. Nesta fase do projeto, o objetivo consiste em aplicar os conceitos vistos até o momento em um estudo de caso real. Portanto, ao longo deste capítulo descreve-se um modelo de arquitetura que será construído para simular um ambiente nesses moldes, onde seja necessário manipular e analisar uma grande quantidade de dados, além de prover escalabilidade linear.

5.1 Motivação

Com o surgimento de políticas públicas voltadas para transparência digital, principalmente após o estabelecimento da Lei de Acesso à Informação[1], a população brasileira tem exercido um papel fundamental no monitoramento das atividades exercidas por seus representantes. A realização de eventos como a maratona Hackathon[2] da Câmara dos Deputados contribui e incentiva ainda mais a adesão da comunidade de desenvolvedores para a construção de softwares voltados para este contexto.

As redes sociais também desempenham uma função essencial nesse processo. Este meio de comunicação deixou de ser utilizado apenas como lazer e atualmente é usado pelos cidadãos para expressar opiniões sobre os mais variados assuntos, princialmente quando refere-se a questões políticas e sociais aplicadas no país.

5.2 Problema

A política brasileira vem sendo bastante criticada nos últimos anos, escândalos políticos, notícias frequentes sobre corrupção, serviços públicos de má qualidade, entre outros fatores, contribuem para a insatisfação do cidadão brasileiro em relação ao governo do país. Tendo em vista que as redes sociais são utilizadas por grande parcela da população como meio para expressar suas opiniões, uma análise minuciosa destas informações pode ser um fator de extrema importância para compreender a percepção das pessoas sobre

[1] <http://www.planalto.gov.br/ccivil_03/_ato2011-2014/2011/lei/l12527.htm>
[2] <http://www2.camara.leg.br/responsabilidade-social/edulegislativa/educacao-legislativa-1/educacao-para-a-democracia-1/hackathon/hackathon>

seus representantes. Processar estas informações não é uma tarefa trivial, pois os dados gerados por este meio podem estar na ordem de terabytes.

5.3 Arquitetura

Nesta seção, será apresentado um modelo para solucionar o problema descrito anteriormente. A quantidade de dados gerados por redes sociais exige uma abordagem voltada para o contexto Big Data, pois, como mostrado nos capítulos anteriores, bancos de dados relacionais e outras tecnologias tradicionais não são adequadas a esse cenário. Portanto, o estudo de caso proposto por este trabalho tem como objetivo a construção de uma aplicação que realize análise de dados de redes sociais, buscando publicações que citem os candidatos que concorreram às eleições do segundo turno para Presidência da República no ano de 2014. As mensagens capturadas devem expressar opiniões sobre estes políticos, permitindo investigar a percepção dos internautas antes, durante e após o período correspondente às eleições. A figura 15 ilustra a arquitetura proposta.

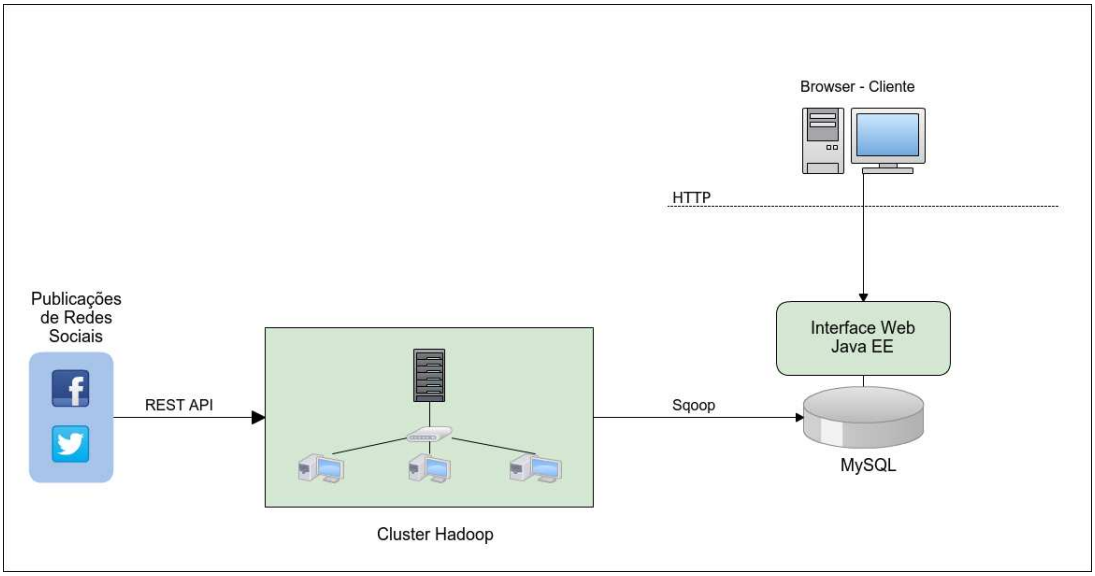

Figura 15 – Arquitetura proposta
Fonte: Autor

A arquitetura apresentada na figura 15 é composta, principalmente, por um *cluster* Hadoop, o qual é responsável por receber e armazenar as mensagens coletadas do Facebook e Twitter, as redes sociais escolhidas como fonte de informações para este trabalho. A captura de publicações é feita utilizando APIs baseadas no protocolo REST[3], que serão detalhadas nas próximas seções. Os dados resididos no *cluster* são processados e os resultados são exportados para um banco de dados relacional MySQL. A transição entre estes dois ambientes ocorre com o auxílio da ferramenta Sqoop. Por fim, uma interface

[3] <http://pt.wikipedia.org/wiki/REST>

web, construída utilizando a linguagem Java, disponibiliza os resultados finais da etapa de análise. A figura 16 apresenta um fluxo com as principais atividades envolvidas neste estudo de caso.

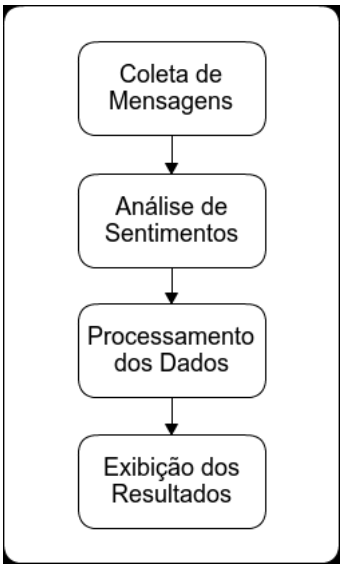

Figura 16 – Atividades envolvidas no estudo de caso
Fonte: Autor

A seguir, é apresentada uma breve descrição sobre as atividades ilustradas na figura 16:

- **Coleta de Mensagens:** Consiste na busca por mensagens publicadas no Facebook e Twitter, onde se utilizam as APIs disponibilizadas por estas redes sociais, as quais são voltadas aos desenvolvedores de aplicações. Nesta etapa, são pesquisadas mensagens que citam os dois candidatos à Presidência da República durante o segundo turno das eleições de 2014. Para este trabalho, foram coletadas publicações realizadas entre janeiro de 2014 e abril de 2015. A ingestão destes dados no *cluster* é realizada com o auxílio da ferramenta Flume, a qual exige uma implementação personalizada que será abordada na seção 5.4.

- **Análise de Sentimentos:** As mensagens coletadas passam por um processo de análise de sentimentos, onde são classificadas de acordo com o sentimento expresso pelo autor de cada publicação a respeito de um determinado candidato. Desta forma, é possível realizar uma breve análise sobre a percepção de uma parcela de usuários sobre estas personalidades públicas.

- **Processamento dos Dados:** Concluídas as etapas anteriores, todas as mensagens resididas no *cluster* são processadas e agrupadas com auxílio da ferramenta de *data warehouse* Hive, a qual permite obter resultados gerais sobre a fase de análise de sentimentos das mensagens de cada candidato. Estes resultados são exportados para o banco de dados relacional MySQL através da ferramenta Sqoop.

- **Exibição dos Resultados:** Com os resultados já persistidos no banco relacional, uma interface *web* permite a visualização destas informações finais, disponibilizando gráficos e tabelas com os resultados do processo de análise de sentimentos para cada um dos candidatos analisados.

As seções seguintes descrevem estas atividades com um maior nível de detalhamento, especificando todos os componentes da arquitetura proposta por este estudo de caso. O *cluster* utilizado neste trabalho, juntamente com as outras ferramentas necessárias, foram configurados em uma única máquina com sistema operacional Ubuntu 12.04. O código fonte das implementações e os arquivos de configuração utilizados estão disponíveis no diretório *implementacao*, localizado no seguinte repositório da ferramenta Github[4]: <https://github.com/guilhermedelima/tcc>.

5.4 Coleta de Mensagens

Nesta seção, são ilustrados os passos necessários para a conclusão da etapa referente a coleta de mensagens. Inicialmente, é apresentada uma breve explicação sobre as APIs do Facebook e Twitter e a definição da metodologia utilizada para pesquisar as publicações. Por fim, apresenta-se o Flume e a implementação construída para capturar as mensagens e armazená-las no *cluster*.

5.4.1 Facebook

Uma das redes sociais de maior sucesso atualmente, o Facebook conta com uma média superior a 890 milhões de usuários diários ativos (COMPANY. . . ,). Além de ser um veículo utilizado para interação entre pessoas, também é comum o seu uso para exposição de opiniões e preferências políticas, religiosas, entre uma infinidade de temas. Devido à sua expressividade, tornou-se comum a utilização do Facebook por pessoas públicas, como políticos e celebridades, seja para divulgação de notícias e informações imparciais, como também para exposição de ideias e opiniões pessoais. Determinadas páginas possuem, inclusive, um selo identificando sua autenticidade, confirmando a utilização daquele meio pela personalidade pública em questão.

Pelas características apresentadas, o Facebook se torna uma fonte de informações indispensável para busca de opiniões. A principal forma de obter dados desta rede social é através da chamada *Graph* API[5], uma plataforma HTTP de baixo nível que permite não apenas consultas, mas também publicar postagens, gerenciar contatos, divulgar fotos, e uma variedade de funcionalidades. O nome desta API deriva da ideia de um grafo

[4] <https://github.com/>
[5] <https://developers.facebook.com/docs/graph-api>

social, onde os vértices são representados por pessoas, fotos, comentários, etc., já arestas expressam a conexão entre esses elementos.

Um dos objetivos deste trabalho é retirar postagens onde pessoas externam opiniões pessoais sobre os candidatos citados anteriormente. A principal forma de um usuário manisfestar estas informações é através de postagens em sua linha do tempo, onde cria-se um tópico específico permitindo, inclusive, respostas de amigos e pessoas autorizadas. A versão 1.0 da *Graph* API possibilitava a busca por postagens públicas, entretanto, por motivos não revelados, esta funcionalidade foi descontinuada pela equipe do Facebook, não sendo possível executá-la.

