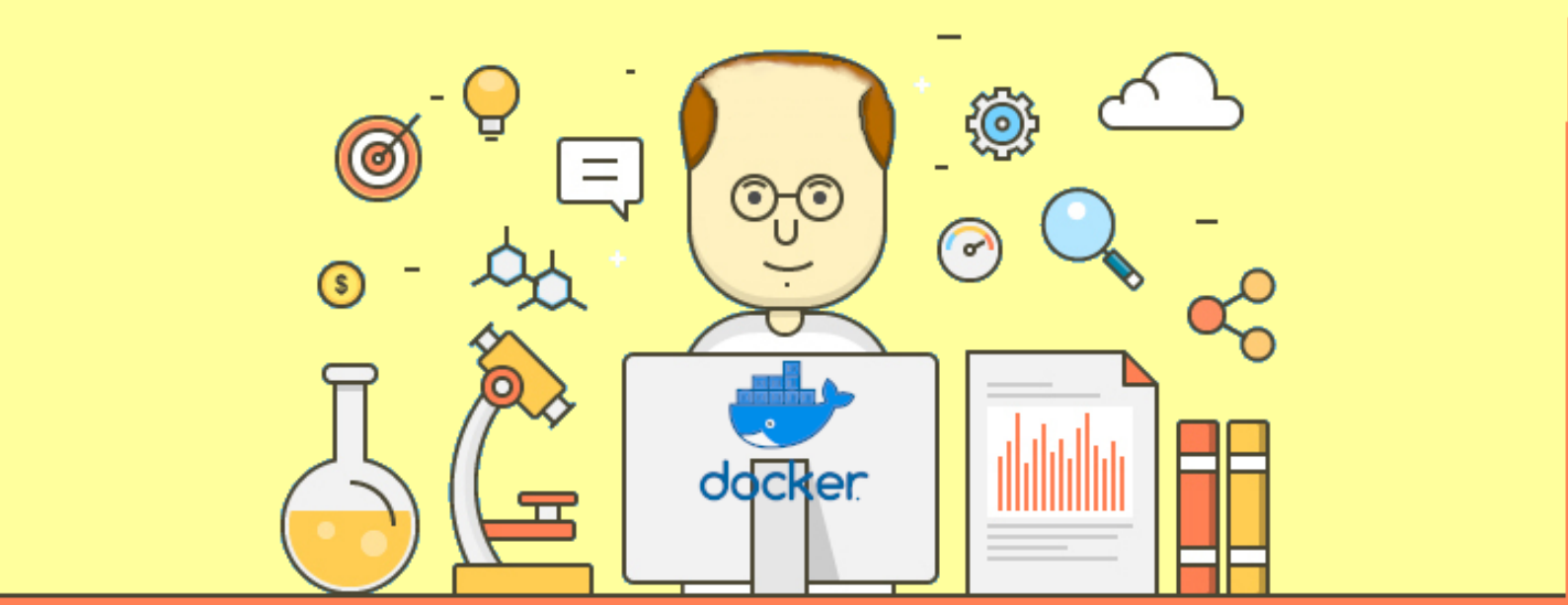
Apache Spark na Prática

Com PySpark, Docker Compose e JupyterLab

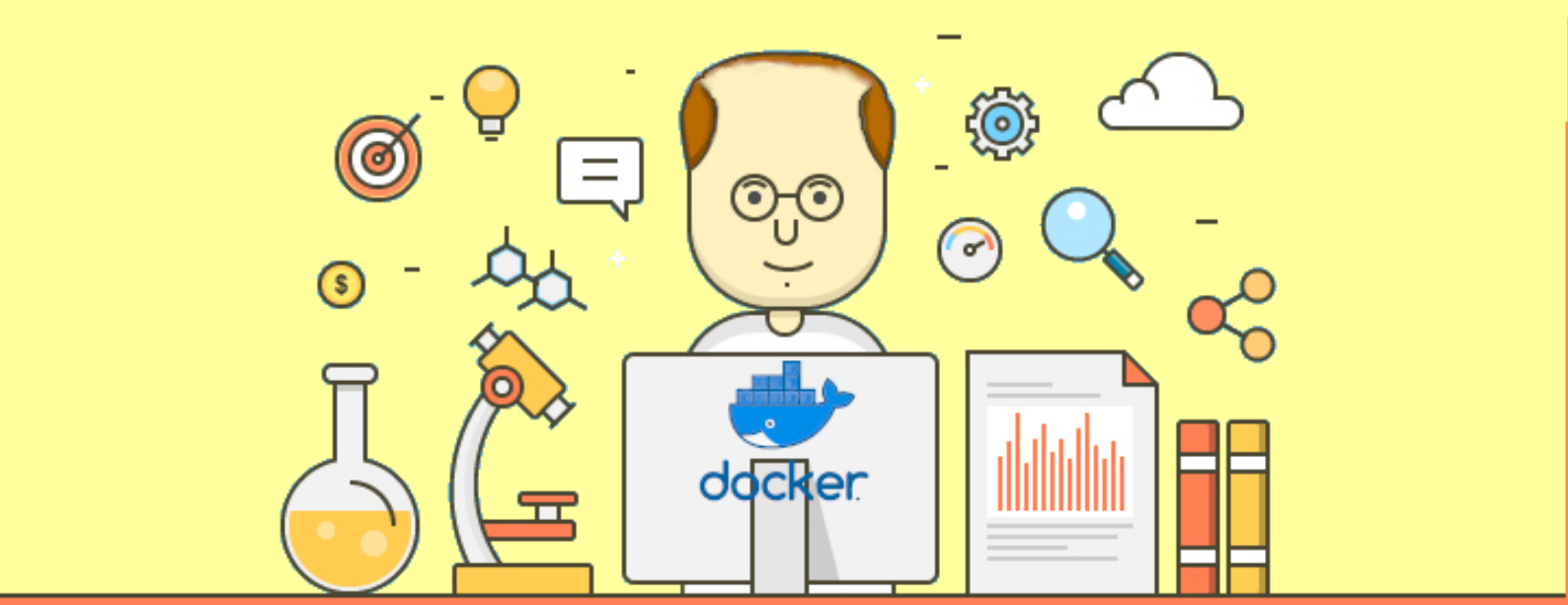
Fernando Anselmo

PUBLICAÇÃO INDEPENDENTE

http:\fernandoanselmo.orgfree.com



Sumário



1. Entendimento Geral

"A tecnologia é uma faísca criativa que ilumina as mentes e transforma a maneira como vemos e interagimos com o mundo."(Steve Jobs)

1.1 Do que trata esse livro?

Apache Spark (a palavra pode ser traduzida para "centelha"ou faísca") pois seu objetivo é dar velocidade e eficiência ao processamento de dados. Assim como uma centelha energética, seu objetivo é de uma maior produtividade e desempenho em projetos de análise e processamento de informações. Com sua capacidade de processar grandes volumes de dados de forma distribuída, Apache Spark acelera o ritmo das descobertas, permite que empresas se destaquem na era da transformação digital. Com essa poderosa ferramenta, é possível acender a faísca da inovação e alcançar resultados surpreendentes, explorando todo o potencial dos dados para impulsionar o crescimento e a competitividade.

Apache Spark (para abreviar pretendo chamá-lo somente de Spark) possui uma poderosa e veloz arquitetura para processamento em tempo real, pois sua principal vantagem reside na capacidade de realizar cálculos em memória, e isso acelera significativamente a análise de dados. Surgindo como resposta à limitação do **Apache Hadoop MapReduce**, que se restringia a processamentos em lote, Spark preencheu a lacuna ao introduzir o processamento de *stream* (fila de dados) em tempo real. Tornou-se inclusive um componente essencial desse ambiente.

