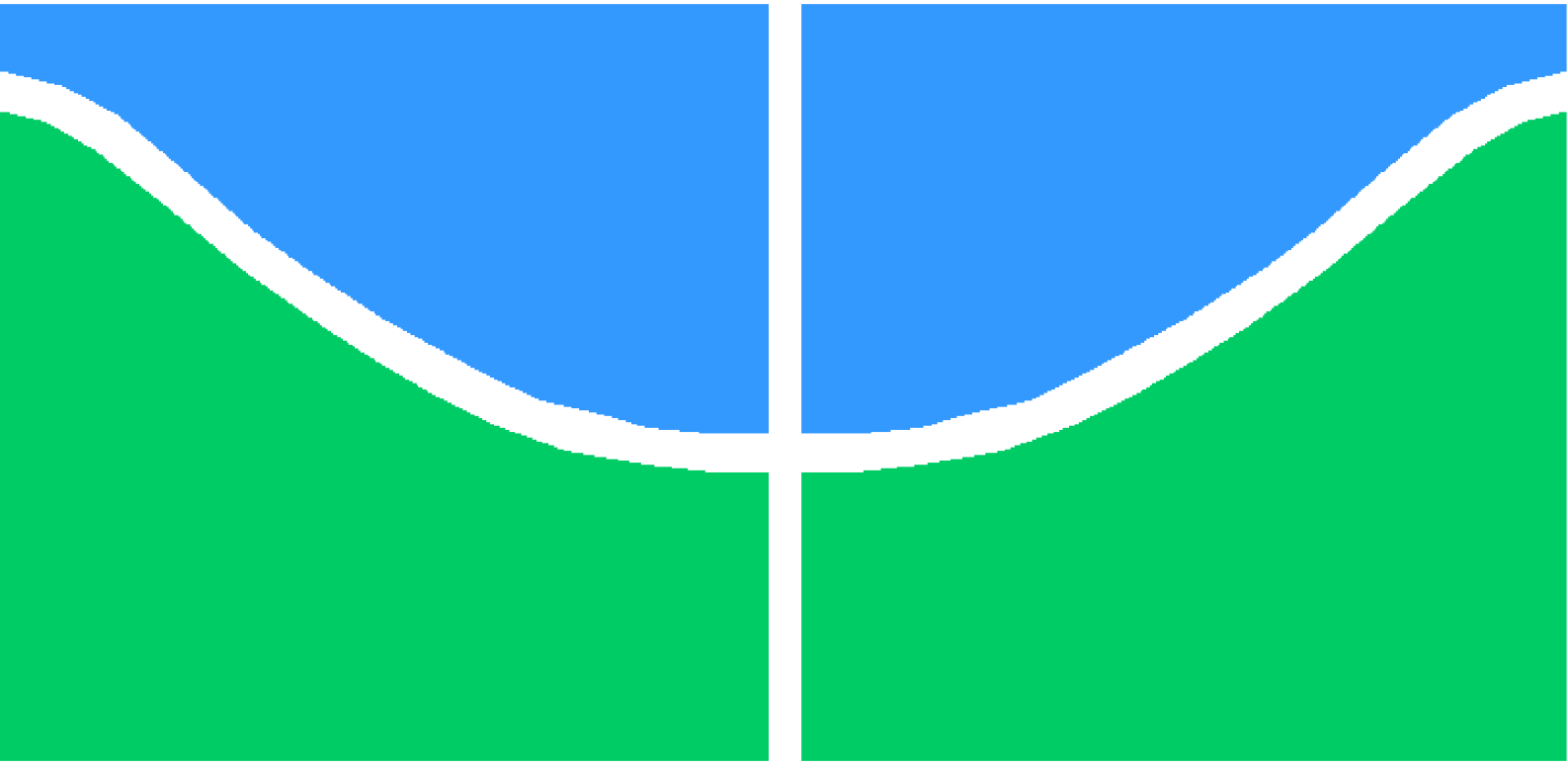

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Desenvolvimento de Software no Contexto Big Data

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

Brasília, DF
2014

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

2014

Guilherme de Lima Bernardes

Desenvolvimento de Software no Contexto Big Data / Guilherme de Lima Bernardes. – Brasília, DF, 2014-

73 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 , 2014.

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
2014

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 por exemplo, 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 Este trabalho tem como objetivo realizar um estudo sobre uma das arquiteturas mais conhecidas para solucionar estes problemas, o software Hadoop. O desenvolvimento para o paradigma MapReduce é abordado juntamente com os projetos que são construídos no topo do sistema Hadoop, provendo serviços em um nível de abstração maior. Ao fim da pesquisa é proposta uma arquitetura de aplicação Big Data voltada para um estudo de caso real, o qual envolve a extração e análise de publicações de redes sociais com foco voltado para políticas públicas.

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 This work aims to conduct a study on one of the most known architectures to solve these problems, Hadoop software. The development MapReduce paradigm is discussed together with the projects that are built on top of Hadoop system, which provides services at a higher level of abstraction. At the end of the research we propose an architecture for implementing Big Data faces a real case study, which involves the extraction and analysis of social networks with publications focus toward public policy.

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

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

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

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 por exemplo, 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 a 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 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 a tomadas de decisões em vários contextos. A utilização desta tecnologia associada a algoritmos de aprendizagem de máquinas, como 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 com 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 apresentado um estudo sobre a arquitetura e o modelo proposto por este software. Ao fim da etapa de pesquisa estes conceitos são aplicados em um estudo de caso real, onde será proposta uma arquitetura modelo para o cenário apresentado.

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 políticas públicas abordadas em redes sociais.

1.1.2 Específicos

Os objetivos específicos desse trabalho são apresentados a seguir:

1. Analisar a arquitetura e funcionamento do núcleo do Hadoop.
2. Descrever os passos necessários para o desenvolvimento de aplicações utilizando o paradigma MapReduce.
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.