Outra forma na qual usuários podem interagir consiste na postagem em páginas de pessoas públicas, empresas, entre outros. Apesar da semelhança com a linha do tempo de um usuário comum, estas páginas possuem um caráter integralmente público, onde qualquer indivíduo tem permissão para ler e escrever postagens. Nestes locais, é possível identificar comentários onde usuários expressam opiniões e participam de discussões. Para contornar as limitações descritas no parágrafo anterior e realizar a coleta dos dados, optou-se por buscar todos os comentários publicados nas páginas oficiais de cada candidato analisado. Portanto, todas as postagens retiradas do Facebook são oriundas, exclusivamente, das páginas públicas e autenticadas. A tabela 7 apresenta as páginas utilizadas para coleta das mensagens.

Candidato	Página no Facebook
Aécio Neves	AecioNevesOficial
Dilma Rousseff	SiteDilmaRousseff

Tabela 7 – Páginas utilizadas para pesquisa
Fonte: Autor

Como mencionado anteriormente, a *Graph* API é baseada apenas no protocolo HTTP. Uma das funcionalidades presentes nesta arquitetura é a busca por páginas do Facebook, para isso basta montar uma requisição informando o *id* de uma determinada página. O resultado desta solicitação será um objeto JSON[6] contendo todos os campos do elemento requerido, neste caso, a própria página. A tabela 8 apresenta alguns destes campos.

[6] Acrônimo para JavaScript *Object Notation*. Especifica um formato leve para transferência de dados computacionais, derivado de um subconjunto da notação de objeto da linguagem JavaScript.

Campo	Descrição
id	Inteiro que identifica a página.
about	Informação da página.
category	Categoria da página, por exemplo, tecnologia, política e serviços.
name	Nome da página.
likes	Número de usuários que curtiram a página.

Tabela 8 – Campos de uma página pública do Facebook
Fonte: Adaptado de (FACEBOOK. . . ,)

Uma página representa um vértice do grafo social montado pelo Facebook, assim como as postagens, fotos, álbuns, notificações e outros elementos que estão relacionados a mesma. Para obter as publicações efetuadas em uma página é necessário acrescentar à requisição o nome da aresta que faz a ligação entre eles. A figura 17 apresenta como seria uma requisição HTTP para consultar as publicações de uma página qualquer.

```
GET /v2.3/{page-id}/feed HTTP/1.1
Host: graph.facebook.com
```

Figura 17 – Requisição HTTP para obter publicações de páginas do Facebook
Fonte: (FACEBOOK. . . ,)

O resultado da requisição ilustrada na figura 17 também será um objeto JSON com diversos campos, incluindo os comentários públicos efetuados por usuários. Através dos comentários, é possível identificar o autor, data da postagem e o corpo da mensagem. O Facebook utiliza o protocolo *OAuth*[7] para autorizar o uso dos recursos da *Graph* API, portanto, todas as requisições submetidas devem utilizar um *access token*, que pode ser obtido na página voltada aos desenvolvedores de aplicações.

5.4.2 Twitter

Outra rede social em evidência no cenário atual, o Twitter apresenta um número superior a 280 milhões de usuários ativos mensais, além de uma média equivalente a 500 milhões de mensagens publicadas por dia (ABOUT. . . ,). O modelo adotado pelo Twitter tem uma complexidade bem menor, quando comparado ao Facebook, seu proposito consiste em oferecer um microblog onde usuários podem publicar mensagens de no máximo 140 caracteres, conhecidas como *tweets*.

Para coletar *tweets* com opiniões de usuários, optou-se por realizar uma busca por mensagens utilizando filtragem textual e temporal. Ao contrário do Facebook, o Twitter

[7] <http://en.wikipedia.org/wiki/OAuth>

autoriza pesquisas por comentários públicos em páginas pessoais, portanto, o recolhimento dos *tweets* se torna uma atividade mais acessível. A tabela 9 apresenta os termos usados para pesquisar as publicações que expressam opiniões, ou simplesmente citam os candidatos analisados neste trabalho.

Candidato	Termos utilizados
Aécio Neves	aecio, aecio neves
Dilma Rousseff	dilma, dilma rousseff

Tabela 9 – Termos adotados para pesquisa dos candidatos no Twitter
Fonte: Autor

O Twitter permite a extração de publicações através de duas formas distintas. A primeira maneira se dá pelo uso da *Search* API[8], um componente da REST API[9], a qual possibilita execução de consultas HTTP para obter os *tweets* mais recentes e relevantes. Apesar da semelhança com a ferramenta de busca padrão, disponível para os usuários do Twitter, esta API apresenta algumas limitações, entre elas, o número de *tweets* que podem ser adquiridos. Utilizando a *Search* API, é possível obter apenas as mensagens publicadas entre os últimos 6 ou 9 dias, postagens antigas não são indexadas pela API. Também há um limite de 180 requisições realizadas a cada 15 minutos, ao exceder este valor, as solicitações são negadas até o recomeço de uma nova janela de transmissão.

Outra forma de consultar as publicações do Twitter é através da *Streaming* API[10], um módulo que possibilita a captura de mensagens publicadas em tempo real. Diferente da *Search* API, este modelo mantém uma conexão aberta em um processo separado, o qual notifica o programa principal ao receber novos dados. Desta maneira, o servidor não é inundado com requisições, oferecendo uma opção leve para a coleta de mensagens em tempo real.

Semelhante ao Facebook, o Twitter também utiliza o protocolo *OAuth* para autorizar as requisições HTTP que desejam acessar os serviços da rede social, portanto, é necessário obter um *access token* na área voltada aos desenvolvedores do microblog. Ambas as APIs de busca oferecidas retornam as requisições em objetos no formato JSON, onde é possível identificar o autor do comentário, a data de publicação, o texto da mensagem, entre outros campos.

5.4.3 Apache Flume

Assim como discutido em seções anteriores, um dos grandes desafios inerentes ao uso de tecnologias Big Data se dá pela dificuldade na manipulação de quantidades

[8] <https://dev.twitter.com/rest/public/search>
[9] <https://dev.twitter.com/rest/public>
[10] <https://dev.twitter.com/streaming/overview>

elevadas de informações. A solução apresentada pelo Hadoop para o armazenamento em larga escala envolve algumas questões técnicas que o diferenciam dos diversos paradigmas tradicionais presentes na computação. Segundo Hoffman (2013), em sistemas de arquivos POSIX, durante uma operação de escrita os dados inseridos se mantém persistidos no disco antes mesmo do fechamento do arquivo e término de todo procedimento. Isto significa que se outro processo realizar operações de leitura, ele será capaz de ler os dados que estão sendo escritos no momento. Caso ocorra uma interrupção durante um processo de escrita, as informações registradas até o momento continuam no arquivo, de maneira incompleta, porém parcialmente visíveis.

Por outro lado, o HDFS apresenta um modelo dissimilar. Arquivos escritos neste sistema de arquivos distribuídos são interpretados como uma entrada de diretório de tamanho nulo até o período em que a operação é finalizada. Portanto, caso um processo cliente realize alguma operação de escrita de longa duração, e por alguma razão não a conclua, ao término deste cenário o usuário terá acesso apenas a um arquivo vazio.

Realizar a escrita de arquivos pequenos pode parecer uma alternativa para este problema, pois desta forma, as operações são finalizadas rapidamente, e consequentemente reduziria a probabilidade de perdas durante o processo. Entretanto, esta estratégia resulta em considerável perda de performance do HDFS. Segundo White (2012), cada arquivo, diretório ou bloco representa um objeto na memória do *namenode* do *cluster*, ocupando 150 bytes cada. Além do desperdício de memória, a execução de *MapReduce jobs* seria menos eficiente, pois arquivos pequenos representam blocos de tamanhos reduzidos, ocasionando *input splits* reduzidos. Desta forma, seria necessário um número maior de *map tasks*, o que impacta em um maior tempo de execução.

Inserido nesse contexto, o Apache Flume é um projeto com objetivo de oferecer suporte para situações onde é necessário realizar transferência de registros de uma determinada fonte de dados para um destino final, como HDFS. O Flume foi criado para ser uma ferramenta padronizada, simples, robusta e flexível para ingestão de dados dentro do Hadoop (HOFFMAN, 2013). Sua arquitetura é composta por três módulos principais: *source*, *channel* e *sink*. A figura 18 apresenta este modelo.

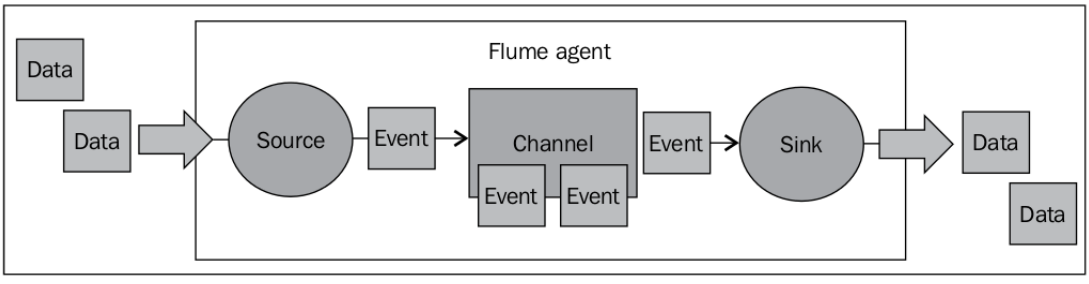

Figura 18 – Arquitetura Flume
Fonte: Retirado de (HOFFMAN, 2013)

O agente Flume apresentado na figura 18 representa o processo executado pela ferramenta, o qual realiza todo procedimento de transferência de informações entre os dois ambientes. Um *source* é uma *thread* responsável por capturar os dados de uma determinada fonte, geralmente um serviço que caracteriza-se por produzir dados a todo instante, como registros de log, publicações de redes sociais, ou similares. Os dados coletados pelo *source* são convertidos para um formato específico do Flume, denominado evento, o qual é composto apenas por um cabeçalho e o conteúdo da mensagem.

Todos os eventos criados por um *source* são escritos em uma área de espera chamada *channel*, responsável pela ligação entre *sources* e *sinks*. Um *channel* representa um *buffer*, podendo armazenar seus dados em memória, ou também em arquivos. Segundo Hoffman (2013), o uso de arquivos permite que um *channel*, antes de disponibilizar os eventos para leitura, efetue todas as alterações no sistema de arquivos local, oferecendo maior espaço físico e tolerância a falhas. Por outro lado, *channels* baseados em memória, apesar da escassez de recursos e volatilidade dos dados, possuem performance superior.

