Processamento de dados utilizando o ecossistema Hadoop

Silas Yunghwa Liu

2023

SUMÁRIO

> Capítulo 1

Capítulo 1. Introdução ao ecossistema Hadoop

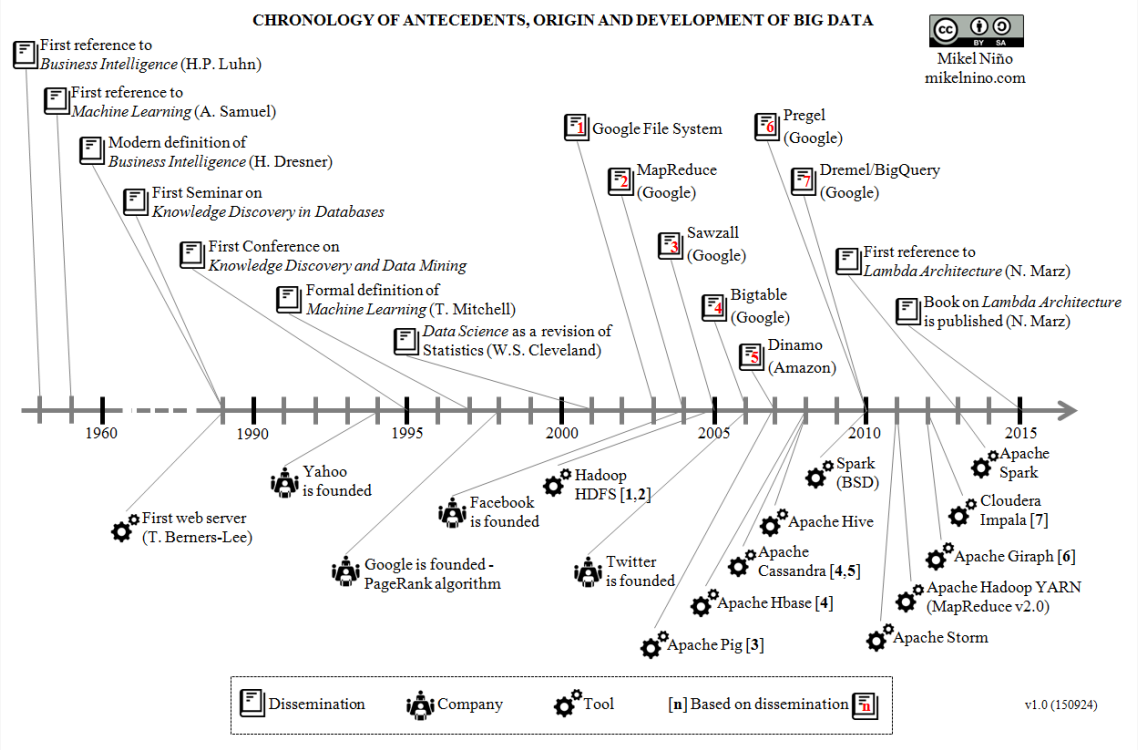
O Hadoop é um ecossistema, de código aberto, projetado para leitura, armazenamento e processamento de dados de forma paralela e distribuída. Devido à sua arquitetura, com o passar dos anos ela se tornou ideal para manipulação de grandes volumes de dados, ou big data. Atualmente existem dezenas de bibliotecas e ferramentas auxiliares, construídas para o Hadoop, com as mais diversas aplicações e utilidades, dentre as quais iremos estudar alguns mais utilizados.

Para entendermos melhor o Hadoop e os frameworks, vamos primeiro observar a linha do tempo na Figura 1, em que mostra a evolução do big data e marca datas importantes do desenvolvimento de sistemas, ferramentas e companhias. Pode-se observar que os grandes feitos se concentram a partir de 1990 e que houve um grande avanço recentemente.

O Hadoop em si nasceu nos meados de 2005 e desde então tem surgido dezenas de ferramentas e frameworks dedicados a ele. O Hadoop é implementado em Java, mas oferece suporte a diversas outras linguagens, tais como Python e SQL. Sua aplicação em uma gama de áreas se deve ao fato de ser um sistema confiável, escalável e distribuído, além de ser tolerante a falhas. Mais adiante voltaremos a esses tópicos e explicaremos o significado de cada um.

Atualmente existem diversas distribuições Hadoop, que oferecem suporte e melhorias. A distribuição gratuita de código aberto e mais utilizada é o Apache Hadoop. Entretanto, existem diversas distribuições comerciais tais como o Cloudera Hadoop, Hortonworks Hadoop, MapR Hadoop, AWS Elastic MapReduce (EMR), Microsoft Azure HDInsight e Google Cloud Dataproc.

Figura 1 – Linha do tempo do big data.



Fonte: https://www.mikelnino.com/2016/03/chronology-big data.html?m=1.

1.1. Introdução ao Hadoop

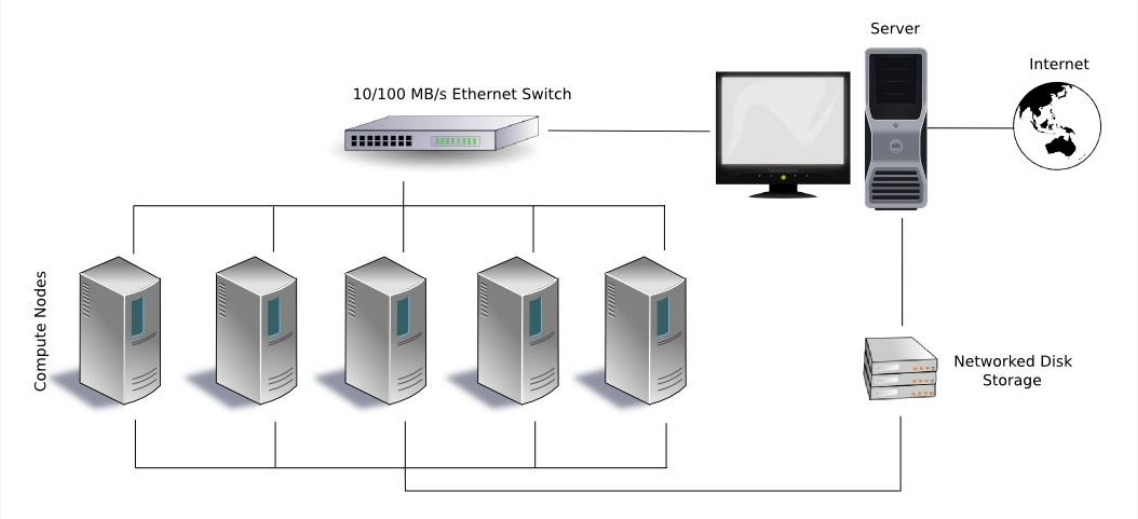
Uma das características do Hadoop é ser escalável. Ele apresenta uma escalabilidade horizontal, o que é muito atrativo atualmente. Sistemas antigos eram escaláveis verticalmente, o que significa que, para melhorar uma máquina, precisavam ser melhorados em hardware, ou seja, terem suas peças trocadas, para oferecer mais memória, mais processamento ou HD. Escalar verticalmente custa caro, pois é necessário comprar mais recursos e peças, e quando não estão sendo usados se tornam ociosos e representam gastos desnecessários.

Escalabilidade horizontal é o desejável nos dias atuais, e muito empregado em ambientes nuvem. A escalabilidade horizontal se caracteriza em disponibilizar máquinas em paralelo, perante a necessidade. Ao se precisar de mais recursos, tais como memória ou processamento, pode-se iniciar e emparelhar uma série de máquinas, usá-las simultaneamente e,

após seu uso, desligá-las do sistema. Essas máquinas são disponibilizadas por plataformas nuvem e não é necessário comprá-las, pagando apenas pelo seu uso.

O Hadoop, portanto, permite a escalabilidade horizontal, dependendo da atividade e dos dados. E isso é alcançado por ser um sistema distribuído, ou seja, através do uso de clusterização. Clusters são agrupamentos de computadores, que trabalham em conjunto para uma operação. Eles geralmente são gerenciados por um único computador (servidor) e provêm armazenamento, processamento e intercâmbio de recursos.

Figura 2 – Cluster de computadores.



Fonte: https://medium.com/lvs-load-balance-clustering-configuration-on/what-is-a-computer-cluster-c4c219f4f6d9.

Em ambientes de clusterização, utilizamos os seguintes conceitos:

- Nodo ou nó: é o nome dado a um computador individual, dentro de um cluster. Existe o nó master (ou driver), que gerencia e distribui o trabalho entre os demais nós, que são chamados de nós workers (ou slaves).

- Daemon: termo que se dá ao programa, ou serviço (ou job), que é executado em um nó. Esses daemons podem ser dos mais diversos como gerenciamento de recursos ou análise de dados.

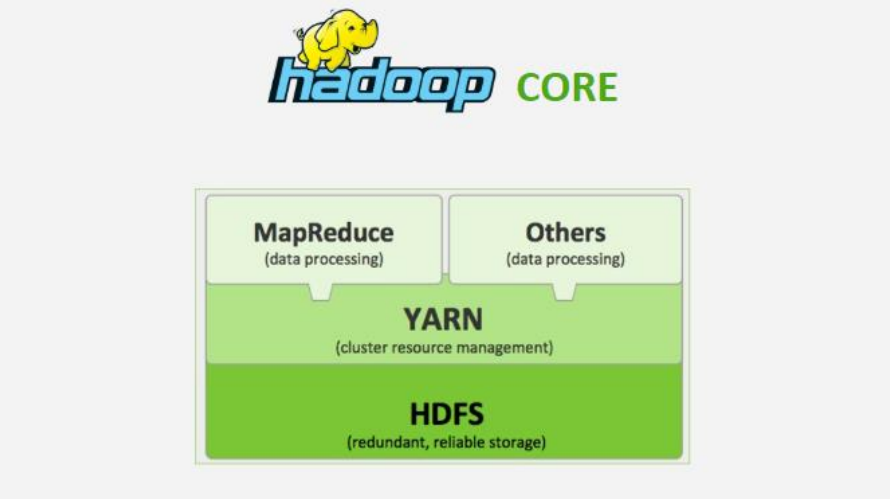
O Hadoop ainda se caracteriza como confiável, ou seja, tolerante a falhas, ao empregar replicação dos dados em nodos diferentes. Isso garante o acesso aos dados e ao processamento mesmo que se perca a comunicação ou disponibilidade de alguns computadores dentro do cluster.

Nos dias atuais, existem dezenas de frameworks para Hadoop, com as aplicações das mais variadas. Há específicos para bancos de dados, processamento, gerenciamento de clusters, segurança, mensageria, fluxo de dados etc. Alguns estão apresentados na Figura 3.

Figura 3 – Frameworks para Hadoop.

É importante conhecer os blocos principais que constituem o Hadoop. Ele é constituído pelo HDFS, seu sistema de armazenamento de dados; o YARN, o gerenciador de recursos; e o MapReduce, o método de processamento de dados.

Figura 4 – Hadoop Core.



Fonte: https://medium.com/dataengineerbr/um-quase-guia-nada-completo-sobre-o-hadoop-a3eeee170deb.

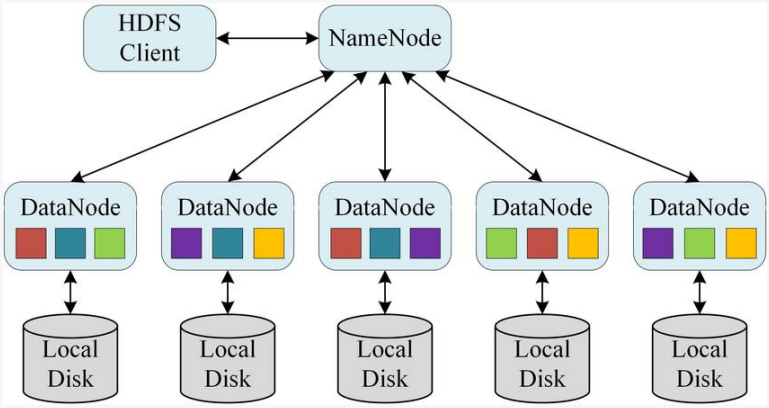
1.2. Introdução ao HDFS

HDFS é a sigla para Hadoop Distributed File System, e é o sistema responsável pelo armazenamento dos dados, no Hadoop. Este sistema foi desenvolvido baseado no GFS (Google File System).

O HDFS se caracteriza como um sistema de armazenamento escalável e tolerante a falhas, conforme já vimos a respeito do Hadoop. Ele aceita tanto dados tabulares como NoSQL. Para garantir a escalabilidade horizontal e clusterização, o sistema funciona baseado em dois tipos de nós, o NameNode e o DataNode:

- NameNode: o nodo que gerencia o Namespace e é responsável por armazenar o metadado dos demais nodos.
- Secondary NameNode: funciona apenas como fator de segurança, realiza pontos de verificação e backup do NameNode.
- DataNode: todos os demais nodos, responsáveis por armazenar os dados.

Figura 5 – Estrutura HDFS.

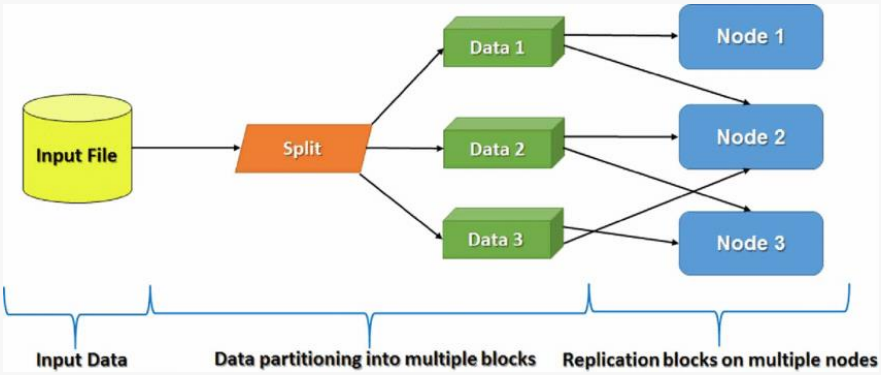


Fonte: https://www.researchgate.net/figure/The-overview-of-the-Hadoop-Distributed-File-System-HDFS_fig4_348387085.

O HDFS armazena um dado ou separa em blocos. O tamanho padrão desses blocos é de 128 MB. Cada bloco é então replicado pelo fator de replicação, que tem o padrão de 3, e cada DataNode armazena uma cópia de cada bloco, enquanto o NameNode armazena qual DataNode possui quais blocos.

No caso de leitura de um dado o HDFS acessa o NameNode e a partir dele consegue recuperar cada bloco, e a partir dos DataNode que possuem os blocos requisitados reconstrói o arquivo original. Qualquer DataNode que possua uma cópia de cada bloco pode ser utilizado, os demais funcionam apenas para redundância e garantem a confiabilidade e tolerância a falhas.

Figura 6 – Processo do HDFS (ex. replicação 2).



Fonte: https://www.researchgate.net/figure/Data-replication-using-HDFS_fig3_325608248.

