



Processamento de dados utilizando o Ecossistema Hadoop

João Paulo Barbosa Nascimento

2021

Processamento de dados utilizando o Ecossistema Hadoop

João Paulo Barbosa Nascimento

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

| | |
|---|----|
| Capítulo 1. O Modelo MapReduce | 5 |
| História e Motivação | 5 |
| Funcionamento | 7 |
| Capítulo 2. Conhecendo o Hadoop | 11 |
| História..... | 11 |
| Ecossistema..... | 12 |
| YARN | 13 |
| Componentes do YARN..... | 15 |
| Instalação..... | 17 |
| Mecanismos e Arquivos de Configuração..... | 25 |
| Funções Map, Reduce e Combine..... | 28 |
| Tolerância a Falhas | 28 |
| Formatos de Entrada e Saída (texto, binário, etc.) | 30 |
| Capítulo 3. O HDFS (Hadoop Distributed File System) | 31 |
| O formato do HDFS: Conceitos e comandos básicos | 31 |
| Namenodes e Datanodes | 32 |
| Operações básicas | 33 |
| Interface gráfica para gerenciamento do HDFS | 35 |
| Capítulo 4. Criando programas Apache Hadoop | 39 |
| Estrutura de um programa Hadoop e a classe Jobconf | 39 |
| A classe MapReduceBase | 42 |
| Criando consultando e excluindo diretórios no HDFS (via aplicação)..... | 44 |
| Lendo e gravando dados do HDFS (via aplicação)..... | 46 |
| Um “Olá mundo!” utilizando o Hadoop..... | 47 |
| Métodos <i>Configure</i> e <i>Close</i> | 51 |

| | |
|---|----|
| Definindo uma função <i>Combine</i> | 53 |
| Depurando um <i>job</i> e os <i>logs</i> do Hadoop..... | 54 |
| Algoritmos interativos com Hadoop | 57 |
| Capítulo 5. Parâmetros de Desempenho..... | 60 |
| Principais parâmetros de desempenho..... | 60 |
| Influência dos parâmetros de desempenho | 62 |
| Medidas de desempenho em <i>cluster</i> (escalabilidade e eficiência) | 62 |
| Capítulo 6. Um pouco mais do ecossistema Hadoop | 65 |
| Hive..... | 65 |
| HBase | 66 |
| Sqoop | 69 |
| Referências..... | 71 |

Capítulo 1. O Modelo MapReduce

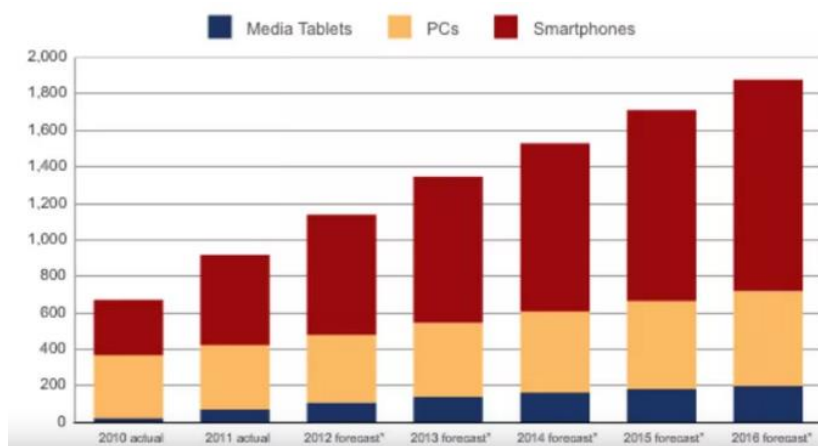
Esse capítulo tem o objetivo de apresentar a história e motivação para a criação do modelo de programação MapReduce, assim como apresentar um exemplo de funcionamento da ferramenta.

História e Motivação

Vivemos atualmente na era dos dados e a tarefa de mensurar a quantidade total de dados armazenados eletronicamente não é fácil. Toda essa imensa quantidade de dados provém de diversas fontes, tais como redes sociais (Facebook, Twitter, etc), experimentos científicos (LHC, Projeto Genoma, etc), mercado de ações (Bovespa, Nasdaq, etc), blogs, notícias, fotos, vídeos dentre muitas outras. A facilidade da computação móvel, trazida pelos Smartphones e Tablets, também contribui de forma determinante para esse aumento dos dados.

Aliado a tudo isso ainda temos, ao longo dos anos, um aumento massivo da capacidade de armazenamento em *Hard Disks* e também um aumento relevante na velocidade de acesso aos dados armazenados nesses dispositivos. Em 1990 podíamos armazenar 1.370MB de dados e ter uma taxa de transferência de 4.4MB/s, podendo assim ler os dados de todo o armazenamento em aproximadamente 5 minutos. Quase 30 anos depois podemos armazenar na média 1TB de dados e temos uma taxa de transferência de 100MB/s, levando mais que duas horas e meia para ler todos os dados armazenados no disco. A redução do custo de armazenamento e a computação em nuvem indiscutivelmente contribuíram também para o que chamamos de fenômeno do Big Data.

Figura 1 – Dispositivos conectados à Internet (em Milhões de Unidades).



Fonte: IDC's Digital Universe Study.

Diante do que foi apresentado até aqui (grande quantidade de dados – Big Data) é necessária uma infraestrutura com capacidade de gerenciar e manipular tanto o armazenamento quanto a recuperação desses dados, de forma eficiente e confiável, tolerante a falhas, adaptável aos problemas rotineiros e com alta disponibilidade. A partir dessa necessidade surgiu o modelo MapReduce e posteriormente a ferramenta Hadoop.

Em 2003 o Google propôs uma ferramenta altamente escalável, tolerante a falhas e com um sistema de arquivos distribuído (DFS) chamado Google File System (GFS). Além disso, essa ferramenta introduziu o modelo de programação MapReduce. O modelo de programação MapReduce tem o objetivo de proporcionar alto desempenho de processamento ao distribuir tarefas para serem processadas em nós de um *cluster* onde os dados da tarefa são armazenados e processados paralelamente. Em 2004 o Google publicou um artigo apresentando o modelo MapReduce ao mundo.

Em 2008 o Hadoop passou a ser um dos principais projetos da Apache, passando a ser utilizado por várias grandes empresas, tais como Yahoo, Facebook e New York Times. Ainda em 2008 o Hadoop quebrou o recorde mundial ao tornar-se o mais rápido sistema a ordenar 1TB de dados, executando em um cluster com 910 nós, realizando a operação em 209 segundos, batendo o recorde anterior que era de 297 segundos. Nesse mesmo ano o Google anunciou que a sua implementação do

MapReduce ordenou a mesma quantidade de dados em 68 segundos. Posteriormente em 2009, o Yahoo anunciou que conseguiu quebrar o recorde, reduzindo o tempo para 62 segundos.

Funcionamento

Para facilitar o entendimento do modelo de programação paralela MapReduce, o seu funcionamento será exemplificado utilizando uma rede que monitora o nível de CO₂ (dióxido de carbono) na atmosfera em diversos pontos do globo terrestre. É esperado que, ao longo do tempo, os dados gerados por essa rede cresçam e as pesquisas fiquem inviáveis do ponto de vista de processamento sequencial. Neste contexto, o modelo MapReduce torna-se um importante aliado para auxiliar nesse processamento. Sensores espalhados por pontos estratégicos no mundo fazem a coleta dos dados e os armazenam em um arquivo em formato texto. Esse problema apresenta-se como um bom candidato para ser resolvido pelo modelo MapReduce/Hadoop, pois os dados estão estruturados em posições pré-definidas dentro do arquivo texto e estes dados são orientados a registro (cada medição consiste em uma linha do arquivo). Iremos usar, apenas a título de exemplo, alguns dados gerados e coletados aleatoriamente pelo site de medição de CO₂, denominado CO₂ Now (<http://www.co2now.org>). Os dados são armazenados usando o formato ASCII, e cada linha corresponde a um registro. Para simplificar o exemplo, focaremos nos elementos básicos do conjunto de dados, a saber, data da medição e nível de CO₂. Cada linha a seguir está composta dos seguintes dados: posição 1 até posição 12, data e hora da medição no seguinte formato: ano (yyyy), mês (mm), dia (dd), hora (hh) e minutos (mm). Da posição 13 até a posição 17 é o nível de CO₂ coletado, representado com duas casas decimais. Os espaços em branco foram inseridos para facilitar a visualização dos dados e sua posição não deve ser contada.

1988 04121402 35707

1988 04251405 35727

1988 05031922 35823

1988 05091357 35719

1987 02231400 35412
1987 03021349 35267
1987 03091400 35350
1987 03161359 35418
1986 01201523 35129
1986 01271518 35520
1986 02031455 35328
1986 02101622 35257
1985 05061943 35234
1985 06241908 35056
1985 07011900 34967
1985 08261528 33733

Suponha que a partir do conjunto de dados apresentado anteriormente, queremos encontrar o maior nível de CO2 captado por ano pelos sensores e disponível no arquivo texto. Os dados a serem analisados estão destacados em ano e nível de CO2 apurado. Primeiramente a função Map recebe as linhas do arquivo e extrai o ano e o nível de CO2 (destacado) e emite os seguintes dados de saída:

(1988, 357.07)
(1988, 357.27)
(1988, 358.23)
(1988, 357.19)
(1987, 354.12)
(1987, 352.67)
(1987, 353.50)
(1987, 354.18)
(1986, 351.29)
(1986, 355.20)
(1986, 353.28)
(1986, 352.57)
(1985, 352.34)
(1985, 350.56)

(1985, 349.67)

(1985, 337.33)

Em seguida, os dados de saída da função Map são processados pelo framework do MapReduce antes de serem enviados para a função Reduce. Esse processamento irá ordenar e agrupar os pares de chave/valor pela chave e ao final a função Reduce receberá o seguinte conjunto de dados:

(1985, [337.33, 349.67, 350.56, 352.34])

(1986, [351.29, 352.57, 353.28, 355.20])

(1987, [352.67, 353.50, 354.12, 354.18])

(1988, [357.07, 357.19, 357.27, 358.23])

Cada ano pode ser processado paralelamente por uma função Reduce e ao final será produzida uma saída composta por ano e seus respectivos níveis de CO₂. Cada função Reduce percorrerá paralelamente todo o vetor de medições e encontrará o maior nível para cada ano. A saída do processamento será a seguinte:

(1985, 352.34)

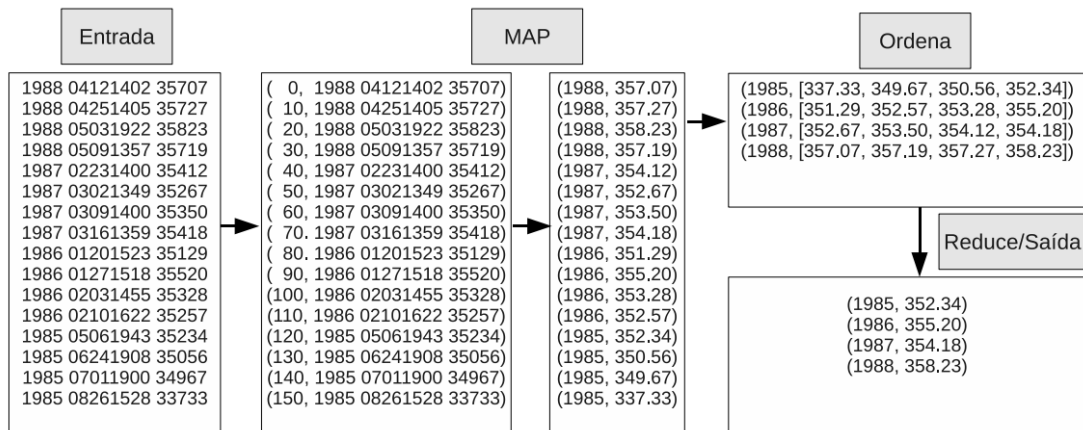
(1986, 355.20)

(1987, 354.18)

(1988, 358.23)

Este é um pequeno exemplo prático da maneira como o modelo paraleliza grandes massas de dados. A Figura 2 ilustra o fluxo de dados do modelo MapReduce, a partir do exemplo explicado acima. Os dados da primeira coluna da função Map (10, 20, 30, ..., 150) são valores de chave gerados automaticamente pelo modelo.

Figura 2 – Exemplo do fluxo de dados no MapReduce/Hadoop.



Fonte: Nascimento (2011).

Capítulo 2. Conhecendo o Hadoop

Esse capítulo tem o objetivo de apresentar a história por trás da criação do framework Hadoop, assim como detalhar alguns componentes que formam o seu ecossistema, tais como YARN, Spark e HBase. Nesse capítulo será abordado também a instalação do Hadoop, os mecanismos de configuração, as funções Map, Reduce e Combine, assim como formatos de entrada e saída de dados.

História

O Apache Hadoop foi criado por Doug Cutting, criador do Apache Lucene, uma biblioteca de busca textual amplamente utilizada.

O Hadoop teve sua origem no Apache Nutch que é um motor de busca *open-source* da Web. O Nutch faz parte do Lucene e surgiu em 2002 como um sistema de crawler e busca na Web. Após a sua criação, percebeu-se que a arquitetura do Nutch não era escalável para ser aplicado na busca em bilhões de páginas Web e a solução para esse caso estava em um artigo publicado em 2003 pelo Google, que descrevia a arquitetura do Google Distributed FileSystem (GFS).

O GFS resolveria a necessidade de armazenamento de arquivos muito grandes, gerado pela coleta e indexação da Web. Em 2004, foi iniciada a implementação *open-source* do GFS e, a partir daí, surgiu o NDFS – Nutch Distributed FileSystem. Ainda em 2004, o Google apresentou o modelo MapReduce para o mundo e, logo em seguida, em 2005, os desenvolvedores do Nutch implementaram o modelo MapReduce na ferramenta.

Em meados de 2005, todos os principais algoritmos do Nutch foram adaptados para utilizar o modelo MapReduce e o sistema de arquivos distribuído NDFS. Em 2006, surgiu o projeto independente Hadoop.

Ecosystem

O Hadoop é um arcabouço (*framework*) que permite o processamento distribuído e paralelo de grandes massas de dados utilizando *clusters* de computadores e proporcionando ao desenvolvedor um modelo de programação simplificado. O Hadoop foi projetado para proporcionar escalabilidade ao utilizá-lo em uma simples máquina até em centenas delas.

O projeto Hadoop inclui alguns módulos, sendo eles:

- **Hadoop Common:** nesse módulo estão os utilitários comuns que suportam todos os outros módulos do Hadoop.
- **Hadoop Distributed File System (HDFS):** módulo que representa o sistema de arquivos distribuído do Hadoop.
- **Hadoop Yarn:** um framework responsável pelo agendamento das tarefas e pelo gerenciamento dos recursos do *cluster*.
- **Hadoop MapReduce:** um sistema baseado no Yarn para processamento paralelo de grandes quantidades de dados.

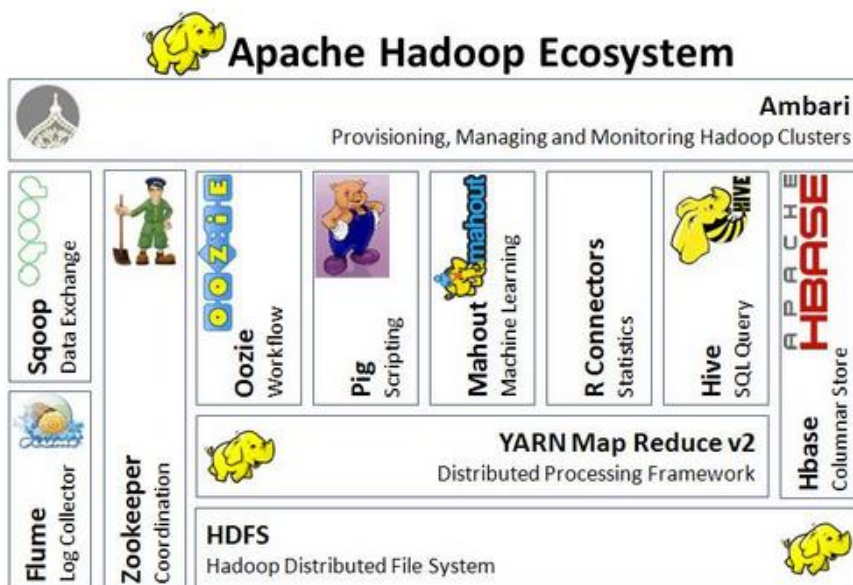
Além dos quatro módulos citados acima o Hadoop possui mais alguns projetos em seu arcabouço, sendo que os principais são:

- **Ambari:** uma ferramenta para monitoramento do cluster Hadoop.
- **HBase:** um banco de dados escalável e distribuído que suporta o armazenamento de dados estruturados em grandes tabelas.
- **Hive:** Uma estrutura de data-warehouse.
- **Mahout:** uma biblioteca que fornece algoritmos de aprendizado de máquina e mineração de dados.
- **Spark:** uma estrutura Hadoop que prioriza o processamento em memória. Está disponível para processamento com algoritmos de machine learning, grafos, ETL e *streaming*.