O *sink* representa o último componente do modelo proposto pelo Flume. Sua execução ocorre em uma *thread* independente, com responsabilidade de coletar eventos disponibilizados por um *channel* e transferir estes dados para uma fonte de destino, geralmente o HDFS. O trabalho de um HDFS *sink* é, continuamente, abrir um arquivo no HDFS, transferir dados para ele, e em algum momento fechá-lo e iniciar um novo (HOFFMAN, 2013).

Com esta arquitetura, é possível criar fluxos complexos e variados, adaptando-se a diferentes cenários. A combinação de *sources*, *channels* e *sinks* tornam o Flume uma ferramenta poderosa para ingestão de dados no HDFS.

5.4.4 Implementação Flume

Para coletar as mensagens das fontes de informações disponíveis, foi necessário a implementação de três *sources* distintos, de acordo com as redes sociais selecionadas e suas APIs de acesso.

- **Facebook:** utiliza a *Graph* API para obter os comentários das páginas oficiais dos candidatos.
- **Twitter Rest:** faz uso da *Search* API para coletar os *tweets* publicados entre os últimos 6 ou 9 dias.
- **Twitter *Streaming*:** captura os *tweets* em tempo real utilizando a *Streaming* API.

A figura 19 ilustra o fluxo da informação durante todo o processo de coleta de mensagens, iniciando na interação com as APIs das redes sociais, passando pelos componentes

do Flume e finalizando com as postagens armazenadas no HDFS.

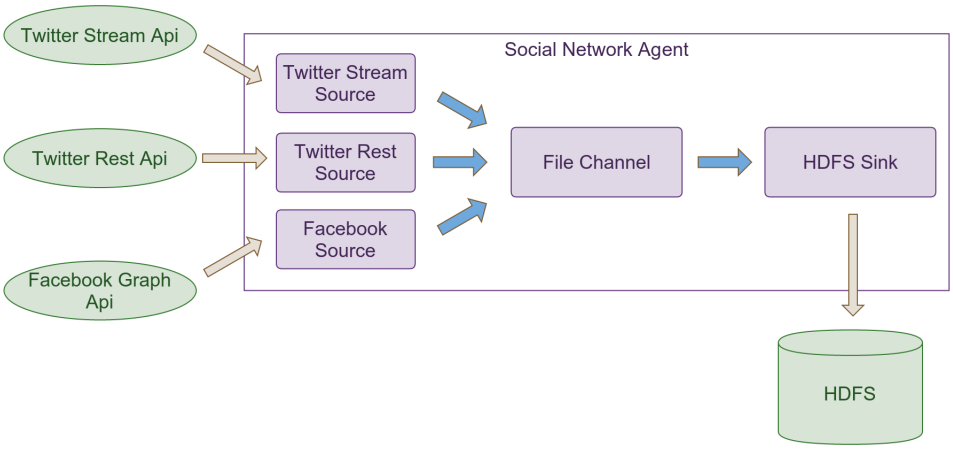

Figura 19 – Arquitetura da coleta de mensagens.
Fonte: Autor

O bloco denominado *Social Network Agent* representa todo o processo que será executado no Flume. Cada *source* diferente, na prática, simboliza uma classe customizada que estende a classe *AbstractSource* e implementa as interfaces *EventDrivenSource* e *Configurable*, pertencentes a API do Flume. A interação destes componentes com as APIs das redes sociais ocorre com o uso das bibliotecas Facebook4j[11] e Twitter4j[12], encapsulando o uso do protocolo REST para facilitar a fase de desenvolvimento.

A imagem 20 apresenta uma arquitetura simplificada dos elementos responsáveis por realizar a captura de mensagens. O pacote *Source* é composto pelas classes que implementam diretamente *sources* do Flume, onde utilizam serviços do pacote *Service*, responsável por se comunicar diretamente com as redes sociais.

[11] <http://facebook4j.org/>
[12] <http://twitter4j.org/>

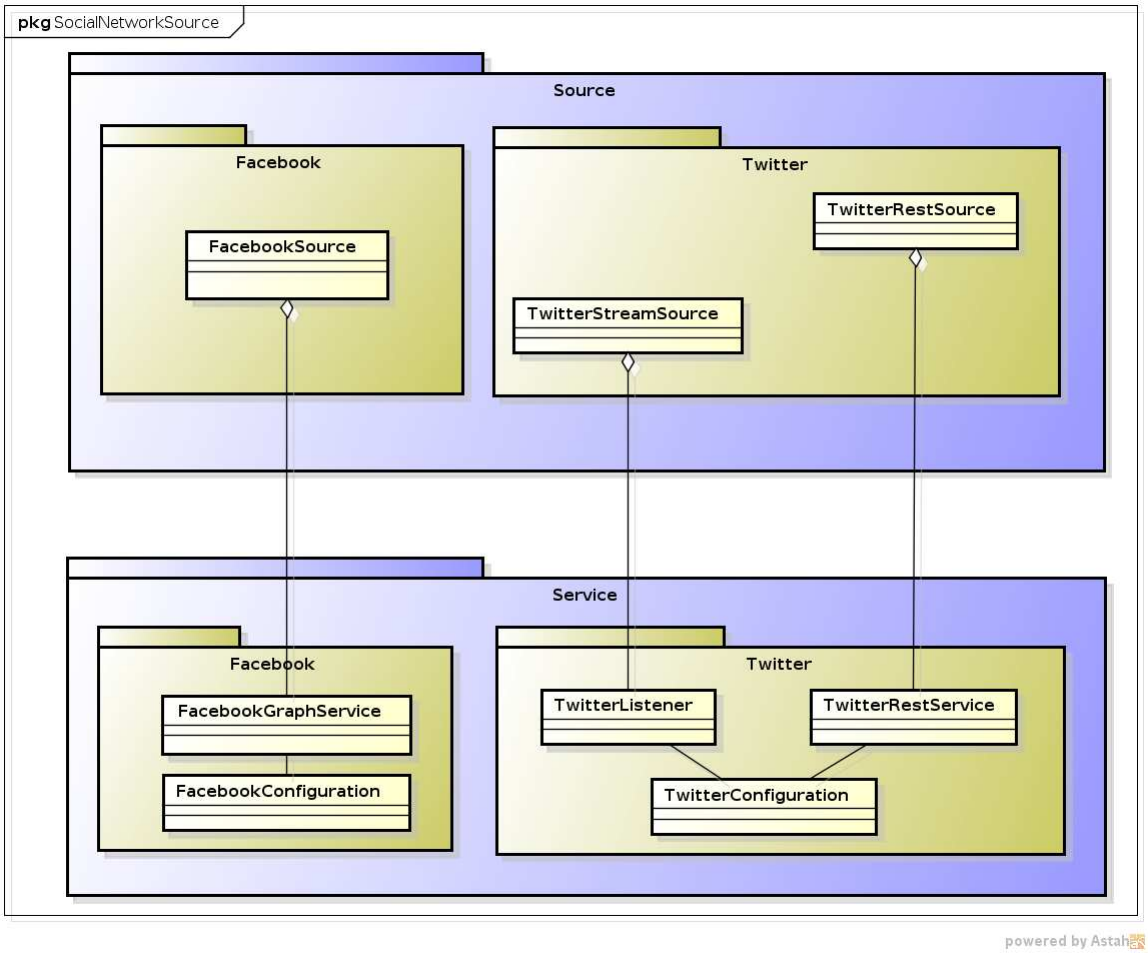

Figura 20 – Arquitetura dos *sources* customizados
Fonte: Autor

Ainda na figura 20, é possível perceber que os eventos gerados pelos diferentes *sources* são escritos em um único *file channel*, o qual conta com um arquivo no próprio disco local para armazenamento temporário. Desta forma, evita-se que na ocorrência de falhas todo o progresso até o momento seja perdido.

Cada evento gerado é composto por um objeto JSON com as informações a respeito da mensagem coletada. Após estarem disponíveis no *file channel*, o HDFS *sink* se encarrega de abrir arquivos no HDFS e transferir os eventos do *buffer* para o *cluster*. A maneira como este *sink* copia os eventos para um arquivo no HDFS pode ser ajustada através de parâmetros presentes no arquivo de configuração do Flume. É importante garantir uma vazão alta juntamente com a criação de arquivos grandes no HDFS. Segundo White (2012), o uso de arquivos pequenos prejudica consideravelmente o desempenho de *MapReduce jobs*, além do uso excessivo de memória pelo *namenode*, pois cada arquivo, diretório e bloco representa um objeto de 150 bytes.

5.5 Análise de Sentimentos

As seções anteriores apresentaram a metodologia utilizada para realizar a coleta das mensagens provenientes do Facebook e Twitter. Nesta fase, será relatado o paradigma adotado para efetuar a análise de sentimentos sobre as publicações armazenadas no HDFS. O objetivo desta etapa é a definição de um modelo, onde mensagens de um candidato específico devem ser processadas e classificadas em dois grupos distintos: postagens que expressam sentimentos positivos ou negativos a respeito do indivíduo.

Uma abordagem comum para análise de sentimentos é a classificação do texto através do emprego de recursos léxicos, onde utiliza-se uma base constituída de termos opinativos previamente computados. O SentiWordNet é um dos métodos mais conhecidos que implementam este modelo, segundo Araújo, Gonçalves e Benevenuto (2013), este classificador usa o dicionário léxico *WordNet* como parâmetro para, posteriormente, atribuir a cada palavra uma pontuação variando de 0 a 1, atribuindo estes valores para cada um dos sentimentos possíveis: negativo, neutro e positivo.

O uso da análise léxica apresenta algumas desvantagens significativas. Sua utilização é dependente, primeiramente, da existência de uma base previamente classificada e disponível para o idioma adotado. O uso de termos pré computados também pode afetar a classificação de uma sentença, pois algumas palavras, quando inseridas em contextos diferentes, podem representar sentimentos opostos. Por estas razões, optou-se por utilizar um método baseado em aprendizado de máquina, livre de restrições relativas a dicionários já existentes.