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

6. Propor uma arquitetura modelo para o problema e definir próximos passos do trabalho.

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 se apresentou como uma das ferramentas mais utilizadas e consolidadas para este contexto. Portanto a próxima etapa consistiu em uma pesquisa para compreender e 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 sessão está nas ferramentes construídas no topo da arquitetura Hadoop, em específico o banco de dados NoSQL HBase e também o *data warehouse* distribuído Hive.

No capítulo 5 é apresentado o estudo de caso proposto para a continuação deste trabalho. Nesta sessão a arquitetura modelo para o problema é definida.

No capítulo 6 são discutidos os resultados obtidos após o término deste trabalho, onde são apresentados os próximos passos para a conclusão do TCC 2.

2 Hadoop

Uma das soluções mais conhecidas para análise de dados em larga escala é o projeto Apache Hadoop, concebido por Doug Cutting, o mesmo criador da biblioteca de busca textual Apache Lucene[1] (WHITE, 2012). Os componentes principais desta ferramenta consistem em um sistema de arquivos distribuídos para ser executado em *clusters*[2] 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 apêncice 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, como 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 arquivos distribuídos chamado Hadoop Distributed File System, ou HDFS.

---

1 <http://lucene.apache.org/core/>

2 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.

O HDFS foi projetado para executar MapReduce *jobs*[3], 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[4] 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.

[3] Um *job* pode ser interpretado como uma unidade de trabalho a ser executada pelo sistema, ou seja, o próprio programa MapReduce.
[4] 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 - Yahoo, extraída de White (2012), 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 *namedode* possui apenas o registro da localização exata de cada bloco no *cluster*. Os conceitos sobre *namenode* e *datanode* são apresentados na sessão 2.1.3.

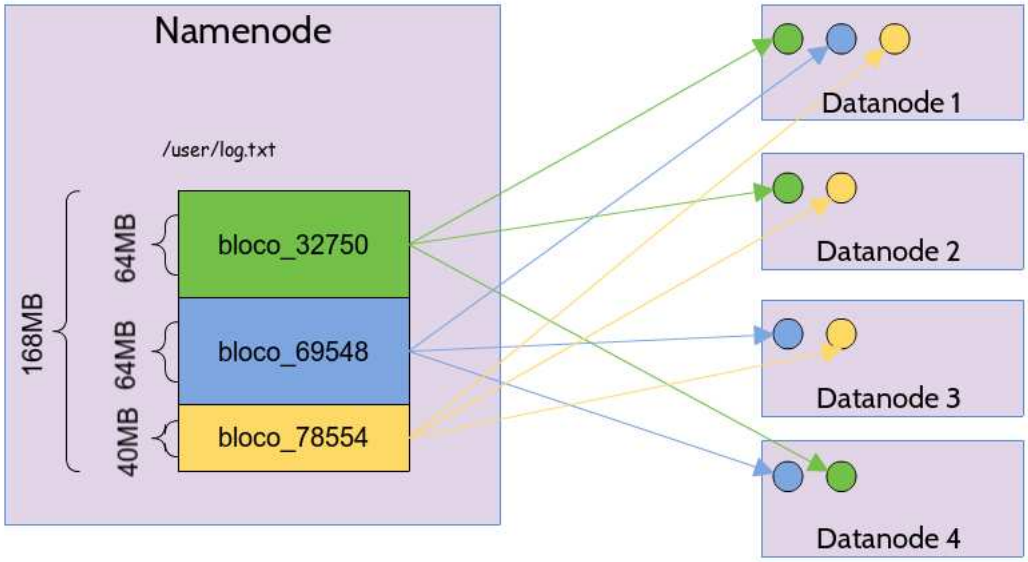

Figura 1 – Divisão de Arquivos em Blocos

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 uma tarefa 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 forma 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[5], 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).

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

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

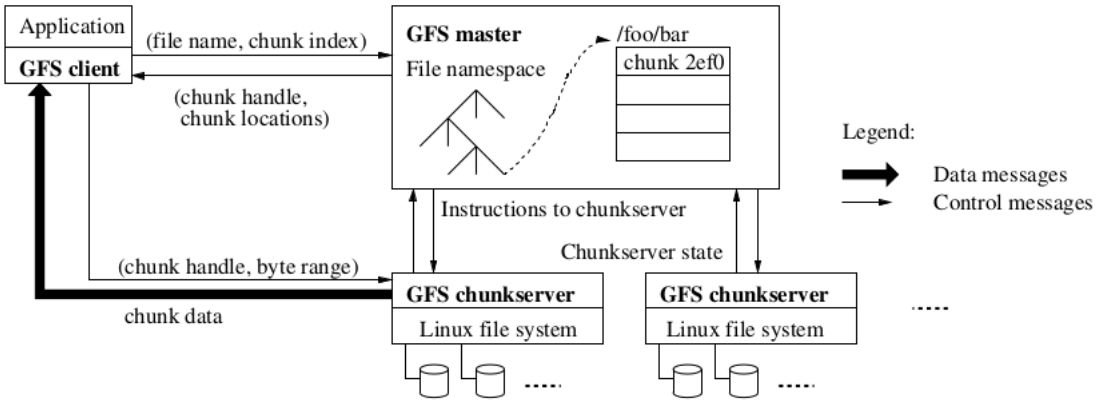

Figura 2 – Arquitetura GFS, retirado de Ghemawat, Gobioff e Leung (2003)

O HDFS foi construído a partir deste modelo proposto por Ghemawat, Gobioff e Leung (2003). Todas as características abordadas nas sessões anteriores, referentes ao HDFS, foram definidas com base neste trabalho. O HDFS apresenta uma alternativa *open-source*[6] 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.

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

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

armazenando, desta forma o *namenode* gerencia todos os *datanodes* presentes na rede. A figura 3 ilustra a arquitetura do HDFS.