- **Zookeeper:** um coordenador de serviço altamente distribuído.

A Figura 3 apresenta os componentes listados acima e outros que fazem parte do ecossistema Hadoop.

Figura 3 – Componentes Hadoop 1.0 e 2.0.



Fonte: hadoop.org.

Alguns desses componentes apresentados podem trabalhar em conjunto com outros ou completamente separados, dependendo da real necessidade do usuário.

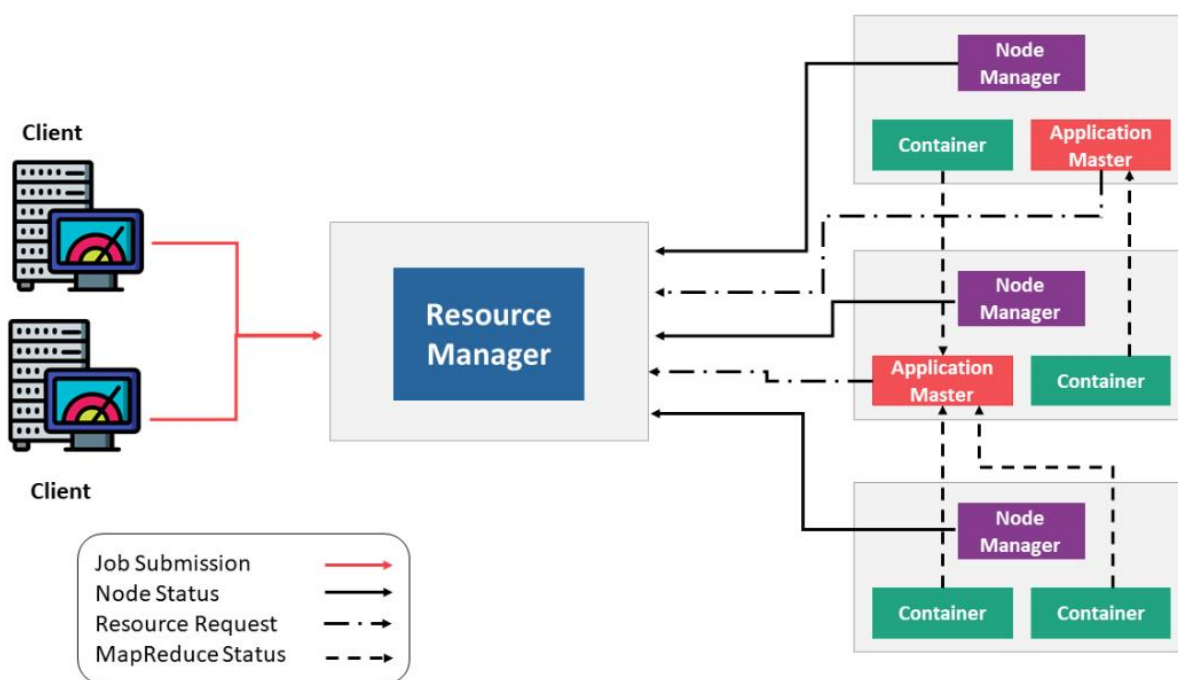
YARN

O Apache YARN (*Yet Another Resource Negotiator*) é um gerenciador de recursos que foi incorporado ao ambiente a partir da Versão 2 do Hadoop com o objetivo de melhorar a implementação do MapReduce. O Yarn funciona como uma espécie de sistema operacional, realizando o gerenciamento dos recursos do Hadoop e do *cluster*, além de gerenciar o escalonamento de tarefas.

O YARN possui um componente global para gerenciar os recursos (Resource Manager- RM) e outro para cada programa executado no ambiente (Application Master - AM). Cada nó do *cluster* possui um gerenciador próprio (Node Manager -

NM), estes são responsáveis pela execução dos contêineres de seu nó, monitorando os recursos utilizados como CPU, Memória, Disco e Rede, e reportando as informações para o Resource Manager. Na Figura 4 é apresentado a execução de duas aplicações pelo YARN, cada uma com seu Application Master e os demais componentes citados acima.

Figura 4 – Arquitetura do YARN



Até a Versão 1 do Hadoop, toda a responsabilidade pelo processamento e gerenciamento de recursos era atribuída ao modelo MapReduce. Esse desenho resultou em um gargalo no processamento. Para superar esse problema, o YARN foi introduzido na Versão 2 do Hadoop, com o objetivo de aliviar o trabalho do modelo MapReduce. Com isso, o YARN passou a assumir a responsabilidade pelo gerenciamento dos recursos e agendamento das tarefas.

Essa mudança trouxe uma revolução dentro do ecossistema do Hadoop, pois a ferramenta tornou-se mais flexível, eficiente e escalável, desacoplando o

monitoramento dos recursos do modelo MapReduce, podendo assim anexar outros componentes ao ecossistema.

Componentes do YARN

Conforme foi apresentado na Figura 4, a arquitetura do Apache YARN é formada por quatro componentes principais, sendo: Resource Manager, Node Manager, Application Master e Container.

O Resource Manager é conhecido por sua autoridade na alocação de recursos do *cluster*. Possui a responsabilidade de otimizar a utilização dos recursos e possui dois componentes principais: o Agendador e o Gerenciador de Aplicativos. O Agendador é o responsável por alocar recursos para vários aplicativos que podem estar em execução, sempre respeitando as restrições de capacidade dos nós e as filas existentes. Além disso, o Agendador não executa nenhum monitoramento ou rastreamento da situação de uma determinada aplicação, ou seja, é um agendador puro. Além disso, o Agendador não reinicia uma tarefa que, porventura, tenha falhado. O segundo componente do Resource Manager é o Gerenciador de Aplicativos. Esse componente é responsável por aceitar ou recusar os trabalhos e realizar a negociação dos recursos durante a sua execução.

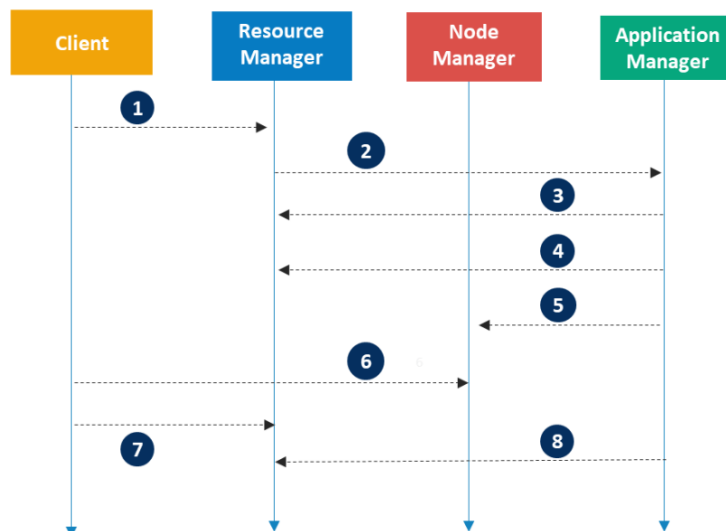
O Node Manager cuida do *cluster* e envia “pulsos” sobre a integridade de um determinado nó. Seu objetivo principal é gerenciar os contêineres dos aplicativos e também realizar o gerenciamento dos *logs* do sistema, monitorando o uso de CPU, memória e disco nos contêineres individuais.

Cada trabalho a ser executado pelo cluster possui um Application Master próprio. Esse componente é o processo que coordena a execução de uma aplicação no *cluster*, gerenciando suas falhas e negociando os recursos necessários (contêineres) com o Resource Manager,

O Container é uma coleção de recursos físicos de um determinado nó. Essa coleção é composta por quantidades de memória RAM, CPU e disco em um único nó.

A Figura 5 apresenta um fluxo de execução de uma aplicação no Apache YARN.

Figura 5 – Fluxo de execução no Apache YARN.



1. O Client submete uma aplicação.
2. O Resource Manager aloca um contêiner para iniciar o Application Manager.
3. O Application Manager registra-se no Resource Manager.
4. O Application Manager solicita contêineres ao Resource Manager.
5. O Application Manager notifica o Node Manager para que os contêineres sejam disponibilizados.
6. O código da aplicação é executado nos contêineres.
7. O Client entra em contato com o Resource Manager para que ele monitore o status da aplicação.
8. O Application Manager encerra seu registro no Resource Manager.

Instalação

A instalação do Hadoop pode ser exemplificada por meio de 10 passos básicos. Esse tutorial foi adaptado do trabalho de Michael Noll disponível em <http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/>.

O primeiro passo diz respeito aos pré-requisitos do ambiente para a instalação da ferramenta Hadoop. Iremos considerar a instalação em um ambiente Linux Ubuntu e deveremos ter previamente instalado uma máquina virtual Java (JVM) de preferência em sua mais recente versão. Para verificar se a JVM está instalada, utilize o comando no *prompt* de comandos do Linux:

```
java -version
```

Ao utilizar esse comando, o sistema operacional irá apresentar a versão da JVM, caso a mesma esteja instalada. Caso isso não aconteça, providencie a instalação da JVM. Além da JVM, iremos precisar do SSH para que o Hadoop possa realizar as suas conexões (mesmo usando o Hadoop localmente). Utilize o comando abaixo para verificar se o SSH está presente na máquina, caso não esteja, providencie a sua instalação.

```
ssh
```

O segundo passo será a criação de um usuário exclusivo para trabalhar com o Hadoop, entretanto esse passo não é obrigatório, pois você pode utilizar o seu usuário de costume. Iremos criar um usuário chamado *hduser* e adicioná-lo ao grupo *hadoop* através do comando abaixo.

```
1 $ sudo addgroup hadoop
2 $ sudo adduser --ingroup hadoop hduser
```

O terceiro passo trata da configuração do SSH. O Hadoop utiliza o SSH para comunicação entre os nós do cluster e para acesso à própria máquina local (*localhost*). Iremos configurar o acesso para o usuário *hduser* à máquina local sem a necessidade de digitação de senha, pois durante o processamento do Hadoop

diversas conexões são feitas e não seria uma boa ideia termos que digitar senha a todo momento, pois isso pararia o processamento. Para esse passo iremos considerar que o SSH está instalado e executando na máquina.

Conecte ao usuário *hduser* por meio do Comando 1, destacado em vermelho na imagem abaixo. Em seguida, gere uma chave com senha vazia para esse usuário através do Comando 2, destacado em verde na imagem abaixo.

```
1 user@ubuntu:~$ su - hduser
2 hduser@ubuntu:~$ ssh-keygen -t rsa -P ""
3 Generating public/private rsa key pair.
4 Enter file in which to save the key (/home/hduser/.ssh/id_rsa):
5 Created directory '/home/hduser/.ssh'.
6 Your identification has been saved in /home/hduser/.ssh/id_rsa.
7 Your public key has been saved in /home/hduser/.ssh/id_rsa.pub.
8 The key fingerprint is:
9 9b:82:ea:58:b4:e0:35:d7:ff:19:66:a6:ef:ae:0e:d2 hduser@ubuntu
10 The key's randomart image is:
11 [...snipp...]
12 hduser@ubuntu:~$
```

Em seguida você deve habilitar o SSH para acessar a máquina local (*localhost*) com o usuário *hduser* sem a necessidade de senha. Para essa tarefa, utilize o comando abaixo:

```
1 hduser@ubuntu:~$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Finalmente podemos testar se o acesso foi configurado corretamente. Para isso, execute o comando abaixo (*ssh localhost*) e, se tudo deu certo, o sistema não poderá solicitar a digitação da senha para acessar o *localhost*.

```
1 hduser@ubuntu:~$ ssh localhost
2 The authenticity of host 'localhost (::1)' can't be established.
3 RSA key fingerprint is d7:87:25:47:ae:02:00:eb:1d:75:4f:bb:44:f9:36:26.
4 Are you sure you want to continue connecting (yes/no)? yes
5 warning: Permanently added 'localhost' (RSA) to the list of known hosts.
6 Linux ubuntu 2.6.32-22-generic #33-Ubuntu SMP wed Apr 28 13:27:30 UTC 2010 i686 GNU/Linux
7 Ubuntu 10.04 LTS
8 [...snipp...]
9 hduser@ubuntu:~$
```

O quarto passo para a instalação e configuração do Hadoop diz respeito ao IPV6. Para realizar essa tarefa, abra o arquivo */etc/sysctl.conf* usando o seu editor de texto preferido e adicione ao final desse arquivo as 3 linhas apresentadas abaixo.

```

/etc/sysctl.conf
1 # disable ipv6
2 net.ipv6.conf.all.disable_ipv6 = 1
3 net.ipv6.conf.default.disable_ipv6 = 1
4 net.ipv6.conf.lo.disable_ipv6 = 1

```

Em seguida pode ser necessário reiniciar a máquina para garantir que as alterações serão recarregadas. Você pode verificar se a tarefa foi realizada com sucesso utilizando o comando abaixo:

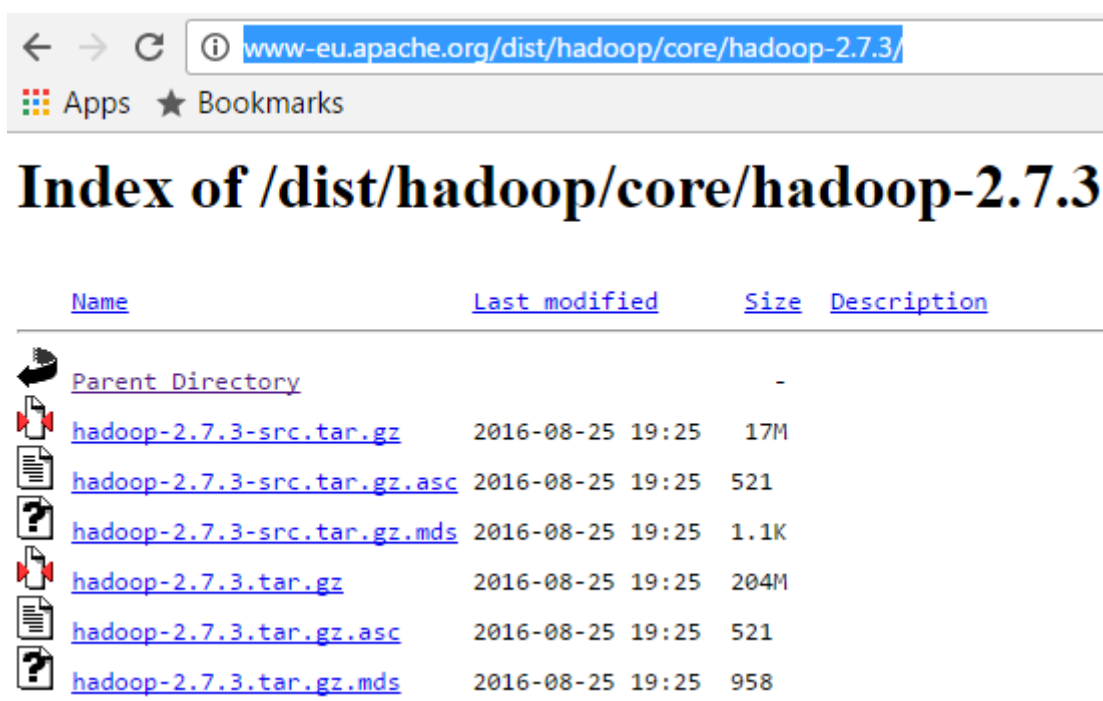
```