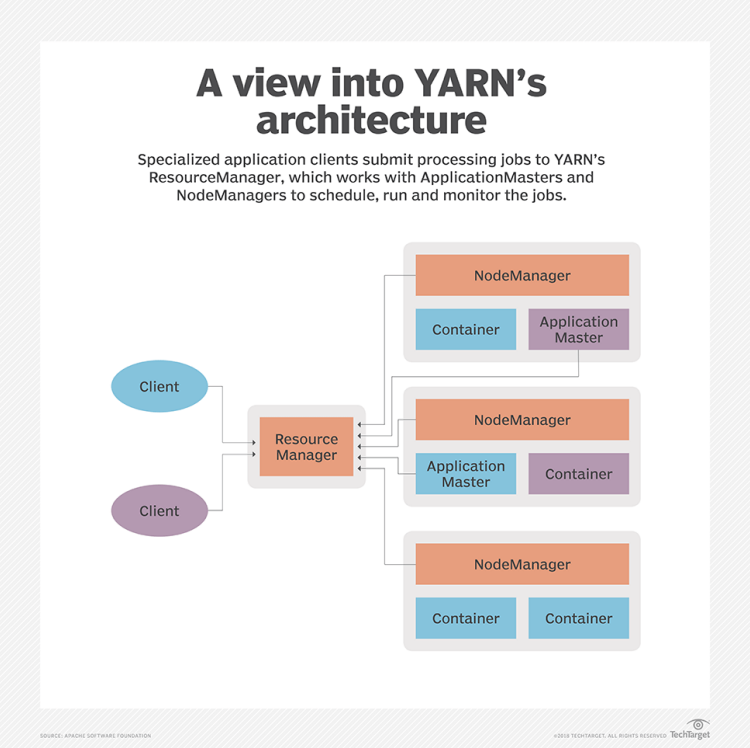
1.3. Introdução ao YARN

YARN é a sigla para Yet Another Resource Negotiator, foi desenvolvido a partir da versão 2.0 do Hadoop e é responsável pelo gerenciamento dos recursos dos nodos. Ele é otimizado para o processamento paralelo e provê os recursos necessários, mediante a requisição de cada job. Isso otimiza a utilização de processamento e memória em múltiplos nodos, dentro do cluster, atribuindo os nodos mais recomendados para cada atividade.

Na estrutura do YARN temos os seguintes conceitos:

- Resource Manager: é o gerenciador global dos jobs e recursos.
- Node Manager: é inicializado em cada nó, monitora os recursos do container e reporta ao Resource Manager.
- Application Master: é inicializado um para cada job, gerenciando a atividade e requisitando recursos pelo Node Manager.
- Container: unidade de alocação de recursos (memória, processamento), controlado pelo Node Manager.

Figura 7 – Estrutura do YARN.



Fonte: https://www.techtarget.com/searchdatamanagement/definition/Apache-Hadoop-YARN-Yet-Another-Resource-Negotiator.

No exemplo da Figura 7 temos 3 computadores (nodos), cada um com seu Node Manager monitorando os respectivos containers. Temos 2 tarefas sendo realizadas (2 Application Master), e todos reportando a um Resource Manager.

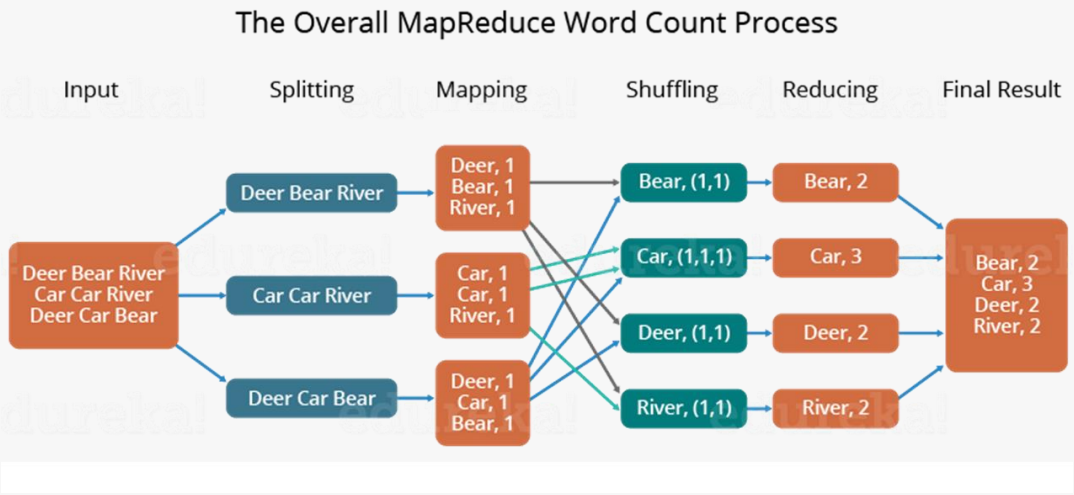
O YARN abstrai esse gerenciamento e requisição de recursos. Temos o papel apenas de configurá-lo durante a instalação do Hadoop. Geralmente mantemos as configurações padrões, mas é possível modificar quaisquer parâmetros. Exploraremos mais adiante, quando apresentarmos os códigos.

1.4. Introdução ao MapReduce

MapReduce é o modelo de programação desenvolvido para processar os dados distribuídos no HDFS. Desta forma, ele também deve ser capaz de operar de forma paralela e distribuída. É projetado para funcionar independentemente da quantidade de nodos dentro de um cluster. Em

comparação à arquitetura convencional, o MapReduce apresenta um ganho considerável na velocidade e tempo de processamento de big data.

Figura 8 – Exemplo de processo MapReduce.



Fonte: https://dkharazi.github.io/notes/de/etl/mapreduce.

Vamos entender como o processo do MapReduce ocorre. Ele se resume a um pipeline de vários processos, em que a saída de um é a entrada da próxima etapa:

1. Splitting (separação): os dados são divididos entre os nodos, conforme a distribuição do HDFS.

2. Mapping (mapeamento): cada nodo faz uma tarefa individual, apenas nos dados que possui.

3. Shuffling (ordenação): os resultados do Mapping de todos os nodos são reunidos e ordenados, em ordem alfabética (ou crescente se forem números).

4. Reducing (redução): com as chaves já ordenadas, realiza-se uma tarefa para extrair os resultados de cada chave.

Deve ficar claro que é dever do desenvolvedor escrever o script das etapas de Mapping e Reducing. Dependendo da operação que se quer

realizar, os scripts podem mudar, mas os processos mais comuns já possuem scripts prontos na internet.

A exemplo, como mostrado na Figura 8, temos o MapReduce de um contador de palavras. Para essa atividade, na etapa de Mapping, imprimimos cada palavra como chave e o número 1 (indicando uma ocorrência para cada palavra). A seguir as palavras (chaves) são ordenadas em ordem alfabética e por fim, na etapa de Reducing, a quantidade de repetições de cada palavra é apresentada como o resultado final.

Pode-se observar que essa lógica permite que o MapReduce seja aplicado a qualquer número de nodos em um cluster, e também independe de como os dados estejam distribuídos. Cada máquina realiza o Mapping de forma independente e o Shuffling tem o papel de reunir a operação de todos os nodos e ordená-los, para, por fim, obter o resultado coerente com o Reducing. Enquanto o Mapping e Reducing precisam ser disponibilizados ou implementados, o Shuffling é realizado de forma automática entre as duas etapas, e sempre ordena os dados de saída do Mapping, alimentando o Reducing.

Abordaremos novamente os scripts de Mapping e Reducing com mais calma adiante, quando tratarmos dos códigos e sua implementação.

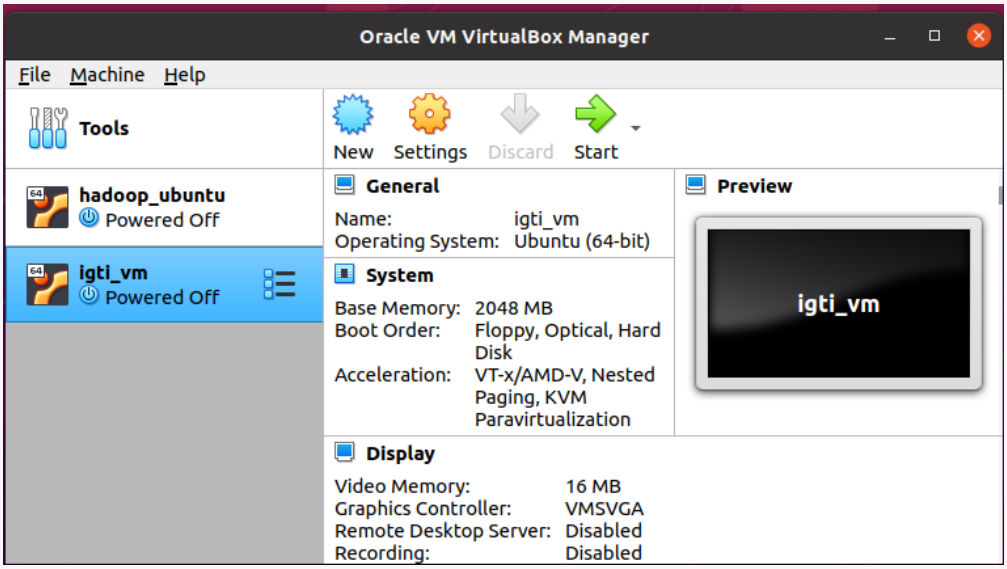
> Capítulo 2

Capítulo 2. Apresentação do Ambiente

Antes de avançarmos e lidarmos com o Hadoop, precisamos apresentar a Máquina Virtual e o ambiente Linux. Estes serão utilizados em nossas aulas futuras, quando analisarmos na prática.

2.1. Máquina Virtual

Figura 9 – Interface do Oracle VirtualBox.



Máquina Virtual (ou Virtual Machine) é uma ferramenta para simular um sistema operacional dentro do computador local. Ele é bastante útil em uma série de aplicações, como teste de sistemas operacionais ou teste de aplicativos e implementações em sistemas diferentes. Ele oferece segurança, pois pode-se isolar totalmente o sistema operacional virtual da máquina externa. Além disso, é possível de se simular praticamente qualquer ambiente Linux, Windows, MAC ou mobile, independente do sistema operacional original.

Como principais softwares de máquina virtual temos o VMWare Workstation, o Oracle VM Virtualbox e o Microsoft Hyper-V. Durante nossas aulas utilizaremos o Oracle VM Virtualbox, por ser gratuito e muito utilizado, mas qualquer um é suficiente. Optamos pelo uso de um sistema operacional

em máquina virtual, pela capacidade de isolar um sistema e podermos testar, manipular arquivos de configuração, instalar e desinstalar bibliotecas, sem o risco de impactar a máquina real. Iremos trabalhar em ambiente Linux, uma vez que alguns frameworks para Hadoop são implementados e otimizados para o Linux, além de termos mais liberdade para configurações. Abordaremos mais o Linux na próxima sessão.

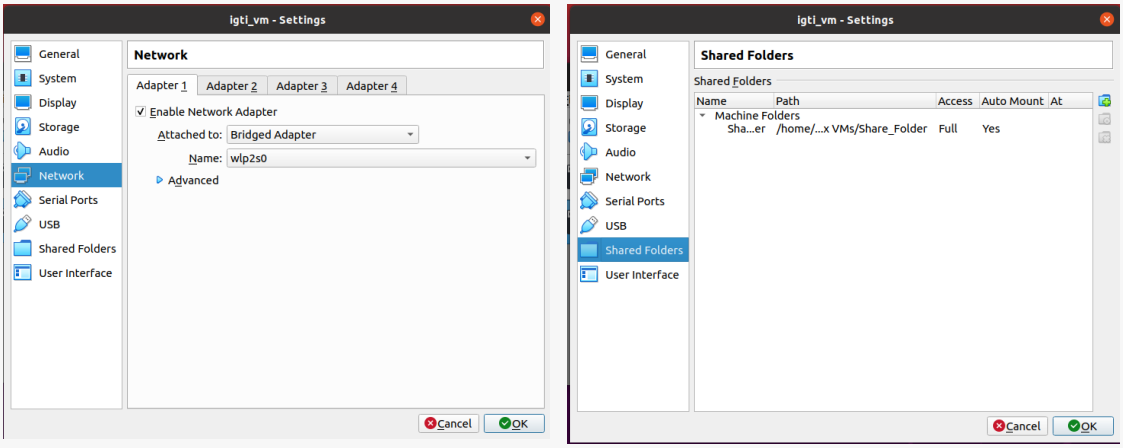
Para criar uma máquina, fazemos isso pelo botão NEW. Deve-se primeiro configurar o hardware da máquina, como se estivesse montando um PC novo, mas limitado pelas configurações de hardware da máquina real. Recomenda-se aplicar metade da memória RAM da máquina e pelo menos 50 GB de HD. Não aloque toda a memória RAM do PC físico para o virtual, uma vez que o físico irá precisar de memória para rodar a máquina virtual, além de outros processos.

É importante dizer que para se instalar um sistema operacional usando a máquina virtual é necessário ter o arquivo .iso daquele sistema operacional. Quanto à distribuição Linux a se instalar, existe uma gama variada de distros e pode-se escolher a maioria deles. Os mais comuns são Ubuntu, Linux Mint ou versões menores como Lubuntu e Linux Lite. Após criar a máquina e configurar seu hardware, rodamos a máquina pelo botão START. Na 1ª vez é necessário instalar o sistema operacional, selecionando onde a imagem .iso que está localizada na máquina. A partir daí, ao inicializar a máquina ele já irá carregar como em um PC comum.

Ao se instalar um sistema operacional, antes de começarmos a trabalhar com ele é sugerido que se realize duas configurações, conforme ilustra a Figura 10. Clicando no botão Settings, na aba Network, configuramos a placa de rede do pc virtual e devemos selecionar ‘Bridged Adapter’ ou ‘Adaptador em Ponte’. Isso possibilita comunicarmos com a máquina virtual a partir da nossa máquina local. A segunda configuração a se fazer é acessando a aba Shared Folders, onde adicionamos um diretório

para compartilhamento de arquivos. Esse diretório se localiza na nossa máquina local e é visível na máquina virtual também, como um HD externo.

Figura 10 – Configurações do Oracle VirtualBox.



2.2. Comandos CLI no Ambiente Linux

Como dito anteriormente, iremos realizar os estudos em ambiente Linux. Isso irá contribuir muito, uma vez que o Hadoop se utiliza de comandos de linha (CLI – Command Line Interface) do Linux. Para nos familiarizarmos com os comandos, iremos aprender aqui os mais usuais, com o uso do Terminal, da interface de CLI do Linux.

Os comandos por linha nos dão total liberdade para realizarmos as mesmas operações que conseguimos através de mouse e clique, como navegar pelos diretórios, ver conteúdo, abrir e editar arquivos, criar, copiar, mover, renomear pastas e arquivos, dentre outros. Mas mais que isso, pelos comandos de linha podemos automatizar e escalonar processos. Os comandos de linha de operações básicas estão apresentados:

$ cd	mudar de diretório (sem argumentos, para a pasta pessoal ~)
$ cd ~	mudar para a pasta pessoal ~
$ cd /	mudar para o diretório raiz /
$ cd dir	mudar para o diretório dir
$ cd ..	subir um diretório
$ ls	mostra o conteúdo do diretório atual
$ cp orig dest	copia a origem ao destino
$ mv orig dest	move origem ao destino

```
$ rm file          deleta o arquivo
$ rm -R dir        deleta recursivo o diretório e todo seu conteúdo
$ cat file         imprime o conteúdo do arquivo
$ mkdir dir        cria um diretório
$ wget url         download de arquivo do url no diretório atual
$ nano file        abre o arquivo no editor de textos nano
```

Na 1ª vez em que o sistema operacional for instalado, será preciso atualizar ele e suas bibliotecas, para permitir a instalação do Hadoop e outros frameworks. Isso pode ser feito com os comandos:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

No caso do Oracle VirtualBox, também será necessário instalar um pacote do VirtualBox para configurar corretamente a pasta compartilhada, para que fique visível como uma mídia externa. Isso é feito com os seguintes comandos:

```
$ sudo apt-get install virtualbox-guest-utils    instalar o guest utils do virtualbox
$ sudo adduser $USER vboxsf
$ sudo reboot
```

Após esses comandos, o sistema irá rebootar e a pasta compartilhada irá aparecer sempre no caminho /media/sf_{NOME_DO_DIR}. No caso dei o nome de Shared_Folder à pasta e ela aparecerá no caminho /media/sf_Shared_Folder.