O classificador *Naive Bayes* pode ser considerado um dos principais métodos conhecidos para problemas relacionados a aprendizagem de máquina. Segundo pesquisa realizada por Mitchell (1997), esta abordagem, para muitos casos, apresenta um nível de confiabilidade semelhante a outros algoritmos complexos, como árvores de decisão e redes neurais. Este modelo faz uso de fórmulas estatísticas e cálculos de probabilidade para realizar a classificação. Através do Teorema de *Bayes*, o classificador calcula a probabilidade de uma evidência $E = e_1, e_2 \dots e_n$ pertencer a uma classe de acordo com a seguinte fórmula.

$$P(classe|E) = \frac{P(e_1 ... e_n|classe) \times P(classe)}{P(e_1 ... e_n)} \tag{5.1}$$

Para definir a classe mais provável de uma evidência, calcula-se a probabilidade de todas as classes existentes, para em seguida, escolher a classe com a probabilidade mais alta de ocorrência. Em termos estatísticos, isso é análogo a maximizar o valor de $P(classe|E)$. O cálculo de $P(e_1 \dots e_n|classe)$ torna-se complexo de ser obtido, pois possivelmente não há uma ocorrência similar presente na base de treinamento. Para este cenário, o classificador *Naive Bayes* realiza uma simplificação, onde "ingenuamente" considera

que as evidências são fatores independentes. Portanto, o cálculo é reduzido ao produto das probabilidades de cada evidência individual. Aplicando estas observações e eliminando a constante $P(e_1 \dots e_n)$, a fórmula pode ser reduzida, como apresentada a seguir.

$$arg\ max\ P(classe|E) = \prod_{i=1}^{n} P(e_i|classe) \times P(classe) \tag{5.2}$$

Apesar de ser possível perceber que a afirmação, na qual as evidências são atributos independentes, é falsa na maioria das vezes, segundo Mitchell (1997), o classificador *Naive Bayes*, mesmo assim, é capaz de produzir resultados consideravelmente satisfatórios. Por estas razões, optou-se por utilizar este método para classificar os comentários coletados nas redes sociais.

Para cada um dos candidatos analisados, foi construída uma base de treinamento independente, com aproximadamente 300 publicações retiradas do Facebook e Twitter, onde cada comentário foi classificado de acordo com o sentimento expresso pelo autor, pertencendo a classe **positivo** ou **negativo**. Após a fase de elaboração das bases de treinamento, iniciou-se a etapa de coleta e classificação das mensagens. A figura 21 apresenta as atividades executadas durante este processo.

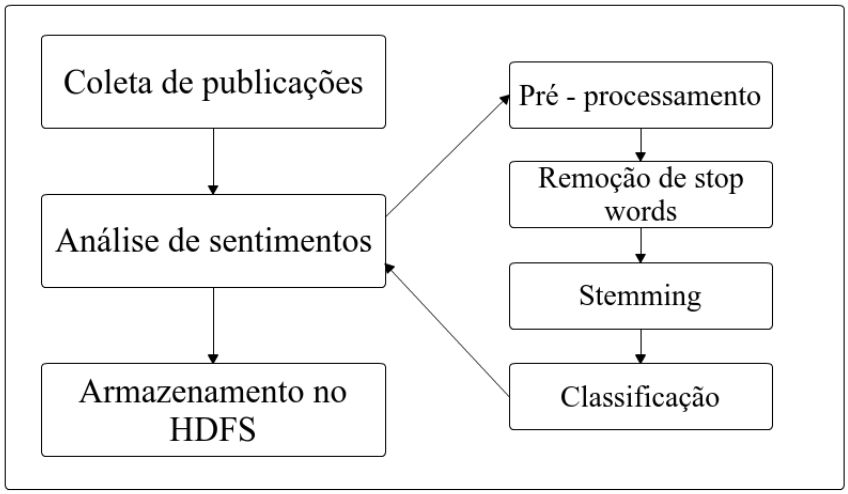

Figura 21 – Processo de análise de sentimentos
Fonte: Autor

Após uma mensagem ser coletada de uma fonte de dados, inicia-se o processo de análise de sentimentos. Primeiramente, o texto recebe uma filtragem inicial, onde são retiradas quebras de linha e caracteres especiais, além da eliminação de *stop words*, ou seja, palavras irrelevantes para o processamento, como preposições. Na etapa de *stemming*, todas as palavras são reduzidas a seu radical, para na sequência, serem submetidas ao classificador *Naive Bayes*. Toda etapa de análise de sentimentos ocorre antes de uma publicação ser escrita no *file channel* do Flume.

Considerando que o escopo deste trabalho não abrange a implementação de um método de classificação, foram utilizados os seguintes projetos *open source* para auxílio na etapa de análise de sentimentos.

- Java *Naive Bayes Classifier*[13]: implementação Java do classificador *Naive Bayes*.
- PTStemmer[14]: biblioteca Java para utilização de *stemming* na língua portuguesa.

A fase de análise de sentimentos ocorre antes de uma publicação ser escrita no *file channel* do Flume, as implementações dos *sources* utilizam as funcionalidades do classificador como uma biblioteca externa.

5.6 Processamento dos Dados

Até esta etapa, foram descritos os passos necessários para que as publicações de redes sociais fossem coletadas, classificadas pelo algoritmo *Naive Bayes* e armazenadas no HDFS. Nesta seção, será apresentada a organização destes arquivos no *cluster*, de forma que os dados possam ser facilmente consultados e agrupados, permitindo uma análise geral dos resultados obtidos.

Após serem salvas no HDFS, as mensagens são persistidas em arquivos no formato JSON e ficam disponíveis em um diretório do sistema de arquivos distribuído. Ao término de uma fase de coleta de dados, todos os arquivos deste diretório são carregados em uma tabela do Hive. Nesse processo, os arquivos são fisicamente transferidos para o diretório de administração da própria ferramenta de *data warehouse*.

Segundo White (2012), cada registro no Hive pode ser do tipo *row format*, ou *file format*. O tipo *row format* especifica como cada linha de uma tabela deverá ser armazenada, para isso é necessário um serializador e deserializador de objetos - *SerDe* - capaz de interpretar este formato. A tabela criada para manter as publicações de redes sociais utiliza um *SerDe*[15] para arquivos no formato JSON. A vantagem de persistir as mensagens desta forma se dá pela capacidade de representar objetos complexos de maneira simples e leve. O quadro 9 apresenta o script para gerar a tabela de mensagens, composta pelos seguintes campos: *author* (autor); *message* (corpo do texto); *politician* (candidato mencionado); *classification* (resultado da análise de sentimentos); *year* (ano); *month* (mês); *day* (dia); *type* (rede social onde a mensagem foi publicada).

```
create table social_messages(
  timestamp STRING,
```

[13] <https://github.com/ptnplanet/Java-Naive-Bayes-Classifier>
[14] <http://code.google.com/p/ptstemmer/>
[15] <https://github.com/rcongiu/Hive-JSON-Serde>

```
  year STRING,
  month STRING,
  day STRING,
  message STRING,
  author STRING,
  politician STRING,
  classification STRING,
  type STRING
)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
STORED AS TEXTFILE;
```

Quadro 9 – Comando HiveQL para criar tabela de mensagens
Fonte: Autor

Após carregados os registros nesta tabela, é possível acessar os dados utilizando a linguagem HiveQL, semelhante ao uso de consultas SQL. Ao solicitar um procedimento HiveQL, a consulta será convertida para um *MapReduce job* e executada no *cluster*, todavia, isso ocorre de forma transparente para o usuário deste serviço. Desta forma, torna-se viável o agrupamento de alguns campos para gerar relatórios, permitindo uma análise geral sobre as mensagens capturadas de cada candidato.

Para obter resultados que apresentem a quantidade de mensagens positivas e negativas de um determinando candidato, em intervalos mensais, foi criada uma tabela para agrupar os campos: *politician*, *classification*, *year*, *month*, *type*. O quadro 10 detalha o *script* utilizado para popular esta tabela, oriunda do agrupamento dos campos e contagem dos valores.

```
INSERT OVERWRITE TABLE analysis_result
select politician, classification, type, year, month, count(*)
from social_messages
group by politician, classification, type, year, month;
```

Quadro 10 – Comando HiveQL para gerar de registros para tabela de resultados
Fonte: Autor

O povoamento desta tabela deve ser realizado sempre que novos registros forem carregados na tabela de mensagens. A cláusula *OVERWRITE* indica que os dados da tabela são excluídos antes da inserção de novas informações.

O objetivo da criação de uma tabela de resultados é permitir que aplicações em tempo real apresentem relatórios sobre os dados analisados. O Hive é uma ferramenta de *data warehouse* poderosa e robusta, porém não foi projetada para prover acesso com baixa latência. Portanto, foi construída uma tabela análoga em uma banco de dados relacional MySQL. A figura 22 ilustra a organização das tabelas no Hive e MySQL.

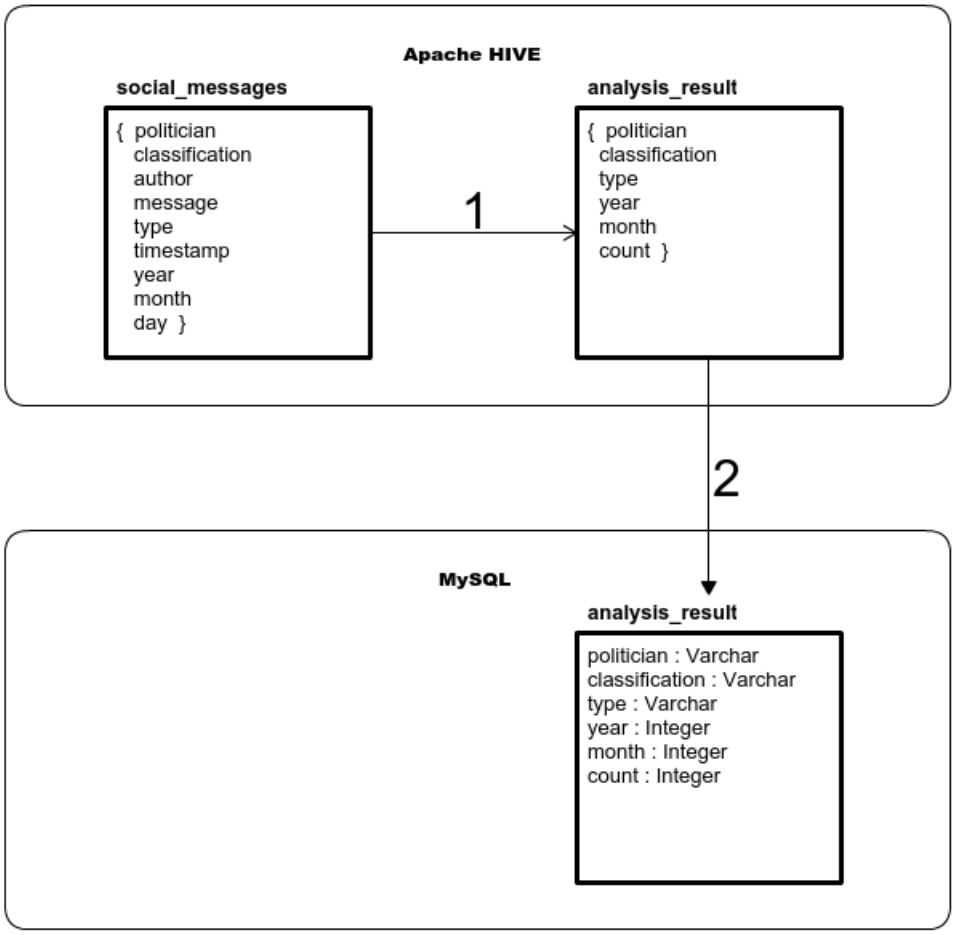

Figura 22 – Organização das tabelas
Fonte: Autor

A imagem 22 também representa o fluxo da informação até chegar ao MySQL. Inicialmente, a grande quantidade de mensagens persistidas na tabela *social_messages* são processadas e armazenadas em uma tabela intermediária no próprio Hive. A seta de número 1 indica a execução do comando HiveQL apresentado no quadro 10. Através do Sqoop, representado pela seta de número 2, realiza-se a importação dos registros da tabela *analysis_result* para uma tabela idêntica, residente no MySQL. Com esta arquitetura, o Hive permite analisar informações em larga escala e de forma linear, enquanto o banco de dados relacional é usado para apresentar os resultados em tempo real.