1 $ cat /proc/sys/net/ipv6/conf/all/disable_ipv6

```

Se o comando retornar zero, o IPV6 ainda está habilitado e pode ser necessário conferir a operação anterior. Se o comando retornar 1, o IPV6 está desabilitado e é realmente isso que desejamos nesse momento.

No quinto passo podemos efetivamente começar a instalação do Hadoop. Para isso, precisamos inicialmente fazer o download da ferramenta no endereço <http://www-eu.apache.org/dist/hadoop/core/hadoop-2.7.3/>. Deve-se selecionar o arquivo *hadoop-2.7.3.tar.gz*.



| Name | Last modified | Size | Description |
|---|------------------|------|-------------|
|  Parent Directory | | - | |
|  hadoop-2.7.3-src.tar.gz | 2016-08-25 19:25 | 17M | |
|  hadoop-2.7.3-src.tar.gz.asc | 2016-08-25 19:25 | 521 | |
|  hadoop-2.7.3-src.tar.gz.mds | 2016-08-25 19:25 | 1.1K | |
|  hadoop-2.7.3.tar.gz | 2016-08-25 19:25 | 204M | |
|  hadoop-2.7.3.tar.gz.asc | 2016-08-25 19:25 | 521 | |
|  hadoop-2.7.3.tar.gz.mds | 2016-08-25 19:25 | 958 | |

Após o download do arquivo, o mesmo deverá ser descompactado em um diretório de sua preferência. Para o exemplo desse tutorial, iremos considerar o diretório */usr/local*. Abaixo apresentamos 4 comandos. No primeiro estamos entrando no diretório */usr/local*, o segundo está efetivamente descompactando o arquivo *hadoop-2.7.3.tar.gz* dentro da pasta */usr/local*. Ao descompactar o arquivo, é criado o diretório padrão do Hadoop (*hadoop-2.7.3*) e iremos renomeá-lo para *hadoop* utilizando o comando 3. O comando 4 está alterando as permissões desse diretório.

- (1) `cd /usr/local`
- (2) `sudo tar xzf hadoop-2.7.3.tar.gz`
- (3) `sudo mv hadoop-2.7.3 hadoop`
- (4) `sudo chown -R hduser:hadoop hadoop`

O sexto passo trata das variáveis de ambiente que precisam ser configuradas. Para atualizar essas variáveis, iremos editar o arquivo *.bashrc* para o usuário *hduser*. Esse arquivo encontra-se no *\$HOME* do seu sistema operacional. As variáveis de ambiente que precisam ser configuradas são: *HADOOP_HOME* e *JAVA_HOME*. Atribua a variável *JAVA_HOME* o diretório de instalação da sua JVM e para *HADOOP_HOME* atribua o diretório de instalação do Hadoop, que no caso desse tutorial é */usr/local/hadoop*. Em seguida atribua a variável *HADOOP_HOME* ao *PATH* (seta vermelha). A figura abaixo apresenta essas configurações:

```

$HOME/.bashrc
1  # Set Hadoop-related environment variables
2  export HADOOP_HOME=/usr/local/hadoop
3
4  # Set JAVA_HOME (we will also configure JAVA_HOME directly for Hadoop later on)
5  export JAVA_HOME=/usr/lib/jvm/java-6-sun
6
7  # Some convenient aliases and functions for running Hadoop-related commands
8  unalias fs &> /dev/null
9  alias fs="hadoop fs"
10 unalias hls &> /dev/null
11 alias hls="fs -ls"
12
13 # If you have LZOP compression enabled in your Hadoop cluster and
14 # compress job outputs with LZOP (not covered in this tutorial):
15 # Conveniently inspect an LZOP compressed file from the command
16 # line; run via:
17 #
18 # $ lzohead /hdfs/path/to/lzop/compressed/file.lzo
19 #
20 # Requires installed 'lzop' command.
21 #
22 lzohead () {
23     hadoop fs -cat $1 | lzop -dc | head -1000 | less
24 }
25
26 # Add Hadoop bin/ directory to PATH
27 export PATH=$PATH:$HADOOP_HOME/bin

```

O sétimo passo trata da configuração da ferramenta Hadoop. Primeiramente você deve criar um diretório para o armazenamento temporário dos dados. Iremos criar um diretório chamado *tmp* dentro do diretório de instalação do Hadoop (*/usr/local/hadoop*) por meio do comando 1, e os comandos 2 e 3 darão permissões de acesso a esse novo diretório.

- (1) `sudo mkdir -p /usr/local/tmp`
- (2) `sudo chown hduser:hadoop /usr/local/hadoop/tmp`
- (3) `sudo chmod /usr/local/hadoop/tmp`

Em seguida podemos alterar os arquivos de configuração do Hadoop (.xml). O primeiro a ser alterado é o *core-site.xml*. Na versão atual do Hadoop, esse arquivo encontra-se em */usr/local/hadoop/etc/hadoop/core-site.xml*. Abra esse arquivo em seu editor de textos preferido e, entre a tag *<configuration>* *</configuration>* inclua a configuração destacada na figura abaixo. A descrição (*description*) é opcional.


```

conf/core-site.xml
1  <property>
2    <name>hadoop.tmp.dir</name>
3    <value>/app/hadoop/tmp</value>
4    <description>A base for other temporary directories.</description>
5  </property>
6
7  <property>
8    <name>fs.default.name</name>
9    <value>hdfs://localhost:54310</value>
10   <description>The name of the default file system. A URI whose
11     scheme and authority determine the FileSystem implementation. The
12     uri's scheme determines the config property (fs.SCHEME.impl) naming
13     the FileSystem implementation class. The uri's authority is used to
14     determine the host, port, etc. for a filesystem.</description>
15 </property>

```

Faça o mesmo para o arquivo *mapred-site.xml*, conforme a figura abaixo. Mais uma vez, o item descrição (description) é opcional.

```

conf/mapred-site.xml
1  <property>
2    <name>mapred.job.tracker</name>
3    <value>localhost:54311</value>
4    <description>The host and port that the MapReduce job tracker runs
5     at. If "local", then jobs are run in-process as a single map
6     and reduce task.
7    </description>
8  </property>

```

Por último altere o arquivo *hdfs-site.xml* conforme a figura abaixo.

```

conf/hdfs-site.xml
1  <property>
2    <name>dfs.replication</name>
3    <value>1</value>
4    <description>Default block replication.
5     The actual number of replications can be specified when the file is created.
6     The default is used if replication is not specified in create time.
7    </description>
8  </property>

```

Estando os arquivos de configuração devidamente alterados, iremos para o oitavo passo da configuração que diz respeito à formatação do HDFS (Sistema de Arquivos Distribuído do Hadoop). O HDFS será tratado com detalhes no próximo capítulo de nosso curso. Por enquanto iremos apenas formatá-lo. Para isso utilize o comando abaixo:

```
1 hduser@ubuntu:~$ /usr/local/hadoop/bin/hadoop namenode -format
```

Se tudo der certo, a saída deverá ser a seguinte:

```
1 hduser@ubuntu:/usr/local/hadoop$ bin/hadoop namenode -format
2 10/05/08 16:59:56 INFO namenode.NameNode: STARTUP_MSG:
3 /*****
4 STARTUP_MSG: Starting NameNode
5 STARTUP_MSG: host = ubuntu/127.0.1.1
6 STARTUP_MSG: args = [-format]
7 STARTUP_MSG: version = 0.20.2
8 STARTUP_MSG: build = https://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20 -i
9 *****/
10 10/05/08 16:59:56 INFO namenode.FSNamesystem: fsowner=hduser,hadoop
11 10/05/08 16:59:56 INFO namenode.FSNamesystem: supergroup=supergroup
12 10/05/08 16:59:56 INFO namenode.FSNamesystem: isPermissionEnabled=true
13 10/05/08 16:59:56 INFO common.Storage: Image file of size 96 saved in 0 seconds.
14 10/05/08 16:59:57 INFO common.Storage: Storage directory .../hadoop-hduser/dfs/name has been
15 10/05/08 16:59:57 INFO namenode.NameNode: SHUTDOWN_MSG:
16 /*****
17 SHUTDOWN_MSG: Shutting down NameNode at ubuntu/127.0.1.1
18 *****/
19 hduser@ubuntu:/usr/local/hadoop$
```

Estando o HDFS formatado, poderemos finalmente ir para o nono e penúltimo passo, que trata de iniciar os serviços do Hadoop. Para isso utilize o comando abaixo. Esse processo pode levar alguns segundos, ou até minutos, dependendo da máquina utilizada.

/usr/local/hadoop/sbin/start-all.sh

A saída do comando acima deve ser algo parecido com a figura abaixo:

```
1 hduser@ubuntu:/usr/local/hadoop$ bin/start-all.sh
2 starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-namenode-ubuntu.out
3 localhost: starting datanode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-datanode-
4 localhost: starting secondarynamenode, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-
5 starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-jobtracker-ubuntu.
6 localhost: starting tasktracker, logging to /usr/local/hadoop/bin/../logs/hadoop-hduser-tasktr
7 hduser@ubuntu:/usr/local/hadoop$
```

Nesse momento é necessário conferir se todos os serviços do Hadoop foram iniciados corretamente. Para isso, utilize o comando *JPS*. Esse comando permite verificar todos os processos da JVM que estão executando na máquina. Caso você não possua o *JPS*, esse deverá ser instalado. Os processos que nos interessam são: **Datanode**, **ResourceManager**, **Namenode**, **SecondaryNameNode** e **NodeManager**.

Finalmente podemos ir para o décimo e último passo da instalação que é executar um experimento de testes. Iremos utilizar o cálculo do Pi que já vem implementado com o Hadoop e é muito simples de ser utilizado. Para isso, utilize o comando abaixo:

```
bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar pi 16 1000
```

A tabela abaixo apresenta o significado do comando de execução do Pi.

| <u>Parte do Comando</u> | <u>Significado</u> |
|--|--|
| bin/hadoop | O arquivo hadoop que fica dentro do diretório bin é o que usamos para execução dos comandos. |
| jar | Informação de que iremos executar um programa Java. |
| /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar | O arquivo jar chamado <i>hadoop-mapreduce-examples-2.7.3</i> contém os exemplos de implementação já incluídos no Hadoop. Iremos utilizar bastante esse arquivo durante a condução da disciplina. |
| pi | Dentro do arquivo <i>hadoop-mapreduce-examples-2.7.3.jar</i> temos vários programas implementados e ao passarmos o parâmetro pi, estamos informando que queremos executar o programa pi. |
| 16 e 1000 | Parâmetros específicos para execução do Pi (números de casas decimais e precisão do cálculo). |

Ao final da execução do teste do Pi, o seguinte resultado poderá ser obtido:


```

HDFS: Number of write operations=309
Map-Reduce Framework
  Map input records=16
  Map output records=32
  Map output bytes=288
  Map output materialized bytes=448
  Input split bytes=2390
  Combine input records=0
  Combine output records=0
  Reduce input groups=2
  Reduce shuffle bytes=448
  Reduce input records=32
  Reduce output records=0
  Spilled Records=64
  Shuffled Maps =16
  Failed Shuffles=0
  Merged Map outputs=16
  GC time elapsed (ms)=828
  Total committed heap usage (bytes)=2527027200
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=1888
File Output Format Counters
  Bytes Written=97
Job Finished in 9.169 seconds
Estimated value of Pi is 3.142500000000000000000000
hhduser@inovaz hadoop1$

```

Mecanismos e Arquivos de Configuração

Diversas implementações de MapReduce são conhecidas e novas implementações também podem ser feitas, ficando a escolha atrelada ao ambiente de execução e ao problema envolvido. Uma implementação pode ser adequada para uma pequena máquina de memória compartilhada ou para uma grande máquina com multiprocessadores ou ainda uma grande rede de computadores.

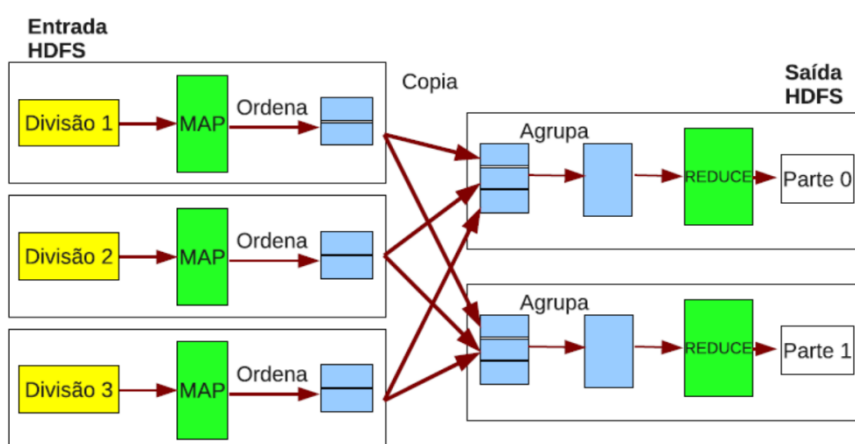
As chamadas à função Map são distribuídas automaticamente em diversas máquinas particionando os dados de entrada em M partições. As partições de entrada podem ser processadas por diferentes máquinas em paralelo. As chamadas à função Reduce são distribuídas particionando-se as chaves intermediárias em R pedaços, usando uma função de particionamento (por exemplo, $hash(key) \bmod R$). O número de partições (R) e a função de particionamento são especificados pelo usuário. Quando o programa inicia a execução de uma tarefa MapReduce, a seguinte sequência de ações ocorre:

1. A biblioteca MapReduce primeiramente divide os arquivos de entrada em M partes, frequentemente de tamanho 64 Megabytes (MB) por parte. Esse valor é controlado pelo usuário através de um parâmetro opcional e o valor default é 64 Megabytes. Esse processo inicia diversas cópias do programa em um cluster de máquinas.
2. Uma das cópias do programa é especial - a master, que divide o trabalho em M tarefas Map e R tarefas Reduce. O resto das cópias são workers que recebem trabalhos atribuídos pelo master. O master escolhe workers inativos e atribui a cada um deles uma tarefa Map ou uma tarefa Reduce.
3. O worker, a quem o master atribuiu uma tarefa Map, lê o conteúdo da partição de entrada, analisa o par chave/valor de cada entrada e passa cada par para a função Map que o usuário definiu. O par intermediário de chave/valor produzido pela função Map é armazenado em *buffer*. O tamanho desse *buffer* também é controlado pelo usuário por meio de um parâmetro opcional.
4. Periodicamente, os pares armazenados em *buffer* são escritos no disco local, particionado em R regiões pela função de particionamento. A localização desses pares em buffer no disco local é passada de volta ao master, que é responsável por encaminhar essas localizações para os *workers* Reduce.
5. Quando o worker Reduce é notificado pelo master sobre essas localizações, ele usa a chamada a um procedimento remoto para ler os dados em buffer do disco local dos workers Map. Quando o worker Reduce ler todos os dados intermediários, ele ordena esses dados pelas chaves intermediárias, de modo que as mesmas chaves são agrupadas. A ordenação é necessária porque frequentemente diferentes chaves Map têm a mesma tarefa Reduce. Se a quantidade de dados intermediários é grande para caber na memória, uma ordenação externa é usada.
6. O worker Reduce interage sobre os dados intermediários ordenados. Para cada chave intermediária única encontrada, o worker passa para a função Reduce definida pelo usuário a chave intermediária e o conjunto de valores intermediários correspondentes à essa chave.

7. Quando todas as tarefas Map e Reduce foram completadas, o master retorna ao programa que o chamou e que estava em modo de espera.

Depois da execução com sucesso, a saída do MapReduce fica disponível em R arquivos de saída, um para cada tarefa Reduce, com nomes especificados pelo usuário. Os usuários não precisam combinar esses R arquivos dentro de um único arquivo, pois eles são frequentemente passados como entrada para outra chamada MapReduce e assim sucessivamente até que uma situação de parada, de_nida pelo usuário, seja alcançada. A Figura 6 ilustra o funcionamento do modelo MapReduce.

Figura 6 - Funcionamento do Modelo MapReduce.



Fonte: Nascimento(2011).

O Hadoop possui diversos arquivos de configuração e os mais importantes estão listados abaixo:

- ***hadoop-env.sh***: trata-se de um arquivo de lotes e que armazena as variáveis de ambiente do Hadoop.
- ***core-site.xml***: arquivo de configuração contendo parâmetros importantes para o núcleo do Hadoop.
- ***hdfs-site.xml***: arquivo de configuração do HDFS contendo parâmetros para o Datanode e Namenode.
- ***mapred-site.xml***: arquivo de configuração para o modelo MapReduce.

- **masters:** uma lista de máquinas (uma por linha) onde cada registro será um Namenode.
- **slaves:** uma lista de máquinas (uma por linha) onde cada registro será um Datanode.

Funções Map, Reduce e Combine

O MapReduce trabalha dividindo o processamento da carga de trabalho em duas fases distintas que são: a fase Map e a fase Reduce. Essas fases, que são definidas pelo programador, dependem de pares chave-valor como entrada e também geram um formato de saída específico. Os tipos de dados desses pares de chaves podem ser definidos pelo programador e um exemplo de funcionamento dessas duas fases será apresentado mais adiante.

A largura de banda disponível para o cluster geralmente é limitada e para diminuir a transferência de dados entre as fases Map e Reduce. O Hadoop permite que seja utilizada uma terceira função denominada *Combiner*. A função *Combiner* é executada após a fase Map e antes dos dados serem repassados para a função Reduce.

Tolerância a Falhas

As aplicações do mundo real podem se deparar com *bugs* no código, falhas de máquinas ou processos que podem ser inesperadamente encerrados. Um dos maiores benefícios do Hadoop é a sua habilidade em lidar com falhas que podem ocorrer inesperadamente e, com isso, poder garantir a conclusão dos trabalhos já agendados.

Durante a execução dos trabalhos podemos ter alguns tipos de falhas que são: falha da tarefa (Task Failure), falha no serviço TaskTracker ou falha no serviço JobTracker.

Ao executar uma tarefa, a forma mais comum de ocorrer uma falha é devido a uma exceção levantada pelo código fonte, dentro de uma tarefa Map ou Reduce. Quando isso acontece, a JVM comunica o acontecido ao TaskTracker antes de encerrar a execução da tarefa. O erro é registrado em um arquivo de log e o TaskTracker marca essa tarefa que falhou, liberando o slot para ser usado por uma nova tarefa.

O sistema de arquivos do Hadoop, que veremos adiante, replica os blocos de dados de arquivos para garantir a tolerância a falhas. Como o Hadoop normalmente trabalha com um conjunto de máquinas funcionando em paralelo, quanto mais máquinas são adicionadas ao cluster, maior a possibilidade de ocorrerem falhas no processamento. Essas falhas podem surgir a partir de problemas de hardware (memória, disco, rede, etc), do sistema operacional, da JVM (Java Virtual Machine) ou mesmo uma exceção dentro do código escrito nas funções Map e Reduce.

Quando um trabalho (job) no Mapreduce é iniciado, ele é distribuído ao longo das máquinas que compõem o cluster e caso alguma dessas tarefas apresente falhas, o Hadoop possui um mecanismo para se recuperar e garantir que o trabalho seja concluído.

Quando o gerenciador de tarefas fica muito tempo sem receber informações de progresso de um determinado nó por um período de tempo, a tarefa será considerada nula e será identificado que aquele nó falhou. Nesse momento a mesma tarefa será agendada novamente e o gerenciador tenta evitar que essa tarefa seja alocada para o mesmo nó que produziu a falha anterior.

Caso uma tarefa falhe sucessivas vezes (esse número é configurado em um parâmetro de configuração), isso poderá resultar no cancelamento da tarefa, nem sempre cancelando o trabalho total. Existe um parâmetro que, quando definido, informa o percentual aceitável de cancelamento de tarefas para que o trabalho total não seja cancelado.

Um outro tipo de falha que pode ocorrer é um problema no nó escravo do cluster. Se um determinado nó sofre uma falha ou está executando os seus trabalhos de forma muito lenta o gerenciador encerra a comunicação com esse nó e ele não

mais receberá tarefas até que a sua execução seja totalmente normalizada. Além disso, o Hadoop mantém uma lista dos nós que falharam mais que a média de falhas apurada e para que um nó saia dessa lista ele terá que ser reinicializado.

Um dos maiores problemas que pode acontecer em um cluster Hadoop é a falha do nó mestre. Por se tratar do nó controlador do cluster, se não existir alguma redundância, o Hadoop não conseguirá se recuperar dessa falha.

Formatos de Entrada e Saída (texto, binário, etc.)

O Hadoop permite que diversos formatos de arquivos sejam utilizados para o processamento, sendo que os mais comuns são os formatos texto (TXT) e os arquivos estruturados e separados por vírgula (CSV), porém o Hadoop tem a capacidade de lidar com arquivos não-estruturados e complexos, tais como vídeo, áudio, imagens, etc.

Existe um fator muito importante para a definição dos tipos dos arquivos que serão usados como entrada ou saída do processamento, que é a capacidade de divisão que esse formato de arquivo tem. Isso se deve ao fato do Hadoop trabalhar com a divisão dos arquivos em blocos. Essa divisão se faz necessária para a efetiva distribuição do processamento ao longo dos nós do cluster.

Os arquivos que serão usados como entrada para o processamento devem estar preparados para serem processados fora da ordem e, os formatos TXT e CSV, normalmente são ideais para isto. Um arquivo XML pode ser diferente, pois não podemos começar a ler o arquivo no meio de uma tag.

O formato SequenceFile tem a capacidade de armazenar os dados em formato binário e é utilizado para o processamento de áudio e vídeo.

Capítulo 3. O HDFS (Hadoop Distributed File System)

Quando um determinado conjunto de dados (Dataset) extrapola a capacidade de armazenamento de uma máquina, torna-se necessário realizar o particionamento desse dataset através de outras máquinas interligadas em um cluster. Os sistemas de arquivos que gerenciam e armazenam dados através de uma rede de máquinas são chamados de Sistemas de Arquivos Distribuídos (*distributed filesystems*). Por se tratar de uma rede de máquinas distribuídas, esse tipo de sistema de arquivos é mais complexo que os sistemas de arquivos comuns que atuam apenas localmente. Um dos maiores desafios nessa arquitetura é gerenciar e tolerar as falhas de um determinado nó da rede evitando a perda de dados. O Hadoop possui o seu próprio sistema de arquivos que é chamado de HDFS (Hadoop Distributed FileSystem).

O formato do HDFS: Conceitos e comandos básicos

O HDFS é um sistema de arquivos que foi projetado para armazenar arquivos grandes distribuídos em grandes clusters. Dentre as principais características do HDFS estão:

- **Grandes arquivos:** nesse contexto estão arquivos com centenas de megabytes, gigabytes ou terabytes de tamanho. Já temos atualmente clusters Hadoop executando petabytes de dados.
- **Acesso aos dados:** o HDFS foi baseado na ideia de que o mais eficiente padrão de processamento de dados é o: escreva uma vez e leia muitas vezes. Um conjunto de dados é geralmente gerado e copiado de uma fonte e então diversas análises podem ser realizadas nesse conjunto de dados ao longo do tempo. Essas análises podem envolver uma porção do conjunto de dados ou mesmo ele todo.
- **Hardware:** o Hadoop não requer um grande, caro e sólido hardware para realizar o processamento. O Hadoop foi projetado para executar em clusters homogêneos, onde todos os nós possuem a mesma configuração ou em clusters heterogêneos, onde temos configurações de máquinas

diferentes. O Hadoop está preparado para continuar trabalhando mesmo que alguma falha ocorra, sem que o usuário perceba esse fato.

O HDFS está preparado também para trabalhar com vários pequenos arquivos e existem diversos estudos que analisaram essa característica, ao compararem o tempo de processamento de aplicações acessando um único arquivo grande no HDFS ou vários pequenos arquivos (granularidade).

Namenodes e Datanodes

Um cluster HDFS possui dois tipos de nós realizando operações no padrão mestre-escravo, um Namenode (que é o mestre) e um número indefinido de datanodes (que são os escravos).

O Namenode gerencia o sistema de arquivos mantendo a árvore de diretórios e arquivos. Essas informações são localmente persistidas por meio de dois arquivos: o namespace image e o arquivo de log. O Namenode também tem acesso a todos os Datanodes do cluster, onde blocos de dados serão enviados para processamento.

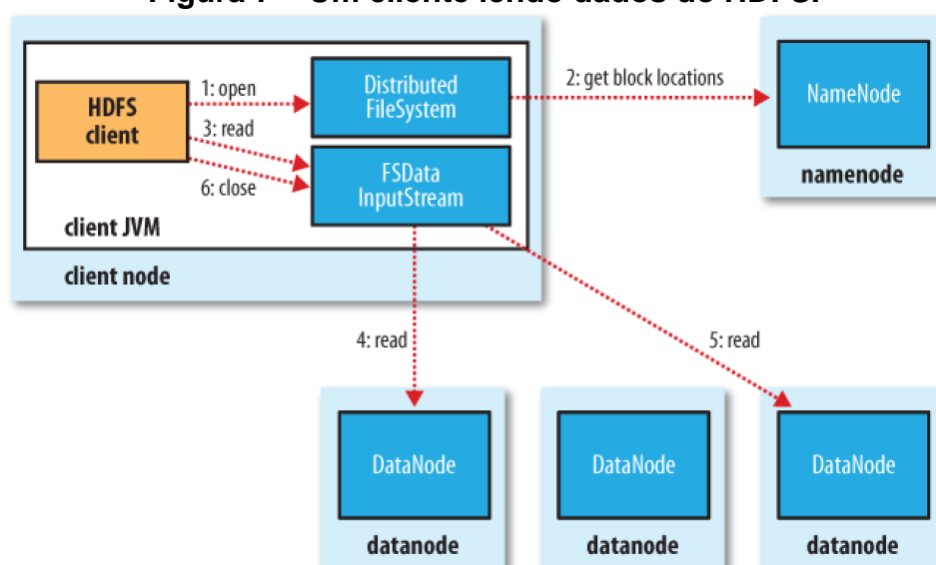
Sem o Namenode o sistema de arquivos não pode ser usado. De fato, se a máquina que faz o papel de Namenode sofrer algum problema grave, todos os arquivos do sistema de arquivos serão perdidos, desde que as informações necessárias para sua reconstrução não estejam completamente armazenadas nos Datanodes. Diante desta situação é muito importante manter o Namenode tolerante a falhas e o Hadoop fornece dois mecanismos para isso: a primeira forma é realizar uma cópia de segurança (backup) dos arquivos que armazenam o estado do sistema de arquivos. O Hadoop pode ser configurado para que o Namenode armazene seus arquivos de estado em múltiplos sistemas de arquivos. A segunda maneira é manter um segundo Namenode idêntico ao primeiro e que funcionará como um backup inteiro da máquina para o caso de alguma falha importante e irreversível. Essa segunda máquina mantém uma cópia completa dos arquivos e pode ser requisitada a qualquer momento.

Os Datanodes são os "trabalhadores" do sistema. Instruídos pelo Namenode, os Datanodes armazenam e recuperam blocos de dados e, periodicamente, reportam aos Namenodes enviando uma lista contendo os blocos de dados que eles estão armazenando.

Operações básicas

A Figura 7 apresenta um panorama da comunicação (fluxo de eventos) e do fluxo de dados entre um nó escravo (Datanode), o nó mestre (Namenode) e o HDFS, ao realizar a operação de leitura de um arquivo.

Figura 7 – Um cliente lendo dados do HDFS.



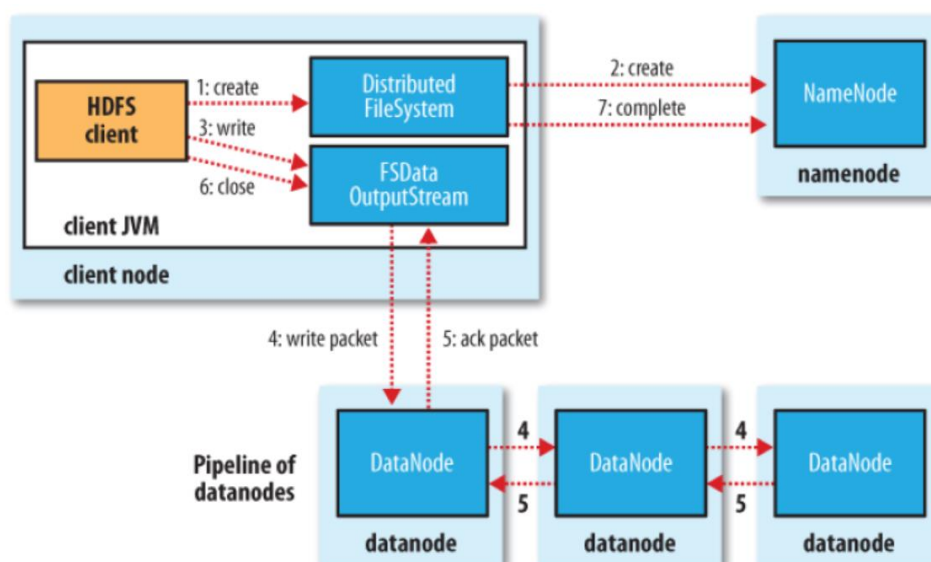
Fonte: White(2009)

O cliente abre o arquivo desejado chamando a função *Open()* no objeto *FileSystem* que é uma instância de *DistributedFileSystem* (Passo 1 da Figura 7). *DistributedFileSystem* chama o *Namenode* para que ele determine a localização dos primeiros blocos do arquivo (Passo 2). Para cada bloco do arquivo, o *Namenode* retorna os endereços dos *Datanodes* que possuem uma cópia desse arquivo. Além disso, os *Datanodes* estão ordenados de acordo com sua proximidade com o cliente, de acordo com a topologia do cluster.

O *DistributedFileSystem* retorna um *FSDDataInputStream* para o HDFS Cliente para que ele possa ler os dados. O Cliente HDFS chama a função *Read()* para a leitura de dados no *stream* (Passo 3). O *DFSInputStream*, o qual possui armazenado o endereço do Datanode, conecta-se ao Datanode mais próximo para ler o primeiro bloco do arquivo. Os dados são enviados do Datanode de volta para o cliente chamando repetidamente a função *Read()* (Passo 4). Quando o fim do bloco é alcançado, o *DfsInputStream* fechará a conexão com o Datanode e então encontrar o primeiro Datanode do próximo bloco (Passo 5).

Em seguida temos o processo de escrita no HDFS (Figura 8):

Figura 8 - Exemplo de escrita de dados no HDFS.



O cliente cria o arquivo ao chamar a função *Create()* no *DistributedFileSystem* (Passo 1 da Figura 8). O *DistributedFileSystem* faz uma chamada ao Namenode para que o novo arquivo seja criado (Passo 2). O Namenode faz várias verificações para certificar-se que o arquivo já não existe e que o cliente tem permissões para a criação desse arquivo. Se existir a permissão o Namenode faz um registro do novo arquivo. Caso não exista permissão, a criação do arquivo falha e o cliente recebe uma exceção. O *DistributedFileSystem* retorna um *FsDataOutputStream* para o cliente para que a escrita dos dados no arquivo possa começar.

A medida que o cliente escreve os dados (Passo 3), o *DFSOutputStream* particiona esses dados dentro de pacotes que serão escritos em uma fila interna,

chamada *Data Queue* (Fila de Dados). O *Data Queue* é consumido pelo *DataStreamer*, o qual tem a responsabilidade de solicitar ao Namenode a alocação de novos blocos, escolhendo em uma lista de Datanodes os mais adequados para armazenar as réplicas. A lista de Datanodes forma um pipeline (fila) e, caso o número de réplicas esteja definido como 3, então teremos 3 nós nesse pipeline. O *DataStreamer* divide os pacotes para o primeiro Datanode do pipeline e ele armazena os dados, e o próprio Datanode repassa os pacotes para o Datanode 2. Ao final o Datanode 2 repassa os dados para o Datanode 3 (Passo 4).

O *DFSOutputStream* também mantém uma fila interna de pacotes que estão aguardando para serem conhecidos pelos Datanodes. Essa fila é chamada de *Ack Queue*. Um pacote é removido da *Ack Queue* apenas quando ele foi conhecido por todos os Datanodes no Pipeline (Passo 5).

Quando o cliente finaliza a escrita dos dados, ele chama a função *Close()* no *stream* (Passo 6). Essa ação descarrega todos os pacotes restantes para o pipeline e aguarda que todos os Datanodes reconheçam antes de entrar em contato com o Namenode indicando que a escrita do arquivo foi completa (Passo 7).

Interface gráfica para gerenciamento do HDFS

O Hadoop traz algumas interfaces gráficas que podem ser utilizadas para o gerenciamento dos serviços e podem ser acessadas pelo *browser*. Uma delas pode ser acessada pela porta 50070 (<http://servidor:9870>) e possui diversas informações sobre o cluster, conforme apresentado na Figura 9.

Figura 9 – Interface gráfica do Hadoop (porta 9870).

| Hadoop | Overview | Datanodes | Datanode Volume Failures | Snapshot | Startup Progress | Utilities |
|---------------------------------------|---|-----------|--------------------------|----------|------------------|-----------|
| Overview 'HadoopMaster:9000' (active) | | | | | | |
| Started: | Sat Apr 30 22:25:45 UTC 2016 | | | | | |
| Version: | 2.7.1, r15ecc87ccf4a0228f35af08fc56de536e6ce657a | | | | | |
| Compiled: | 2015-06-29T06:04Z by jenkins from (detached from 15ecc87) | | | | | |
| Cluster ID: | CID-87116469-144b-456f-b395-ba421b18bfff | | | | | |
| Block Pool ID: | BP-2092665936-10.54.8.6-1462055141567 | | | | | |

A Figura 10 apresenta diversas informações úteis dos Datanodes. Nesse exemplo podemos observar 4 Datanodes ativos, suas capacidades de armazenamento e o quanto está sendo consumido dessas capacidades. Além disso é apresentado quantos blocos eles estão armazenando, quantos volumes falharam e qual versão do Hadoop cada um desses Datanodes está usando.

Figura 10 – Interface gráfica do Hadoop (porta 50070).

Hadoop

Overview

Datanodes

Datanode Volume Failures

Snapshot

Startup Progress

Utilities

Datanode Information

In operation

| Node | Last contact | Admin State | Capacity | Used | Non DFS Used | Remaining | Blocks | Block pool used | Failed Volumes | Version |
|--------------------------------------|--------------|-------------|----------|-----------|--------------|-----------|--------|-------------------|----------------|---------|
| metrca2-3:50010 (10.54.8.4:50010) | 0 | In Service | 393.6 GB | 737.26 MB | 20.35 GB | 372.53 GB | 106 | 737.26 MB (0.18%) | 0 | 2.7.1 |
| metrca2-4:50010 (10.54.8.7:50010) | 0 | In Service | 393.6 GB | 736.43 MB | 20.43 GB | 372.45 GB | 99 | 736.43 MB (0.18%) | 0 | 2.7.1 |
| metrca2-2:50010 (10.54.8.5:50010) | 0 | In Service | 393.6 GB | 820.04 MB | 20.29 GB | 372.51 GB | 103 | 820.04 MB (0.2%) | 0 | 2.7.1 |
| HadoopMaster:50010 (10.54.8.6:50010) | 2 | In Service | 393.6 GB | 830.29 MB | 20.36 GB | 372.43 GB | 114 | 830.29 MB (0.21%) | 0 | 2.7.1 |

Através das interfaces gráficas podemos ainda ter informações acerca do HDFS, tais como diretórios, tamanho do bloco, arquivos, etc. É possível também acessar os arquivos que estão armazenados no HDFS. A Figura 11 apresenta uma visão geral do HDFS.

Figura 11 – Visão do HDFS por meio de interface gráfica.

| Hadoop Overview Datanodes Snapshot Startup Progress Utilities | | | | | | | |
|---|--------|------------|------|---------------------|-------------|------------|----------------------|
| Browse Directory | | | | | | | |
| / | | | | | | | Go! |
| Permission | Owner | Group | Size | Last Modified | Replication | Block Size | Name |
| drwx----- | hduser | supergroup | 0 B | 30/04/2016 19:26:21 | 0 | 0 B | tmp |
| drwxr-xr-x | hduser | supergroup | 0 B | 30/04/2016 19:26:07 | 0 | 0 B | user |

Hadoop, 2015.

Todos os logs de execução do Hadoop também podem ser acessados pela interface gráfica da porta 9870, conforme apresentado na Figura 12.

Figura 12 – Visão dos logs do Hadoop por meio de interface gráfica.

Directory: /logs/

| | | |
|--|--------------|--------------------------|
| SecurityAuth-hduser.audit | 0 bytes | Apr 30, 2016 11:16:14 PM |
| hadoop-hduser-datanode-HadoopMaster.log | 68991 bytes | Apr 30, 2016 11:17:51 PM |
| hadoop-hduser-datanode-HadoopMaster.out | 718 bytes | Apr 30, 2016 11:16:19 PM |
| hadoop-hduser-namenode-HadoopMaster.log | 216054 bytes | Apr 30, 2016 11:17:52 PM |
| hadoop-hduser-namenode-HadoopMaster.out | 718 bytes | Apr 30, 2016 11:16:14 PM |
| hadoop-hduser-secondarynamenode-HadoopMaster.log | 22743 bytes | Apr 30, 2016 11:17:29 PM |
| hadoop-hduser-secondarynamenode-HadoopMaster.out | 718 bytes | Apr 30, 2016 11:16:24 PM |
| userlogs/ | 4096 bytes | Apr 30, 2016 11:17:01 PM |
| yarn-hduser-nodemanager-HadoopMaster.log | 194842 bytes | Apr 30, 2016 11:17:49 PM |
| yarn-hduser-nodemanager-HadoopMaster.out | 702 bytes | Apr 30, 2016 11:16:32 PM |
| yarn-hduser-resourcemanager-HadoopMaster.log | 548947 bytes | Apr 30, 2016 11:17:51 PM |
| yarn-hduser-resourcemanager-HadoopMaster.out | 702 bytes | Apr 30, 2016 11:16:31 PM |

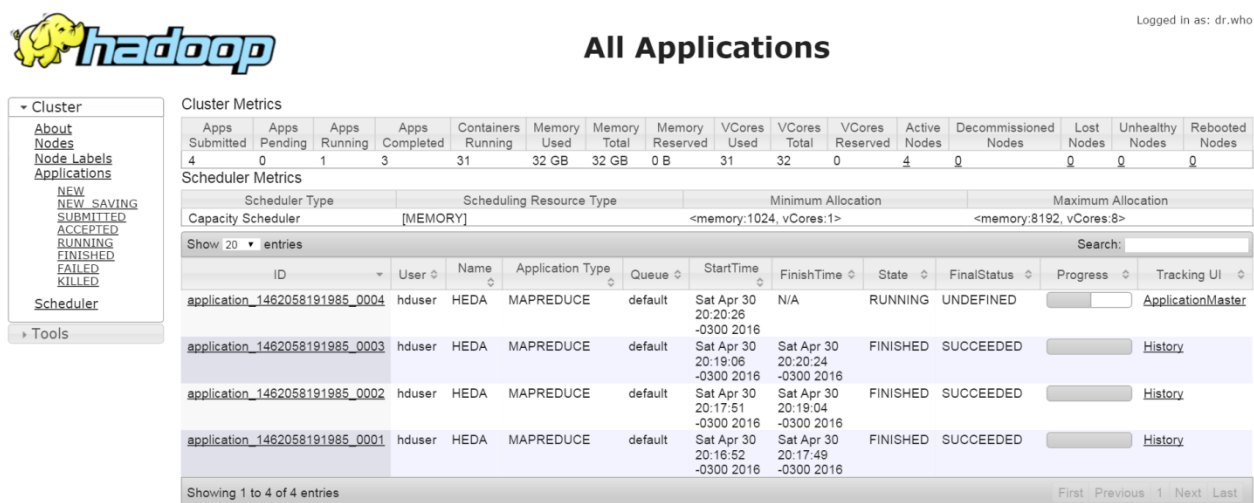
Outra interface gráfica que auxilia no desenvolvimento e na execução de programas usando o Hadoop é a tela de aplicações. Essa tela pode ser acessada por meio da porta 8088 (<http://servidor:8088>). A Figura 13 apresenta essa interface que traz informações muito importantes, tais como:

- **Trabalhos:** todos os trabalhos que já foram executados pelo Hadoop com nome, fila, hora de início, hora de fim, situação atual do trabalho, situação final

e progresso. Clicando no histórico do trabalho ainda é possível consultar diversas outras informações como os logs de execução por tarefa.

- **Métricas do cluster:** muitas outras informações sobre o cluster são apresentadas, tais como: número de trabalhos submetidos, pendentes, executando, completados, etc. Além disso temos a quantidade total de nós no cluster, memória total disponível e utilizada, núcleos usados e disponíveis, etc.

Figura 13 – Interface gráfica de aplicações do Hadoop.



Capítulo 4. Criando programas Apache Hadoop

O objetivo desse capítulo é apresentar a estrutura de um programa Apache Hadoop. Será discutido aqui cada uma das classes básicas para implementação desse programa, inclusive com a apresentação de alguns exemplos de implementação.

Estrutura de um programa Hadoop e a classe Jobconf

A estrutura de um programa Hadoop segue a mesma estrutura de um programa Java comum. É necessário definir previamente uma classe principal e dentro desta o método *main()*, bem conhecido pelos programadores Java.

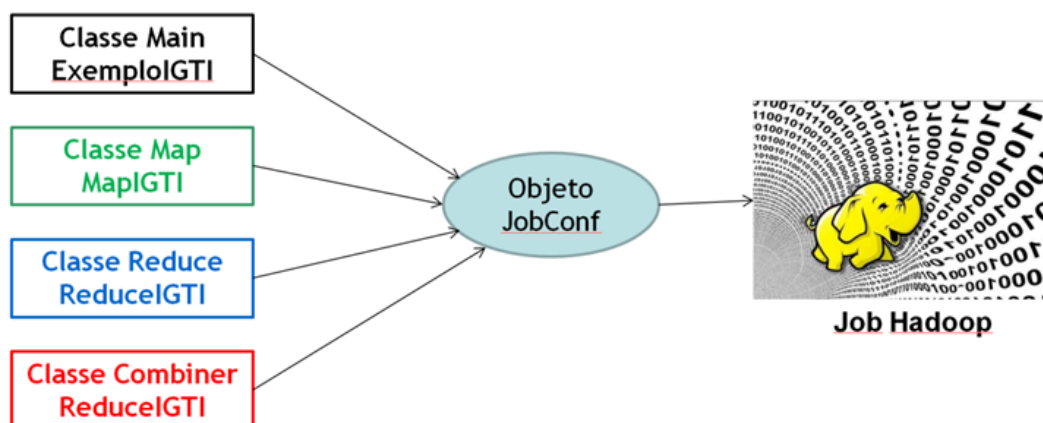
Além da classe principal é necessário criar também uma classe exclusiva para o mapeamento dos dados. Essa será a responsável por realizar a implementação do método *Map()* e iremos referenciá-la aqui como *Mapper*. Outra classe que pode ser criada é aquela que ficará responsável por conter a implementação do método *Reduce()* e iremos referenciar essa classe como *Reducer*. Essa mesma classe pode ser utilizada como uma classe *combiner*.

O Hadoop fornece ainda um objeto que é responsável por manter todos os dados referentes ao *job* que será executado. A classe *JobConf* é a interface primária para o usuário descrever um *job MapReduce* e enviar para o *framework* Hadoop para execução. O *framework* ao receber o *JobConf* tenta executar o *job* conforme descrito pelo objeto.

O *JobConf* especifica quais as classes *Mapper*, *Combiner* (se existir), *Reducer*, e formatos de entrada e saída dos *jobs*. Alguns parâmetros de configuração podem ser atribuídos pelo *JobConf*, porém não todos, uma vez que o Hadoop possui mais de 200 parâmetros configuráveis do *JobConf*. Abaixo estão listados alguns exemplos de métodos constantes na classe e alguns deles exigem o reinício dos serviços, o que não é possível realizar por meio *JobConf*:

- **setStrings:** enviar dados da classe principal para as classes *Mapper* ou *Reducer*.
- **setNumMapTasks:** atribui o número de tarefas *Map* para cada *job*.
- **setNumReduceTasks:** atribui o número de tarefas *Reduce* para cada *job*.
- **getJobName:** recupera o nome do Job que foi atribuído pelo usuário.
- **getMaxMap[Reduce]Attempts:** recupera o número de tentativas de execução da tarefa Map/Reduce definida pelo usuário.
- **setJobName:** método para atribuição de um nome para o *job*.
- **setOutputKeyClass (Class<?> theClass):** atribui a classe para a saída dos dados do *job* (dados da chave – Key).
- **setOutputValueClass(Class<?> theClass):** atribui a classe para a saída dos dados do *job* (dados do valor - *value*).
- **setMapperClass:** atribui uma classe *Mapper* para o *job* (minha classe).
- **setReducerClass:** atribui uma classe *Reducer* para o *job* (minha classe).
- **setCombinerClass:** atribui uma classe *Combiner* para o *job*. Geralmente a mesma classe *Reducer*.

Figura 14 – Estrutura básica de um programa Hadoop.



A Figura 14 apresenta a estrutura de um programa Hadoop, com a classe principal *ExemploIGTI* contendo o método *Main()*, a classe *Mapper* (*MapIGTI*), a classe *Reducer* (*ReduceIGTI*) e a classe *Combiner*, que no exemplo apresentado utilizou a mesma classe *Reducer*. O objeto *JobConf* (no meio da figura) possui as informações sobre todas essas quatro classes e ele é o responsável por enviar esses dados para o *job* que será executado pelo Hadoop.

A Figura 15 apresenta uma implementação Java básica do esquema apresentado na Figura 14. Podemos observar que a implementação possui a classe principal *ExemploIGTI* (linha 4) que herda a classe *Configured* e implementa a interface *Tool*. Essa classe possui uma instância do objeto *JobConf* que chamamos de *conf* (linha 11). Utilizamos o método *setJobName* para atribuir o nome do nosso *job* Hadoop (linha 12), atribuímos os tipos de saída das nossas chaves e valores (*Text*) e atribuímos nossas classes *Mapper* (*MapIGTI*), *Reducer* (*ReduceIGTI*) e *Combiner* (*ReduceIGTI*) por meio dos métodos *setMapperClass*, *setReducerClass* e *setCombinerClass*, respectivamente.

Figura 15 – Exemplo de utilização da classe JobConf.