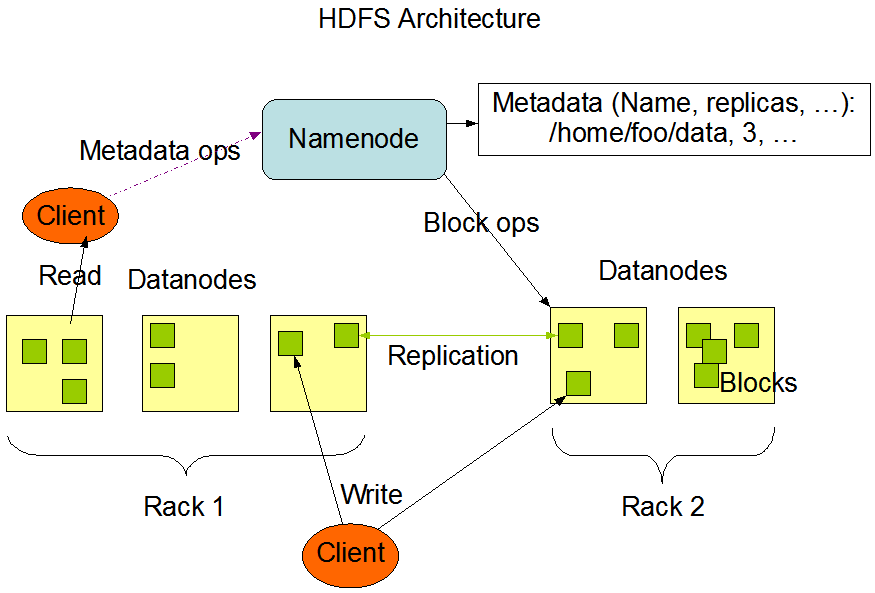

Figura 3 – Arquitetura HDFS, retirado de HDFS... (2013)

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.

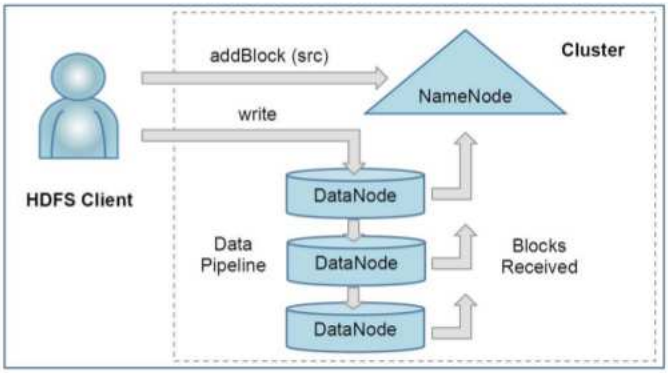

Figura 4 – Operação de escrita, extraído de Shvachko et al. (2010)

Para cada bloco do arquivo é realizado este 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*[7] 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

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

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 código 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)
```

Código 1 – Entradas e saídas - MapReduce, extraído de Dean e 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, valor}*. Cada *map task* realizada uma chamada a função *map* definida pelo usuário.

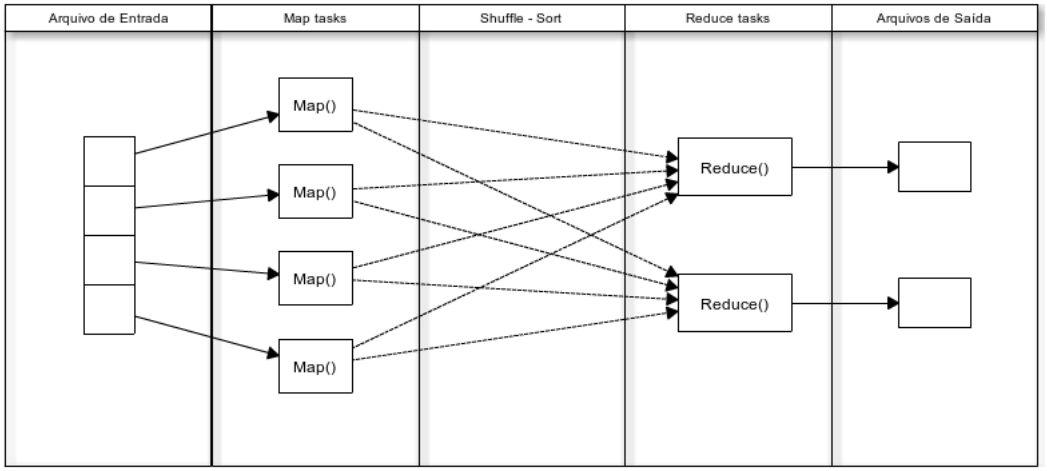

Figura 5 – Fluxo de um programa MapReduce

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 palavras 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 código 2.