5.7 Exibição dos Resultados

Nesta seção, será apresentada a última etapa do estudo de caso proposto por este trabalho. O objetivo desta fase é construir uma interface *web* que apresente os resultados obtidos após a análise de sentimentos e processamento de todas as mensagens coletadas. Os requisitos deste componente podem ser expressos nos seguintes itens.

- Apresentar a quantidade total de mensagens retiradas por cada uma das redes sociais usadas durante a etapa de busca por publicações.
- Apresentar graficamente os resultados da análise de sentimentos concluída sobre as mensagens dos candidatos observados.

A partir destas necessidades, se iniciou a fase de elaboração e implementação da aplicação. A construção deste componente foi feita com uso da linguagem de programação Java. Para auxílio no desenvolvimento *web*, foi utilizado o *framework* Vraptor[16], elaborado pela empresa Caelum[17]. Uma das principais características desta desta tecnologia é o foco em alta produtividade e na simplicidade do código, onde o tempo para configurar a ferramenta e escrever códigos repetitivos são otimizados para que o desenvolvedor possa trabalhar especialmente na lógica do negócio. A alta curva de aprendizado e a clareza do código, facilitando a manutenção do sistema, também são atributos positivos deste modelo.

Ao contrario de paradigmas de desenvolvimento *web* baseados em componentes, como JSF[18], o Vraptor é um *framework* MVC[19] que trabalha de maneira *RESTFul*, ou seja, é possível associar uma determinada *URL* a um método de uma controladora, que posteriormente torna-se responsável por executar a lógica da aplicação. Uma das grandes vantagens desta tecnologia é a automatização da conversão de parâmetros da requisição HTTP em objetos esperados pela controladora, além da facilidade de controle da navegação das páginas.

Para o desenvolvimento da interface gráfica, utilizou-se o *framework* Twitter *Bootstrap*[20]. Este projeto *open source* é uma das ferramentas mais conhecidas para customização e criação de páginas HTML. Para incluir o *Bootstrap* no projeto, basta importar seus arquivos CSS[21] e suas funções escritas em Javascript. Com isso, é possível usar seus componentes reutilizáveis e extensivos, permitindo o desenvolvimento de interfaces responsivas e estilizadas.

A imagem 23 apresenta a arquitetura planejada para este módulo *web*.

16 <http://www.vraptor.org/pt/>
17 <https://www.caelum.com.br/>
18 <http://pt.wikipedia.org/wiki/JavaServer_Faces>
19 Padrão de projeto de software que separa a lógica da aplicação da interação do usuário com o sistema.
20 <http://getbootstrap.com/>
21 Mecanismo usado para adicionar estilos a documentos *web* escritos em linguagem de marcação, como HTML ou XML.

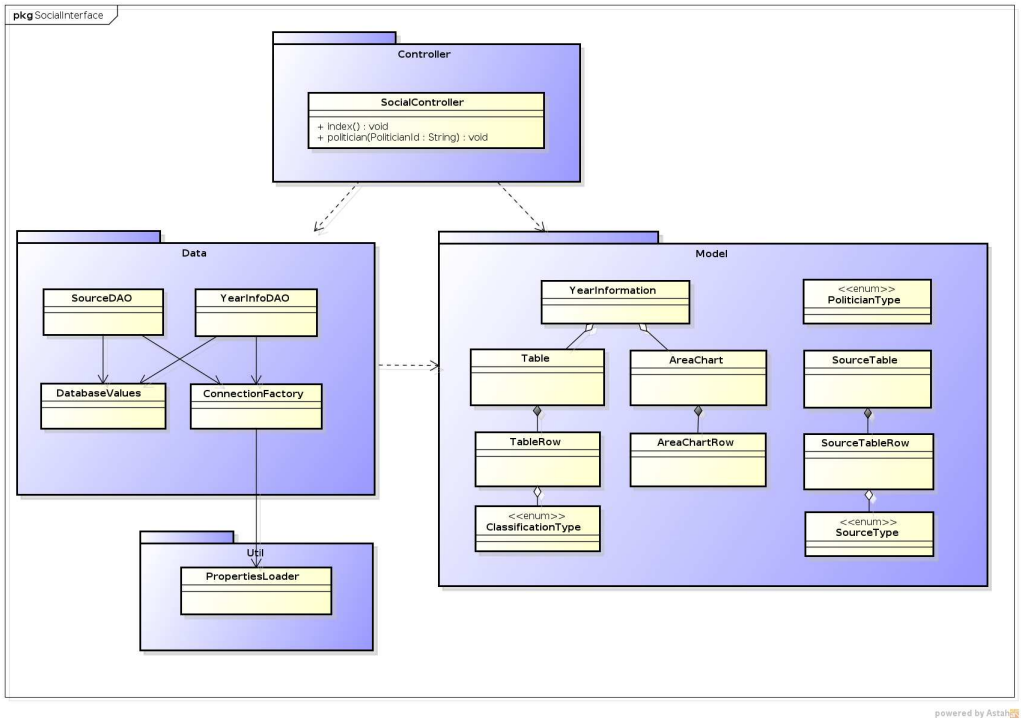

Figura 23 – Arquitetura da interface *web*
Fonte: Autor

O pacote *model* contém as classes de negócio, responsáveis por armazenar todas as informações que serão exibidas na interface gráfica. Já o pacote *data*, engloba as classes que realizam consultas à tabela de resultados persistida no MySQL, além de converter estes dados para objetos do pacote *model*. A classe *PropertiesLoader* permite carregar as propriedades do banco de dados, que estão registradas em um arquivo físico no diretório raiz do projeto. Por fim, no pacote *controller* está presente a controladora do Vraptor, responsável por gerenciar e responder as requisições feitas à aplicação.

6 Resultados e Discussões

A partir das especificações descritas no capítulo 5, foram implementados todos os componentes que fazem parte da arquitetura proposta para o estudo de caso deste trabalho. Neste capítulo apresentam-se os resultados obtidos após a etapa de desenvolvimento. Na seção 6.1, são exibidos os resultados encontrados após a etapa de coleta de dados. Na seção 6.2, discute-se a fase de análise de sentimentos das mensagens pesquisadas. Por fim, na seção 6.3, são apresentados os resultados do processamento ocorrido no Hive e também da construção da interface *web*.

6.1 Coleta de Dados

Durante o processo de busca por postagens provenientes das redes sociais escolhidas como fonte de pesquisa, foram coletados aproximadamente 600MB de arquivos no formato JSON. Estes dados correspondem às mensagens publicadas no período entre janeiro de 2014 e abril de 2015, além de conter outras informações relevantes, como especificado no modelo de dados descrito na figura 22. A tabela 10 apresenta a quantidade exata de mensagens obtidas a partir das APIs do Facebook e Twitter, combinadas com a implementação de *souces* do Flume.

Fonte de dados	Quantidade de mensagens
Facebook	1612701
Twitter	113990

Tabela 10 – Quantidade de mensagens coletadas
Fonte: Autor

A busca pela coleta de uma grande quantidade de mensagens acabou se tornando um dos principais obstáculos encontrados nesta fase do trabalho. As limitações impostas pelas APIs das redes sociais influenciaram diretamente para que o número de publicações obtidas não apresentasse um resultado mais expressivo, quando comparado ao potencial destes meios. O Twitter, como mencionado anteriormente, apenas permite buscas por mensagens recentes, além de limitar o número de requisições submetidas a Search API. Durante uma janela de 30 minutos, constatou-se que é possível obter uma média de 4MB de informações, até o momento em que a busca falha, retornando um código de erro indicando que o limite de requisições foi atingido. Portanto, a pesquisa realizada no Twitter resultou em um número limitado de mensagens, as quais compreendem apenas o mês de abril de 2015, período no qual se iniciou o processo de busca.

O uso da Graph API permitiu a coleta de todas as publicações presentes nas páginas oficiais dos candidatos no Facebook. Diferente do Twitter, não houve restrições temporais ou quantitativas sobre estes dados. Entretanto, a principal desvantagem desta abordagem ocorre devido à proibição e limitação na busca por mensagens publicadas em páginas pessoais, a qual representa uma fonte de informações significativamente superior, quando comparado ao número restrito de páginas permitidas e utilizadas neste trabalho.

6.2 Análise de Sentimentos

Todas as publicações capturadas pelo Flume foram classificadas como uma sentença positiva ou negativa, considerando o candidato pesquisado e a opinião expressa no corpo do texto a seu respeito. Nesse contexto, a análise de sentimentos se apresentou como um método que viabilizou o processamento de grande volume de informações, permitindo análises e reflexões pessoais sobre os resultados encontrados. A tabela 11 apresenta algumas mensagens coletadas e suas respectivas classificações obtidas após a etapa de análise de sentimentos.