> Capítulo 3

Capítulo 3. Hadoop na Prática

Vamos prosseguir agora com a instalação do Hadoop na nossa máquina virtual e a seguir, começar a trabalhar e fazer análises com ele.

3.1. Instalação do Hadoop

Para instalar o Hadoop bem como os futuros frameworks, buscamos pela versão que desejamos e fazemos o download do arquivo binário comprimido, com extensão .tar.gz. O arquivo binário já possui todos os arquivos necessários para execução, e não precisam ser instalados, necessitando apenas serem extraídos e configurados de maneira correta. O hadoop pode ser baixado de sua página: https://hadoop.apache.org .

Instalação do Java

Uma vez que o Hadoop é programado em Java, a máquina precisa ter o Java instalado. Os comandos a seguir verificam a versão do Java instalado e instalam ele, caso necessário. Existem diversas versões e distros do Java, a versão 8 é o suficiente para nós.

```
$ java -version                                  imprime a versão do java se instalado
$ sudo apt-get install openjdk-8-jdk-headless  instalação do java 8 openjdk
```

Após sua instalação é necessário setar as variáveis de ambiente com o diretório onde o java foi instalado (usamos aqui o diretório padrão da versão 8):

```
$ sudo nano /etc/environment
    JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64"
    JRE_HOME="/usr/lib/jvm/java-8-openjdk-amd64/jre"
```

Por fim, precisamos recarregar o arquivo e podemos testar:

```
$ source /etc/environment                recarrega as variáveis de ambiente
$ echo $JAVA_HOME                        imprime a variável JAVA_HOME
```

Extração do binário Hadoop

Após fazer o download do binário, vamos extraí-lo e renomear a pasta:

```
$  tar -xvf hadoop-3.3.3.tar.gz          extrair o binário do tar.gz
$  mv hadoop-3.3.3 hadoop                renomear o diretório para simplificar
```

O próximo passo é incluir os diretórios no arquivo .bashrc, que também armazena variáveis de ambiente. Dependendo da versão de Linux, o arquivo irá sugerir utilizar um arquivo secundário, exclusivo para variáveis, usando o nome bash_aliases.

```
$  nano .bash_aliases                    editar / criar o arquivo .bash_aliases
      export HADOOP_HOME=$HOME/hadoop
      export HADOOP_CONF_DIR=$HOME/hadoop/etc/hadoop
      export HADOOP_MAPRED_HOME=$HOME/hadoop
      export HADOOP_COMMON_HOME=$HOME/hadoop
      export HADOOP_HDFS_HOME=$HOME/hadoop
      export YARN_HOME=$HOME/hadoop
      export PATH=$PATH:$HOME/hadoop/bin
      export PATH=$PATH:$HOME/hadoop/sbin
      export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
      export PATH=$PATH:/usr/lib/jvm/java-8-openjdk-amd64/bin
```

Após a inserção, recarregamos o arquivo.

```
$  source .bash_aliases
```

Configurações do Hadoop

Após a instalação, precisamos configurar de maneira adequada os diversos arquivos de configuração do Hadoop, com extensão xml. Eles se encontram no diretório ~/hadoop/etc/hadoop. Também pode ser preciso configurar arquivos com extensão sh, de script. As propriedades dos arquivos xml seguem o padrão html com a estrutura apresentada abaixo. Uma tag configuration engloba todas as propriedades e cada propriedade deve repetir as tags property, name e value. As indentações não fazem diferença no arquivo xml mas os fazemos para melhorar a visibilidade.

```
<configuration>
```

```
  <property>
    <name>NOME_DA_PROPRIEDADE</name>
    <value>VALOR_DA_PROPRIEDADE</value>
  </property>
  <property>
    ....
  </property>
</configuration>
```

Portanto, vamos mudar para o diretório de configurações e editar os arquivos necessários:

```
$  cd ~/hadoop/etc/hadoop                    ir ao diretório de configurações
```

- core-site.xml: configurações de localização do NameNode e I/O.

```
$  nano core-site.xml
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

- hdfs-site.xml: configurações dos daemons HDFS (NameNode, Secondary NameNode, DataNode), fator de replicação. Substitua 'user' pelo seu nome de usuário.

```
$  nano hdfs-site.xml
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.permission</name>
    <value>false</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>file:///home/user/hadoop/hadoop_data/name</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>file:///home/user/hadoop/hadoop_data/data</value>
  </property>
</configuration>
```

- mapred-site.xml: configurações do MapReduce, número de JVMs em paralelo, tamanho dos processos mapper e reducer.

```
$ nano mapred-site.xml
<configuration>
  <property>
    <name>mapred.framework.name</name>
    <value>yarn</value>
  </property>
  <property>
    <name>yarn.app.mapreduce.am.env </name>
    <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
  </property>
  <property>
    <name>mapreduce.map.env</name>
    <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
  </property>
  <property>
    <name>mapreduce.reduce.env</name>
    <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
  </property>
</configuration>
```

- yarn-site.xml: configurações de Resource Manager e Node Manager.

```
$ nano yarn-site.xml
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-
services.mapreduce_shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
</configuration>
```

- hadoop-env.sh: configurações de variáveis de ambiente no script Hadoop.

```
$ nano hadoop-env.sh
        export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Conexão com acesso SSH

Ainda é necessário setar mais uma configuração antes de rodarmos nosso sistema Hadoop. Os nodos do cluster comunicam entre si através do

acesso SSH, sigla para Secure Shell. O SSH é um protocolo seguro e muito utilizado para realizar comunicação e acesso entre servidores e sistemas remotos, como em redes confiáveis ou serviços nuvem. Os serviços Hadoop funcionam como uma rede própria, onde cada nodo funciona em uma porta diferente, e é preciso setar o SSH para que eles comuniquem entre si.

Cada acesso utiliza uma chave SSH, então para o Hadoop devemos gerar uma chave SSH e cadastrá-la nos serviços de acesso e log de chaves autorizadas. Durante a criação da sua chave SSH tome cuidado para não sobrescrever outras já existentes, atribuindo um nome diferente para a chave a ser criada. Aqui vamos usar o nome padrão e não vamos colocar senha, para facilitar o acesso. Realize os seguintes comandos no terminal:

```
$ sudo apt-get remove openssh-client openssh-server      desinstalar primeiro o ssh
$ sudo apt-get install openssh-client openssh-server     reinstalar o ssh corretamente
$ ssh-keygen                                             criar chave ssh (definir nome,
  senha)
$ eval`ssh-agent`                                        ativar o ssh-agent
$ ssh-add $HOME/.ssh/id_rsa                              ou usar o nome da chave criada
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys        adicionar ao log
$ sudo service ssh restart                               recarregar as chaves ssh
$ ssh localhost                                          acessar com chave ssh e selecionar
  yes
```

Formatar o NameNode

Para o primeiro acesso, deve-se primeiro formatar o NameNode. Isso é realizado uma única vez, com o seguinte comando:

```
$ ~/hadoop/bin/hadoop namenode -format
```

Após formatar o NameNode, podemos finalmente inicializar os nodos e começar a trabalhar com o Hadoop.

3.2. Manipulando o HDFS

Para nossos estudos com o Hadoop, iremos utilizar um conjunto de dados de filmes e séries, retirado do Kaggle. O Kaggle é uma plataforma gratuita muito utilizada na comunidade de dados e contém uma quantidade

significativa de dados, análises e até competições. Os dados que usaremos é o dataset Netflix TV Shows and Movies, no link: https://www.kaggle.com/datasets/victorsoeiro/netflix-tv-shows-and-movies.

Esses dados constituem-se de dois arquivos no formato csv, o credits e o titles. É preciso dizer que devemos fazer um pré-processamento no titles antes de importá-lo no HDFS. Arquivos csv usam a vírgula como separador de colunas, mas uma de suas colunas, o description, possui vírgulas (,) no meio do texto, bem como quebras de linha (\n). Em nosso pré-processamento removemos as quebras de linha e salvamos o arquivo com tab (\t) como separador de colunas.

Para inicializar o sistema Hadoop e todos os seus nodos, rodamos o seguinte comando. Isso será tratado com mais detalhes na próxima sessão.

```
$ ~/hadoop/sbin/start-all.sh
```

Como mencionado anteriormente, os comandos no Hadoop utilizam basicamente os mesmos comandos CLI do Linux. Para empregar o HDFS usamos os comandos hdfs dfs - precedendo o comando Linux. Temos assim, portanto:

```
$ hdfs dfs -put local dir-hdfs          copia o arquivo local para dir no hdfs
$ hdfs dfs -get dir-hdfs                copia o arquivo do dir hdfs para diretório
  local
$ hdfs dfs -ls /                        mostra o conteúdo do diretório raiz hdfs
$ hdfs dfs -ls /dir-hdfs                mostra o conteúdo do diretório hdfs
$ hdfs dfs -help                        chama o arquivo de ajuda do hdfs
$ hdfs dfs -mkdir /dir      -hdfs       cria o diretório hdfs
$ hdfs dfs -cp orig dest                copia a origem ao destino no hdfs
$ hdfs dfs -mv orig dest                move origem ao destino no hdfs
$ hdfs dfs -cat /file                   imprime o arquivo armazenado no hdfs
$ hdfs dfs -head /file                  imprime as primeiras linhas do arquivo no
  hdfs
$ hdfs dfs -touchz /file                cria o arquivo vazio no hdfs
$ hdfs dfs -rm /file                    deleta o arquivo no hdfs
$ hdfs dfs -rm -R /dir-hdfs             deleta o diretório hdfs e todo seu
  conteúdo interno
```

Quanto à estrutura e ao uso dos comandos CLI no HDFS, ele se porta como no Linux. Seu diretório raiz é o barra (/), como no Linux. Todo o mecanismo de partição em blocos, replicação dos blocos, armazenamento nos DataNodes e anotação no NameNode ocorre, mas não precisamos nos preocupar com a sua execução.

Portanto, para importarmos no HDFS e lermos as primeiras linhas do arquivo credits.csv, por exemplo, realizamos os seguintes comandos:

```
$ cp /media/sf_Shared_Folder/credits.csv ~          Copia o arquivo da pasta
      compartilhada para o diretório do usuário
$ hdfs dfs -put ~credits.csv /          copia o arquivo para a raiz do hdfs
$ hdfs dfs -ls /                        visualiza o conteúdo do diretório raiz hdfs
$ hdfs dfs -head /credits.csv           mostra as primeiras linhas do arquivo
```

3.3. Comandos com o YARN

Lembremos que o YARN é responsável por gerenciar os recursos dos nodos no cluster. Ele começa a atuar desde o momento que inicializamos os nodos do Hadoop. Há duas maneiras de se inicializar os nodos.

A primeira maneira, que é a recomendada para se utilizar em produção, é inicializar cada nodo individualmente, chamando os arquivos script em separado:

```
$ cd ~/hadoop/sbin                          muda para o diretório dos scripts
$ ./start-dfs.sh                            inicializa os nodos e jobs do HDFS
$ ./start-yarn.sh                           inicializa os nodos e jobs do YARN
$ mapred –daemon start historyserver        inicializa o job de log do servidor
```

A segunda maneira, recomendada apenas para ambiente de teste, é usar um único comando, que inicializa todos os nodos e jobs. Esse comando não é recomendado para o ambiente de produção, pois dependendo do número e ordem de frameworks, pode ser importante especificar uma ordem de inicialização e esperar que um complete para dar continuidade a outro.

```
$ ~/hadoop/sbin/start-all.sh                inicializa todos os nodos e jobs
```

Uma vez que o Hadoop tenha sido inicializado, pode-se verificar os jobs Java ativos, usando o comando seguinte. Se tudo estiver funcionando, devem aparecer todos: o NameNode, Secondary NameNode, DataNode, Node Manager e Resource Manager.

```
$ jps                                   verifica processos Java ativos
```

Quando terminar a utilização e desejar desligar a máquina virtual ou PC, é importante finalizar os nodos, para liberar memória e recursos. Análogo à inicialização, pode-se rodar um script individual para cada ou usar o comando geral.

```
$ cd ~/hadoop/sbin                      muda para o diretório dos scripts
$ ./stop-dfs.sh                         finaliza os nodos e jobs do HDFS
$ ./ stop -yarn.sh                      finaliza os nodos e jobs do YARN
$ mapred –daemon stop historyserver     finaliza o job de log do servidor
$ ~/hadoop/sbin/stop-all.sh             finaliza todos os nodos e jobs
```

Enquanto o Hadoop estiver ativo ou realizando alguma atividade, podemos acompanhar seu funcionamento de duas maneiras.

Uma maneira é verificar, através do navegador (browser), pelo endereço e porta dos nodos. Na máquina virtual, os nodos estarão no localhost, conforme configurado no core-site.xml. Pode-se acessar, portanto, pelo navegador dele:

Serviço	IP
Hadoop	http://localhost:9870 (antes da versão 3.0, a porta era 50070)
YARN NameNode	http://localhost:8042
YARN ResourceManager	http://localhost:8088

Para cada job uma porta é liberada durante a sua execução, onde pode-se acompanhá-lo, pelo navegador.

Outra maneira de se acompanhar os recursos de uma tarefa é acessando o log do YARN. Para isso utiliza-se o comando a seguir, junto com o número identificador da tarefa (X representa o número).

```
$ yarn logs -applicationId application_XXXXXXXXXXXXX_XXXX
                                                    acessa o log
$ yarn logs -applicationId application_XXXXXXXXXXXXX_XXXX > processo.log
                                                    salva o log no arquivo
```

3.4. Aplicando o MapReduce

Vamos agora analisar como aplicamos o MapReduce para extrair informações dos nossos dados. O Hadoop oferece liberdade e flexibilidade ao desenvolvedor através do Hadoop Streaming. Essa ferramenta permite que os scripts de Mapper e Reducer sejam escritos em qualquer linguagem de programação, desde que o computador tenha a engine para interpretar aquele código. Dessa maneira, o desenvolvedor pode criar seus próprios scripts, na linguagem que lhe for mais conveniente, como Python, R ou até o próprio Java. Neste curso vamos focar no Python, portanto os códigos apresentados aqui são nessa linguagem. A estrutura básica do comando Hadoop Streaming é da seguinte forma:

```
$ mapred streaming -input <INPUT_DIR>
                   -output <OUTPUT_DIR>
                   -mapper <SCRIPT_MAPPER>
                   -reducer <SCRIPT_REDUCER>
```

O INPUT_DIR é um diretório no HDFS, onde deve conter dentro deles todos os dados em que se deseja realizar a análise. O OUTPUT_DIR é o diretório no HDFS onde ficará salvo o resultado do processo MapReduce que será realizado. Este diretório não pode existir antes de se rodar o código, ele será gerado automaticamente. O SCRIPT_MAPPER e SCRIPT_REDUCER são os arquivos de script contendo os códigos, no PC local. Ao final desta sessão iremos mostrar este comando com os devidos argumentos, em um exemplo.

Vamos analisar então como vamos escrever nossos códigos de Mapper e Reducer. Para cada análise pretendida, o desenvolvedor deve pensar e criar o código, seguindo a operação que pretende realizar. Vamos começar com uma análise bastante simples: um MapReduce para contar palavras.

O nosso Mapper.py pode ser escrito da seguinte forma:

```
#! /usr/bin/env python3

import sys

try:
        for line in sys.stdin:
                words = line.split()
                for word in words:
                        print('{0}\t{1}'.format(word, 1))
except Exception as e:
        raise(e)
```

É importante observar primeiro como ocorrem a entrada e saída dos códigos Mapper e Reducer. Em Python, a entrada do código ocorre pelo sys.stdin (arquivos ou texto, que são argumentos da função chamada) e a saída ocorre pelo próprio comando print. Como o processo inteiro de MapReduce ocorre através de uma pipeline, a saída de um funciona como a entrada da próxima etapa.