```
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));
```

Código 2 – Pseudo código MapReduce, retirado de Dean e Ghemawat (2008)

No código 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.

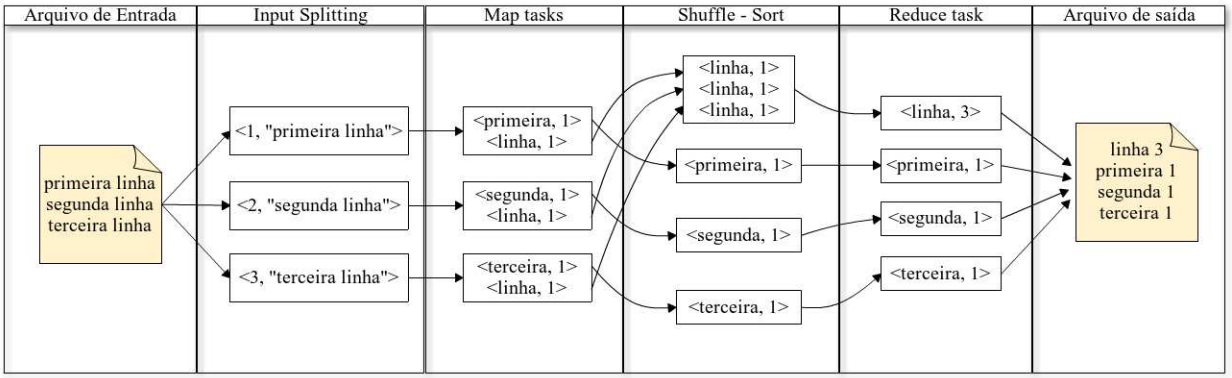

Figura 6 – Fluxo de atividades para o contador de palavras

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 o 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*.

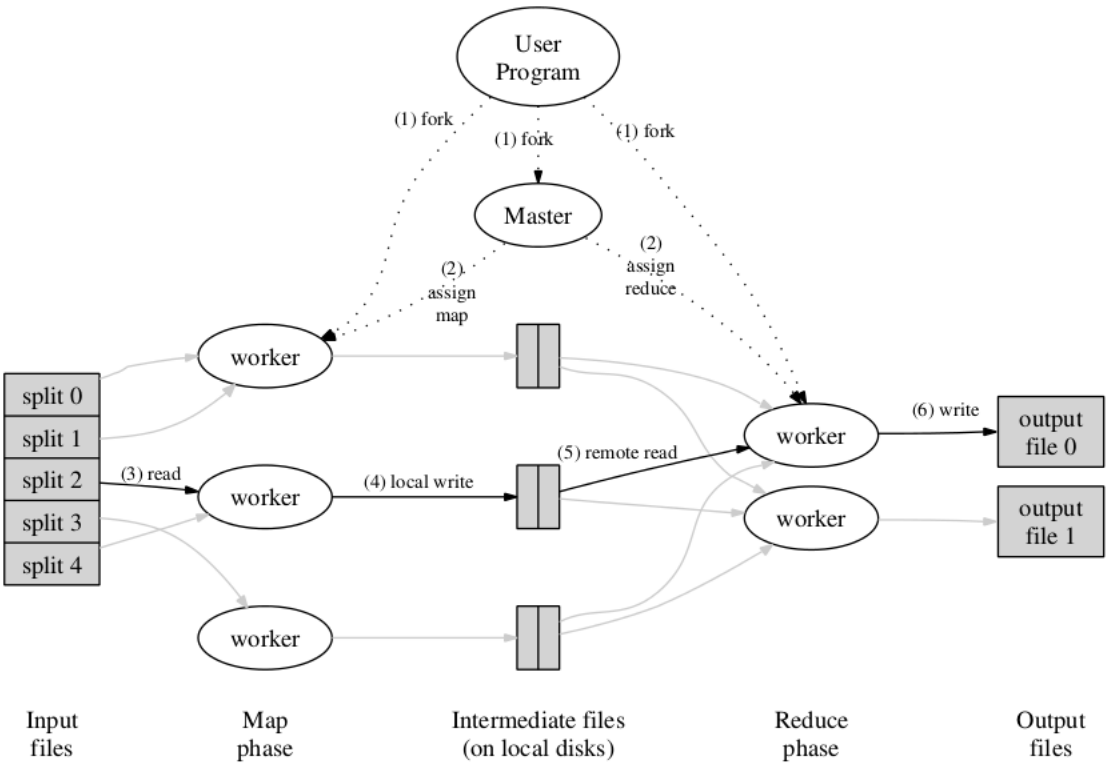

Figura 7 – Arquitetura MapReduce, retirado de Dean e Ghemawat (2008)

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.

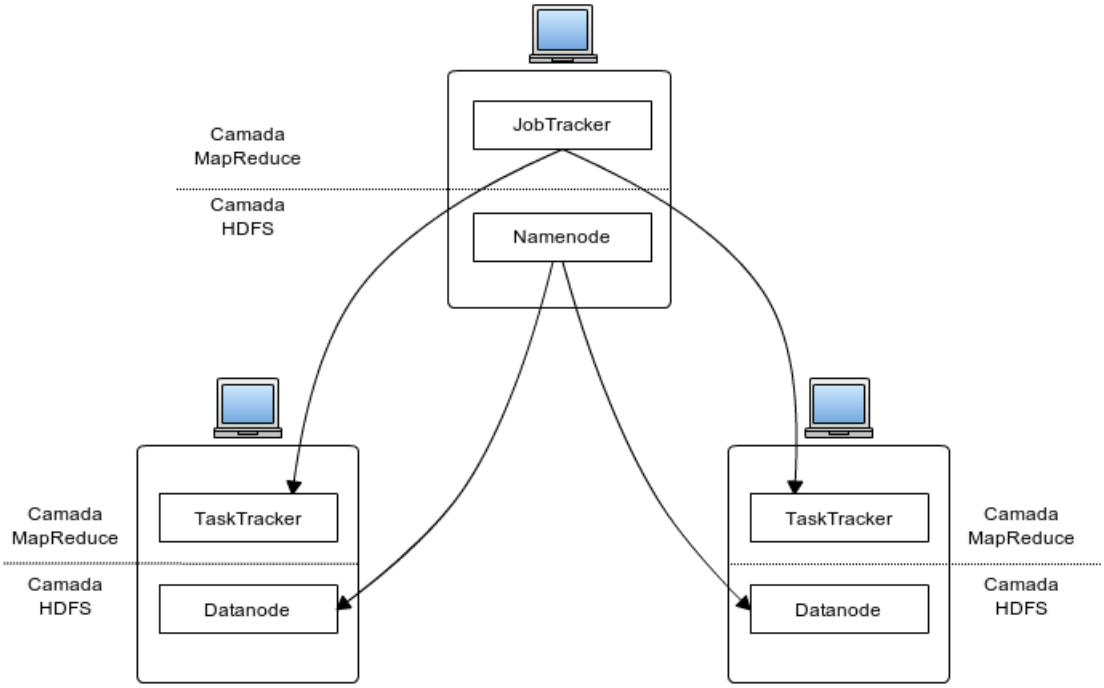

Figura 8 – Arquitetura de um cluster Hadoop

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*.

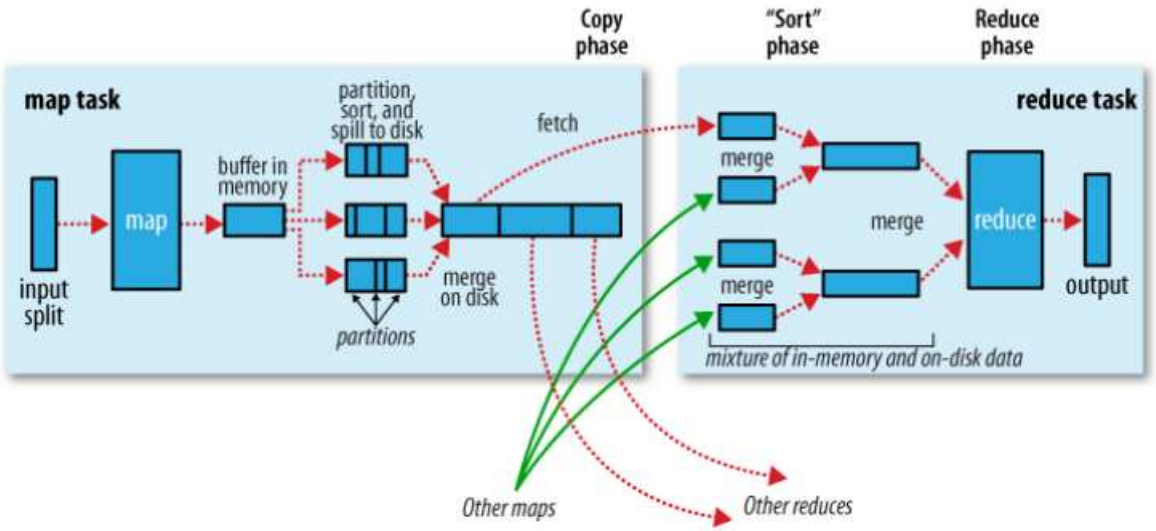

Figura 9 – Fluxograma das etapas Shuffle e Sort, extraído de White (2012)

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. Desta forma as entradas para as funções de redução estão prontas para serem computadas.

3 Desenvolvimento MapReduce

Neste capítulo discutimos sobre a construção de aplicações baseadas no modelo MapReduce. Anteriormente foi apresentada uma introdução a este novo paradigma, porém o foco desta sessão 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 apêncice 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*, como por exemplo, Python e Ruby, usando o utilitário *Hadoop Stream*. Para a implementação em C++ é utilizada 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 sessão 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 código 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>