Candidato	Mensagem	Classificação Esperada	Classificação Obtida
Aécio Neves	A sanha golpista do tucano Carlos Sampaio impede-o de enchergar a um palmo do nariz. Até o neogolpista Aécio Neves busca sair pela direita.	Negativo	Negativo
Aécio Neves	O derrotado Aécio das Neves o golpista ressentido excremento das magoas de seu criador FHC o rejeitado que quebrou o Brasil por 3 vezes.	Negativo	Positivo
Aécio Neves	Parabéns, Aécio. Continue a divulgar a sua simpatia.	Positivo	Positivo
Aécio Neves	O estado de Goiás está com Aécio!	Positivo	Negativo
Dilma Rousseff	fora Dilma Rousseff pra sempre longe de nos. Fora palhaços do PT.	Negativo	Negativo
Dilma Rousseff	A Dilma Rousseff vai entregar o Brasil para o Bradesco? Puxa vida, acabei de ver na TV as taxas de juros a.a. #ForaPT	Negativo	Positivo
Dilma Rousseff	Dilma 13 com certeza. E a melhor opção. Se Deus quiser ela vence. Ias já ganhou.	Positivo	Positivo
Dilma Rousseff	Cai mais um juiz contratado pra perseguir e prender petistas! Viva Lula, viva nós, viva Dilma Rousseff!	Positivo	Negativo

Tabela 11 – Amostra dos resultados obtidos
Fonte: Autor

O baixo número de mensagens previamente classificadas, gerando uma base de treinamento pequena para ambos os candidatos, e também os diferentes contextos onde as mensagens foram publicadas podem ter influenciado diretamente na qualidade dos resultados obtidos pela análise de sentimentos. Entretanto, o escopo deste trabalho não abrange a verificação e validação da eficácia e confiabilidade na utilização do classificador *Naive Bayes*.

6.3 Processamento dos Dados e Exibição dos Resultados

Após todo processo de coleta de dados, as mensagens citadas na tabela 10 foram armazenadas na tabela *social_messages*, persistida na ferramenta de *data warehouse* Hive. A etapa de processamento destas informações consiste no agrupamento de alguns campos desta tabela, transferindo os resultados para uma tabela intermediária chamada *analysis_result*. Apesar da quantidade de registros não ter alcançado o valor esperado para uma aplicação Big Data, a arquitetura proposta neste trabalho foi construída para garantir escalabilidade linear, ou seja, a estrutura do modelo se torna independente do volume de dados analisado. Portanto, o uso desta abordagem proporcionou a simulação de um ambiente voltado para situações onde é necessário processamento de grande quantidade de dados.

Após a execução do comando HiveQL para agrupar a tabela *social_messages*, o Hive converte esta instrução para um programa *MapReduce*, que será submetido ao *cluster* onde estão persistidos os dados. Esta etapa resultou em um *MapReduce job* com 3 *map tasks* e 2 *reduce tasks*, o qual foi finalizado em aproximadamente 34 segundos. A figura 24 apresenta a tela de execução deste comando HiveQL, descrito no quadro 10.

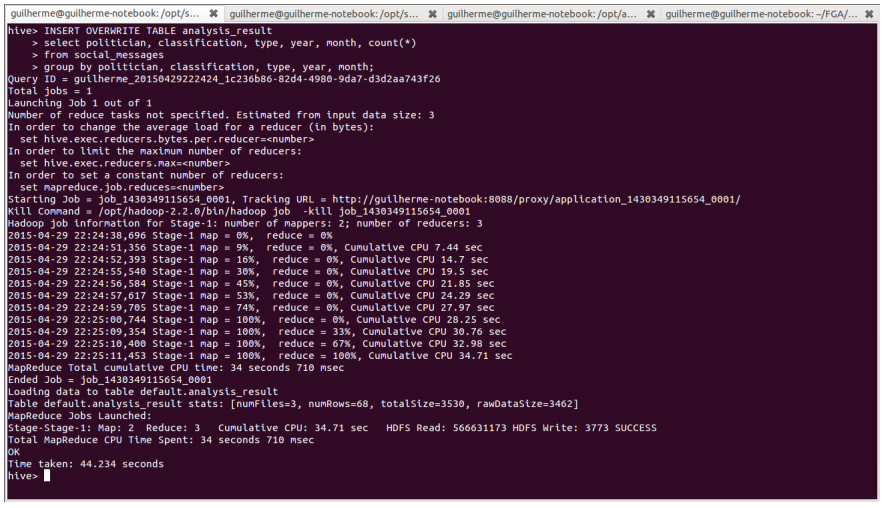

Figura 24 – Tela de execução do comando HiveQL
Fonte: Autor

Os resultados obtidos após a execução do comando HiveQL foram armazenados na tabela *analysis_result* e posteriormente transferidos para uma tabela análoga localizada no banco relacional MySQL. A transição entre os dois ambientes, Hive e MySQL, ocorreu com a utilização da ferramenta Sqoop. Os valores desta tabela são exibidos graficamente através da interface *web* construída de acordo com a especificação contida no capítulo 5. Para cada candidato, é possível visualizar a quantidade de mensagens positivas e negativas para os anos de 2014 e 2015. A figura 25 apresenta a página de um dos candidatos analisados.

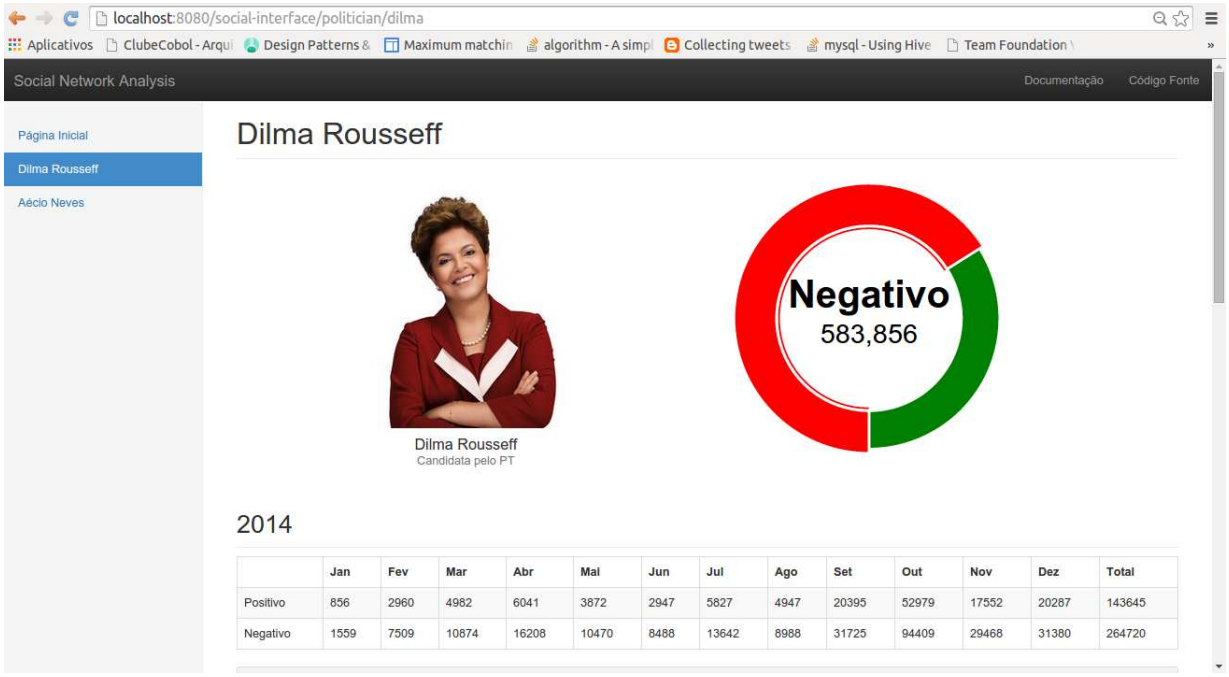

Figura 25 – Tela de resultados para a candidata Dilma Rousseff
Fonte: Autor

Cada página de um candidato, além do gráfico com os resultados de todas as mensagens analisadas, também possui, para cada ano, uma tabela com os valores mensais e um gráfico de área ilustrando a análise de sentimentos ao longo dos meses. A figura 25 apresenta o gráfico de área referente ao ano de 2015 para o mesmo candidato mostrado na imagem anterior.

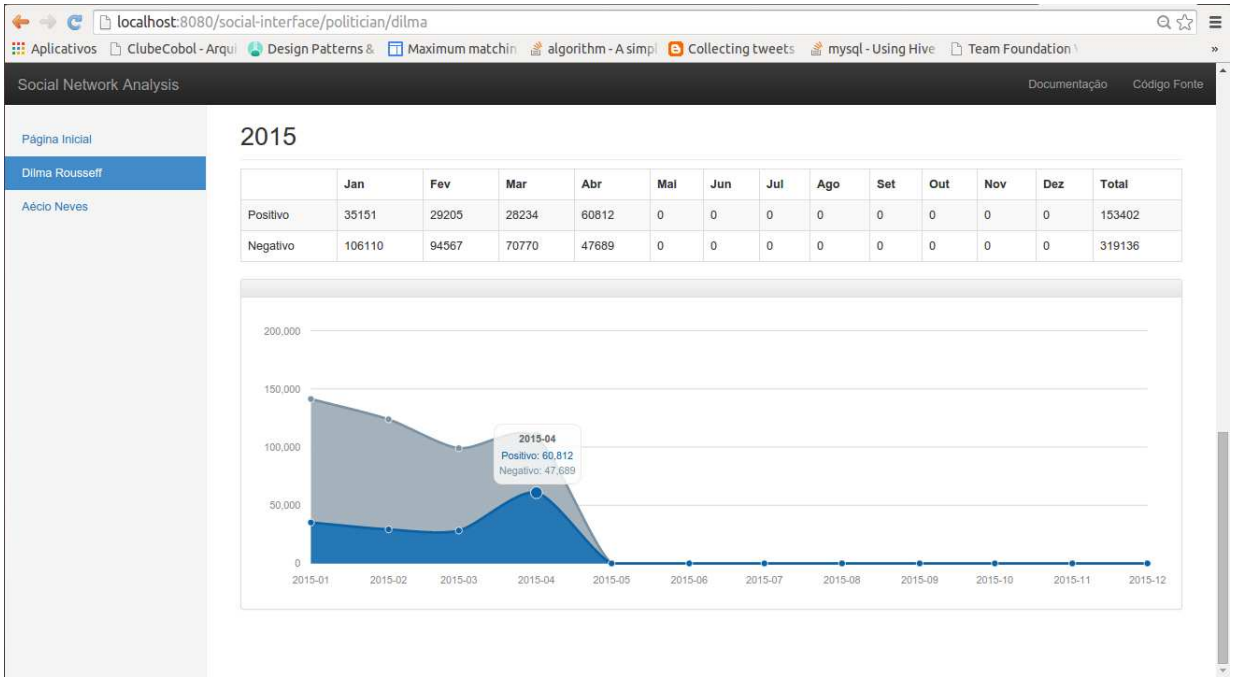

Figura 26 – Gráfico de Área para o ano de 2015
Fonte: Autor

Na figura 27, é ilustrada a página referente a outro candidato analisado, a qual apresenta o mesmo padrão de tela evidenciado na figura 25.