Este código é bastante simples. Para cada linha da entrada ele separa as palavras pelos espaços, pelo comando split() e para cada palavra individual, ele imprime a palavra e 1, indicando uma ocorrência. A saída do Mapper é o conjunto impresso de todas as palavras e 1 para cada ocorrência. Podemos modificar para que ele passe a imprimir o número de ocorrências de uma determinada coluna, por exemplo, de uma tabela.

```
#! /usr/bin/env python3

import sys

try:
        for line in sys.stdin:
                data = line.split(",")
                print('{0}\t{1}'.format(data[1], 1))
except Exception as e:
        raise(e)
```

Neste caso, o código separa as linhas por vírgulas e não mais espaços. A vírgula foi escolhida como separadora das colunas, mas estes podem ser ponto e vírgula, tabulação ou outros, dependendo dos dados. O

print não imprime mais as palavras e sim o valor na 2ª coluna, que é indexada por 1 (já que o Python começa do 0).

Na próxima etapa, do Shuffling, o sistema ordena todas as palavras em ordem alfabética. Esse conjunto ordenado então alimenta a etapa de Reducer. Para o nosso contador, podemos implementar o Reducer da seguinte maneira:

```
#! /usr/bin/env python3

import sys

curr_key = None
curr_count = 0

try:
        for line in sys.stdin:
                key, count = line.split("\t", 1)
                count = int(count)
                if key == curr_key:
                        curr_count += count
                else:
                        if curr_key:
                                print('{0}\t{1}'.format(curr_key, curr_count))
                        curr_count = count
                        curr_key = key
        if curr_key == key:
                print('{0}\t{1}'.format(curr_key, curr_count))
except Exception as e:
                raise(e)
```

Esta não é a única maneira de se implementar o Reducer. O código em si fica da liberdade de cada um, para se implementar. O importante aqui é fazer um contador, para que imprima cada palavra-chave e a soma de ocorrências de cada palavra.

Apesar de possuírem o comando print, que normalmente faz com que algo seja impresso na tela, os scripts Mapper e Reducer não imprimem os estágios intermediários na tela do usuário. Esses comandos são rodados e o próprio Hadoop armazena os estágios intermediários em arquivos temporários, sem poluir o ambiente do usuário. O resultado final é

armazenado em arquivos, no diretório especificado pelo argumento –out, dentro do HDFS.

Durante a execução do MapReduce, ele disponibiliza a porta para o usuário poder acompanhá-lo, junto aos recursos utilizados pelo sistema, para tal operação. No final do processo, o diretório especificado é criado, então, no HDFS. Podemos acessar o seu conteúdo a partir dos comandos abaixo.

```
$ hdfs dfs -ls /out                    mostrar conteúdo do diretório /out
$ hdfs dfs -cat /out/part-00000        imprime a saída do MapReduce
```

Após o processo de MapReduce, ele vai criar dentro do diretório de saída dois arquivos. O arquivo _SUCCESS não contém dados e é utilizado como um flag, apenas para identificar que o processo rodou com sucesso. Dependendo do tamanho dos dados de saída, eles podem ser armazenados em um ou mais partições, identificados começando por part-00000 e numerados em ordem. Para ler a saída, basta ler esses arquivos.

Concluindo nosso exemplo, para rodar o comando MapReduce, com os arquivos Mapper e Reducer em python armazenados no diretório principal do usuário, usamos o seguinte comando:

```
$ mapred streaming -input /input -output /out -mapper ~/mapper.py -reducer
  ~/reducer.py
```

> Capítulo 4

Capítulo 4. Framework Hive

Diante do que foi apresentado, vimos como é possível armazenar, manipular e fazer contas com dados no Hadoop. Mas vimos também que a manipulação através do MapReduce exige um certo esforço do desenvolvedor, para cada operação. Diante dessas dificuldades, surgiu o framework Hive, que abstrai parte da complexidade e facilita as análises a partir de uma linguagem semelhante ao SQL.

4.1. Introdução ao Hive

Hive é um framework construído em cima do Hadoop, que abstrai toda a dificuldade e necessidade de se implementar cada processo individualmente, e facilita o uso do MapReduce, dando liberdade ao desenvolvedor para realizar análises mais complexas. O Hive foi desenvolvido em 2007 pelo Facebook e em 2008 se tornou um projeto código aberto.

O diferencial do Hive é que ele emprega uma linguagem de programação própria, o HQL (Hive Query Language), bastante semelhante ao SQL, mas com recursos mais avançados de orientação a objeto. Uma vez que a linguagem SQL é muito difundida e utilizada para lidar com bancos de dados, as operações no Hadoop se tornam muito mais fáceis e familiares. Apesar disso, as operações ainda realizam processos MapReduce por trás.

Desta maneira, o Hive emprega um padrão de dados relacional, para organizar e armazenar os dados. É necessário selecionar e instalar um dos possíveis schemas de bancos de dados relacionais, que será utilizado pelo framework. Dentre os mais famosos, ele possibilita o uso do MySQL, PostgreSQL e Derby. Iremos entrar em mais detalhes quando apresentarmos a instalação e configuração do Hive.

Quanto à linguagem HQL, alguns detalhes são importantes. A linguagem em si não diferencia maiúscula de minúscula, mas nomes de variáveis e classes Java diferenciam. Portanto, empregaremos o padrão de usar maiúsculas para comandos HQL, para diferenciar comandos de variáveis, mas por pura estética. Além disso, cada comando deve terminar em ponto e vírgula. Deve-se atentar ainda para o uso de espaço, pois o HQL utiliza o espaço para separar comandos de seus argumentos. Por fim, o HQL suporta a maioria dos comandos SQL tais como CREATE, DROP, SELECT, WHERE etc.

4.2. Metastore e modelagem dos dados

O sistema de armazenamento de dados do Hive funciona através do Metastore, a estrutura de banco de dados selecionada. Para nossos estudos iremos empregar o Derby. Enquanto o MySQL e PostgreSQL sejam mais utilizados em ambiente de produção, o Apache Derby é mais simples, possui algumas desvantagens, mas será o mais fácil de utilizarmos em nossos estudos.

Dentro do Metastore ele armazena os metadados. Estes são as informações essenciais das tabelas, que são armazenadas de fato no HDFS. O caminho padrão no HDFS, onde ficam todos os dados do Hive, é: /user/hive/warehouse/

Dentro do diretório warehouse o usuário pode definir bancos de dados (databases) e dentro dos bancos de dados ficam as tabelas (tables). Os bancos de dados e tabelas aparecem no HDFS como diretórios, da maneira:

- /user/hive/warehouse/<database>
- /user/hive/warehouse/<database>/<table>

Ao se criar uma tabela no Hive, é necessário declarar o seu schema, com os nomes e tipos de variável de cada coluna. O Hive suporta uma quantidade considerável de tipos de variáveis, de forma a otimizar o uso de memória para a tabela.

- Numéricos:
  - Tinyint
  - Smallint
  - Int / Integer
  - Bigint
  - Float
  - Double
  - Double precision
  - Decimal
  - Numerical
- Datas:
  - Timestamp
  - Date
  - Interval
- Strings:
  - String
  - Varchar

- Char

- Misc:
  - Boolean
  - Binary

Para as tabelas, existem dois tipos, o External e o Managed:

- External:
  - A forma usual de se empregar tabelas.
  - A tabela é um metadado dos dados armazenados no HDFS.
  - Se a tabela for dropada (apagada), os dados continuam.
- Managed:
  - Útil em ocasiões especiais, como tabelas temporárias.
  - A tabela corresponde aos próprios dados no HDFS.
  - Se a tabela for dropada, os dados são apagados.

O usual é empregarmos sempre a tabela External, que também é a mais segura, para não correr o risco de se perder os dados.

4.3. Instalação do Hive

Para fazer a instalação do Hive, primeiro precisamos fazer o download dos arquivos binários, tanto do Hive quanto do Derby, a estrutura de banco de dados que utilizaremos. É possível empregar as outras estruturas como MySQL e PostgreSQL, o download e configuração são semelhantes, mas ilustramos aqui o Derby.

Os dois podem ser baixados de suas páginas oficiais, seguindo a versão correta:

- https://hive.apache.org/
- https://db.apache.org/derby/

Para extrair os arquivos binários e renomeá-los, rodamos os seguintes códigos:

```
$  tar -xvf apache-hive-3.1.3-bin.tar.gz          extrair o binário do hive
$  tar -xvf db-derby-10.14.2.0-bin.tar.gz         extrair o binário do derby
$  mv apache-hive-3.1.3-bin hive                  renomear o diretório hive
$  mv db-derby-10.14.2.0-bin derby                renomear o diretório derby
```

O próximo passo é adicionar esses diretórios às variáveis de ambiente:

```
$  nano .bash_aliases
      export HIVE_HOME=$HOME/hive
      export PATH=$PATH:$HIVE_HOME/bin
      export DERBY_HOME=$HOME/derby
      export PATH=$PATH:$DERBY_HOME/bin
```

Também é preciso adicionar o diretório do Hadoop ao script do Hive-config:

```
$  cd ~/hive/bin
$  nano hive-config.sh
      export HADOOP_HOME=$HOME/hadoop
```

Para configurar corretamente o Hive, precisamos criar o arquivo hive-site.xml do template e setar corretamente suas variáveis:

```
$  cd ~/hive/conf
$  cp hive-default-xml-template hive-site.xml    criar hive-site.xml do template
$  nano hive-site.xml
      <property>
        <name>system:java.io.tmpdir</name>
        <value>/tmp/hive/java</value>
      </property>
      <property>
        <name>system:user.name</name>
        <value>${user.name}</value>
```

</property>

Além de adicionar essas duas propriedades, precisamos realizar mais duas modificações, ainda dentro do editor de textos nano. Uma é remover os caracteres '&#8;' da linha 3215 do arquivo (eles ocasionam erro na hora de rodar) e outra é modificar o valor da propriedade javax.jdo.option.ConnectionURL. Para fazer a busca por linha, no editor nano, usamos o comando CTRL + SHIFT + -. Para buscar por texto, usamos o comando CTRL + W. Edite para a seguinte propriedade:

```
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:derby:;databaseName=metastore/metastore_db;create=true</value>
</property>
```

Uma das propriedades descritas aqui é o caminho para o diretório onde o Metastore armazenará todos os metadados. Uma das desvantagens do Derby em relação às outras estruturas de dados, é que este caminho não é global, ele é relativo à posição do usuário. Portanto, lembre-se de chamar o Hive sempre localizado na pasta principal do usuário. Desta maneira, o Metastore será armazenado no caminho ~/metastore/metastore_db. Se o Hive for chamado de outro caminho, você pode reiniciar o Derby apagando o diretório, onde for criado.

Para completar nossas configurações, precisamos recarregar as variáveis de ambiente e criar o diretório dentro do HDFS, onde os dados serão armazenados:

```
$ cd
$ source .bash_aliases                          recarregar as variáveis de ambiente
$ hdfs dfs -mkdir -p /user/hive/warehouse       cria o diretório para o hive
$ hdfs dfs -chmod g+w /user/hive/warehouse      dá permissão para escrita e
  execução
```

Após as configurações, na primeira vez precisamos iniciar o schema da estrutura de dados, no caso o Derby. Isso é feito a partir do seguinte comando:

```
$  ~/hive/bin/schematool -dbType derby -initSchema   inicializa o schema
```

Para verificar a instalação, pode-se pedir a versão do Hive:

```
$  hive --version                                          verifica a versão do Hive
```

4.4. Hive na prática

Para acessar o Hive basta entrar com o seguinte comando no terminal:

```
$  hive
```

Isso fará com que o sistema entre no Hive e os comandos passem a ser em HQL. O terminal de comandos também mudará de $ para > indicando que você está dentro do Hive. Para sair, use o comando 'quit;'.

Para criar uma tabela, deve-se selecionar um banco de dados. Para criar, visualizar e deletar eles, os comandos são iguais ao SQL. Mostramos a seguir os comandos mais básicos para rotinas comuns:

```
>  SHOW DATABASES;                          mostra os bancos de dados disponíveis
>  CREATE DATABASE nome_db;                 cria o banco de dados nome_db
>  USE nome_db;                             seleciona o banco de dados nome_db
>  SET hive.cli.print.current.db=true; mostra o banco de dados ativo
>  SHOW TABLES;                             mostra as tabelas disponíveis
>  CREATE TABLE nome_tb( );                 cria a tabela managed nome_tb
>  CREATE EXTERNAL TABLE nome_tb( );        cria a tabela external nome_tb
>  CREATE TABLE nome_tb(nome_col1 INT,
                        nome_col2 STRING,
                        nome_col3 FLOAT);
                                           cria a tabela com 3 colunas
>  CREATE TABLE nome_tb( )
   ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
                                  cria a tabela com , como separador das colunas
>  CREATE TABLE nome_tb( )
   TLBPROPERTIES("skip.header.line.count"= "1");
                                  cria a tabela ignorando a 1ª linha (header)
>  INSERT INTO TABLE nome_tb VALUES(1, 'Aaa', 1.0);
       insere dados manualmente
>  LOAD DATA INPATH '/file' OVERWRITE INTO TABLE nome_tb;
   lê um arquivo e importa para a tabela
>  DESC nome_tb;                            descrição das colunas e tipos da tabela
>  DESCRIBE FORMATTED nome_tb;              descrição detalhada da tabela
>  DROP TABLE nome_tb;                      deleta uma tabela
```

```
> DROP DATABASE nome_db;              deleta um banco de dados
> SHOW CREATE TABLE nome_tb;          mostra os comandos do hive para criar a tb
```

Mostramos agora alguns comandos possíveis de se realizar com o comando SELECT, que seleciona determinados campos da tabela:

```
> SELECT * FROM nome_tb;              mostra toda a tabela
> SELECT * FROM nome_tb LIMIT 5;      mostra as primeiras 5 linhas da tabela
> SELECT COUNT(*) FROM nome_tb;       conta o número de ocorrências (linhas) da tb
> SELECT COUNT(*) FROM nome_tb WHERE col = 1;
      conta o número de ocorrências de acordo com um filtro
> SELECT col1 AS col2 FROM nome_tb;           mostra a col1, mas usa o nome col2
> SELECT * FROM nome_tb GROUP BY col;             agrupa por col
> SELECT * FROM nome_tb ORDER BY col;             ordena crescente por col
> SELECT * FROM nome_tb ORDER BY col DESC;        ordena decrescente por col
> SELECT AVG(col) FROM nome_tb;               mostra média da coluna col
> SELECT MAX(col) FROM nome_tb;               mostra o máximo da coluna col
> SELECT MIN(col) FROM nome_tb;               mostra o mínimo da coluna col
```

Pode-se ver que com os comandos HQL temos total liberdade, tal como com SQL em banco de dados relacionais. Podemos realizar filtros e contagens mais avançadas também. O Hive interpreta o conjunto de comandos e realiza as operações necessárias em MapReduce.