```
HashMap<String, Integer> wordsMap;
InputStream isFile;
BufferedReader reader;
String line, token;
int tempCount;

isFile = new FileInputStream(PATH + FILE);
reader = new BufferedReader(new InputStreamReader(isFile));
wordsMap = new HashMap<String, Integer>();

while( (line=reader.readLine())!=null ){
  StringTokenizer tokens = new StringTokenizer(line);

  while(tokens.hasMoreTokens()){
    token = tokens.nextToken();

    if( !wordsMap.containsKey(token) ){
      wordsMap.put(token, 1);
    }else{
      tempCount = wordsMap.get(token);
      wordsMap.put(token, tempCount+1);
    }
  }
}

for(Entry<String, Integer> entry : wordsMap.entrySet()){
  System.out.println(entry.getKey() + " " + entry.getValue());
}

reader.close();
isFile.close();
```

Código 3 – Algoritmo convencional para contador de palavras

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, extraída de Venner (2009)

No código 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 sessão.

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, editado de Venner (2009)

3.1 Mapper

No código 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 código 4 é apresentada uma função *map* para o problema proposto.

```
public static class TokenizerMapper extends Mapper<LongWritable, Text,
    Text, IntWritable>{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context) throws
   IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Código 4 – Classe Mapper

Os parâmetros presentes na assinatura da classe representam os tipos *{chave, valor}* de entrada e saída referentes a função *map*. No código 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, 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*.

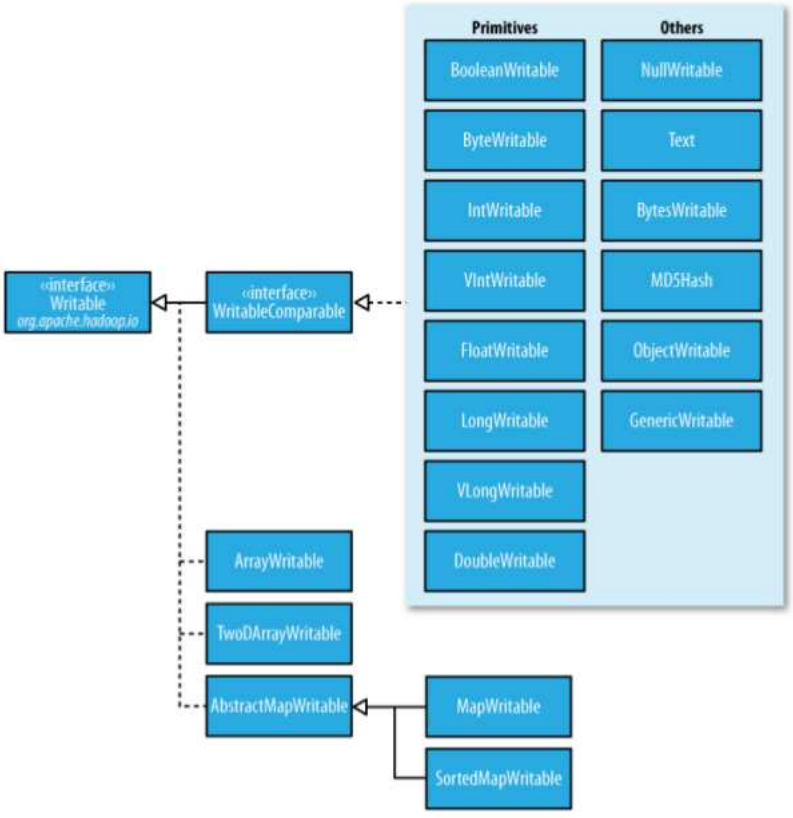

Figura 10 – Hierarquia Writable, extraído de White (2012)

A função *map* definida no código 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 possí-

[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).

vel 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 código 5 apresenta a implementação da classe *Reducer*.