Possui uma arquitetura distribuída, isso significa que pode ser escalado horizontalmente em vários nós de computação, e assim possibilitar o processamento eficiente para grandes volumes de dados. Adequado para lidar com conjuntos de dados que varia de terabytes ou até petabytes.

1.2 O que é Apache Spark?

A arquitetura do Spark foi projetada para superar as limitações do **Hadoop MapReduce** e oferecer um ambiente de processamento de dados mais rápido e flexível. Ao contrário do **Hadoop MapReduce**, que requer a leitura e escrita repetida de dados no disco, Spark utiliza uma estrutura de processamento em memória, isso permite operações mais ágeis e eficientes. Essa abordagem executa análises mais complexas em tempo real, acelera o tempo de resposta e abre novas possibilidades de aplicação.

Spark é uma plataforma com processamento de dados que possui seu próprio gerenciador de cluster, isso permite hospedar aplicativos diretamente nele. Além disso, faz uso do **Hadoop**, aproveitando o sistema de arquivo distribuído **HDFS** (*Hadoop Distributed File System*) para armazenamento e o **YARN** para executar os aplicativos distribuídos.

Originalmente foi desenvolvido para usar a linguagem de programação **Scala**, porém expandiu suas funcionalidades para oferecer suporte a **Python** por meio de uma ferramenta chamada **PySpark**. Com essa ferramenta, os usuários podem trabalhar com **RDD** (*Resilient Distributed Dataset*), que é uma abstração de dados resiliente distribuída, na linguagem de programação Python. E isso amplia as possibilidades de desenvolvimento e análise de dados, permite aos programadores utilizar a linguagem de sua preferência para aproveitar o poder e a flexibilidade do Spark.