É possível agrupar uma série de comandos HQL para uma única operação. A exemplo, se tivermos uma tabela com uma coluna col1 com palavras-chave, podemos agrupar por col1, contar a quantidade de ocorrências como col2 e pedir para ordenar em ordem decrescente de col2, isso vai nos retornar as palavras-chave de col1, que têm mais ocorrência. Isso pode ser realizado pelo seguinte comando:

```
> SELECT col1,COUNT(1) AS col2 FROM nome_tb GROUP BY col1 ORDER BY col2
  DESC;
```

O Hive ainda permite que usemos comandos de JOIN, muito usado em SQL para mesclar tabelas por chaves. Para aplicar com HQL, selecionamos o tipo que queremos:

- JOIN / FULL OUTER JOIN
- LEFT OUTER JOIN

- RIGHT OUTER JOIN

```
> SELECT tab01.col1, tab01.col2, tab02.col3          colunas a mostrar após o join
  FROM nome_tb1 tab01          nome da tabela a esquerda, aqui usamos alias tab01
  JOIN                                               tipo de join a usar
  nome_tb2 tab02               nome da tabela a direita, aqui usamos alias tab02
  ON (tab01.col_x = tab02.col_y);                    quais colunas farão o join
```

As colunas que são utilizadas como chave primária e secundária, para realizar os Joins, não precisam aparecer na linha SELECT. O uso de *alias* é apenas para facilitar, pode-se usar os nomes originais das tabelas.

4.5. Formatos de arquivos

As tabelas podem ser comprimidas e salvas em formatos diferentes, próprios para otimizar o uso de HD, velocidade de acesso, operação ou transferência dos dados, de acordo com a aplicação. Essas compressões podem se tornar necessárias de acordo com o crescimento do volume e tamanho das tabelas.

Para a compressão dos dados, utiliza-se algum codec, algoritmos próprios para a compressão dos dados. Os mais eficientes são o LZ4 e o Snappy.

Os formatos de arquivo, para os quais podemos comprimir, são Parquet, Avro e ORC. Abaixo apresentamos as principais características de cada:

- Parquet:
  - Formato colunar desenvolvido pela Cloudera e Twitter.
  - Reduz o espaço de armazenamento.
  - Aumenta a performance.
  - Mais eficiente com inserção de dados em batch (grande volume).

- Ideal para dados colunares.

- Avro:
  - Armazenamento em formato de linhas.
  - Salva os dados em formato JSON.
  - Os dados são salvos em binário, otimizando a compactação.
  - Funciona com dados semiestruturados.

- Apache ORC:
  - Otimizado para armazenamento eficiente de dado.
  - Sua estrutura armazena diversas linhas em colunas.
  - Pode realizar cálculos em colunas, referente a diversas linhas.

De maneira geral, temos que o Parquet é mais recomendado para dados tabulares, já armazenados em colunas, e é melhor para leitura de dados e buscas em colunas. O Avro se torna mais recomendado para escrita de dados e possui mais liberdade de schema, que pode ser modificado livremente. O ORC é mais recomendado para compressão dos dados e método de Spark Filter / predicate pushdown, métodos avançados de busca.

Para se salvar uma tabela em um formato de compressão, pode-se especificar o formato e o codec a se utilizar:

```
> CREATE TABLE nome_tb( )
  STORED AS PARQUET
  TBLPROPERTIES(“parquet.compression”= “SNAPPY”);
```

4.6. Particionamento

O Hive oferece o particionamento nas tabelas, que pode ser utilizado para melhorar a organização e acesso aos dados. Quando se declara um particionamento, essa partição é armazenada como uma coluna adicional na tabela. Entretanto, no HDFS, aparece como um diretório adicional, dentro da tabela.

Ao se utilizar o particionamento, o desenvolvedor deve tomar cuidado para não particionar demais a tabela, pois pode sobrecarregar o namenode do cluster, que precisará acessar um grande número de diretórios a cada operação.

Ao se declarar uma partição, esta funciona como uma variável comum, portanto deve-se declarar seu nome e o tipo de variável. Ao se inserir dados em uma tabela com partição, deve-se declarar qual seu valor. Veja o exemplo abaixo:

```
> CREATE TABLE nome_tb( )
  PARTITIONED BY (data STRING);
                              criação da tabela com partição chamada data do tipo string
> INSERT INTO TABLE nome_tb
  PARTITION (data='01-01-2022')
  VALUES( );                          inserindo dados em uma tb com partição
> LOAD DATA INPATH 'file' INTO TABLE nome_tb
  PARTITION (data='01-02-2022'); importando dados em uma tb com partição
```

No caso do exemplo acima, criamos duas partições e elas aparecerão no HDFS como dois diretórios dentro da tabela.

```
$ hdfs dfs -ls /user/hive/warehouse/nome_db/nome_tb
$ hdfs dfs -ls /user/hive/warehouse/nome_db/nome_tb/data=01-01-2022
$ hdfs dfs -ls /user/hive/warehouse/nome_db/nome_tb/data=01-02-2022
```

Acessando os dados da tabela, pelo Hive, será possível verificar que a tabela possuirá uma coluna adicional chamada data e com os respectivos valores das partições.

> Capítulo 5

Capítulo 5. Framework Impala

Impala é um outro framework desenvolvido para lidar com HDFS, que também permite a manipulação dos dados através de comandos SQL. Muitas vezes é comparado ao Hive, por ambos permitirem comandos em SQL. Porém a sua arquitetura se diferencia bastante do Hive.

5.1. Introdução ao Impala

O Impala pode ser descrito como uma engine MPP (*massive parallel processing*) open source, que atua no HDFS. Ele foi inspirado no projeto Dremel do Google e atua direto nos daemons, sem a necessidade de implementar o MapReduce. Devido à sua arquitetura diferente, o Impala acaba se sobressaindo como um motor de SQL de alta performance, com baixa latência (em torno de milissegundos) e ideal para processos que exijam muita velocidade de resposta. Comparado ao tradicional MapReduce, ele oferece uma velocidade de 5x a 50x maior.

5.2. Impala x Hive

Comparando o Impala com Hive, temos recomendações de quando se utilizar um ou outro. A primeira grande diferença é que o Hive salva os estágios intermediários dos processos em disco, resultado do processo de MapReduce, que é constituído de várias etapas. E essa é uma das razões pela qual o Hive é mais lento. Entretanto, isso garante ao Hive uma vantagem: ele é mais recomendado para lidar com lotes de dados e quantidades de dados maiores. Uma outra vantagem do Hive é a sua confiabilidade e tolerância a falhas. Devido à estrutura, já apresentada, há fatores de replicação e garantia da disponibilidade dos dados e, portanto, é o recomendado para operações em que seja necessário garantir a integridade dos dados e das operações.

O Impala se apresenta como a melhor solução para processamento em tempo real e consultas ad-hoc. Entretanto, com o crescimento da quantidade de dados a ser manipulado, aumenta o risco de algum nodo falhar e no caso de falha, perde-se todo o job realizado, não havendo tolerância a falhas. Um outro fator é que o Impala consome muita memória e não é executado de forma eficiente para operações pesadas, pois ele tenta salvar tudo em memória, podendo sobrecarregar o sistema e travá-lo. De fato, o Impala exige pelo menos 4 GB de RAM, mas recomenda-se pelo menos 8 GB de RAM para ser executado. Em uma máquina virtual, isso quer dizer que nossa máquina física deve possuir pelo menos em torno de 12 GB para rodar o essencial, uma configuração bastante exigente, do contrário do Hive que pode ser rodado em qualquer máquina.

Em resumo, para sistemas mais robustos, com muita memória e que não exigem integridade dos dados, mas velocidade de resposta ou processamento em tempo real, o Impala se torna o mais atrativo. Não iremos estudá-lo na prática aqui, mas seus comandos são todos em SQL, semelhantes aos comandos já estudados para o Hive.

> Capítulo 6

Capítulo 6. Introdução às Plataformas Nuvem

As plataformas nuvem surgiram por volta de 2005 e atualmente se apresentam como soluções no estado da arte em se tratando de escalabilidade, confiabilidade e disponibilidade de sistemas. Aliados à facilidade de se lidar com big data, se tornam ferramentas ideais para esse universo. E obviamente os serviços de nuvem possuem suas aplicações em Hadoop também.

6.1. Apresentação das plataformas nuvem

Atualmente as três maiores plataformas nuvem são a AWS (*Amazon Web Services*), o Google Cloud e o Microsoft Azure. Cada uma dessas plataformas disponibiliza dezenas e dezenas de serviços, que englobam todas as necessidades de um produto, tais como bancos de dados, nodos em clusters, segurança, streaming, coordenação, aplicativos de IA, Business Intelligence, monitoramento etc.

Plataformas nuvem cobram por seus serviços, pois são soluções comerciais, mas oferecem opções de se testar serviços selecionados por um determinado tempo. Só é preciso tomar cuidado para não utilizar recursos que cobram ou se esquecer e deixar máquinas ligadas, consumindo os recursos grátis.

Não iremos tratar de todos os serviços disponíveis, mas iremos destacar os seus serviços correspondentes ao Hadoop. O AWS possui o EMR (*Elastic MapReduce*), o Google Cloud possui o Dataproc e o Microsoft Azure possui o HDInsight. Cada um tem suas peculiaridades de configuração e programação, mas todos funcionam baseados na estrutura Hadoop que já estudamos.

Em cada serviço é preciso primeiro selecionar as máquinas que irão compor o cluster, de forma análoga a quando criamos e instalamos o sistema operacional na nossa máquina virtual. No AWS, estas são os EC2s (*Elastic Compute Cloud*), no Google Cloud são os Compute Engine, no Microsoft Azure são as Máquinas Virtuais Azure. A vantagem de se testar em serviços nuvem é que podemos testar clusters com vários computadores.

De maneira análoga à nossa máquina virtual, onde tivemos de instalar o Hadoop e configurar a conexão SSH, é necessário instalar o Hadoop e configurar a conexão SSH em cada computador do cluster. Mas no caso de múltiplos computadores compondo um nodo, podemos testar propriedades como fator de replicação e nodos diferentes, funcionando como NameNode ou DataNode.

Isso encerra nossa breve introdução a esses sistemas nuvem. Quem se interessar mais, há bastante conteúdo em guias criados pelas próprias plataformas, que envolvem muito mais outros tópicos, tais como segurança, comunicação e configuração. Na próxima sessão iremos apresentar uma outra plataforma nuvem, que ainda não foi citada, por ser específica para big data, o Databricks.

6.2. Plataforma Databricks

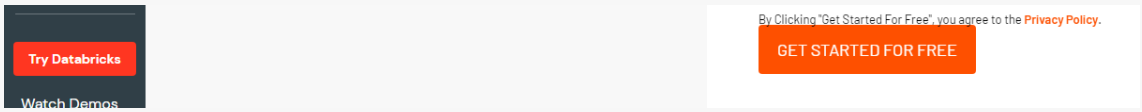
O Databricks é uma plataforma que engloba arquitetura Data Lake, com processamento Spark e notebooks próprios, semelhante ao Jupyter. Ele funciona como as outras plataformas nuvem, mas especializado para big data e sistemas distribuídos. A sua aplicação é mais visada à engenharia e análise de dados.

Felizmente o Databricks também oferece contas gratuitas, para explorar e aprender a navegar pela sua plataforma, através do Databricks Community Edition. Nosso próximo framework, que iremos estudar, será o Spark e o aprenderemos através do Databricks.

Para iniciar e criar a conta gratuita, acessamos a página principal do Databricks: https://databricks.com/.

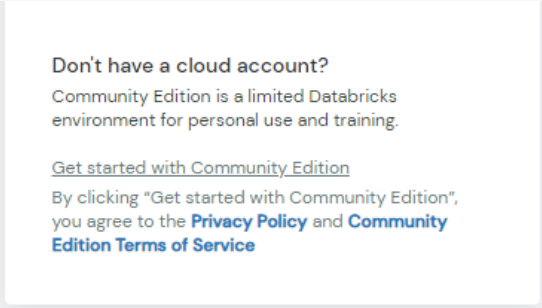
No site deles, clique no botão vermelho Try Databricks, preencha com seus dados e, posteriormente, clique em GET STARTED FREE.

Figura 11 – Botão de criar conta no Databricks.



No final da página haverá um link escrito “Get started with Community Edition”. É muito importante clicar nesse botão e NÃO selecionar o botão azul Get started. Ao se criar uma conta no Community Edition, você terá acesso gratuito por tempo indeterminado a recursos básicos limitados, mas o suficiente para estudar e aprender a utilizar o Spark. Caso você clique no botão azul Get started, você estará criando uma conta no próprio Databricks, que estará atrelado a um dos sistemas nuvem, apresentados anteriormente, e terá acesso gratuito à plataforma inteira, mas por apenas 14 dias, passando a pagar após esse prazo. Clicando para se cadastrar no Community Edition, o sistema irá enviar um e-mail para você pelo qual poderá confirmar seu cadastro.

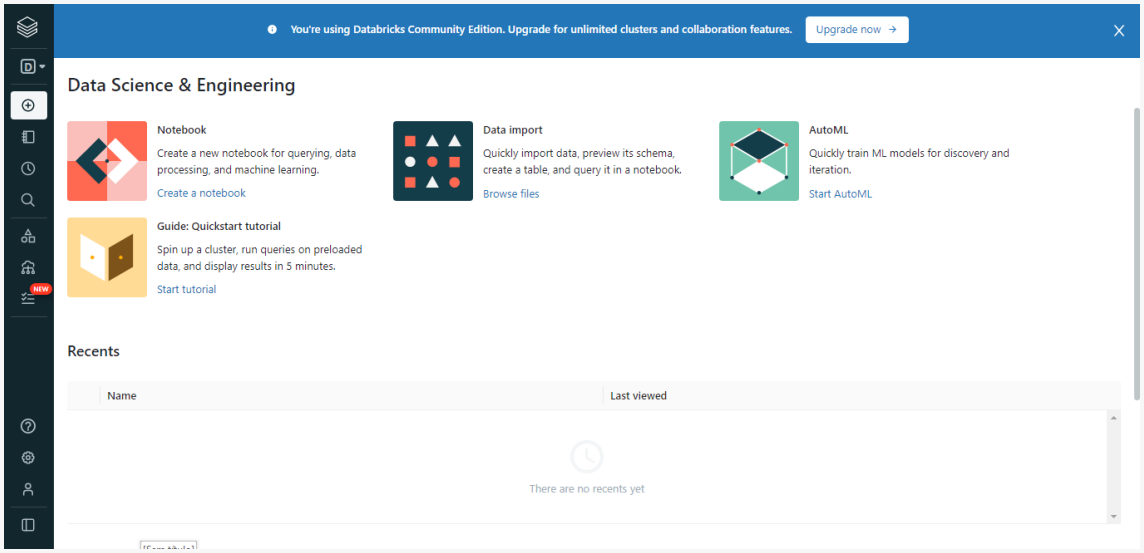
Figura 12 – Link criar conta no Community Edition.



Você pode acessar o Databricks Community Edition através do link: https://community.cloud.databricks.com/ . Este sistema é bastante amigável e facilita o aprendizado de novos usuários. O usuário não precisa instalar

nenhum sistema e nem configurar o Hadoop ou Spark. Os notebooks já possuem o Spark pré-instalado e podem ser aplicados de imediato.

Figura 13 – Página inicial Databricks Community Edition.