```

1 package exemploigti;
2 import org.apache.hadoop.conf.*;
3
4 public class ExemploIGTI extends Configured implements Tool {
5     public static void main (final String[] args) throws Exception {
6         int res = ToolRunner.run(new Configuration(), new ExemploIGTI, args);
7     }
8
9     public int run (final String[] args) throws Exception {
10         ...
11         JobConf conf = new JobConf(getConf(), ExemploIGTI.class);
12         conf.setJobName("Nome do Job");
13
14         conf.setOutputKeyClass(Text.class);
15         conf.setOutputValueClass(Text.class);
16
17         conf.setMapperClass(MapIGTI.class);
18         conf.setReducerClass(ReduceIGTI.class);
19         conf.setCombinerClass(ReduceIGTI.class);
20         ...
21     }
22 }

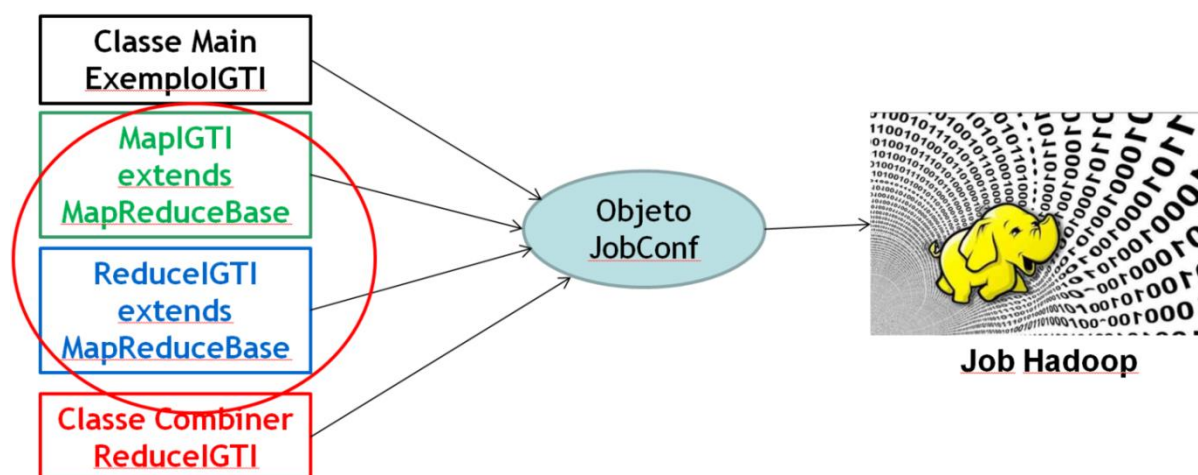
```

A classe MapReduceBase

A classe *MapReduceBase* é a classe base para implementação das nossas classes *Mapper*, *Reducer* e *Combiner* e encontra-se em `org.apache.hadoop.mapred.MapReduceBase`.

As classes *Mappers* e *Reducers* são implementadas pelo usuário, possuem os métodos *Map()* e *Reduce()* e são herdadas da classe *MapReduceBase*. A classe *MapReduceBase* possui apenas dois métodos que serão discutidos mais a frente: *Configure(JobConf job)* e *Close()*. A Figura 16 destaca, dentro do esquema que estamos exemplificando, as classes que herdam de *MapReduceBase* (*MapIGTI* e *ReduceIGTI*).

Figura 16 – Classes que herdam de *MapReduceBase*.



A Figura 17 apresenta a atribuição das classes *MapIGTI*, e *ReduceIGTI* usando os métodos *setMapperClass*, *setReducerClass* e *setCombinerClass*, todos da classe *JobConf*.

Figura 17 – Exemplo de utilização da classe MapReduceBase – Atribuição.

```

1 package exemploigti;
2 import org.apache.hadoop.conf.*;
3
4 public class ExemploIGTI extends Configured implements Tool {
5     public static void main (final String[] args) throws Exception {
6         int res = ToolRunner.run(new Configuration(), new ExemploIGTI, args);
7     }
8
9     public int run (final String[] args) throws Exception {
10         ...
11         JobConf conf = new JobConf(getConf(), ExemploIGTI.class);
12         conf.setJobName("Nome do Job");
13
14         conf.setOutputKeyClass(Text.class);
15         conf.setOutputValueClass(Text.class);
16
17         conf.setMapperClass(MapIGTI.class);
18         conf.setReducerClass(ReduceIGTI.class);
19         conf.setCombinerClass(ReduceIGTI.class);
20         ...
21     }
22 }

```

A Figura 18 apresenta a implementação da classe *MapIGTI*, com o seu método *map()*. Observe que *MapIGTI* herda de *MapReduceBase*.

Figura 18 – Exemplo de utilização da classe MapReduceBase – Mapper.