1.3 Montagem do Ambiente

Atualmente, presenciamos uma revolução no campo da Ciência de Dados, com constantes avanços e atualizações nas ferramentas utilizadas nessa área. No entanto, é importante destacar que essas atualizações rápidas podem trazer consigo problemas que afetam diretamente o sistema operacional. Surge então a pergunta: Como podemos nos manter atualizados e seguros ao mesmo tempo? A resposta que se destaca como a mais coerente é a utilização da técnica de conteinerização para resolver esse problema.

A conteinerização é uma abordagem que permite empacotar um aplicativo e todas as suas dependências em um contêiner isolado. Com isso, é possível garantir que as atualizações e modificações necessárias sejam feitas dentro do contêiner, sem afetar o nosso sistema operacional. Essa abordagem proporciona uma camada adicional de segurança, uma vez que os contêineres são isolados uns dos outros e do ambiente de hospedagem.

Ao utilizar a conteinerização, é possível manter-se atualizado com as últimas versões das ferramentas, aproveitando os benefícios das atualizações sem correr o risco de afetar a estabilidade do sistema operacional. Além disso, os contêineres podem ser facilmente replicados e distribuídos em diferentes ambientes, garantindo consistência e portabilidade.

Uma das ferramentas populares para conteinerização é o **Docker**, que permite criar, implantar e gerenciar contêineres de forma eficiente. Com o Docker, é possível construir imagens personalizadas contendo as bibliotecas e dependências necessárias para suas aplicações de Data Science, facilitando a reprodução e compartilhamento do ambiente de desenvolvimento.

Figura 1.1: Docker Compose para Gerenciamento de Contêineres

Docker Compose é uma ferramenta que permite definir e gerenciar múltiplos contêineres com uma

simples configuração de arquivos **YAML**. Projetado para simplificar o processo de criação e execução de aplicativos compostos por vários contêineres Docker interagindo entre si. Podemos simplesmente descrever a configuração dos aplicativos, incluindo quais contêineres serão usados, as redes e volumes necessários, bem como as variáveis de ambiente e outras configurações específicas.

Docker Compose oferece uma série de benefícios. Simplifica o processo para a implantação de aplicativos compostos por múltiplos contêineres, pois tudo é definido em um único arquivo. Isso torna o compartilhamento e a colaboração mais fáceis, pois toda a configuração está documentada em um local central. Além disso, facilita a escalabilidade do aplicativo. Podemos especificar número de réplicas para determinado serviço e o Docker Compose se encarrega de criar e gerenciar os contêineres necessários para atender a essa demanda.

Porquê vários contêineres? Pelo simples fato que precisamos de um editor (no caso o escolhido foi o JupyterLab), um **Spark Master** e alguns **Spark Workers** além obviamente de um **HDFS** para realizar as transações. Basicamente funciona assim:

- No JupyterLab podemos dar o comando para realizar qualquer atividade.
- Essa é interceptada pelo **Spark Master** que a repassará para os **Spark Workers** disponíveis que se encarregarão de sua execução.

Graficamente seria essa imagem:

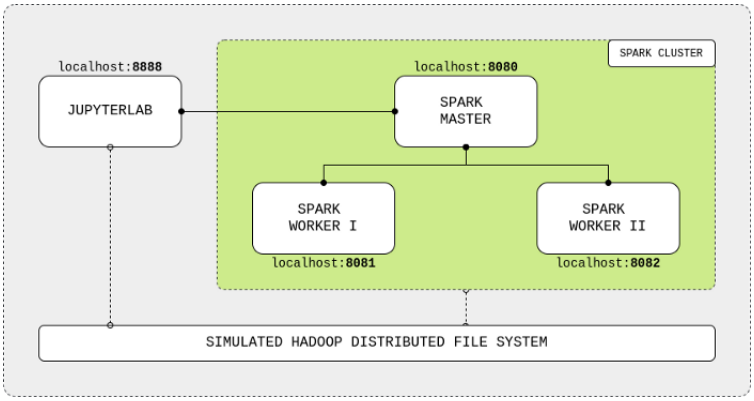


Figura 1.2: Arquitetura do Ambiente

A ideia original veio do **André Perez** que consiste na montagem de imagens Docker e a criação de um ambiente através do Docker Compose. O único detalhe que adicionei foi transpor de forma mais fácil para o iniciante. Primeiro devemos criar uma pasta para comportar os arquivos necessários que conterão os códigos para a criação das dos contêineres necessários e suas respectivas imagens.