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,
    IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values, Context
   context) throws IOException,
   InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

Código 5 – Classe Reducer

Após a leitura do capítulo 2 fica claro que o MapReduce procura otimizar a quantidade de dados que são transmitidos pelo 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 código 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 código 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*().

```
public class WordCount {
  public static void main(String[] args) throws Exception {

    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).
   getRemainingArgs();

    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }

    Job job = Job.getInstance(conf);
    job.setJobName("word count");

    job.setJarByClass(WordCount.class);

    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Código 6 – Classe para execução do programa MapReduce

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 software 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.
Avro[1]	Sistema para serialização de dados.
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[2]	Linguagem de procedimentos de alto nível para grandes bases de dados. Executada em clusters HDFS e MapReduce.
Hive[3]	Um data warehouse distribuído. Gerencia arquivos no HDFS e provê linguagem de consulta baseada em SQL.
HBase[4]	Banco de dados distribuído orientado a colunas. Utiliza o HDFS para armazenamento dos dados.
ZooKeeper[5]	Coordenador de serviços distribuídos.
Sqoop[6]	Ferramenta para mover dados entre banco relacionais e o HDFS.

Tabela 4 – Ecossistema Hadoop, retirado de White (2012), Shvachko et al. (2010)

Apesar do Hadoop apresentar uma boa alternativa para processamento em larga escala, ainda existem algumas limitações em seu uso. Como discutido anteriormente o

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

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 forma utilizam-se do Hadoop para prover serviços com um nível de abstração maior para o usuário. Neste capítulo discutimos sobre a ferramenta de data warehouse distribuído Hive, e também sobre o banco de dados orientado a colunas HBase.

4.1 Hive

Antes de iniciar a discussão proposta por esta sessão, 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 resolver 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, como por exemplo, contagem de linhas e cálculos de médias aritméticas. Esta situação prejudicava a produtividade da equipe, pois nem todos eram familiarizadas 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, 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 código 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>>);
```