```

27 ...
28 public static class MapIGTI extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {
29     public void configure (JobConf job) {
30         ... //sua implementação
31     }
32
33     public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter)
34     throws IOException {
35         ... //sua implementação
36     }
37
38     public void close(){
39         ... //sua implementação
40     }
41 }

```

A Figura 19 destaca a implementação da classe *ReduceIGTI*, com o seu método *reduce()*. Observe que *ReduceIGTI* herda também da classe *MapReduceBase*.

Figura 19 – Exemplo de utilização da classe MapReduceBase – Reducer.

```

44 public static class ReduceIGTI extends MapReduceBase implements Reducer<Text, Text, Text, Text>
45 {
46     ... //sua implementação
47 }
48
49 public void reduce (Text key, Iterator<Text> values, OutputCollector<Text, Text> output,
50 Reporter reporter) throws IOException {
51     ... //sua implementação
52 }
53
54 public void close(){
55     ... //sua implementação
56 }
57

```

Os métodos `map()` e `reduce()` vêm das interfaces *Mapper* e *Reducer*. Os parâmetros desses métodos são:

- *key*: a chave do par *key/value*.
- *Values (reduce)*: uma lista com os valores referentes à cada *key*.
- *Value (map)*: valor que compõe o par *key/value*.
- *Output*: dados de saída dos métodos
- *Reporter*: uma maneira de reportar o andamento da execução do *job*.

Criando consultando e excluindo diretórios no HDFS (via aplicação)

O Hadoop permite que o usuário manipule e execute operações básicas de diretórios a partir de sua própria aplicação. Essa tarefa é extremamente importante, uma vez que isso traz dinamismo para os nossos programas Hadoop.

A classe que utilizamos para realizar essa manipulação de diretórios é a *FileSystem*, que possui dezenas de métodos. A documentação dessa classe pode ser consultada em:

<https://hadoop.apache.org/docs/r2.7.1/api/index.html?org/apache/hadoop/fs/FileSyst%20em.html>.

Alguns métodos úteis da classe *FileSystem*:

- **exists(Path f)** – verifica se um determinado caminho existe.
- **getHomeDirectory()** – retorna o diretório raiz (home) no HDFS.
- **isDirectory (Path p)** – verifica se o caminho p é um diretório.
- **isFile(Path f)** – verifica se p é um arquivo.
- **mkdirs(Path p)** – permite a criação de um diretório.
- **delete(Path p)** – permite a exclusão de um diretório.

A Figura 20 apresenta um exemplo de utilização da classe *FileSystem*. Aqui criamos um objeto *FileSystem*, verificamos se o diretório IGTI existe (comando *fs.exists(p)*). Se o diretório não existe, ele é imediatamente criado com o comando *fs.mkdirs(p)* e se existe ele é deletado com o método *fs.delete(p)*. A variável *p* é do tipo *Path* e determina o diretório que será manipulado.

Figura 20 – Exemplo de utilização da classe *FileSystem*.

```
public int run (final String[] args) throws Exception {
    try{
        JobConf conf = new JobConf(getConf(), ComandosHDFS.class);
        conf.setJobName("Ola Mundo");

        final FileSystem fs = FileSystem.get(conf);

        Path p = new Path("IGTI"); /* caminho no HDFS: /user/hduser/IGTI */

        if (fs.exists(p)) { /* verifica se o caminho p existe */
            System.out.println("Diretorio IGTI encontrado");
            fs.delete(p); /* como o diretório já existe, vamos exclui-lo */
        }
        else {
            System.out.println("Diretorio inexistente");
            fs.mkdirs(p); /* como o diretório não existe, vamos cria-lo */
        }
    }
    catch ( Exception e ) {
        throw e;
    }
    return 0;
}
```

O resultado dos métodos da classe *FileSystem* podem ser consultados via linha de comando ou pela ferramenta de gerenciamento do Hadoop no endereço <http://localhost:9870>.

Lendo e gravando dados do HDFS (via aplicação)

É muito importante para uma aplicação Hadoop ler e gravar dados no HDFS. Essa interação (aplicação x HDFS) é extremamente útil, principalmente se estivermos lidando com um algoritmo iterativo que manipula o HDFS o tempo todo.

A classe *FileSystem*, que vimos anteriormente, fornece vários métodos para que possamos lidar com os arquivos. Além de *FileSystem*, podemos utilizar duas outras importantes classes, que são: *FileInputFormat* e *FileOutputFormat*.

A classe *FileInputFormat* permite ao usuário definir qual o(s) diretório(s) serão atribuídos como entrada para um *job* Hadoop. Abaixo alguns métodos úteis:

- **static setInputPaths(JobConf conf, Path inputPaths):** atribui caminhos como entrada para o job MapReduce.
- **static setInputPaths(JobConf conf, String commaSeparatedPaths):** lista de entradas para o job MapReduce.
- **static getInputPaths(JobConf conf):** retorna uma lista de Paths para o job MapReduce.

A classe *FileOutputFormat* permite que o usuário defina o diretório do HDFS onde os dados de saída do *job* Hadoop serão gravados. Alguns métodos úteis da classe *FileOutputFormat*.

- **static setOutputPath(JobConf conf, Path outputDir):** atribui um caminho para o diretório de saída do job MapReduce.
- **setCompressOutput(JobConf conf, boolean compress):** define se os dados de saída do *job* serão compactados.

A Figura 21 apresenta a utilização das classes *FileInputFormat* e *FileOutputFormat*. Foram criados dois diretórios com os nomes “entrada” e “saída”.

Figura 21 – Exemplo de utilização das classes *FileInputFormat* e *FileOutputFormat*.

```
public class ExemploIGTI extends Configured implements Tool
{
    public static void main (final String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new ExemploIGTI(), args);
        System.exit(res);
    }

    public int run (final String[] args) throws Exception {
        try{
            JobConf conf = new JobConf(getConf(), ExemploIGTI.class);
            conf.setJobName("Ola Mundo");

            String caminhoEntrada = "entrada";
            FileInputFormat.setInputPaths(conf, caminhoEntrada);
            FileOutputFormat.setOutputPath(conf, new Path("saida"));

            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(Text.class);
            conf.setMapperClass(MapIGTI.class);
            conf.setReducerClass(ReduceIGTI.class);
            JobClient.runJob(conf);
        }
        catch ( Exception e ) {
            throw e;
        }
        return 0;
    }
}
```

Um “Olá mundo!” utilizando o Hadoop

Para a construção de um programa básico utilizando o Hadoop, iremos seguir os seguintes passos:

- Configurar um objeto *JobConf*.
- Criar um diretório de entrada com a classe *FileSystem*.
- Incluir um arquivo para processamento (do sistema de arquivos do Linux para o HDFS), com o método *copyFromLocalFile*.
- Atribuir um diretório de entrada para o *Job*.
- Atribuir um diretório de saída para o *Job* (output).
- Atribuir as classes *Mapper* e *Reducer*.
- Apresentar a implementação dos métodos *Map* e *Reduce*.

- Executar o *Job*.
- Apresentar e analisar os resultados.

Primeiramente iremos criar um objeto *JobConf* e atribuir o nome da aplicação, conforme apresentado na Figura 22.

Figura 22 – Criação do objeto *JobConf*.

```
JobConf conf = new JobConf(getConf(), ExemploIGTI.class);
conf.setJobName("Ola Mundo");
```

Em seguida, por meio da classe *FileSystem*, iremos criar um diretório de entrada (*fs.mkdirs*), verificando antes se o diretório já existe (*fs.exists*). Em seguida, por meio do método *copyFromLocalFile* iremos copiar um arquivo do sistema de arquivos do Linux para o diretório de entrada do HDFS, conforme apresentado na Figura 23.

Figura 23 – Criação de um diretório com a classe *FileSystem*.

```
final FileSystem fs = FileSystem.get(conf);
Path diretorioEntrada = new Path("Entrada")

/* Criar um diretorio de entrada no HDFS */
if (!fs.exists(diretorioEntrada))
    fs.mkdirs(diretorioEntrada);

/* Adicionar um arquivo para ser processado */
fs.copyFromLocalFile(new Path("/usr/local/hadoop/Dados/arquivoBigData.txt"), diretorioEntrada);
```

O conteúdo do arquivo *arquivoBigData.txt* é apresentado na Figura 24. O arquivo refere-se às compras realizadas por um cliente em uma empresa. Destacamos as posições do código do cliente em azul e a quantidade de vendas em vermelho. Esses serão os valores para os nossos pares chave/valor (*key/value*).

Figura 24 – Conteúdo do arquivo arquivoBigData.txt.

[illegible]

Por meio das classes *FileInputFormat* e *FileOutputFormat* iremos definir quais são os diretórios de entrada e saída no HDFS usados pelo *job* Hadoop. A Figura 25 apresenta o código para atribuição desses diretórios.

Figura 25 – Atribuindo os diretórios de entrada e saída para o *job* Hadoop.

```
/* Atribuindo os diretorios de Entrada e Saida para o Job */
FileInputFormat.setInputPaths(conf, diretorioEntrada);
FileOutputFormat.setOutputPath(conf, diretorioSaida);
```

Após, os *Mappers* e *Reducers* serão atribuídos. Iremos utilizar as classes `MapIGTI` e `ReduceIGTI`. Ambas as classes são de responsabilidade dos desenvolvedores e possuem as implementações dos métodos *Map* e *Reduce* respectivamente. A Figura 26 apresenta a atribuição das classes.

Figura 26 – Atribuição do *Mapper* e *Reducer*.

```
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setMapperClass(MapIGTI.class);
conf.setReducerClass(ReduceIGTI.class);
JobClient.runJob(conf);
```

Em seguida a Figura 27 destaca a implementação do método *Map* na classe MapIGTI. Observe que o trabalho do *Map* é “recortar” o código do cliente e a quantidade de itens comprados. O código do cliente será enviado para a função *Reduce* como *key* e a quantidade de itens será enviada como *value*.

Figura 27 – Implementação da classe MapIGTI.

```
public static class MapIGTI extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {
    public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        Text txtChave = new Text();
        Text txtValor = new Text();

        String codigoCliente = value.toString().substring(58, 61);
        String qtdeItens = value.toString().substring(76, 84);

        txtChave.set(codigoCliente);
        txtValor.set(qtdeItens);

        output.collect(txtChave, txtValor);
    }
}
```

A Figura 28 apresenta a implementação da classe Reducer. O objetivo da função *Reduce* é encontrar a média de itens comprado por cada cliente. Para isso ela utiliza a variável *contaVendas* para contar a quantidade de vendas de cada cliente, além de um acumulador chamado *acumuladorItens*, que tem o objetivo de somar todos os itens comprados por um mesmo cliente. Ao final é calculada a média de itens e gravado o resultado com o método *output.collect(..)*. O “código do cliente” será gravado como chave (*key*) e o e a “média de itens” será gravada como valor (*value*).

Figura 28 – Implementação da classe *ReduceIGTI*.

```
public static class ReduceIGTI extends MapReduceBase implements Reducer<Text, Text, Text, Text> {
    public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        double media = 0.0;
        int acumuladorItens = 0, contaVendas = 0;
        Text value = new Text();

        while (values.hasNext()) {
            value = values.next();
            contaVendas++;
            acumuladorItens += Integer.parseInt(value.toString());
        }

        media = acumuladorItens / new Double(contaVendas);
        value.set(String.valueOf(media));
        output.collect(key, value);
    }
}
```

O resultado do processamento pode ser consultado usando as ferramentas de gerenciamento do Hadoop, através das portas 8088 e 9870. A Figura 29 apresenta o resultado do processamento com o arquivo resultado que se encontra dentro do diretório saída.

Figura 29 – Arquivo com o resultado do processamento.

| | |
|-----|-----------|
| 001 | 18664.428 |
| 004 | 6644.5 |
| 007 | 37482.75 |
| 008 | 2235.0 |
| 009 | 21051.75 |

Métodos *Configure* e *Close*

Os métodos *Configure* e *Close* estão presentes na classe *MapReduceBase* que é a classe que herdamos em nossos *Mappers* e *Reducers*.

O método *Configure* possui como parâmetro o objeto *JobConf*. Essa é uma forma que temos de enviar dados da nossa classe principal (onde está nosso método *main()*) para os nossos *Mappers* ou *Reducers*. Além disso, o método *Configure* é executado antes dos métodos *Map()* ou *Reduce()* e isso nos permite preparar algo antes da execução dos métodos principais das nossas classes *Mapper* ou *Reducer*. Muitas vezes utilizamos o método *Configure()* como uma forma de preparar os dados antes da execução efetiva da rotina principal. Podemos, por exemplo, instanciar objetos que serão usados posteriormente, atribuir valores iniciais para variáveis ou mesmo realizar o registro de *logs*.

A Figura 30 destaca uma forma de enviar dados da classe principal para os *Mappers* ou *Reducers*. Isso é feito por meio do método *setString*, onde informamos o nome do registro (que no exemplo foi “codigoExecucao”) e o valor a ser enviado (que no nosso exemplo foi 337).

Figura 30 – Exemplo de envio de dados da classe principal para um *Mapper* ou *Reducer*.

```
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class ExemploIGTI extends Configured implements Tool
{
    public static void main (final String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new ExemploIGTI(), args);
        System.exit(res);
    }

    public int run (final String[] args) throws Exception {
        try{
            JobConf conf = new JobConf(getConf(), ExemploIGTI.class);
            conf.setJobName("Ola Mundo");
            conf.setStrings("codigoExecucao", Integer.toString(337)); ←
        }
    }
}
```

A Figura 31 apresenta uma maneira de recuperarmos os dados que foram enviados pelo objeto *JobConf* e método *setStrings*. Observe que podemos acessar esses dados no método *Configure()* que é quem recebe o *JobConf* como parâmetro. No nosso exemplo apenas recuperamos o valor de *codigoExecucao* e enviamos para o método *System.out.println(...)* que se encarregará de registrar um *log*.

Figura 31 – Exemplo de recuperação de dados enviados pela classe principal.

```
public static class ReduceIGTI extends MapReduceBase implements Reducer<Text, Text, Text, Text> {
    Text value;
    public void configure(JobConf job) { ←
        value = new Text();
        System.out.println(job.getStrings("codigoExecucao")); ←
    }

    public void reduce (Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
    }
}
```

O método *Close()* não possui parâmetros e é chamado pelo *frameworkHadoop* quando todas as entradas tiverem sido processadas pelos métodos *Map* ou *Reduce*. Geralmente utilizamos o método *Close* para realizar alguma tarefa que arremate o processamento dos dados, ou seja, fechar algum arquivo que foi aberto durante a execução, efetivar alterações ou mesmo registrar algum *log* adicional/final. Observe que no método *Close* nós não temos acesso ao objeto *JobConf*, entretanto é possível obter esse acesso atribuindo o conteúdo do *JobConf* a uma variável global da classe *Mapper* ou *Reducer* e utiliza-la dentro do método *Close*. A Figura 32 destaca a utilização do método *Close*. Nesse exemplo apenas registramos uma frase no *log* por meio do método *System.out.println(...)*.

Figura 32 – Exemplo de utilização do método *Close*.