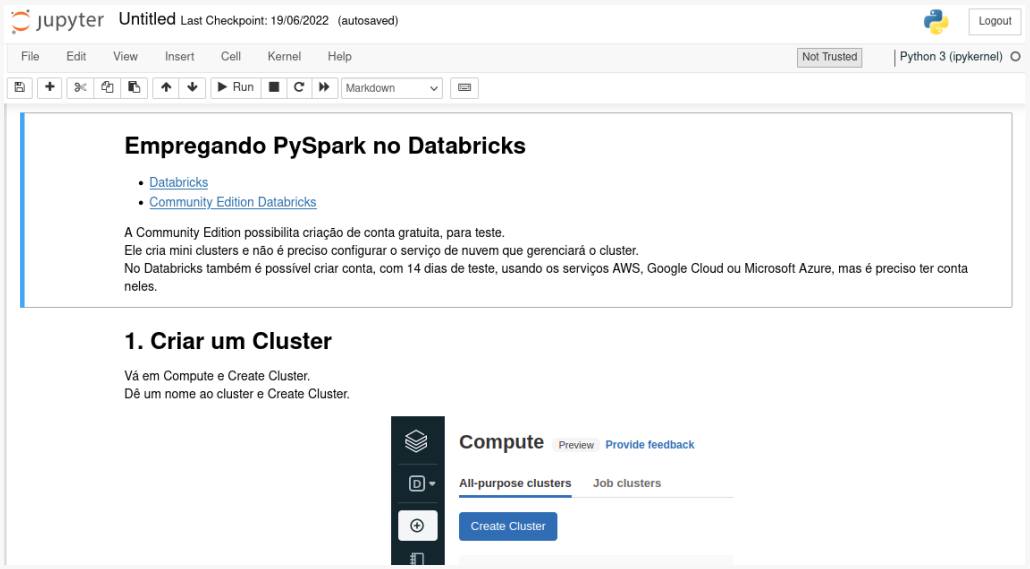


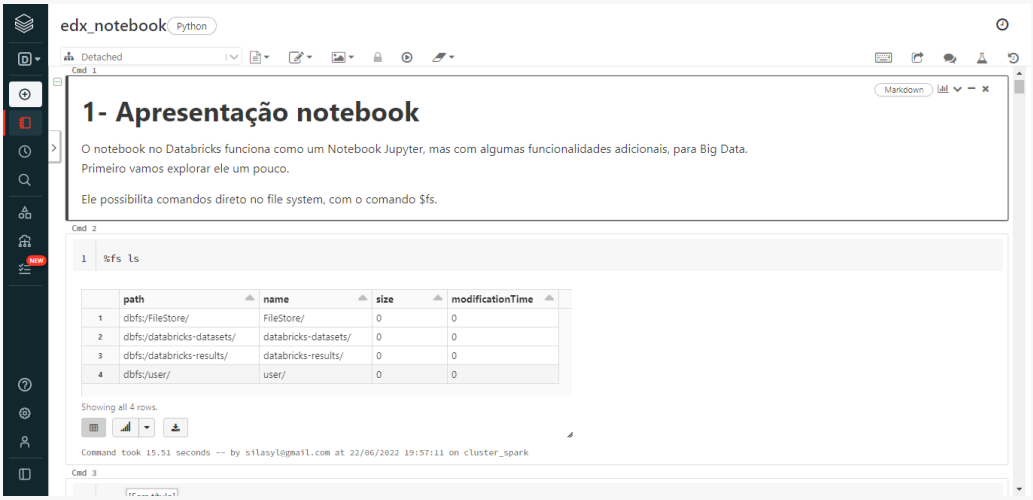
Os botões iniciais mostram as opções de se criar um novo notebook; procurar por arquivos (para importar dados); aplicar o AutoML para treinar modelos de Machine Learning; ou acessar o tutorial deles. Na aba à esquerda, tem-se acesso às ferramentas como criação de notebooks, tabelas, clusters ou pipeline; área de trabalho; busca; dados e outros.

Assim como em uma máquina virtual, o primeiro passo é criar um computador para rodar os códigos. Isso é feito através do botão Compute e Create Cluster. No Databricks o computador (nodo) é chamado de Cluster. Após inicializar o Cluster e ativá-lo, basta associá-lo a um notebook ou criar um novo.

O notebook databricks é bastante semelhante ao Jupyter, mas com algumas modificações e otimizações para o uso do Spark. Para aqueles não familiarizados com notebooks, o Jupyter é uma ferramenta bastante empregada em ciência de dados. Ele possibilita o uso mesclado de blocos de código e texto, além de ser o ideal para montagem de relatórios rápidos e apresentação de gráficos e análises mescladas aos códigos.

Figura 14 – Notebooks Jupyter e Databricks.

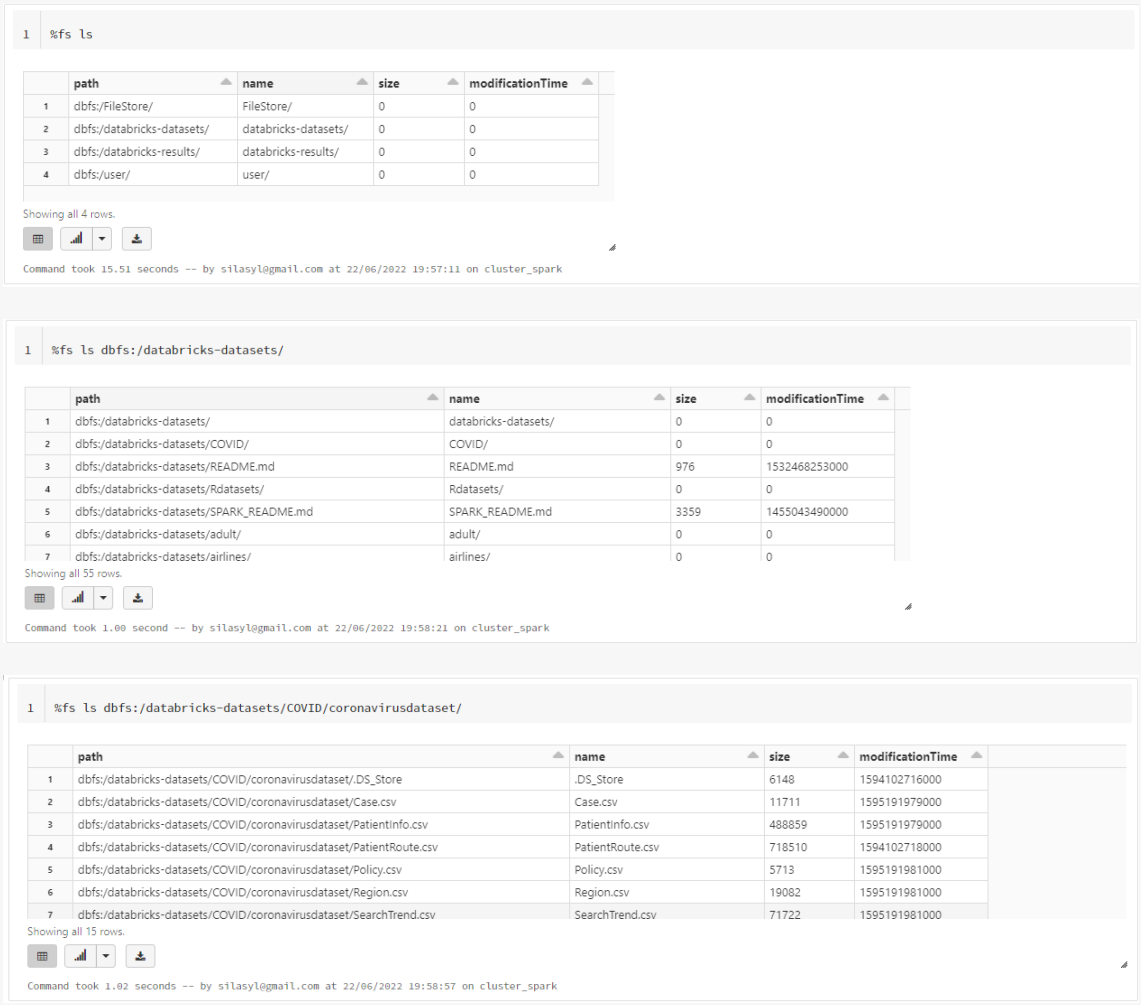




Os blocos do notebook databricks aceitam linguagens Scala, Python, R, SQL, além de linguagem de marcação de texto Markdown. Além disso, também é possível realizar comandos CLI no file system, através do comando %fs. Navegando no diretório raiz do databricks, podemos visualizar que ele possui internamente dados que podem ser utilizados. Vamos estudar os próximos capítulos utilizando seus dados de Covid:

```
%fs ls
%fs ls dbfs:/databricks-datasets/
%fs ls dbfs:/databricks-datasets/COVID/coronavirusdataset/
```

Figura 15 – Comandos no file system do databricks.



```
1  %fs ls
```

	path	name	size	modificationTime
1	dbfs:/FileStore/	FileStore/	0	0
2	dbfs:/databricks-datasets/	databricks-datasets/	0	0
3	dbfs:/databricks-results/	databricks-results/	0	0
4	dbfs:/user/	user/	0	0

Showing all 4 rows.

Command took 15.51 seconds -- by silasyl@gmail.com at 22/06/2022 19:57:11 on cluster_spark

```
1  %fs ls dbfs:/databricks-datasets/
```

	path	name	size	modificationTime
1	dbfs:/databricks-datasets/	databricks-datasets/	0	0
2	dbfs:/databricks-datasets/COVID/	COVID/	0	0
3	dbfs:/databricks-datasets/README.md	README.md	976	1532468253000
4	dbfs:/databricks-datasets/Rdatasets/	Rdatasets/	0	0
5	dbfs:/databricks-datasets/SPARK_README.md	SPARK_README.md	3359	1455043490000
6	dbfs:/databricks-datasets/adult/	adult/	0	0
7	dbfs:/databricks-datasets/airlines/	airlines/	0	0

Showing all 55 rows.

Command took 1.00 second -- by silasyl@gmail.com at 22/06/2022 19:58:21 on cluster_spark

```
1  %fs ls dbfs:/databricks-datasets/COVID/coronavirusdataset/
```

	path	name	size	modificationTime
1	dbfs:/databricks-datasets/COVID/coronavirusdataset/.DS_Store	.DS_Store	6148	1594102716000
2	dbfs:/databricks-datasets/COVID/coronavirusdataset/Case.csv	Case.csv	11711	1595191979000
3	dbfs:/databricks-datasets/COVID/coronavirusdataset/PatientInfo.csv	PatientInfo.csv	488859	1595191979000
4	dbfs:/databricks-datasets/COVID/coronavirusdataset/PatientRoute.csv	PatientRoute.csv	718510	1594102718000
5	dbfs:/databricks-datasets/COVID/coronavirusdataset/Policy.csv	Policy.csv	5713	1595191981000
6	dbfs:/databricks-datasets/COVID/coronavirusdataset/Region.csv	Region.csv	19082	1595191981000
7	dbfs:/databricks-datasets/COVID/coronavirusdataset/SearchTrend.csv	SearchTrend.csv	71722	1595191981000

Showing all 15 rows.

Command took 1.02 seconds -- by silasyl@gmail.com at 22/06/2022 19:58:57 on cluster_spark

> Capítulo 7

Capítulo 7. Framework Spark

O Spark foi desenvolvido por volta de 2009, se tornando código aberto em meados de 2010 e passando a fazer parte do projeto Apache em 2013. Semelhante ao Impala, que executa as operações em memória, o Spark é otimizado para operar na memória RAM. Esse tipo de memória, por ser extremamente mais rápida que o usual HD, torna o Spark até 100x mais rápido que o MapReduce tradicional.

7.1. Introdução ao Spark e PySpark

O Spark é um dos grandes frameworks atuais, muito utilizado quando a quantidade de dados se torna muito grande, e qualquer ganho de tempo em acesso, leitura e escrita, se torna significativamente importante. A sua engine é baseada em RDDs (*resilient distributed datasets*), uma estrutura de dados imutáveis (não podem ser sobrescritos) e distribuídos em nodos, com tolerância a falhas. Em versões mais modernas, o uso de RDDs foi substituído pelo Dataframe API, e posteriormente pelo Dataset API, mas ambos ainda executam os RDDs por trás.

O principal atrativo, portanto, é sua velocidade e capacidade de lidar com dados verdadeiramente grandes, de maneira muito rápida. Para se ter uma ideia, em 2014, o Spark ganhou a competição de *2014 Gray Sort Benchmark*, e foi capaz de ordenar 100 TB de dados em apenas 23 minutos, sendo que o recorde anterior era do Hadoop MapReduce, com 72 minutos. Além disso, em sua execução, o Spark utilizou apenas 1/10 de máquinas usadas pelo Hadoop.

Para se aproveitar ao máximo de sua capacidade, utilizamos máquinas de plataformas nuvem, com muita memória RAM. A memória RAM é uma memória cara e sai extremamente caro para se utilizar o Spark em

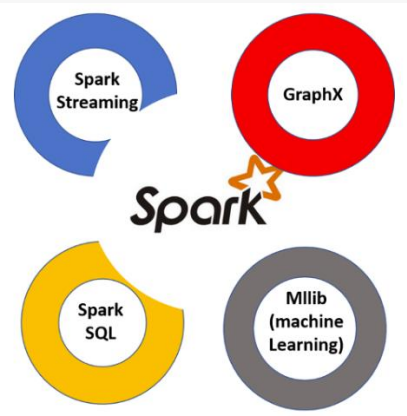
uma máquina local, mas utilizar máquinas existentes de plataformas nuvem sai barato.

Além disso, o Spark é ainda mais flexível que o Impala. O framework é desenvolvido em linguagem Scala, mas aceita comandos em Scala, Java, Python, R e SQL. Ele também permite trabalhar com os mais diversos sistemas de armazenamento como HDFS, Hive, HBase, dentre outros. Isso dá muita mais liberdade ao desenvolvedor. Além disso, o Spark pode ser configurado para rodar junto a uma interface de Notebook, tais como o Jupyter, Zeppelin ou Databricks, não necessitando de comandos no Terminal. Isso resulta em muito mais facilidade para se trabalhar com outros dados como DataFrames do Pandas, e em ambiente de notebook. Iremos utilizar em nossas aulas o framework PySpark, que integra o Spark e possibilita o uso de linguagem Python, para nossos comandos.

O Spark é constituído basicamente de quatro bibliotecas principais:

- Spark SQL: responsável pelo armazenamento e acesso aos dados;
- Spark Streaming: responsável pelo fluxo de dados em streaming;
- MLlib: biblioteca com modelos de Machine Learning;
- GraphX: biblioteca gráfica.

Figura 16 – Bibliotecas do Spark.



Quando rodando em vários nodos, dentro de um cluster, o Spark também precisa de um serviço de governança dos recursos. Ele pode ser configurado para ser monitorado pelo YARN, da mesma maneira que os outros frameworks estudados anteriormente. Mas o Spark também suporta outros serviços, para realizarem a mesma atividade de monitoramento, tais como o Apache Mesos ou Kubernetes. Não vamos entrar nesse detalhamento, mas focar na sua aplicação.

O primeiro passo, para fazer a conexão com o Spark, é preciso conectar ao nodo master. Isso é feito criando-se um objeto da classe SparkContext. Esse objeto é responsável por fazer a comunicação entre o programa e o ambiente e armazena as propriedades e configurações aplicadas. Como o Spark já vem instalado e configurado no Databricks, não é necessário executar esse passo nele, mas o SparkContext pode ser visualizado.

```
import findspark
findspark.init()

from pyspark.context import SparkContext
sc = SparkContext.getOrCreate()
```

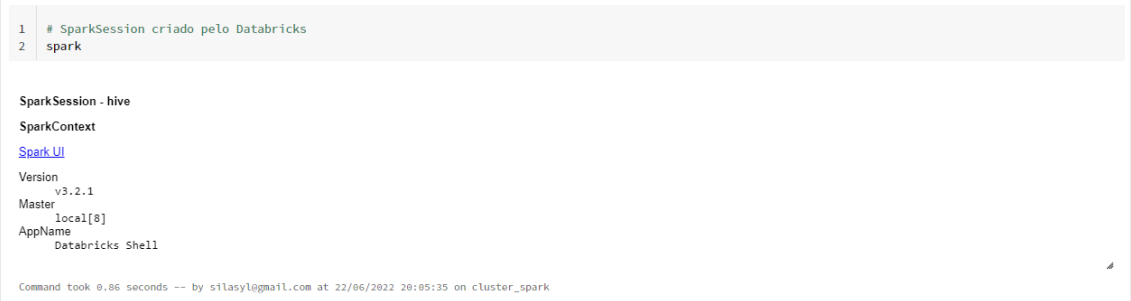
Figura 17 – Visualizando o SparkContext.