	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez	Total
Positivo	6148	8049	13831	19749	15216	18368	15142	21096	45347	218154	79567	63998	524665
Negativo	809	1393	2289	3497	2721	2432	2075	2625	5557	24624	9064	6544	63630

Figura 27 – Tela de resultados para o candidato Aécio Neves
Fonte: Autor

7 Considerações Finais

Ao término de todas as etapas realizadas ao longo deste trabalho, conclui-se que os objetivos almejados foram alcançados com êxito.

Na etapa inicial, apresentou-se um estudo em torno da arquitetura e funcionamento do projeto Hadoop, uma das soluções mais conhecidas entre as alternativas existentes para análise de dados no contexto Big Data. A compreensão do novo paradigma proposto pelo modelo *MapReduce* é o primeiro passo necessário para a inserção de um engenheiro de software no desenvolvimento de aplicações voltadas para processamento de dados em larga escala. Mesmo em situações onde são utilizadas ferramentas com um nível de abstração maior, como o Hive, o conhecimento deste modelo torna-se indispensável, pois grande parte dos projetos construídos no topo do Hadoop utilizam internamente o *MapReduce*.

Os principais pontos sobre a arquitetura e funcionamento do Hadoop foram elencados e detalhados exaustivamente, contribuindo para o amadurecimento sobre o uso desta tecnologia na resolução de problemas voltados para esse contexto. Também foram abordadas as ferramentas essenciais que compõe o ecossistema Hadoop, apresentando os possíveis cenários em que podem ser utilizadas.

Com este trabalho, foi possível perceber que os objetivos e necessidades do negócio impactam diretamente no tipo de arquitetura que deve ser escolhida para realizar análise de dados em larga escala. O *framework MapReduce* mostrou-se uma alternativa muito interessante para a construção de aplicações que exigem processamento em *batch* de um grande volume de informações. Porém o grande ponto negativo deste modelo está na limitação para operações de escritas e leituras randômicas em arquivos, ou seja, para situações em que são exigidas interações com o usuário a melhor opção é a escolha de um banco de dados NoSQL, como o HBase.

Em casos onde os requisitos se encaixam com as características propostas por uma ferramenta de *data warehouse*, o software Hive é uma escolha a se considerar, pois através de uma linguagem baseada em SQL é possível elaborar consultas poderosas sem a necessidade de codificar *MapReduce jobs*, pois esse processo é feito internamente.

O período de especificação e implementação da arquitetura do estudo de caso possibilitou que fossem exploradas várias etapas na construção de um modelo de aplicação completo, permitindo a simulação de um ambiente real, iniciando na busca por mensagens em APIs de redes sociais, passando pelo processamento dessas informações e finalizando com a apresentação dos resultados por uma interface *web*.

A execução da fase de definição e construção do modelo descrito no capítulo 5

também foi acompanhada pela continuação da etapa de pesquisa, sendo necessária a investigação de novas ferramentas, tecnologias e métodos que incrementassem e viabilizassem a implementação do estudo de caso proposto. Portanto, a atividade de pesquisa se perpetuou por, praticamente, todo o trabalho, se diferenciando de outras metodologias de desenvolvimento de software, onde a etapa de especificação segue um padrão mais rígido e detalhado, pois, geralmente, já existe o conhecimento teórico necessário para a etapa de construção.

Outro aspecto importante foi identificado ao longo da etapa de implementação do estudo de caso. Durante o período de coleta de informações, não foi possível capturar um número expressivo de mensagens das redes sociais, quando comparado ao volume de dados que se espera de aplicações voltadas para o contexto Big Data. As limitações impostas pelas APIs de busca se tornaram o principal fator para que se alcançasse esses resultados. Este cenário revelou a dificuldade ao tentar obter informações em larga escala de companhias privadas e produtoras de conteúdo.

7.1 Trabalhos Futuros

Ao longo de todas as etapas deste trabalho, foram exploradas diversas áreas que, apesar de já terem sido abordadas, possibilitam a expansão dos resultados apresentados. A seguir, são listadas algumas sugestões para continuação da pesquisa iniciada neste trabalho.

1. Ampliar a coleta de informações para que seja possível adquirir uma quantidade de dados próxima aos requisitos exigidos por aplicações voltadas para o contexto Big Data. Após esta etapa, verificar e validar a escalabilidade linear oferecida pelo Hadoop.

2. Incrementar a base de treinamento utilizada no algoritmo *Naive Bayes*, definindo métodos de classificação mais sofisticados para as diferentes redes sociais existentes.

3. Alterar o agrupamento da tabela de mensagens na ferramenta Hive, de forma que seja possível identificar os resultados da análise de sentimentos em um pequeno espaço de tempo. Portanto, pode ser possível relacionar variações nos resultados obtidos, com fatos e acontecimentos no mesmo período.

4. Utilizar o HBase para permitir consultas a uma parcela das mensagens persistidas no *cluster*, possibilitando a exibição através da interface *web*.

Referências

ABOUT Twitter, Inc. | About. Disponível em: <https://about.twitter.com/company>. Citado na página 64.

ARAúJO, M.; GONçALVES, P.; BENEVENUTO, F. Métodos para análise de sentimentos no twitter. 2013. Citado na página 70.

CATTELL, R. Scalable sql and nosql data stores. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 39, n. 4, p. 12–27, may 2011. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/1978915.1978919>. Citado 2 vezes nas páginas 53 e 54.

CHANG, F. et al. Bigtable: A distributed storage system for structured data. In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. Berkeley, CA, USA: USENIX Association, 2006. (OSDI '06), p. 15–15. Disponível em: <http://dl.acm.org/citation.cfm?id=1267308.1267323>. Citado na página 55.

COMPANY Info | Facebook Newsroom. Disponível em: <http://newsroom.fb.com/company-info/>. Citado na página 62.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos - 4ed: Conceitos e Projeto*. [S.l.]: BOOKMAN COMPANHIA ED, 2007. ISBN 9788560031498. Citado na página 23.

DARWIN, I. F. *Java Cookbook*. Third edition. O'Reilly Media, 2014. ISBN 9781449337049. Disponível em: <http://amazon.com/o/ASIN/144933704X/>. Citado na página 41.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. *Commun. ACM*, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>. Citado 7 vezes nas páginas 23, 30, 32, 33, 34, 35 e 43.

DING, M. et al. More convenient more overhead: The performance evaluation of hadoop streaming. In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*. New York, NY, USA: ACM, 2011. (RACS '11), p. 307–313. ISBN 978-1-4503-1087-1. Disponível em: <http://doi.acm.org/10.1145/2103380.2103444>. Citado na página 37.

FACEBOOK Page. Disponível em: <https://developers.facebook.com/docs/graph-api/reference/v2.3/page>. Citado na página 64.

GANTZ, D. R. J. *Extracting Value from Chaos*. Framingham, MA, 2011. Disponível em: <http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf>. Citado 2 vezes nas páginas 19 e 20.

GEORGE, L. *HBase: The Definitive Guide*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2011. ISBN 9781449396107. Citado 4 vezes nas páginas 53, 55, 56 e 57.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. In: *ACM SIGOPS Operating Systems Review*. [S.l.]: ACM, 2003. v. 37, p. 29–43. Citado 3 vezes nas páginas 23, 26 e 27.

HDFS Architecture. 2013. Disponível em: <http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>. Citado na página 28.

HOFFMAN, S. *Apache flume distributed log collection for Hadoop : stream data to Hadoop using Apache Flume*. Birmingham: Packt Pub, 2013. ISBN 1782167919. Citado 2 vezes nas páginas 66 e 67.

INMON, W. *Building the Data Warehouse*. [S.l.]: Wiley, 2005. (Timely, practical, reliable). ISBN 9780764599446. Citado na página 48.

KIMBALL, R.; ROSS, M. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. [S.l.]: Wiley, 2013. ISBN 9781118732281. Citado na página 48.

MapReduce Tutorial. 2013. Disponível em: <http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html>. Citado na página 35.

MITCHELL, T. M. *Machine Learning*. New York: McGraw-Hill, 1997. (McGraw-Hill series in computer science). ISBN 0070428077. Citado 2 vezes nas páginas 70 e 71.

OAKS, S. *Java Performance: The Definitive Guide*. [S.l.]: "O'Reilly Media, Inc.", 2014. Citado na página 38.

SHVACHKO, K. et al. The hadoop distributed file system. In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA: IEEE Computer Society, 2010. (MSST '10), p. 1–10. ISBN 978-1-4244-7152-2. Disponível em: <http://dx.doi.org/10.1109/MSST.2010.5496972>. Citado 4 vezes nas páginas 24, 25, 29 e 47.

TANENBAUM, A. *Sistemas Operacionais Modernos*. 2. ed. [S.l.]: PRENTICE HALL (BRASIL), 2003. ISBN 9788587918574. Citado na página 23.

THUSOO, A. et al. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, VLDB Endowment, v. 2, n. 2, p. 1626–1629, aug 2009. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1687553.1687609>. Citado 4 vezes nas páginas 48, 49, 51 e 52.

VENNER, J. *Pro Hadoop*. 1st. ed. Berkely, CA, USA: Apress, 2009. ISBN 1430219424, 9781430219422. Citado 5 vezes nas páginas 24, 35, 39, 40 e 42.

WELCOME to Apache™ Hadoop®! 2014. Disponível em: <http://hadoop.apache.org/>. Citado na página 87.

WHITE, T. *Hadoop: The Definitive Guide*. Third edition. Beijing: O'Reilly, 2012. ISBN 9781449311520. Citado 27 vezes nas páginas 19, 20, 23, 24, 25, 26, 27, 29, 30, 35, 36, 37, 38, 39, 40, 41, 43, 44, 47, 49, 50, 55, 56, 66, 69, 72 e 87.

ZIKOPOULOS, P.; EATON, C. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 1st. ed. [S.l.]: McGraw-Hill Osborne Media, 2011. ISBN 0071790535, 9780071790536. Citado 2 vezes nas páginas 19 e 20.

APÊNDICE A – Manual de Instalação – Hadoop

Este documento tem como objetivo apresentar os passos necessários para a instalação de um *cluster* Hadoop versão 2.2.0. Este ambiente será constituído de uma máquina mestre, na qual estará localizado o *namenode* do HDFS e o *resource manager* do *framework MapReduce*, e também de máquinas escravas representando os *datanodes* e *node managers*. O Hadoop pode ser instalado em sistemas operacionais Unix ou Windows. Neste manual, o foco será a instalação em uma distribuição Linux, todas as máquinas utilizadas possuem Ubuntu versão 12.04 LTS.