```
public void close() {
    System.out.println("Processamento encerrado ");
}
```

Definindo uma função *Combine*

Durante a execução de um *job* Hadoop/MapReduce temos um grande gargalo no processamento que é a largura de banda disponível no *cluster*. A troca de informações entre *Mappers* e *Reducers* causam um tempo adicional (*overhead*) ao framework que pode, em algumas situações, ser amenizado com a utilização da função *Combine*.

A função *Combine* recebe como entrada os dados que foram processados pela função *Map* e aplica uma função sobre esses ainda no *Datanode*, com o objetivo de reduzir a quantidade de dados que chega para a função *Reduce*. Devemos utilizar a função *Combine* com cautela, pois em alguns casos o uso dessa função pode afetar o resultado do processamento. A Figura 33 apresenta a utilização da função *Combine* sem alterar o resultado do processamento:

Figura 33 – Utilização de função *Combine* sem alteração no resultado do processamento.

| <u>Node 1</u> | <u>Node 2</u> | <u>Entrada para Reduce</u> | <u>Saída de Reduce</u> |
|--|--------------------------|-----------------------------|------------------------|
| (1950, 0) (1950, 20) (1950, 10) | (1950, 25) (1950, 15) | (1950, [0, 20, 10, 25, 15]) | (1950, 25) |
| | | | |
| <u>Entrada para Reduce com Combine</u> | | | |
| (1950, [20, 25]) | | | |

Observe que a função *Reduce* deveria receber como entrada 5 pares chave/valor (3 do Node1 e 2 do Node2), porém essa quantidade de valores foi reduzida para apenas dois valores. O que a função *Combine* fez foi aplicar um pré-reduce e já encontrar o maior valor ainda no próprio *Datanode*. A função *Reduce* pode ser descrita mais claramente na Figura 34.

Figura 34 – Função Reduce.

$$\text{max}(0, 20, 10, 25, 15) = \text{max}(\text{max}(0, 20, 10), \text{max}(25, 15)) = \text{max}(20, 25) = 25$$

Dois fatores devem ser levados em consideração no momento da decisão pela utilização de uma função *Combine*. O primeiro é o cuidado para não alterar o valor final e correto dos dados e o segundo é que precisamos levar em conta que a função *Combine* tem um custo computacional e esse custo deve ser compensado com a redução no tempo total de processamento, ou seja, a função *Combine* deve ter um bom custo-benefício.

Se utilizarmos os mesmos dados do exemplo anterior e, se ao invés de buscarmos o maior valor buscarmos a média dos valores, poderemos ter alteração no valor final. Se fizermos a média dos elementos $\text{mean}(0, 20, 10, 25, 15)$ obteremos o valor 14. Se fizermos a média utilizando uma função *Combine*, teremos para o Node1 $\text{mean}(0, 20, 10) = 10$ e para o Node2 $\text{mean}(25, 15) = 20$. Em seguida aplicaremos a função *Reduce* para os valores resultantes da aplicação das duas funções *Combine* anteriores: 10, 20. Finalmente, $\text{mean}(10, 20) = 15$. Observe que tivemos alteração no resultado. Sem usar funções *Combine*, obtivemos o resultado 14 e com o uso o resultado foi 15.

A função *Combine* não substitui a utilização da função *Reduce*, pois os dados processados por ela são locais e não globais, daí o apelido de pré-reduce. O que a função *Combine* pode fazer é reduzir a quantidade de dados que serão mesclados após a execução da função *Map*. Na grande maioria das vezes utilizamos para a função *Combine* a mesma função que utilizamos em nossos *Reducers*.

Depurando um *job* e os *logs* do Hadoop

Existem várias formas de depurar um programa Hadoop. As três principais são: Registro de Logs, *Counters* e *Plugins*.

Na parte de Registro de Logs fazemos o registro de cada linha desejada e, após a execução, consultamos os *logs* do Hadoop. Podemos fazer esses registros

utilizando os métodos `System.out.println(..)` e `System.err.println()`. A Figura 35 apresenta uma forma de realizarmos um registro de *log*, caso a média da quantidade de itens comprado por um cliente seja maior que 15.000.

Figura 35 – Exemplo de registro de Log utilizando o método `System.out.println`.

```
public static class ReduceIGTI extends MapReduceBase implements Reducer<Text, Text, Text, Text> {

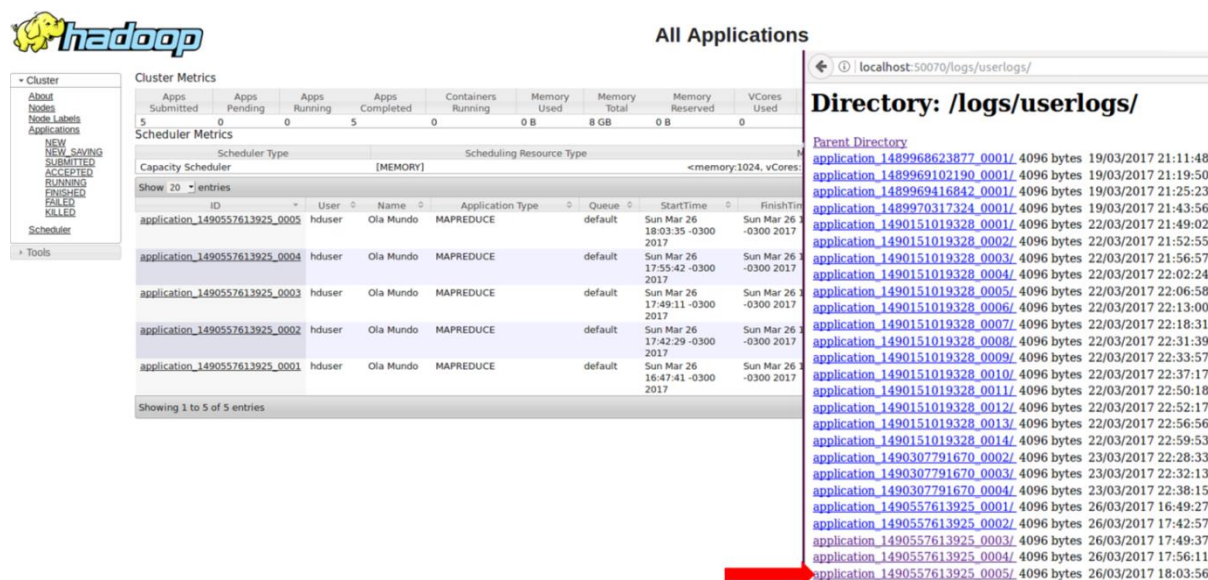
    public void reduce (Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        double media = 0.0;
        int acumuladorItens = 0, contaVendas = 0;
        Text value = new Text();

        while (values.hasNext()) {
            value = values.next();
            contaVendas++;
            acumuladorItens += Integer.parseInt(value.toString());
        }
        media = acumuladorItens / new Double(contaVendas);

        if (media > 15000)
            System.out.println("Chave " + key.toString() + " possui mais de 15000 itens ( " + String.valueOf(media) + " ) ");
        value.set(String.valueOf(media));
        output.collect(key, value);
    }
}
```

Para consultar os *logs* que foram gerados devemos acessar uma das ferramentas de gerenciamento do Hadoop (<http://localhost:9870>) indo até o menu *Utilities* → *Logs*. O arquivo que contém os registros dos *logs* deve ser localizado por meio de seu ID, conforme apresentado na Figura 36.

Figura 36 – Tela de logs.



Outra forma de depurar um programa Hadoop é utilizando os *Counters*, por meio do objeto *Reporter*, disponível nos métodos *Map* e *Reduce*. Para o nosso exemplo iremos criar um tipo enumerado que denominaremos *Classificação*. Nesse tipo teremos três opções: *MAIOR_15000*, *MENOR_15000* e *TODOS*. A forma de

realizar o registro dos *counters* (contadores) é muito simples, pois apenas utilizamos o método *getCounter* do objeto *reporter* e incrementamos o seu valor, passando um dos itens do tipo enumerado que criamos. A Figura 37 demonstra a forma de realizar esse registro de depuração:

Figura – 37 – Utilização de *counters* para depuração.

```
static enum Classificacao { MAIOR_15000, MENOR_15000, TODOS }
public static class ReduceIGTI extends MapReduceBase implements Reducer<Text, Text, Text, Text> {
    public void reduce (Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        double media = 0.0;
        int acumuladorItens = 0, contaVendas = 0;
        Text value = new Text();

        while (values.hasNext()) {
            value = values.next();
            contaVendas++;
            acumuladorItens += Integer.parseInt(value.toString());
        }
        media = acumuladorItens / new Double(contaVendas);

        if (media > 15000)
            reporter.getCounter(Classificacao.MAIOR_15000).increment(1);
        else
            reporter.getCounter(Classificacao.MENOR_15000).increment(1);

        reporter.getCounter(Classificacao.TODOS).increment(1);

        value.set(String.valueOf(media));
        output.collect(key, value);
    }
}
```

Observe que, caso a média de vendas seja maior que 15000, incrementamos o *counter* com a identificação MAIOR_15000. Por outro lado, se a média de vendas for menor ou igual a 15000, incrementaremos o *counter* com a identificação MENOR_15000. Por último, iremos incrementar sempre o *counter* TODOS. A Figura 38 apresenta os resultados finais dos *counters* que foram criados. Esses resultados somente são apresentados para o usuário após o final da execução do *job*:

Figura 38 – Resultado dos *counters*.