Parte 1 - Criar um arquivo chamado **cluster-base.Dockerfile** (**ATENÇÃO**: Respeite as letras maiúsculas e minúsculas dos nomes e de seu conteúdo, tudo aqui é *case-sensitive*) que servirá como camada base com o Sistema Operacional e o Python 3, através da seguinte codificação:

```
FROM openjdk:8-jre-slim

RUN mkdir -p /opt/workspace && \
    apt-get update -y && \
    apt-get install -y python3 && \
```

```
    ln -s /usr/bin/python3 /usr/bin/python && \
    rm -rf /var/lib/apt/lists/*

ENV SHARED_WORKSPACE=/opt/workspace
VOLUME /opt/workspace
CMD ["bash"]
```

Para sua execução digite o seguinte comando:
`$ docker build -f cluster-base.Dockerfile -t cluster-base .`

Parte 2 - Criar um arquivo chamado **spark-base.Dockerfile** que é a nossa camada Spark, através da seguinte codificação:

```
FROM cluster-base

RUN apt-get update -y && \
    apt-get install -y curl && \
    curl
    https://archive.apache.org/dist/spark/spark-3.4.1/spark-3.4.1-bin-hadoop3.tgz -o
    spark.tgz && \
    tar -xf spark.tgz && \
    mv spark-3.4.1-bin-hadoop3 /usr/bin/ && \
    mkdir /usr/bin/spark-3.4.1-bin-hadoop3/logs && \
    rm spark.tgz

ENV SPARK_HOME /usr/bin/spark-3.4.1-bin-hadoop3
ENV SPARK_MASTER_HOST spark-master
ENV SPARK_MASTER_PORT 7077
ENV PYSPARK_PYTHON python3
WORKDIR ${SPARK_HOME}
```

Para sua execução digite o seguinte comando:
`$ docker build -f spark-base.Dockerfile -t spark-base .`

Parte 3 - Criar um arquivo chamado **spark-master.Dockerfile** que é o nosso Spark Master, através da seguinte codificação:

```
FROM spark-base
EXPOSE 8080 ${SPARK_MASTER_PORT}
CMD bin/spark-class org.apache.spark.deploy.master.Master >> logs/spark-master.out
```

Para sua execução digite o seguinte comando:
`$ docker build -f spark-master.Dockerfile -t spark-master .`

Parte 4 - Criar um arquivo chamado **spark-worker.Dockerfile** que é o nosso Spark Worker, através da seguinte codificação:

```
FROM spark-base
EXPOSE 8081
CMD bin/spark-class org.apache.spark.deploy.worker.Worker
    spark://${SPARK_MASTER_HOST}:${SPARK_MASTER_PORT} >> logs/spark-worker.out
```

Para sua execução digite o seguinte comando:
`$ docker build -f spark-worker.Dockerfile -t spark-worker .`

Parte 5 - Criar um arquivo chamado **jupyterlab.Dockerfile** que é o nosso editor, através da seguinte codificação:

```
FROM cluster-base

RUN apt-get update -y
RUN apt-get install -y python3-pip
RUN pip3 install pyspark==3.4.1
RUN pip3 install jupyterlab

EXPOSE 8888
WORKDIR ${SHARED_WORKSPACE}
CMD jupyter lab --ip=0.0.0.0 --port=8888 --no-browser --allow-root
    --NotebookApp.token=spark
```

Para sua execução digite o seguinte comando:
`$ docker build -f jupyterlab.Dockerfile -t jupyterlab .`

Após essa etapa, temos cinco imagens que possuem uma dependência interligada, representada graficamente da seguinte forma:

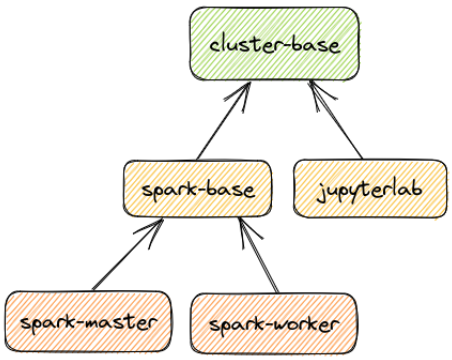


Figura 1.3: Dependência das imagens geradas

Basicamente, precisamos de três delas: **jupyterlab** (editor de códigos), **spark-master** e **spark-worker**. No entanto, é mais conveniente criar uma imagem base chamada **cluster-base**, que contém as informações do sistema operacional utilizado. Além disso, uma segunda imagem fundamental para o Spark é obtida pela **spark-base**, que instala uma instância do Spark sobre o Hadoop,

> **Dica 1.1: Porquê no Linux?.** Sinto se você é fã do Windows ou Mac, nada contra esses sistemas operacionais, porém preciso de um ambiente robusto e que o Docker seja nativo, se conseguir transpor todos os comandos mostrados aqui no shell para esses ambientes não terá muitos problemas. Aqui utilizei o **Linux Ubuntu**.