O Hadoop foi construído utilizando a linguagem de programação Java, portanto o primeiro pré requisito consiste na instalação e configuração de uma JVM, versão 6 ou superior, em todas as máquinas. Segundo White (2012), a utilização da implementação da Sun é a mais comum neste contexto, portanto todas as máquinas foram configuradas com a JDK 7 da própria Sun.

O próximo passo é realizar o download do software Hadoop. A versão utilizada nesta instalação será a 2.2.0, caracterizada como uma versão estável. O programa pode ser obtido a partir do seguinte *link* disponibilizado pela documentação oficial (WELCOME..., 2014): <http://ftp.unicamp.br/pub/apache/hadoop/common/hadoop-2.2.0/hadoop-2.2.0.tar.gz>. O arquivo compactado contendo o Hadoop deve ser extraído para o local de sua preferência, neste caso foi escolhido o diretório /opt.

```
$ tar xzvf hadoop-2.2.0.tar.gz -C /opt
```

Para cada nó no *cluster* o Hadoop necessita de um diretório para armazenar suas informações. O *namenode* registra os metadados de todo o sistema, já um *datanode* utiliza este diretório para armazenar os blocos de arquivos. Diversos arquivos temporários também podem ser criados quando o *cluster* estiver em funcionamento, por isso o conteúdo deste ficheiro não deve ser alterado. O diretório especificado para este ambiente pode ser identificado no comando a seguir.

```
$ mkdir /opt/hadoop-hdfs/tmp
```

É recomendável a criação de um usuário e um grupo próprio para a utilização do Hadoop pelas máquinas do *cluster*. Esta abordagem facilita a administração dos diretórios e arquivos criados e gerenciados pelo Hadoop.

```
$ addgroup hadoop
$ sudo adduser --shell /bin/bash --ingroup hadoop hduser
```

O próximo passo é atribuir a este novo usuário as permissões para o diretório do Hadoop e também do diretório criado para armazenamento de seus arquivos.

```
$ chown -R hduser:hadoop /opt/hadoop-2.2.0 /opt/hadoop-hdfs
```

O arquivo /.bashrc deve ser atualizado com algumas variáveis de ambiente utilizadas pelo Hadoop, entre elas o caminho para JVM descrito na variável JAVA_HOME. Neste exemplo foi colocado o caminho para a JDK 7 da Oracle. Outro ponto importante é a atualização da variável PATH com os diretórios dos executáveis do Hadoop. As linhas a seguir devem ser inclusas ao fim do arquivo /.bashrc do usuário hduser.

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle
export HADOOP_HOME=/opt/hadoop-2.2.0
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export PATH=$PATH:$HADOOP_HOME/bin/:$HADOOP_HOME/sbin/
```

O arquivo hadoop-env.sh está localizado em $HADOOP_CONF_DIR e também deve ser alterado. A variável JAVA_HOME deve ser inclusa juntamente com a *flag* para desabilitar o protocolo IPV6, pois o Hadoop não tem suporte para tal recurso. As linhas a seguir devem ser inclusas no arquivo (a variável JAVA_HOME deve ser a mesma citada no arquivo /.bashrc).

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle
export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
```

O próximo passo é garantir que todas as máquinas da rede estejam realmente interconectadas. O *cluster* montado neste roteiro é composto por duas máquinas: um nó mestre (*namenode* e *resource manager*) e dois nós escravos (*datanode* e *node manager*), onde a máquina mestre também possui um nó escravo. O endereço IP da máquina mestre é representado pelo nome *master*, enquanto o endereço do escravo possui o nome *slave*.

Para que o *cluster* seja iniciado ou desligado o Hadoop executa um *script* que realiza chamadas SSH em todas as máquinas para então inicializar ou parar o serviço que cada uma delas executam. Portanto todos os nós do *cluster* devem ter um serviço de SSH

disponível.

As chamadas SSH realizadas pelo Hadoop devem realizar autenticação por criptografia assimétrica e não por senha. Para isso é necessário gerar um par de chaves pública/privada para o nó mestre, que será responsável por inicializar e interromper o *cluster* Hadoop. O comando a seguir é utilizado para criar um par de chaves RSA[1] e deve ser aplicado ao nó mestre.

```
$ ssh-keygen -t rsa -P ""
```

A chave pública é registrada no arquivo /.ssh/id_rsa.pub e deve ser copiada para o arquivo /.ssh/authorized_keys de todos os nós da rede. Desta forma a máquina mestre estará apta a realizar chamadas SSH sem autenticação por senha em todos os nós do *cluster*.

O próximo passo é no diretório $HADOOP_CONF_DIR editar os arquivos de configuração. As imagens a seguir ilustram o conteúdo que estes arquivos devem apresentar para todos os nós da rede.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
     <name>fs.default.name</name>
     <value>hdfs://master:9000</value>
  </property>
  <property>
      <name>hadoop.tmp.dir</name>
      <value>/opt/hadoop-hdfs/tmp</value>
  </property>
</configuration>
```

Quadro 11 – Arquivo core-site.xml

[1] <http://pt.wikipedia.org/wiki/RSA>

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
     <name>dfs.replication</name>
     <value>2</value>
   </property>
   <property>
     <name>dfs.permissions</name>
     <value>false</value>
   </property>
</configuration>
```

Quadro 12 – Arquivo hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <configuration>
     <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
     </property>
  </configuration>
</configuration>
```

Quadro 13 – Arquivo mapred-site.xml

O Hadoop 2.2.0 é constituído por uma versão mais atual do *framework MapReduce* denominada *MapReduce* 2.0, também conhecida como YARN. A principal diferença está no *jobtracker*, que foi dividido em dois *daemons* distintos: *resource manager* e *application master*. O antigo *tasktracker* foi substituído pelo *node manager*. O *resource manager* está presente no nó mestre e cada um dos escravos possui um *node manager*. A configuração do YARN é descrita nas imagens a seguir.

```
<?xml version="1.0"?>
<configuration>
    <property>
     <name>yarn.nodemanager.aux-services</name>
     <value>mapreduce_shuffle</value>
  </property>
  <property>
     <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
     <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
      <name>yarn.resourcemanager.resource-tracker.address</name>
      <value>master:8025</value>
  </property>
  <property>
      <name>yarn.resourcemanager.scheduler.address</name>
      <value>master:8030</value>
  </property>
  <property>
      <name>yarn.resourcemanager.address</name>
      <value>master:8045</value>
  </property>
  <property>
      <name>yarn.nodemanager.address</name>
      <value>master:8060</value>
  </property>
</configuration>
```

Quadro 14 – Arquivo yarn-site.xml para nó mestre

```
<?xml version="1.0"?>
<configuration>
    <property>
     <name>yarn.nodemanager.aux-services</name>
     <value>mapreduce_shuffle</value>
  </property>
  <property>
     <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
     <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
      <name>yarn.resourcemanager.resource-tracker.address</name>
      <value>master:8025</value>
  </property>
  <property>
      <name>yarn.resourcemanager.scheduler.address</name>
      <value>master:8030</value>
  </property>
  <property>
      <name>yarn.resourcemanager.address</name>
      <value>master:8045</value>
  </property>
  <property>
      <name>yarn.nodemanager.address</name>
      <value>slave:8060</value>
  </property>
</configuration>
```

Quadro 15 – Arquivo yarn-site.xml para nó escravo

O arquivo slaves do nó mestre é a última configuração a ser editada, ele está localizado no diretório $HADOOP_CONF_DIR. Deve ser incluso o endereço IP de todas as máquinas escravas do *cluster*. No ambiente montado para este manual a máquina mestre também é um escravo, portanto são inseridos os endereços *master* e *slave*.

Após toda a configuração do *cluster* Hadoop é necessário formatar o HDFS pela primeira vez. Este é o última passo da instalação e deve ser executado com o comando a seguir.

```
$ hadoop namenode -format
```

Nesta etapa todos os nós do *cluster* estão configurados e o HDFS está pronto para uso. Para iniciar o Hadoop os comandos a seguir devem ser executados.

```
$ hadoop-daemon.sh start namenode
$ hadoop-daemons.sh start datanode
$ yarn-daemon.sh start resourcemanager
$ yarn-daemons.sh start nodemanager
$ mr-jobhistory-daemon.sh start historyserver
```

Com o comando **$ jps** é possível verificar se os *daemons* ao longo do *cluster* foram iniciados com sucesso. A máquina mestre deve apresentar os seguintes processos: ResourceManager, DataNode, JobHistotyServer, Namenode e NodeManager. Para a máquina escrava os processos são: NodeManager e DataNode. O Hadoop também oferece alguns serviços HTTP que podem ser acessadas pelos endereços abaixo.

```
<http://master:50070/dfshealth.jsp>
<http://master:8088/cluster>
<http://master:19888/jobhistory>
```

Para que o *cluster* Hadoop seja desligado corretamente os comandos a seguir devem ser executados na ordem apresentada.

```
$ mr-jobhistory-daemon.sh stop historyserver
$ yarn-daemons.sh stop nodemanager
$ yarn-daemon.sh stop resourcemanager
$ hadoop-daemons.sh stop datanode
$ hadoop-daemon.sh stop namenode
```

APÊNDICE B – Manual de Instalação – *Plugin* Hadoop

A utilização de ferramentas de desenvolvimento é um fator que pode ser muito importante para prover uma maior produtividade e agilidade durante a construção de um software. O eclipse é uma das IDEs mais conhecidas entre os programadores java, uma das suas principais vantagens é a flexibilidade para adição de *plugins*, os quais podem ser desenvolvidos por qualquer pessoa e compartilhados com a comunidade de desenvolvedores.

A implementação de aplicações *MapReduce* pode ser facilitada com a utilização de um *plugin* disponível em um dos repositórios da ferramenta github. Para sua instalação é necessário realizar o download do código fonte no seguinte endereço: <https://github.com/winghc/hadoop2x-eclipse-plugin>.

Após efetuar o download os arquivos devem ser extraídos e o código fonte compilado. Desta forma será gerado um arquivo jar que contém o *plugin* para o Hadoop. Este arquivo deve ser inserido na pasta plugins/ dentro do diretório raiz do eclipse a ser instalado. Os comandos para compilar o código fonte são descritos a seguir.

```
$ unzip hadoop2x-eclipse-plugin-master.zip
$ cd  hadoop2x-eclipse-plugin-master/src/contrib/eclipse-plugin/
$ ant jar -Dversion=2.2.0 -Declipse.home=/opt/eclipse/ -Dhadoop.home=/
   opt/hadoop-2.2.0/
```

As diretivas “-Declipse.home” e “-Dhadoop.home” indicam a localização do eclipse e Hadoop, respectivamente. Após a compilação o arquivo jar referente ao *plugin* é gerado no diretório build/contrib/eclipse-plugin.