Código 7 – Comando HiveQL

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

4.1.2 Arquitetura

Na sessão 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. Desta forma 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 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 código 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'
```

Código 8 – Uso da cláusula STORED AS, extraído de 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. (2009), as consultas também podem ser submetidas através do uso de um serviço web

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

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.

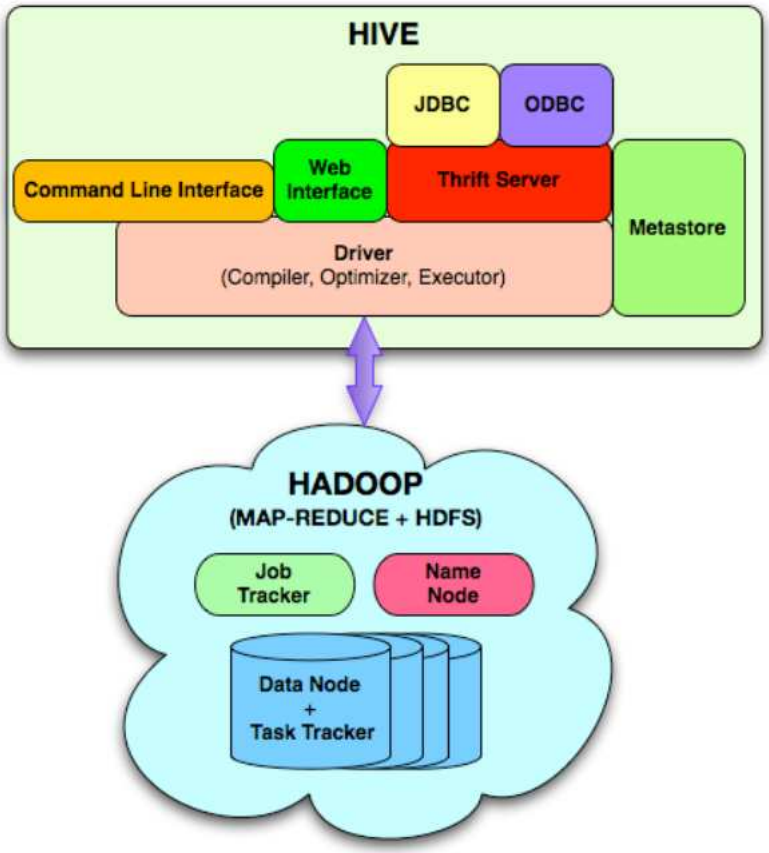

Figura 11 – Arquitetura Hive, retirado de Thusoo et al. (2009)

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 inúmeros exemplos. Este tipo de arquitetura oferecida pelos SGBDs foram construídas de

9

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 sessão 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 terminologia padrão. A pesquisa realizada por Cattell (2011) classifica estas novas tecnologias de acordo

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 sessão 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

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 escala (WHITE, 2012). O propósito desta ferramenta apresenta diferenças se comparado

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/>

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.

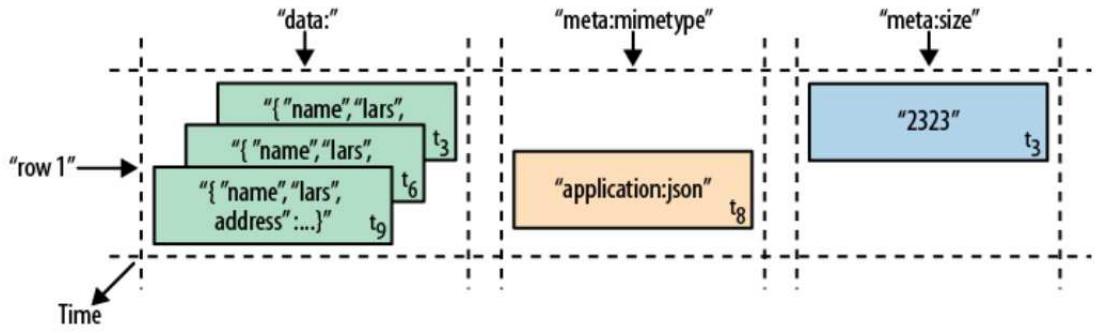

Figura 12 – Linha armazenada no HBase, extraído de George (2011)

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, extraído de 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.

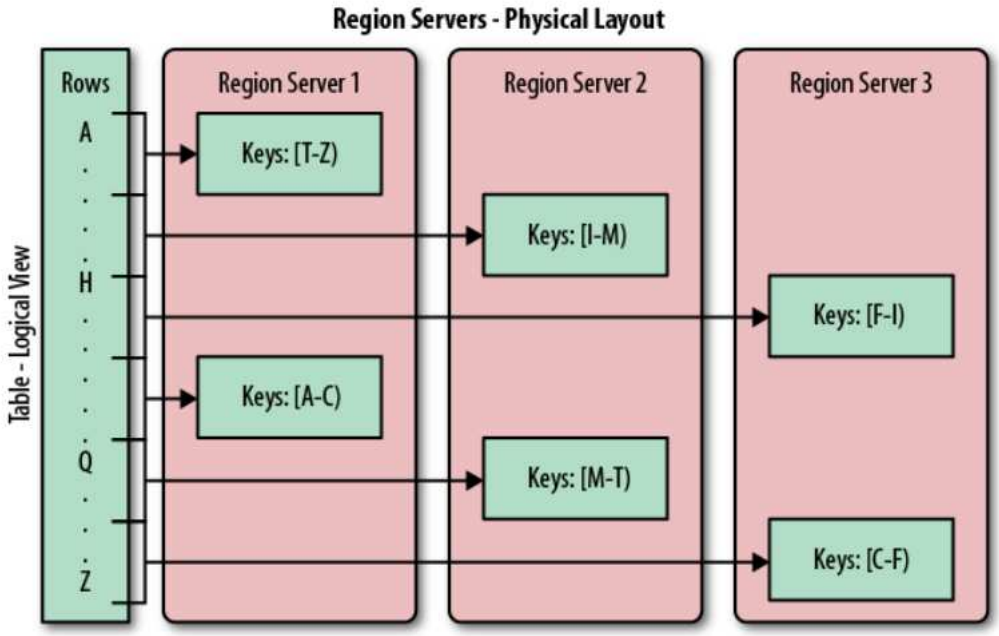

Figura 14 – Regiões HBase, extraído de George (2011)

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. O próximo passo deste trabalho é aplicar os conceitos vistos até o momento em um estudo de caso real. Portanto o objetivo deste capítulo é propor um modelo de arquitetura que deverá ser projetado e construído na sequência desta pesquisa.

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 do governo federal. 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 neste processo. Este meio de comunicação deixou de ser utilizado apenas para lazer e atualmente é usado pelos cidadãos para expressar opiniões dos 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 revelar os principais motivos que geram esse descontentamento. Processar estas informações não é uma tarefa trivial, pois os dados gerados por este meio podem estar na ordem de terabytes.

[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>

5.3 Arquitetura

Nesta sessão será apresentado um modelo de aplicação 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 para este tipo de problema. 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 voltados para políticas públicas. Resumidamente o sistema deverá ser capaz de executar as seguintes atividades:

- Extrair mensagens relacionadas a políticas públicas postadas por usuários de redes sociais.
- Prover escalabilidade linear para a quantidade de registros que são armazenados.
- Analisar as publicações e classificá-las de acordo com o conteúdo.
- Exibir para o usuário final uma interface web para visualização dos resultados obtidos após a análise.

A figura 15 apresenta o modelo de arquitetura que será adotado pela solução proposta neste trabalho.

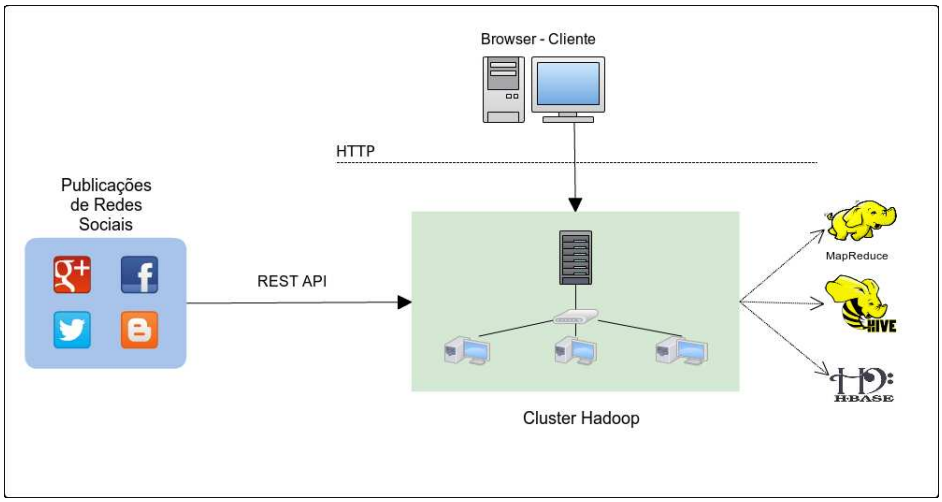

Figura 15 – Arquitetura da solução

A extração de publicações das redes sociais deverá ocorrer através das APIs disponibilizadas pelas redes sociais Facebook[3], Twitter[4] e YouTube[5]. Todas estas ferramentas

[3] <https://developers.facebook.com/docs/graph-api/quickstart/v2.0>
[4] <https://dev.twitter.com/docs/streaming-apis/streams/public>
[5] <https://developers.google.com/youtube/2.0/developers_guide_protocol_comments>

possibilitam capturar postagens públicas através de serviços REST[6]. Estas informações devem ser armazenadas e analisadas em um *cluster* Hadoop. Os resultados obtidos pela análise serão disponibilizados através de um serviço web.

O processo de análise das mensagens que estarão armazenadas no *cluster* ainda está sendo definido. Até o momento, o único método pesquisado procura identificar os sentimentos que um determinado usuário expressa em suas mensagens de redes sociais, este modelo é definido como análise de sentimentos. Araújo, Gonçalves e Benevenuto (2013) destacou abordagens para análise de sentimentos em postagens retiradas de redes sociais. Algumas delas são apresentadas a seguir:

- **Emoticons[7]:** Análise baseada na presença de *emoticons* na corpo da sentença. Uma das abordagens mais ineficientes, segundo Araújo, Gonçalves e Benevenuto (2013).