O último passo é o mais simples de todos e aí que entra o **docker compose**, precisamos a partir dessas imagens subir um contêiner da imagem do **jupyterlab**, mais um com a imagem do **spark-master** e ao menos dois (pode futuramente adicionar novos se assim desejar) com **spark-worker** e ligá-las na

rede além de passar alguns passos para sua criação. Isso tudo será feito através de um arquivo chamado **docker-compose.yml** que possui o seguinte conteúdo:

```
version: "3.6"
services:
  jupyterlab:
    image: jupyterlab
    container_name: jupyterlab
    ports:
      - 8888:8888
    volumes:
      - /home/fernando/meu-spark:/opt/workspace
  spark-master:
    image: spark-master
    container_name: spark-master
    ports:
      - 8080:8080
      - 7077:7077
    volumes:
      - /home/fernando/meu-spark:/opt/workspace
  spark-worker-1:
    image: spark-worker
    container_name: spark-worker-1
    environment:
      - SPARK_WORKER_CORES=1
      - SPARK_WORKER_MEMORY=512m
    ports:
      - 8081:8081
    volumes:
      - /home/fernando/meu-spark:/opt/workspace
    depends_on:
      - spark-master
  spark-worker-2:
    image: spark-worker
    container_name: spark-worker-2
    environment:
      - SPARK_WORKER_CORES=1
      - SPARK_WORKER_MEMORY=512m
    ports:
      - 8082:8081
    volumes:
      - /home/fernando/meu-spark:/opt/workspace
    depends_on:
      - spark-master
```

A pasta "/home/fernando/meu-spark"se refere a um caminho na minha máquina, modifique-o conforme seu sistema e endereçamento.

Para cada **spark-worker** estabelecemos memória de 512 Mb, isso será necessário para que tenham boa performance, mas como citado aumente o número conforme a necessidade. O primeiro usará a porta 8081 e segundo a 8082. O **spark-marter** responde nas portas 8080 e 7077.

Quanto a rede, para rodarmos os contêineres iremos necessitar de uma rede, o nome dessa será a pasta que se encontra este arquivo YAML, sendo assim, vamos supor que a pasta se chama "pyspark-docker",

então o seguinte comando deve ser executado:
`$ docker network create pyspark-docker_default`

Para criar os nossos contêineres:
`$ docker-compose create`

Para executá-los:
`$ docker-compose start`

Como resposta deve ter recebido:
```
Starting jupyterlab ...  done
Starting spark-master ...  done
Starting spark-worker-1 ...  done
Starting spark-worker-2 ...  done
```

Se digitar o comando:
`$ docker ps`

Verá que os quatro contêineres estão ativos em suas respectivas portas, para pará-los:
`$ docker-compose stop`

Lembre-se que sempre estes comandos devem ser dados nesta pasta específica aonde está o arquivo **docker-compose.ylm**.

> **Dica 1.2: Deu erro!.** Provavelmente o nome da sua rede está errado, verifique se o erro é esse aqui: ERROR: for jupyterlab Cannot start service jupyterlab: network "Nome da Rede"not found.

Com os contêineres ativos, abrir o navegador no endereço `http://localhost:8888` e obtemos o seguinte resultado:

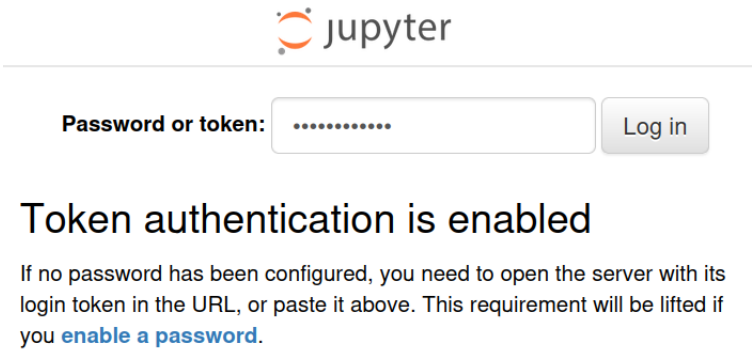


Figura 1.4: Jupyter solicitando o Token

A senha do token está definida no arquivo de criação da imagem do JupyterLab como "spark", após informá-la o JupyterLab pronto para trabalharmos:

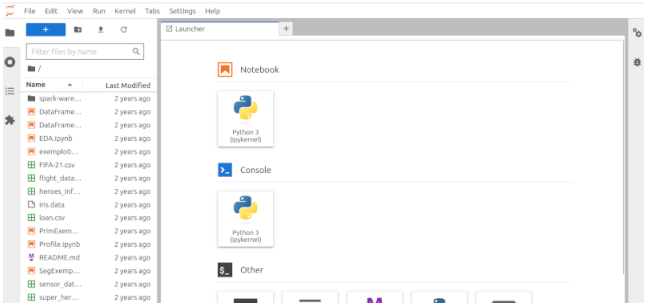


Figura 1.5: Jupyter Lab pronto

Verificar qual a versão do Python que está a nossa disposição, crie um Notebook e em uma célula digite:

```
!python --version
```

Ao pressionarmos Ctrl+Enter será mostrada a versão 3.9.2. Agora em outra aba do navegador acesse o endereço `http://localhost:8080`, e temos:

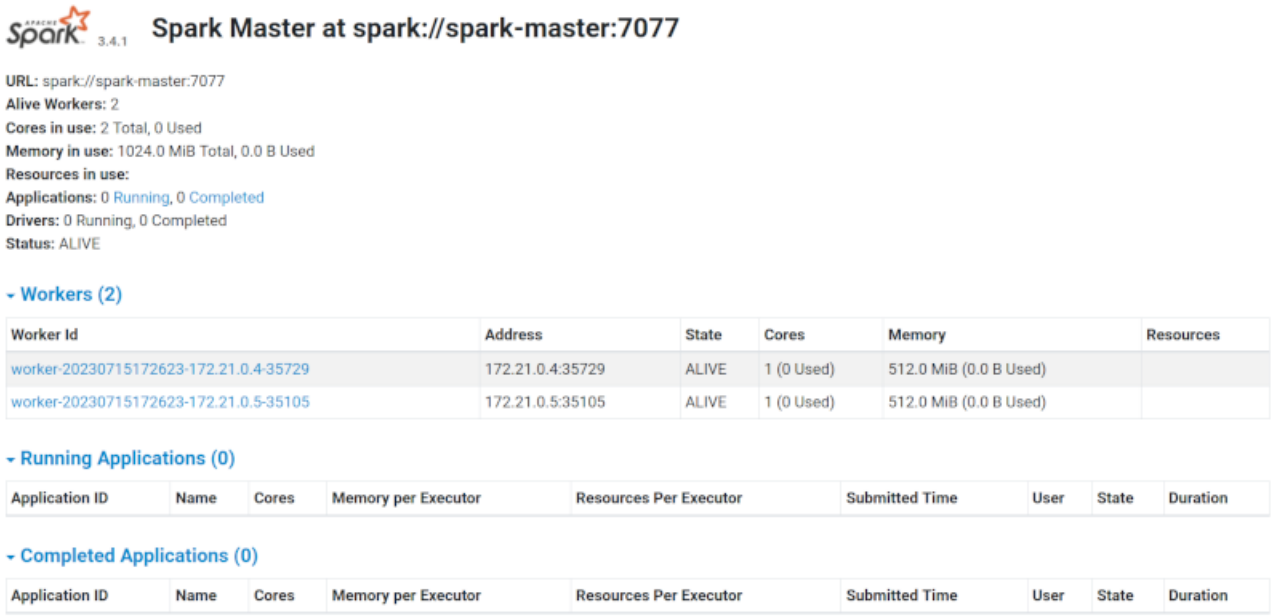


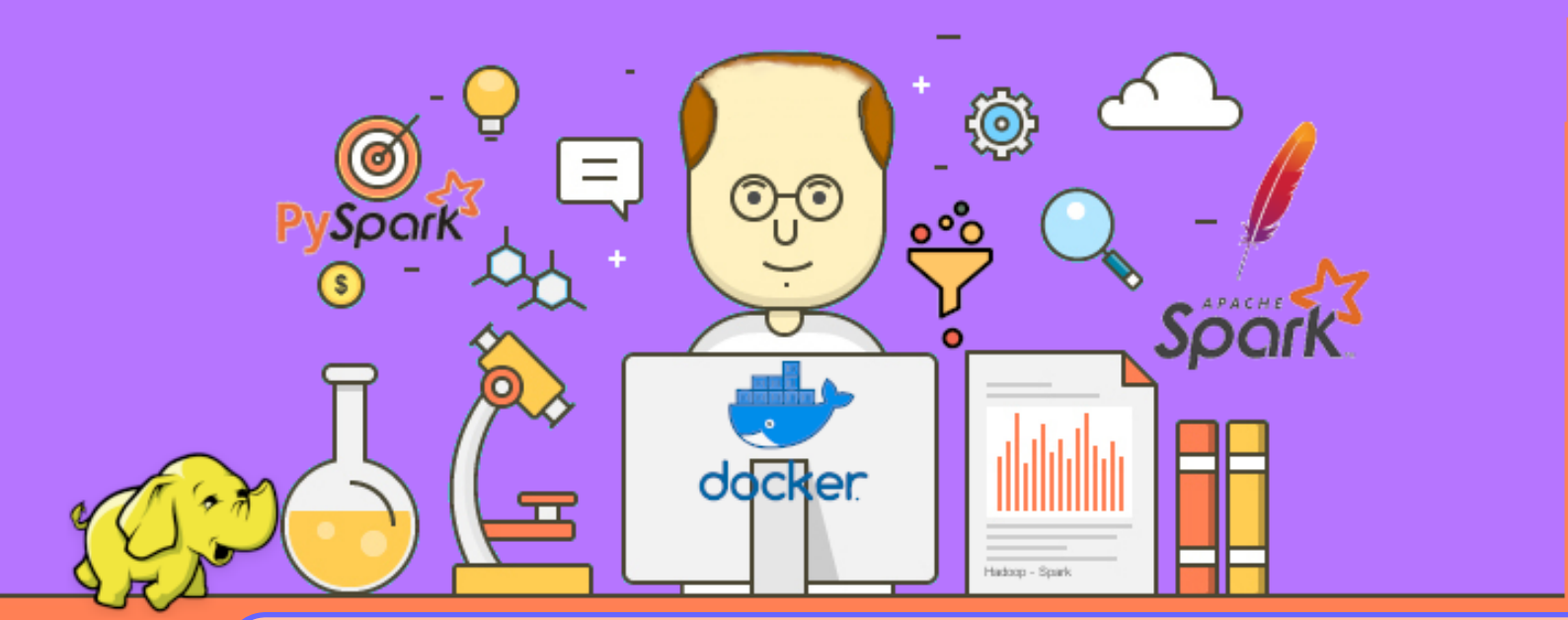
Figura 1.6: Spark Master pronto

Isso indica que nosso Spark Master está pronto e possui 2 Spark Workers a disposição. Além de suas capacidades em tempo real, Spark também é responsável pelo processamento em lote, assim obtemos uma solução abrangente e flexível. Rapidamente tornou-se a escolha popular para empresas que buscam lidar tanto com cargas de trabalho em tempo real quanto com análises em grande escala. Outro aspecto notável é o suporte e execução de consultas complexas e a aplicação de algoritmos iterativos em larga escala, o que é fundamental para tarefas como aprendizado de máquina e mineração de dados. Com essa funcionalidade, esta é uma ferramenta versátil para uma ampla gama de aplicações que exigem o processamento de dados.

Como dito anteriormente uma das principais motivações para a criação do Spark como substituto do

Hadoop MapReduce foi justamente atender às demandas crescentes por velocidade e flexibilidade no processamento de dados. Spark introduziu o conceito de Resilient Distributed Datasets (RDDs), que são estruturas de dados imutáveis e distribuídas, armazenadas em memória e que podem ser processadas de forma paralela. Essa abordagem inovadora permite que o Spark otimize a execução de operações complexas, como transformações e ações, reduzindo a latência e aumentando a eficiência geral do processamento de dados. Veremos isso no próximo capítulo.