```
Map-Reduce Framework
  Map input records=20
  Map output records=20
  Map output bytes=260
  Map output materialized bytes=312
  Input split bytes=226
  Combine input records=0
  Combine output records=0
  Reduce input groups=5
  Reduce shuffle bytes=312
  Reduce input records=20
  Reduce output records=5
  Spilled Records=40
  Shuffled Maps =2
  Failed Shuffles=0
  Merged Map outputs=2
  GC time elapsed (ms)=295
  CPU time spent (ms)=2940
  Physical memory (bytes) snapshot=651509760
  Virtual memory (bytes) snapshot=5749903360
  Total committed heap usage (bytes)=467140608
IGTI.ExemploIGTI$Classificacao
  MAIOR_15000=3
  MENOR_15000=2
  TODOS=5
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=3084
File Output Format Counters
  Bytes Written=71
```

Outras formas de depurar um *job* Hadoop é a utilização de *plugins*. Existem *plugins* disponíveis para Eclipse e Netbeans, além de outros *plugins* específicos para o Hadoop.

Algoritmos interativos com Hadoop

Um algoritmo iterativo consiste em uma sequência de instruções que são executadas passo a passo, algumas das quais repetidas em ciclos cuja execução recebe o nome de iteração e à repetição damos o nome de processo iterativo.

Cada iteração utiliza resultados das iterações anteriores e efetua determinados testes, chamados de critérios de parada. Esses testes permitem verificar se foi atingido um resultado próximo o suficiente do resultado esperado, ou seja, que atenda a precisão desejada.

Executa-se um conjunto de ações em forma completa, verifica-se a condição de saída e se é necessário volta-se a executar o conjunto de ações em forma completa. Uma importante classe de algoritmos iterativos são os denominados

Métodos Iterativos Estacionários de Passo Um. Nessa classe de métodos, sempre estarão presentes os seguintes elementos.

- Uma tentativa inicial para a solução do problema desejado.
- Uma equação de iteração, conforme a Equação 1 onde Φ é uma função a uma variável, denominada função de iteração, que varia de método para método.
- Um teste de parada, por onde se decide quando o método iterativo deve parar (um critério de convergência ou uma condição de parada).

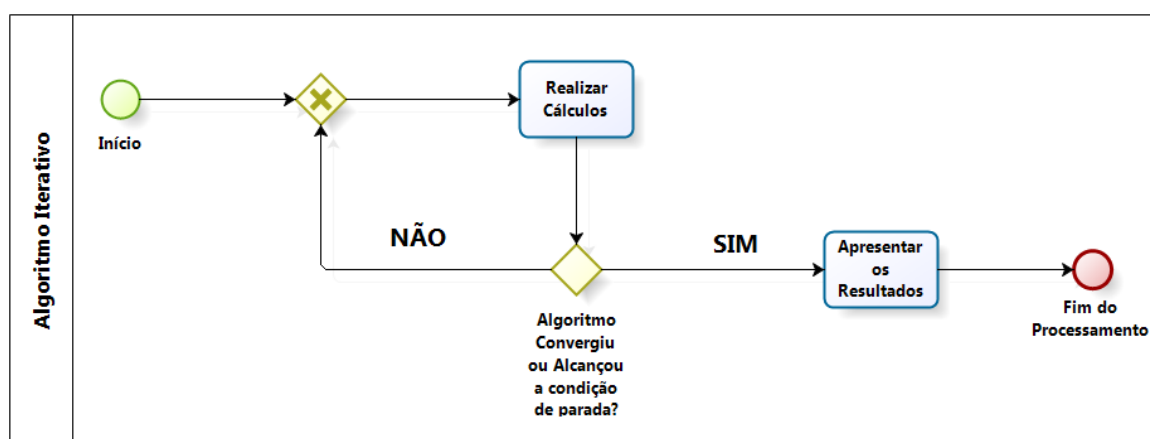
Para a equação de iteração, constrói-se uma sequência conforme apresentado na Equação 2.

$$x = \Phi(x) \quad (1)$$

$$x_1 = \Phi(x_0), x_2 = \Phi(x_1), x_3 = \Phi(x_2), \dots, x_{n+1} = \Phi(x_n), \dots, \quad (2)$$

Uma importante propriedade que deve estar presente em qualquer algoritmo iterativo é um critério de parada bem definido, para quando as iterações terminarem.

Figura 39 – Fluxo de um Algoritmo Iterativo.



A Figura 39 apresenta o funcionamento de um algoritmo iterativo de propósito geral, onde podemos observar que o algoritmo inicia, realiza seus cálculos e em seguida a condição de saída é verificada. Enquanto a condição de saída não for

alcançada, o algoritmo realizará suas iterações e quando essa condição é finalmente alcançada, o algoritmo apresenta os resultados e encerra.

O modelo MapReduce e o Apache Hadoop não são bons para processamento de algoritmos iterativos e isso se deve a duas razões principais a saber: (i) a sobrecarga *overhead* causada pela leitura e gravação de dados em disco (HDFS) e a troca de dados entre o disco e a memória RAM para cada uma das iterações; (ii) a falta de vida longa dos *jobs* no Hadoop. Tipicamente, existe uma checagem da condição de terminação que é executada no *job* MapReduce para determinar se o processamento está completo. Isso causa a inicialização de um novo *job* MapReduce que precisa ser inicializado para cada iteração. A sobrecarga (*overhead*) causada por essa inicialização pode causar uma significativa perda de desempenho.

O MapReduce/Hadoop não possui em sua versão tradicional a capacidade de manter os dados em memória durante a transição de iterações do algoritmo e isso dificulta o reuso dos dados. Devido a isso, o modelo lê os mesmos dados iterativamente e materializa os resultados intermediários nos discos locais a cada iteração, necessitando de uma grande quantidade de acesso ao disco, I/O e processamento desnecessário. Devido a essa deficiência que o modelo apresenta, surgem inúmeras oportunidades de trabalhos que possam propor melhorias e formas alternativas de utilizar o MapReduce/Hadoop.

Capítulo 5. Parâmetros de Desempenho

Ferramentas para construção de aplicações que manipulam dados em larga escala, como o Hadoop, têm ganhado grande popularidade. Entender todos os recursos de desempenho oferecidos por elas é uma tarefa útil, porém pode ser um tanto difícil devido às diversas combinações que podem ser elaboradas alterando seus parâmetros de ajuste de desempenho. Neste tópico apresentamos e discutimos alguns dos parâmetros configuráveis do Hadoop, com ênfase para os que mais impactaram o desempenho das aplicações.

Principais parâmetros de desempenho

O Hadoop, em sua versão 2, possui mais de 200 parâmetros ajustáveis e que podem alterar o desempenho de um determinado trabalho. Porém, ajustar esses parâmetros em um cluster é uma atividade bem diferente de ajustá-los em um nó individual e esse trabalho deve ser planejado adequadamente. Um mesmo parâmetro pode apresentar ganho ou perda de desempenho, quando aplicado a dois conjuntos de dados diferentes. Existem parâmetros que precisam ser observados em separado ao ajustar o cluster inteiro. Dentre esses parâmetros estão o número de *jobs* em execução, o volume de dados a ser processado, a quantidade dos dados a ser replicada, o número de réplicas desses dados, o tamanho do *buffer* para ordenação dos dados e o controle de utilização dos recursos disponíveis no cluster Hadoop.

A seguir estão listados os principais parâmetros da ferramenta Hadoop com uma breve descrição sobre cada um deles, assim como a apresentação de seus valores padrão.

mapred.tasktracker.map/reduce.tasks.maximum - Esse parâmetro define a quantidade máxima de tarefas Map e de tarefas Reduce que executarão simultaneamente em um nó escravo (TaskTracker). O Hadoop considera o valor de duas tarefas para a função Map e duas tarefas para a função Reduce como valores default.

mapred.child.java.opts - Além das otimizações normais do código Java, a quantidade de memória atribuída à JVM, para cada tarefa, também afeta o desempenho. O parâmetro ***mapred.child.java.opts*** pode ser utilizado para alterar várias características da JVM e, no caso da memória, o valor a ser atribuído é - *XmxYYYm*, onde YYY corresponde a quantidade de memória RAM que se deseja atribuir à JVM e os caracteres *-Xmx* significam que estamos querendo alterar o tamanho do Heap Size da JVM. O caractere *m*, ao final, significa que o valor atribuído ao parâmetro está em Megabytes. Por padrão o Hadoop já considera o valor de 200 Megabytes.

io.sort.mb - Esse parâmetro de configuração é utilizado para definir o tamanho do *buffer* de memória usado pelas tarefas Map para ordenar seus dados de saída e que serão enviados à função Reduce. O valor padrão para essa propriedade é de 100 Megabytes.

io.sort.spill.percent - Esse parâmetro deverá ser utilizado em conjunto com o parâmetro *io.sort.mb* pois ele define o momento em que os dados que estão armazenados no *buffer* deverão ser descarregados em disco. Essa tarefa é executada por meio de uma *thread* que é disparada no momento que o *buffer* atinge o valor definido no parâmetro. Esse valor é descrito em formato de percentual e seu valor padrão é 0.80 ou 80 por cento.

mapred.local.dir - Quando os dados do *buffer* atingem o valor definido em *io.sort.spill.percent* eles precisam ser descarregados em disco e o caminho onde esses dados serão descarregados pelo nó (*TaskTracker*) está definido nesse parâmetro. Esse valor pode ser apenas um endereço de diretório ou uma lista de endereços separados por vírgula.

io.sort.factor - Antes de uma determinada tarefa Map terminar os arquivos referentes a essa tarefa precisam ser mesclados em um único arquivo ordenado de saída. Esse parâmetro controla o número máximo de *streams* que será utilizado por vez para mesclar os dados de diversos arquivos em apenas um único arquivo. O valor padrão é 10.

mapred.reduce.parallel.copies - Os dados de saída de uma função Map são copiados para as funções Reduce no momento em que uma tarefa Map termina sua execução. As tarefas Reduce têm uma quantidade definida de threads que tem o objetivo de buscar os dados em paralelo. Essa quantidade de *threads* é definida no parâmetro ***mapred.reduce.parallel.copies*** e tem o valor padrão de 5 *threads*.

Influência dos parâmetros de desempenho

Como apresentado no tópico anterior, os parâmetros de desempenho têm papel fundamental no tempo de processamento e na escalabilidade dos sistemas paralelos e/ou distribuídos. Alguns estudos apontam até 35% de melhora no tempo de processamento e também melhora considerável nos resultados de *Speedup* e Eficiência, chegando em alguns casos a apresentar *Speedup* Linear e Superlinear. Os conceitos de *Speedup* e eficiência serão apresentados a frente.

Medidas de desempenho em *cluster* (escalabilidade e eficiência)

A escalabilidade de um algoritmo paralelo, em uma arquitetura paralela, é a medida da capacidade da utilização efetiva dos recursos pelo algoritmo, ao aumentar o número de processadores. A análise da escalabilidade de uma combinação algoritmo-arquitetura pode ser usada para uma variedade de propósitos como, por exemplo, selecionar a melhor combinação entre o algoritmo e a arquitetura para a solução de um determinado problema que tenha várias restrições quanto ao aumento do tamanho desse problema e do número de processadores. A escalabilidade pode ser usada também para prever o desempenho de um algoritmo paralelo e uma arquitetura paralela para um grande número de processadores, por meio do desempenho conhecido em um ambiente com poucos processadores. Uma baixa escalabilidade pode resultar em um sistema com baixo desempenho, necessitando de uma reengenharia ou mesmo da duplicação dos sistemas.

Uma medida que nos diz o quão mais rápido uma tarefa irá executar usando um computador (*cluster*) com alguma melhoria em relação ao computador (*cluster*)

original é o *Speedup* (Figura 40). Suponha que uma melhoria seja a possibilidade de executar em paralelo. Uma formulação da lei de Amdahl é demonstrada abaixo, onde *pctPar* é a porcentagem de tempo de execução que irá executar em paralelo e *p* é o número de núcleos nos quais a aplicação irá executar em paralelo:

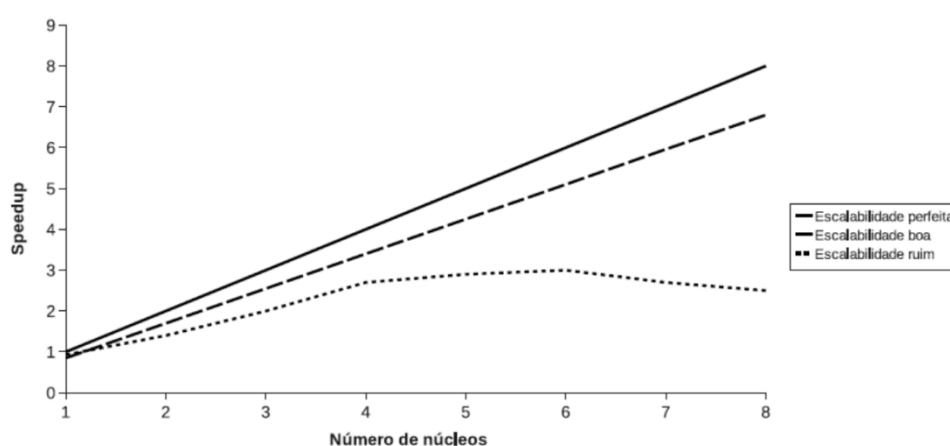
Figura 40 – Fórmula do Speedup.

$$S(p) = \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}$$

Fonte: Nascimento (2011).

Observamos que o speedup é proporcional à fração de tempo em que a melhoria pode ser utilizada e ao número de processadores que executarão essa melhoria. A Figura 41 apresenta exemplos de curvas de speedup. O speedup perfeito ocorre quando o speedup calculado é igual ao número de núcleos (linha sólida). Para uma aplicação ter uma boa escalabilidade, o speedup deveria aumentar próximo ou na mesma proporção que novos núcleos são adicionados (linha tracejada), ou seja, se o número de núcleos é duplicado, o speedup também deve aumentar na mesma proporção. Se o speedup de uma aplicação não acompanha bem essa proporção (linha pontilhada), dizemos que a aplicação não escala bem.

Figura 41 – Exemplos de curvas de Speedup.



Fonte: Nascimento (2011).

Outra métrica relacionada ao speedup é a eficiência. Enquanto o speedup nos dá uma métrica para determinar quão mais rápido é uma aplicação paralela em comparação com sua implementação sequencial, a eficiência nos diz quão bem os recursos computacionais estão sendo utilizados. Para calcular a eficiência de uma execução paralela, deve-se dividir o speedup pelo número de núcleos utilizado. Esse número é expresso em forma de porcentagem. Por exemplo se temos um speedup de valor 48 executando em uma máquina com 64 núcleos, a eficiência alcançada é 75 por cento, ou seja, $48/64 = 0.75$. Isso significa que, em média, durante o curso da execução, cada um dos núcleos está inativo durante 25 por cento do tempo. O cálculo da eficiência é apresentado em seguida na Figura 42:

Figura 42 – Cálculo da Eficiência.

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$$

Capítulo 6. Um pouco mais do ecossistema Hadoop

Esse capítulo tem o objetivo de apresentar outros componentes que fazem parte do ecossistema Apache Hadoop, tais como Hive, HBase e Sqoop.

Hive

O Apache Hive surgiu a partir da necessidade do Facebook de gerenciar e manipular suas grandes massas de dados, que eram armazenadas todas as noites em Sistemas Gerenciadores de Banco de Dados (SGBD's). Sobre esses dados, que cresciam a uma grande velocidade, era preciso realizar operações de ETL (Extração, Transformação e Carga) frequentemente. Existia a necessidade de gerenciar esses dados eficientemente.

O Hive foi a solução encontrada para esse caso e se trata de uma ferramenta de Data Warehouse. O grande segredo por trás do Hive é a sua capacidade de compilar e transformar consultas SQL em *jobs* MapReduce e, em seguida, executá-los em *cluster*.

Os dados do Hive são armazenados no HDFS e sua arquitetura é dividida em:

- *ClientAPI*: executa sentenças HiveQL.
- *Metastore*: armazena o catálogo completo do sistema.
- API (JDBC e ODBC): responsável por fornecer conectividade ao Hive.
- Aplicação de linha de comando: um shell permite manipular as tabelas Hive.
- Web Interface: permite gerenciar o sistema por meio de uma interface gráfica.

O modelo de dados do Hive possui tabelas com seus respectivos tipos (inteiro, ponto flutuante, cadeia de caracteres, tipo data e booleano). Além disso,

existem as Partições (Partitions) que são responsáveis por armazenar fragmentos das tabelas.

O Apache Hive não realiza nenhum tipo de transformação nos dados ao carregar uma tabela, ou seja, a operação de carga é puramente uma operação de cópia que move arquivos de dados para as tabelas Hive.

O Hive é um banco de dados para análise e, portanto, não tem a indicação para ser um banco de dados para aplicações de tempo real.

HBase

O Apache HBase é um banco de dados não-relacional, distribuído e *open-source*, que foi desenvolvido em Java a partir do Google BigTable. O HBase é orientado a colunas, executa sobre o HDFS e tem a capacidade de armazenamento de grandes volumes de dados esparsos. Surgiu em 2007 e, durante esses anos, diversas versões foram lançadas, tornando o HBase um projeto Top-Apache em 2010.

Da perspectiva do usuário, o HBase se assemelha a um banco de dados ou uma planilha tradicional, com linhas e colunas armazenando valores, entretanto o HBase pode ter um *schema* diferente por linha, chamado *schema-less*. Nesse caso o acesso é feito pela chave da linha e pelo nome da coluna, conforme apresentado na Figura 43.

Figura 43 – Schema do HBase.

| | col-A | col-B | col-Foo | col-XYZ | foobar |
|--------|--------------|--------------|--------------|--------------|--------------|
| row-1 | | | | | |
| | col-A | col-D | col-Foo2 | col-XYZ | col-XYZ2 |
| row-10 | | | | | |
| | 20130423 | 20130424 | 20130425 | 20130426 | 20130427 |
| row-18 | | | | | |
| | MaxVal - ts5 | MaxVal - ts4 | MaxVal - ts3 | MaxVal - ts2 | MaxVal - ts1 |
| row-2 | | | | | |

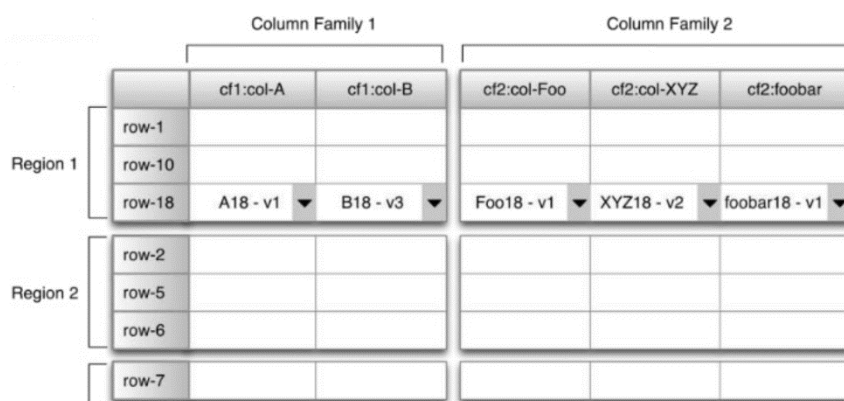
Cada conjunto de linha e coluna é marcado com um número de versão, portanto o HBase permite valores multi-versionados, onde a API da ferramenta permite ao usuário pesquisar os dados pelas versões. A Figura 44 apresenta os valores multi-versionados.

Figura 44 – Valores multi-versionados do HBase.

| | col-A | col-B | col-Foo | col-XYZ | foobar |
|--------|------------|------------------------|--------------|--------------|-----------------|
| row-1 | | | | | |
| row-10 | | | | | |
| row-18 | A18 - v1 ▼ | B18 - v3 ▼ | Foo18 - v1 ▼ | XYZ18 - v2 ▼ | foobar18 - v1 ▼ |
| row-2 | | Peter - v2 Bob - v1 | | Mary - v1 | |
| row-5 | | | | | |
| row-6 | | | | | |
| row-7 | | | | | |

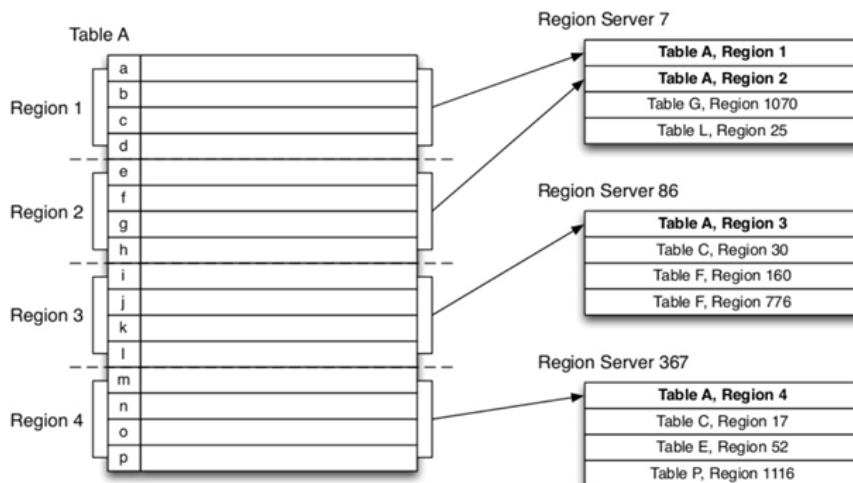
O HBase possui o conceito de Regiões (*regions*), onde os dados de uma tabela são particionados e distribuídos ao longo do *cluster*. As Regiões dividem a tabela em fragmentos de tamanho fixo e as Famílias são conjuntos de colunas dentro das Regiões. A Figura 45 exemplifica os conceitos de Regiões e Famílias do HBase.

Figura 45 – Regiões e Famílias no HBase.



Uma tabela do HBase pode ser formada por um conjunto de Regiões, sendo estas a unidade de trabalho do HBase. Além disso, existe o conceito de Region Server. Uma Região é servida exatamente por uma Region Server. Uma Region Server pode atender muitas Regiões. A Figura 46 apresenta o conceito de Region Server.

Figura 46 – Region Server do HBase.



A utilização do HBase é mais indicada para conjuntos de dados grandes, esparsos, com registros sem normalização e com muitos acessos (clientes) concorrentes. A tecnologia deve ser evitada, obviamente, para pequenos conjuntos de dados e para registros que são altamente relacionados entre si, devido a distribuição que é feita entre as Regiões.

O HBase fornece acesso por meio do HBase Shell e pela sua própria API Java e REST. Além disso, a ferramenta possui integração com outros componentes do ecossistema Hadoop, tais como Apache Spark, Pig e Hive.

Sqoop

O Apache Sqoop é uma ferramenta desenvolvida a partir da necessidade de realização e transferência de dados entre um *cluster* Hadoop e bancos de dados de armazenamento estruturado e relacionais, tais como: Oracle, DB2, MSSql, MySQL, etc. O projeto Sqoop foi iniciado em 2009 pela Cloudera e virou um projeto Top da Apache em 2012.

O Sqoop realiza a transferência das grandes massas de dados a partir da criação de *jobs* MapReduce, tanto do HDFS para os bancos de dados relacionais, quanto fazendo o caminho contrário.

Além de realizar a transferência das grandes quantidades de dados, o Sqoop permite realizar o processo de transformação desses dados, armazenando o resultado no HDFS ou em tabelas Hive ou HBase. O processo de transformação pode realizar a conversão de tipos de campos, a categorização de valores ou a inserção de valores faltantes em campos. O Sqoop aceita a conexão com diversas tecnologias de bancos de dados, via JDBC.

Por utilizar jobs MapReduce para transferir e transformar os dados, é garantido o processamento paralelo, distribuído e tolerante a falhas.

O Sqoop pode ser utilizado para realizar importações no modo incremental, ou seja, buscando/recuperando apenas novas linhas para a importação. Para isso são fornecidos dois tipos de importação incremental:

- **Append:** usado para importar novas linhas. Para cada tabela especifica-se uma coluna contendo um identificador (*check-column*). Diante disto, o Sqoop importará apenas linhas onde o *check-column* tiver um valor maior

que o valor especificado no campo *last value*. Nesse caso, o valor de *last value* é atribuído ao final da última importação realizada.

- **Last Modified:** pode ser usado quando as linhas da tabela fonte precisam ser atualizadas pela tabela destino. Para isso, o Sqoop atualiza um campo do tipo *timestamp*, ou seja, as tabelas destino que possuírem um valor para esse campo mais antigo que na tabela origem, terão seus valores atualizados.

Referências

HAMSTRA M. et al. *Learning Spark: Lightning-Fast Big Data Analysis*. 1. ed. O'Reilly Media, Inc. 2015

NASCIMENTO, J. P. B; MURTA, C. D. *Um Algoritmo Paralelo em Hadoop para Cálculo de Centralidade em Grafos Grandes*. SBRC XXX Simpósio Brasileiro de Redes de Computadores. 2012. Ouro Preto. Brasil.

WHITE, Tom. *Hadoop: The Definitive Guide*. 1. ed. O'Reilly Media, Inc. 2008.