Após ter feito a conexão com o SparkContext, deve-se estabelecer uma interface ao nodo, usando o SparkSession. Esse passo também é pré-executado no Databricks e não precisa ser executado. O SparkSession também pode ser visualizado.

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.enableHiveSupport().getOrCreate()
```

Figura 18 – Visualizando o SparkSession.



7.2. Spark DataFrames

Vamos entender agora como o Spark SQL armazena os dados no Spark. Temos acesso aos objetos Spark Dataframes, que são objetos de alto nível, abstraindo as tabelas no nível RDD. Para aqueles familiares com o DataFrame da biblioteca Pandas, o Spark Dataframe é bastante semelhante em certos pontos.

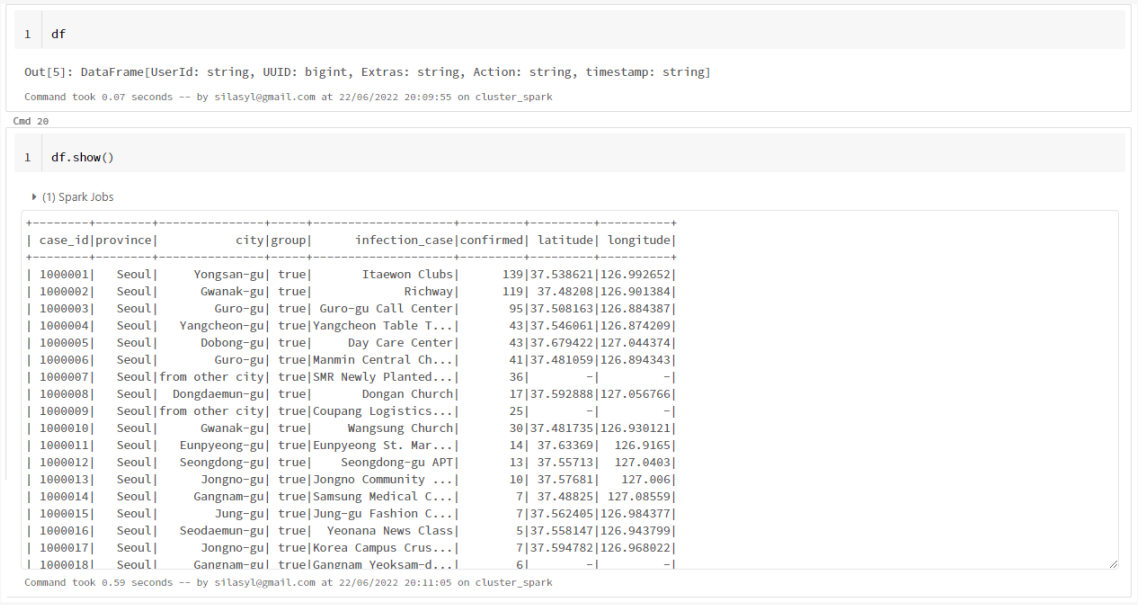
O Spark Dataframe possibilita empregarmos tanto linguagem Python para manipulação dos dados, como SQL. Ele funciona como uma tabela local e é visível apenas à máquina que a está manipulando. Na realidade, todas as operações em tabelas no Spark retornam um Spark Dataframe. Como dito anteriormente, os arquivos em RDD são imutáveis. Isso significa que a cada operação ou modificação, uma nova tabela Spark Dataframe é criada.

Para fazer leitura de um arquivo, podemos usar as diversas funções do spark.read ou usar o comando genérico, que aceita formatos diferentes. A seguir mostramos alguns dos argumentos possíveis de se usar.

```
df = spark.read.csv("file:///home/name.csv", header="true", inferSchema="true")
df = spark.read.json("file:////home/name.json")
df = spark.read.format("json").load("file:///home/name.json")
df = spark.read.format("csv").option("sep", ",").option("header", "true")
      .load("file:///home/name.csv")
```

Diferente do Pandas DataFrame, a chamada do Spark Dataframe retorna apenas o tipo do objeto. Para visualizar o seu conteúdo, usamos o comando df.show(). Para visualizar o schema da tabela, podemos usar o comando df.printSchema().

Figura 19 – Visualizando o Spark Dataframe.



```
1  df
```

```
Out[5]: DataFrame[UserId: string, UUID: bigint, Extras: string, Action: string, timestamp: string]
Command took 0.07 seconds -- by silasyl@gmail.com at 22/06/2022 20:09:55 on cluster_spark
```

```
Cmd 20
1  df.show()
```

▸ (1) Spark Jobs

case_id	province	city	group	infection_case	confirmed	latitude	longitude
1000001	Seoul	Yongsan-gu	true	Itaewon Clubs	139	37.538621	126.992652
1000002	Seoul	Gwanak-gu	true	Richway	119	37.48208	126.901384
1000003	Seoul	Guro-gu	true	Guro-gu Call Center	95	37.508163	126.884387
1000004	Seoul	Yangcheon-gu	true	Yangcheon Table T...	43	37.546061	126.874209
1000005	Seoul	Dobong-gu	true	Day Care Center	43	37.679422	127.044374
1000006	Seoul	Guro-gu	true	Manmin Central Ch...	41	37.481059	126.894343
1000007	Seoul	from other city	true	SMR Newly Planted...	36	-	-
1000008	Seoul	Dongdaemun-gu	true	Dongan Church	17	37.592888	127.056766
1000009	Seoul	from other city	true	Coupang Logistics...	25	-	-
1000010	Seoul	Gwanak-gu	true	Wangsung Church	30	37.481735	126.930121
1000011	Seoul	Eunpyeong-gu	true	Eunpyeong St. Mar...	14	37.63369	126.9165
1000012	Seoul	Seongdong-gu	true	Seongdong-gu APT	13	37.55713	127.0403
1000013	Seoul	Jongno-gu	true	Jongno Community ...	10	37.57681	127.006
1000014	Seoul	Gangnam-gu	true	Samsung Medical C...	7	37.48825	127.08559
1000015	Seoul	Jung-gu	true	Jung-gu Fashion C...	7	37.562405	126.984377
1000016	Seoul	Seodaemun-gu	true	Yeonana News Class	5	37.558147	126.943799
1000017	Seoul	Jongno-gu	true	Korea Campus Crus...	7	37.594782	126.968022
1000018	Seoul	Gangnam-gu	true	Gangnam Yeoksam-d...	6	-	-

```
Command took 0.59 seconds -- by silasyl@gmail.com at 22/06/2022 20:11:05 on cluster_spark
```

Figura 20 – Visualizando o schema do Spark Dataframe.

```
1  # Schema do df em formato de árvore
2  df.printSchema()
```

```
root
 |--  case_id: integer (nullable = true)
 |-- province: string (nullable = true)
 |-- city: string (nullable = true)
 |-- group: boolean (nullable = true)
 |-- infection_case: string (nullable = true)
 |-- confirmed: integer (nullable = true)
 |-- latitude: string (nullable = true)
 |-- longitude: string (nullable = true)

Command took 0.05 seconds -- by silasyl@gmail.com at 22/06/2022 20:11:41 on cluster_spark
```

Podemos realizar uma série de operações com o Spark Dataframe:

df.show()	mostra o dataframe
df.show(10)	mostra as primeiras 10 linhas
df.withColumn("col_nova", valor_col_nova)	operação com coluna
df.select("col")	seleciona uma coluna
df.select(df["col1"], df["col2"])	seleciona múltiplas colunas
df.filter(df['col'] > 0)	filtrar com condição

Como mencionado anteriormente, o Spark Dataframe aceita comandos em Python e em SQL. Para a aplicação do filtro, por exemplo, ele

aceita as duas notações Python, além de poder utilizar expressões SQL entre aspas simples ou dupla:

- df.filter(df['col'] > 0)
- df.filter(df.col > 0)
- df.filter('col > 0')

Podemos ainda aplicar operações de agrupamento, contagem, ordenação, além de valores médio, mínimo e máximo:

```
from pyspark.sql.functions import desc

df.groupBy('col')                          agrupa por coluna
df.groupBy('col').count()                  contar por agrupamento
df.groupBy('col').sum()                    somar por agrupamento
df.groupBy('col').sum().sort('sum')        ordenar por soma
df.groupBy('col').sum().sort(desc('sum'))  ordenar decrescente por soma
df.groupBy('col').min('col')               mínimo de uma coluna
df.groupBy('col').max('col')               máximo de uma coluna
df.groupBy('col').avg('col')               média de uma coluna
```

O Spark ainda permite que apliquemos comandos de join entre Spark Dataframes.

```
df_1.join(df_2, on='col', how='leftouter')    realiza left join
```

7.3. Spark Tables

Enquanto o Spark Dataframe abstrai as tabelas em alto nível, podemos acessar os Spark Tables também, manipulando diretamente nas tabelas armazenadas no RDD. Essas tabelas, também chamadas de views, são armazenadas no seu catálogo (*Catalog*). Para lidar com os Spark Tables, utilizamos unicamente comandos SQL, semelhante ao que era feito no Hive, mas como argumento do comando spark.sql().

As tabelas/views são a única forma de compartilhar dados com outros nodos no cluster. Existem os views temporários que são apagados quando o cluster é desligado e funcionam como tabelas temporárias e as

views externas, que permanecem salvas. Podemos visualizar as views disponíveis no RDD:

```
spark.catalog.listTables()
```

É possível criar uma view temporária a partir de um Spark Dataframe ou o contrário, criar um Spark Dataframe a partir de uma view:

```
df.createOrReplaceTempView("name_view")          criar view de df
spark_df = spark.table("name_view")              criar df de view
```

É possível inclusive salvar uma view em Pandas Dataframe ou ainda converter esse Pandas Dataframe em Spark Dataframe:

```
query = "SELECT * FROM name_view"
pandas_df = spark.sql(query).toPandas()          criar pd df de view
spark_df = spark.createDataFrame(pandas_df)      criar spark df de pandas df
```

Para realizar manipulações nas Spark Tables, devemos empregar o comando spark.sql("expressão"). As expressões são escritas em SQL, as operações são realizadas nas views, mas o retorno das operações é sempre no formato de Spark Dataframe. Podemos realizar as mesmas operações mostradas na sessão anterior com Spark Dataframe.

```
spark.sql("SELECT * FROM name_view")                    seleciona toda a tabela
spark.sql("SELECT * FROM name_view LIMIT 10")           mostra primeiras 10 linhas
spark.sql("SELECT col FROM name_view")                  seleciona uma coluna
spark.sql("SELECT col1, col2 FROM name_view")           seleciona múltiplas colunas
spark.sql("SELECT * FROM name_view WHERE col > 0")      filtra por condição

spark.sql("SELECT * FROM name_view GROUP BY col")       agrupa por coluna
spark.sql("SELECT COUNT(*) FROM name_view GROUP BY col")
                                                        contar por agrupamento
spark.sql("SELECT SUM(*) FROM name_view GROUP BY col")
                                                        somar por agrupamento
spark.sql("SELECT SUM(*) AS sum FROM name_view GROUP BY col ORDER BY sum")
                                                        ordenar por soma
spark.sql("SELECT SUM(*) AS sum FROM name_view GROUP BY col ORDER BY sum
DESC")                                                  ordenar decrescente por soma

spark.sql("SELECT MIN(col) FROM name_view GROUP BY col")
                                                        mínimo de uma coluna
spark.sql("SELECT MAX(col) FROM name_view GROUP BY col")
                                                        máximo de uma coluna
spark.sql("SELECT AVG(col) FROM name_view GROUP BY col")
                                                        média de uma coluna
```

Pode-se ver que o Spark é uma ferramenta bastante poderosa, além de possibilitar total liberdade ao desenvolvedor, que pode realizar operações tanto em Python quanto em SQL.

> Capítulo 8

Capítulo 8. Streaming de dados

Até agora tratamos de frameworks e análises de dados em batch, ou seja, em arquivos fechados, contendo os dados. Com o avanço da tecnologia e velocidade de novas informações, algumas vezes precisamos implementar soluções empregando processamento de dados em tempo real, ou streaming de dados. Consideramos processamento em tempo real quando há novas informações sendo atualizadas a intervalos de no máximo 30 segundos, mas estes podem ser tão pequenos quanto necessários, como a exemplo na casa de milissegundos.

8.1. Streaming de dados

Com o advento da tecnologia, atualmente encontramos cada vez mais oportunidades e necessidades de aplicarmos processamento em tempo real. A exemplo, temos sensores médicos; aparelhos de comunicação e transmissão de sinais; sensores de monitoramento; GPS para localização de veículos ou celulares ou ainda sistemas de videovigilância. Temos cada vez mais necessidade de projetar esses sistemas, que também passam a demandar mais em processamento e em velocidade.

Quando precisamos realizar processamento em tempo real, nosso sistema acaba por encontrar um trade-off entre a velocidade de processamento e a precisão do resultado. O sistema precisa receber, manipular, analisar e retornar uma resposta em tempo hábil e quanto menor é a janela de atualização dos dados ou quanto maior é a quantidade de dados, mais comprometido o sistema fica. O ideal é que o sistema realize o processamento no máximo na mesma taxa de atualização dos dados.

Para o processamento em tempo real ocorrer, é preciso definir uma janela temporal, a qual irá armazenar parcialmente os dados recebidos e constituirá os dados a serem analisados a cada passada. Os frameworks

próprios para streaming de dados realizam marcações de checkpoints e os dados vão sendo recuperados a cada checkpoint.

Plataformas nuvem oferecem serviços adicionais, que permeiam a base de um serviço de streaming de dados, mas que podem ser tão importantes quanto. Esses serviços podem ser segurança na comunicação; redundância e disponibilidade dos dados; lidar com dados críticos em checkpoints; adequação dos dados entre janelas e até governança dos processos paralelos.

Com a implementação do Spark, vários processos que antes eram impossíveis de se realizar em tempo real passaram a ser possíveis, devido à grande velocidade e capacidade de lidar com grande volume de dados. De fato, o Spark possui uma biblioteca própria para lidar com processamento em tempo real, o Spark Streaming. Antes de estudá-lo em detalhes, vamos estudar agora como empregá-lo.

8.2. Spark Streaming

O Spark Streaming é a biblioteca desenvolvida em cima do Spark para prover seus serviços, mas para processamento em tempo real. A sua estrutura, de fato, é bem semelhante à estrutura básica do Spark, o que o torna uma ferramenta bastante fácil de se aprender e utilizar.

No Spark Streaming utilizamos o StreamingContext que faz o mesmo papel do SparkContext, mas no streaming. Ao invés de utilizarmos os RDDs e armazenarmos dados estáticos, utilizamos os DStreams (*discretized streams*), que são estruturas RDDs em janelas temporais de dados. O StreamingContext ainda precisa do SparkContext para manter a comunicação e as configurações. No StreamingContext podemos configurar parâmetros específicos do streaming, como o tamanho da janela. Caso necessário, podemos empregar pós-processamento em RDD através do Spark convencional.

Uma vez que o streaming de dados utiliza um fluxo contínuo de dados como entrada dos DStreams, o StreamingContext possui configurações adicionais para realizar as configurações de comunicação. Normalmente, as fontes de dados são APIs de streaming (como do Twitter), soquetes TCP ou sistemas de mensagens. A comunicação com soquetes TCP ocorre através do socketTextStream, pelo seu hostname e porta:

```
StreamingContext.socketTextStream(hostname,
                                  port,
                                  storageLevel)
```

A seguir vemos um exemplo de como conectar um StreamingContext a um Spark Context e configurar a linha de conexão a uma porta TCP:

```
from pyspark.streaming import StreamingContext

ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream('localhost', 9999)
```

Após fazer a conexão pelo StreamingContext, definimos as operações que deverão ser realizadas. No exemplo a seguir, realizamos as operações de Mapper e Reducer do Spark Streaming, que são operações análogas ao Mapper e Reducer do MapReduce, mas são executados em memória RAM, nos DStreams. Na nossa operação, fazemos a contagem de palavras. Empregamos o uso do lambda no Python para que as operações sejam replicadas por todas as palavras da janela de dados:

```
counts = lines.flatMap(lambda line: line.split(" ").
                 map(lambda word: (word, 1)).
                 reduceByKey(lambda a, b: a + b)
counts.pprint()
```

Por fim, iniciamos o processo pelo comando ssc.start(). Para interromper o processo, podemos programar previamente com o uso do comando ssc.stop() ou esperar pelo fim do processo com ssc.awaitTermination(). É muito importante programar corretamente o

encerramento do processo streaming para não haver consumo excessivo de recursos desnecessários.