> **Dica 1.3: Não sabe nem por onde começar com o Docker?.** Não se desespere, baixe um paper sobre o Docker gratuitamente na minha página no Academia.edu (`https://iesbpreve.academia.edu/FernandoAnselmo`).



2. Primeiros Exemplos

"Inovação é a faísca que acende o fogo da mudança e impulsiona o progresso da tecnologia."(Bill Gates)

2.1 Meu Hello World?

Como todo nosso ambiente montado vamos criar um novo caderno, a primeira ação que devemos realizar é criar uma seção e um contexto para conversação com o Spark. Importamos a biblioteca necessária:

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

Criamos o objeto de seção:

```
spark = SparkSession.\
        builder.\
        appName("pyspark-notebook").\
        master("spark://spark-master:7077").\
        config("spark.executor.memory", "512m").\
        getOrCreate()
```

Damos um nome para esta seção: **pyspark-notebook**, indicamos aonde está o Spark Master (conforme o contêiner Docker): spark-master:7077 e quanta memória deve utilizar: 512 Mb. Agora se voltarmos ao gerenciador veremos que os **Spark Workers** foram alocados para responder a essa seção:

Running Applications (1)

Application ID		Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20230715182639-0000	(kill)	pyspark-notebook	2	512.0 MiB		2023/07/15 18:26:39	root	RUNNING	1.2 min

Figura 2.1: Seção Running Applications do Spark Master

Por fim criamos um contexto de conversação:

```
sc = spark.sparkContext
```

Com essa célula executando perfeitamente temos o nosso Hello Wolrd pronto! Concordo, não teve a menor graça e parece que ficou faltando algo, bem wm paralelismo é assim mesmo que acontece, mas vamos em frente e verificarmos o que são objetos RDD.