- **LIWC(Linguistic Inquiry and Word Count):** Ferramenta comercial que utiliza um dicionário de palavras separadas em categorias para estimar os componentes emocionais, cognitivos e estruturais de um texto.
- **SentiStrength:** Utiliza métodos baseados em aprendizado de máquinas, na qual compara métodos de classificação supervisionadas com não supervisionadas.
- **SentiWordNet:** Este método calcula três índices (positivo, negativo, neutro) que indicam o grau – variando de 0 a 1 – de cada um destes sentimentos presentes no texto. Utiliza um dicionário léxico WordNet[8] e aprendizagem de máquina semi-supervisionada.
- **SenticNet:** Método para mineração de opinião e análise de sentimentos que explora técnicas de Inteligência Artificial e Web Semântica.
- **SASA:** Ferramenta *open-source* que se baseia em aprendizado de máquina.

[6] <http://pt.wikipedia.org/wiki/REST>
[7] Sequencia de caracteres tipográficos que expressam emoções.
[8] <http://wordnet.princeton.edu/>

6 Considerações Finais

Neste trabalho foi apresentado 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 por exemplo, Hive, o conhecimento deste modelo torna-se indispensável, pois grande parte dos projetos construídos no topo do Hadoop utilizam internamente o MapReduce.

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 em realizar 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 por exemplo, 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.

6.1 Trabalhos Futuros

Os próximos passos deste trabalho consistem na especificação e implementação da arquitetura proposta no capítulo 5. Para tal foram definidas as seguintes macro atividades que devem ser realizadas durante o TCC2 e o cronograma a ser seguido.

1. Complementar pesquisas sobre APIs para redes sociais.
2. Definição e implementação do método de extração de dados das redes sociais.
3. Continuar pesquisa de algoritmos para análise de publicações de redes sociais.
4. Instalação e configuração do cluster Hadoop.
5. Definir ferramentas e implementar a análise dos dados obtidos pela fase de extração.

6. Desenvolver aplicação web para apresentar aos usuários finais os resultados obtidos.

7. Documentar pesquisa.

Atividades	Jul 2014	Ago 2014	Set 2014	Out 2014	Nov 2014	Dez 2014
1						
2						
3						
4						
5						
6						
7						

Tabela 7 – Cronograma

Referências

ARAúJO, M.; GONçALVES, P.; BENEVENUTO, F. Métodos para análise de sentimentos no twitter. 2013. Citado na página 59.

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 na página 51.

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 53.

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 10 vezes nas páginas 9, 13, 23, 29, 30, 31, 32, 33, 34 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.

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 5 vezes nas páginas 9, 51, 53, 54 e 55.

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 4 vezes nas páginas 9, 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 3 vezes nas páginas 9, 27 e 28.

INMON, W. *Building the Data Warehouse*. [S.l.]: Wiley, 2005. (Timely, practical, reliable). ISBN 9780764599446. Citado na página 46.

KIMBALL, R.; ROSS, M. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. [S.l.]: Wiley, 2013. ISBN 9781118732281. Citado na página 46.

MapReduce Tutorial. 2013. Disponível em: <http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html>. Citado na página 34.

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 7 vezes nas páginas 9, 11, 24, 25, 28, 29 e 45.

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 6 vezes nas páginas 9, 13, 46, 47, 49 e 50.

VENNER, J. *Pro Hadoop*. 1st. ed. Berkely, CA, USA: Apress, 2009. ISBN 1430219424, 9781430219422. Citado 6 vezes nas páginas 11, 24, 35, 39, 40 e 42.

WELCOME to Apache™ Hadoop®! 2014. Disponível em: <http://hadoop.apache.org/>. Citado na página 65.

WHITE, T. *Hadoop: The Definitive Guide*. Third edition. Beijing: O'Reilly, 2012. ISBN 9781449311520. Citado 27 vezes nas páginas 9, 11, 19, 20, 23, 24, 25, 26, 27, 28, 29, 34, 35, 37, 38, 39, 40, 41, 43, 44, 45, 47, 48, 52, 53, 54 e 65.

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 (Java Virtual Machine), 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 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>
```

Código 9 – Arquivo core-site.xml

```
<?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>
```

Código 10 – 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>
```

Código 11 – 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>
```

Código 12 – 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>
```

Código 13 – 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 MapReduces 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.