```
ssc.start()
ssc.awaitTermination()
```

8.3. Structured Streaming

A partir da versão 2.2 do Spark, lançado em 2021, a equipe desenvolveu o Structured Streaming, uma engine baseada no Spark SQL, para otimizar e solucionar alguns dos problemas encontrados no Spark Streaming. Um dos maiores problemas encontrados pelo Spark Streaming era que ele ainda realizava as operações em batches, cada DStream era uma abstração da janela temporal e o sistema rodava operações em RDD, correspondendo a cada janela. Apesar de tudo, para processos que exigiam resposta realmente em tempo real, era uma dificuldade de se atingir. Uma outra dificuldade era a programação dos processos internos, para processamento em cada janela, não era intuitivo e tão fácil de se fazer como as operações em Spark Dataframes e Tables, como vimos. O Structured Streaming surgiu para otimizar e melhorar esses pontos.

Apesar do Structured Streaming se sobressair, ainda está bem recente e o Spark Streaming ainda não está obsoleto, portanto, Spark Streaming ainda pode ser aplicado. O Structured Streaming traz uma série de novas funcionalidades e facilidades. Sem entrar muito em detalhes, a grande diferença do Structured Streaming é que ele realiza operações de forma individual e cada saída é uma linha, que é adicionada a uma tabela imutável no RDD. Isso permite mais rapidez e se aproxima ainda mais do processamento em tempo real. Além disso, ele aceita comandos em SQL, como aplicamos ao Spark Tables. O Structured Streaming também suporta, além do Spark Dataframes, o Spark Datasets. O Dataset é uma estrutura semelhante ao Dataframe e funciona de modo semelhante, mas contém melhorias como segurança para tipos de variáveis e suporte a operações mais complexas.

A utilização do Structured Streaming é bem semelhante aos comandos usuais no Spark, com Dataframes. Para fazer a leitura de dados, em batch, usamos o comando spark.read. No caso do Structured Streaming, usamos o comando spark.readStream e os argumentos são quase os mesmos. Veja um exemplo a seguir:

```
static_df = spark.read.
      schema(jsonSchema).
      json(input_data)

stream_df = spark.readStream.
      schema(jsonSchema).
      option("maxFilesPerTrigger", 1).
      json(input_data)
```

Nesse exemplo empregamos um argumento a mais para a função spark.readStream, a fim de configurar a quantidade de dados a serem lidos por janelamento, mas o resto dos argumentos são os mesmos.

Para as operações no Structured Streaming, podemos utilizar os comandos análogos aos comandos SQL:

```
df.select("col")
df.filter("col > 0")
df.where("col = 0")
df.withColumn("col_nova", valor_col_nova)
```

Da mesma forma que estudado no Spark Dataframe, podemos realizar operações de agregações e matemáticas como mínimo, máximo e média.

> Capítulo 9

Capítulo 9. Spark MLlib

Como já vimos, o Spark tem a vantagem de ser capaz de manipular e analisar quantidades volumosos de dados com muita velocidade. Na área de dados, as técnicas de Machine Learning são de suma importância, mas muitas vezes seu processamento e tempo de execução crescem quase que exponencialmente com a quantidade de dados. O MLlib disponibiliza bibliotecas prontas para aplicar diversos modelos de Machine Learning, se aproveitando da estrutura e implementação em Spark.

9.1. Introdução ao MLlib

O MLlib é a biblioteca Spark que contém implementações de diversos algoritmos de Machine Learning, os quais podem ser aplicados diretamente nos dados em RDD e utilizar o processamento em memória RAM para as análises.

Dentre os modelos dessa biblioteca, tem-se modelos para todas as aplicações em ML:

- Classificação;
- Regressão;
- Clusterização;
- Modelagem;
- Decomposição;
- Testes estatísticos;
- Redes neurais.

A biblioteca MLlib se aproveita do poder e velocidade de processamento e realiza as operações necessárias em memória RAM de maneira distribuída. Isso dá uma grande vantagem às análises necessárias.

O nosso intuito aqui, entretanto, não é aprender esses modelos e nem como aplicá-los, mas é necessário conhecer o pré-processamento necessário aos dados, que partem de uma tabela comum no RDD ou Dataframe, para alimentar os modelos do MLlib. Com esse conhecimento, é possível construir estruturas para adequar os dados e estarem prontos para serem usados por modelos do MLlib.

9.2. Preparação dos dados

Vamos aprender aqui como construir uma pipeline simples de Machine Learning, utilizando as principais ferramentas do MLlib, a fim de adequar os dados que temos para os diversos modelos do MLlib.

O módulo principal que iremos utilizar é o pyspark.ml e ele utiliza duas classes: o Transformer e o Estimator. Quase todas as outras classes da biblioteca são construídas em cima de uma dessas duas classes.

A classe Transformer possui o método .transform() que realiza alguma operação em um Spark Dataframe e retorna um novo Dataframe. Exemplos de métodos que empregam o Transformer são métodos que adicionam ou somam colunas, tais como o Bucketizer, que cria uma coluna com valores discretos a partir de valores contínuos; ou métodos que extraem valores novos como o PCA, que reduz a dimensionalidade de um dataset e o representa em novas colunas.

A classe Estimator possui o método .fit(), que representa métodos quando precisamos rodar operações em cima de um grupo de dados. Esse método usa Spark Dataframes como entrada, mas não retorna um Dataframe e sim um objeto do tipo modelo. Normalmente esses métodos

compreendem o treinamento de modelos, e os dados são usados para se extrair parâmetros de um modelo.

Um fator extremamente importante é que os modelos de Machine Learning do MLlib lidam apenas com valores numéricos. Com essa implementação, a biblioteca tem a vantagem de ganho de velocidade e processamento. Ele não precisa perder tempo detectando e lidando com diferentes tipos de variáveis. É papel do engenheiro/arquiteto de dados adequar os dados para que eles possam alimentar os modelos sem conflitos.

Diante disso, vamos estudar como transformar nossos dados, que podem ter os mais variados formatos, como texto (*string*), booleanos ou classes, todos em numéricos. Esses procedimentos podem ser realizados dentro do MLlib, ele possui ferramentas próprias para esse fim.

A primeira coisa a se saber é que os modelos MLlib usam dois tipos numéricos: os inteiros (*integer*) e os números fracionários (*double*). Todos os outros formatos precisam ser transformados para um desses dois. A função .cast() modifica uma coluna no formato desejável, então usaremos .cast('integer') ou .cast('double'). Isso é útil para transformar colunas que tenham números em formato string, mas podem ser transformados em números.

```
df = df.withColumn("col_numerico", df.col_string.cast('integer')
```

Para transformar dados booleanos em numéricos é fácil. Primeiro utilizamos uma lógica para detectar a presença do verdade/falso, seja por comparação com alguma lógica ou emprego de true/false. A seguir convertemos essa coluna em integer, uma vez que o próprio true/false do Dataframe é armazenado como 1/0.

```
df = df.withColumn("res_bool", df.col > 10)
df = df.withColumn("bool", df.res_bool.cast('integer'))
```

Para lidarmos com variáveis do tipo texto é preciso um pouco mais de trabalho. Nós utilizamos as estruturas chamadas one-hot vectors, que se

constituem de vetores de zero, com 1 para ocorrência em cada palavra diferente. O vetor one-hot vector tem o comprimento igual ao número de palavras a serem convertidas e é uma matriz esparsa, ou seja, em sua maioria com zeros.

Para realizar essa operação, fazemos duas etapas. A primeira etapa é empregar um StringIndexer, um Estimator, seguido de um Transformer, que retorna um Dataframe com os números correspondentes a cada string a ser mapeado. A segunda etapa é empregar o OneHotEncoder, que constitui de um Estimator, seguido de Transformer, que retorna o one-hot vector correspondente aos números encontrados no passo anterior. Isso tudo pode ser envelopado por um objeto Pipeline.

```
import pyspark.ml.feature

indexer = feature.StringIndexer(inputCol='col', outputCol='index')
encoder = feature.OneHotEncoder(inputCol='index', outputCol='factor')
```

Observe que aqui não declaramos o Dataframe a se utilizar, estamos montando apenas a estrutura. O Dataframe irá entrar posteriormente no pipeline. No nosso exemplo, a coluna 'col' possui valores em string, que serão indexados em números, representados na coluna 'index', que serão transformados em one-hot vector na coluna 'factor'.

O passo seguinte é construir um objeto a partir da função VectorAssembler, que irá agrupar todas as colunas de features e a saída do nosso modelo. Estes podem ser de um Dataframe ou de vários diferentes, mas se vierem de fontes diferentes, precisam passar por Joins primeiro.

```
import pyspark.ml.feature

assembler = feature.VectorAssembler(inputCols=['col1', 'col2', 'col3'],
                                    outputCol='labels')
```

Por fim, montamos nossa pipeline com todos os processos. Nele colocamos em ordem as funções que queremos. Podemos juntar vários

indexers e encoders, um para cada coluna do nosso Dataframe. Ao final juntamos o assembler.

```
import pyspark.ml.feature

pipe = feature.Pipeline(stages=[indexer, encoder, assembler])
```

Para executar a Pipeline de fato, usamos seus comandos de fit() e transform() empregando o Dataframe de dados que possuímos.

```
data = pipe.fit(df).transform(df)
```

A partir daí os dados encontrados em data estão prontos para serem usados pelo cientista de dados. A biblioteca MLlib possui ainda outras funções para etapas posteriores, tais como separação de dados em treino/teste, cross-validation, as técnicas de Machine Learning etc., mas nossa função aqui está concluída, como arquitetos de big data.

Referências

AMAZON WEB SERVICES. AWS, 2022. Disponível em: <https://aws.amazon.com/pt/>. Acesso em: 05 ago. 2022.

BAKSHI, Aphish. Apache Hadoop HDFS Architecture. Edureka, 2021. Disponível em: <https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/>. Acesso em: 05 ago. 2022.

BOUAMAMA, Samah. Research Gate, 2018. Disponível em: <https://www.researchgate.net/figure/Data-replication-using-HDFS_fig3_325608248>. Acesso em: 05 ago. 2022.

CARDOSO, Carlos Alberto Rocha. Um (quase) guia (nada) completo sobre o Hadoop. Medium: DATA ENGINEER BR, 2019. Disponível em: <https://medium.com/dataengineerbr/um-quase-guia-nada-completo-sobre-o-hadoop-a3eeee170deb>. Acesso em: 05 ago. 2022.

DATABRICKS. Databricks Community Edition. Disponível em: <https://community.cloud.databricks.com/login.html>. Acesso em: 05 ago. 2022.

DATABRICKS. Home. Disponível em: <https://databricks.com/>. Acesso em: 05 ago. 2022.

DIRICK, Lore; SOLOMON, Nick. Introduction to PySpark. Datacamp. Disponível em: <https://app.datacamp.com/learn/courses/introduction-to-pyspark/>. Acesso em: 26 jun. 2022.

DOMMATA, Sreehas. Computer cluster. Medium, 2017. Disponível em: <https://medium.com/lvs-load-balance-clustering-configuration-on/what-is-a-computer-cluster-c4c219f4f6d9>. Acesso em: 05 ago. 2022.

JIANBO, Zhang. Research Gate, 2021. Disponível em: <https://www.researchgate.net/figure/The-overview-of-the-Hadoop-Distributed-File-System-HDFS_fig4_348387085>. Acesso em: 05 ago. 2022.

KAGGLE. Disponível em: <https://www.kaggle.com/>. Acesso em: 05 ago. 2022.

KHARAZI, Darius. MapReduce. Disponível em: <https://dkharazi.github.io/notes/de/etl/mapreduce>. Acesso em: 05 ago. 2022.

MANMOHAN. Crayon. Components of Hadoop Architecture & Frameworks used for Data Science. Crayon, 2018. Disponível em: <https://www.crayondata.com/basic-components-of-hadoop-architecture-frameworks-used-for-data-science/>. Acesso em: 05 ago. 2022.

NIÑO, Mikel. Chronology of antecedentes, origin and development of Big Data. Blog Pessoal, 2016. Disponível em <https://www.mikelnino.com/2016/03/chronology-big data.html?m=1/>. Acesso em: 05 ago. 2022.

ORGERA, Scott. The 6 Best Virtual Machine Software Programs of 2022. Lifewire, 2021. Disponível em: <https://www.lifewire.com/best-virtual-machine-software-4147437>. Acesso em: 05 ago. 2022.

SAVE IN CLOUD. Java na nuvem, com alta disponibilidade e escalabilidade! 2021. Disponível em <https://saveincloud.com/pt/blog/web-aplicacao/java-escalavel-e-com-alta-disponibilidade-na-nuvem/>. Acesso em: 05 ago. 2022.

SINGH, Ranvir. How to SSH into your VirtualBox Guest. Linuxhint, 2018. Disponível em <https://linuxhint.com/ssh_virtualbox_guest/>. Acesso em: 05 ago. 2022.

SINHA, Shubham. Install Hadoop: Setting up a Single Node Hadoop Cluster. Edureka, 2022. Disponível em <https://www.edureka.co/blog/install-hadoop-single-node-hadoop-cluster/>. Acesso em: 05 ago. 2022.

SOFTWARE TESTING HELP. Hadoop HDFS – Hadoop Distributed File System. 2022. Disponível em <https://www.softwaretestinghelp.com/hadoop-distributed-file-system/>. Acesso em: 05 ago. 2022.

STACK EXCHANGE INC. Stack Overflow. 2022. Disponível em <https://stackoverflow.com/>. Acesso em: 05 ago. 2022.

STEDMAN, Craig. Apache Hadoop YARN. TechTarget, 2020. Disponível em: <https://www.techtarget.com/searchdatamanagement/definition/Apache-Hadoop-YARN-Yet-Another-Resource-Negotiator>. Acesso em: 05 ago. 2022.

THE APACHE SOFTWARE FOUNDATION. Apache Hadoop. 2022. Disponível em: <https://hadoop.apache.org/>. Acesso em: 05 ago. 2022.

THE APACHE SOFTWARE FOUNDATION. Apache Hive. 2014. Disponível em: <https://hive.apache.org/>. Acesso em: 05 ago. 2022.

THE APACHE SOFTWARE FOUNDATION. Apache Impala. 2022. Disponível em: <https://impala.apache.org/>. Acesso em: 05 ago. 2022.

THE APACHE SOFTWARE FOUNDATION. Apache Spark. 2018. Disponível em: <https://spark.apache.org/>. Acesso em: 05 ago. 2022.

THE APACHE SOFTWARE FOUNDATION. PySpark Documentation. Disponível em: <https://spark.apache.org/docs/latest/api/python/>. Acesso em: 05 ago. 2022.