2.2 Objeto RDD

Por definição, cada aplicativo **Spark** consiste em um programa que executa várias operações paralelas em um **cluster**. A abstração principal que **Spark** oferece é um conjunto chamado **RDD** (*Resilient Distributed Dataset*), este é uma coleção de elementos particionados nos nós do **cluster**. Os RDDs são criados a partir de um arquivo no sistema de arquivos **Hadoop** (ou qualquer outro sistema de arquivos compatível com Hadoop). As características deste são:

Figura 2.2: Características do RDD

Podem ser mantidos em memória, são trabalhados de forma tardia (Lazy, assim não alocam recursos do programa principal), possuem tolerância a falhas, são objetos imutáveis, são particionados, podem realizar persistência dos dados (guardá-los para recuperação posterior) e possuem baixa granularidade (coarse-grained service) e reduz o número de chamadas necessárias para concluir uma tarefa.

Vamos criar um RDD de forma bem simples através de uma lista:

```
rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
print(type(rdd))
```

E como resposta temos:
`<class 'pyspark.rdd.RDD'>`

Para vermos os elementos usamos a função collect():

```
print(rdd.collect())
```

E temos como resposta:
`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

2.3 Uso de Arquivos Texto

Também podemos ler um arquivo texto, para este exemplo criar um arquivo texto na mesma pasta com o nome: teste.txt e seguinte conteúdo:

```
Apache Spark é um mecanismo de análise unificado para processamento de dados em
    grande escala.
Fornece APIs para linguagens de alto nível como Java, Scala, Python e R
Permite um mecanismo otimizado para suporte a gráficos de execução geral.
Possui um excelente conjunto de ferramentas, tais como:
    Spark SQL para SQL e processamento de dados estruturados;
    MLlib para aprendizado de máquina;
    GraphX para processamento de gráfico; e
    Streams estruturados para computação incremental e processamento de fluxo.
```

Para ler este arquivo usamos:

```
rdd = sc.textFile('teste.txt')
print(type(rdd))
```

E observamos que é o mesmo tipo do objeto criado anteriormente, se desejar contar quantos elementos temos:

```
print(rdd.count())
```

E retorna que temos 8 elementos, como assim? Spark utiliza as linhas como elementos, se quisermos verificar como o RDD foi montado é só usar o método collect()

```
print(rdd.collect())
```

Obtemos toda sua configuração utilizada com o comando:

```
print(sc.getConf().getAll())
```

Temos elementos que atrapalham nossos próximos passos, assim para remover vírgulas, dois pontos, ponto final e outros sinais de pontuação do texto em um RDD utilizamos as funções para a manipulação de strings disponíveis no Spark, importamos para isso a biblioteca **re**:

```
import re
```

E para substituir esses caracteres por espaços em branco ou removê-los completamente

```
rdd_sem_pontuacao = rdd.map(lambda linha: re.sub(r'[^\w\s]', '', linha))
for linha in rdd_sem_pontuacao.collect():
    print(linha)
```

Utilizamos a biblioteca do Python para utilizar a função **sub** e substituir os sinais de pontuação por espaços em branco. A expressão regular mostrada corresponde a qualquer caractere que não seja alfanumérico (w) ou espaço em branco (s).

Dessa forma, a transformação **map** é aplicada em cada linha do RDD para substituir os sinais de pontuação por espaços em branco. Em seguida, utilizamos o método **collect()** para coletar e imprimir todas as linhas do RDD sem pontuação.E temos como resultado as linhas livres de pontuação.

Agora podemos contar a quantidade de ocorrências para cada palavra no texto, porém ainda temos um pequeno problema que são as palavras sem significado próprio como "e", "é", "de"e por aí vai. vamos removê-las então:

```
# Definir a lista de palavras indesejadas
palavras_indesejadas = ["e", "é", "um", "para", "em", "como", "a", "tais", "como",
    "de", ""]

# Dividir cada linha em palavras e criar um RDD com todas as palavras
palavras = rdd_sem_pontuacao.flatMap(lambda linha: linha.split(" "))

# Filtrar as palavras indesejadas
palavras_filtradas = palavras.filter(lambda palavra: palavra not in
    palavras_indesejadas)

# Contar as ocorrências de cada palavra
contagem_palavras = palavras_filtradas.countByValue()

# Imprimir as palavras únicas e suas contagens
print("Palavras únicas encontradas e suas contagens:")
for palavra, contagem in contagem_palavras.items():
    print(palavra, "->", contagem)
```

Definimos uma lista de palavras que desejamos remover da contagem. Em seguida, aplicamos o método **filter()** no **RDD** que não contém a pontuação usando uma condição lambda e assim filtrar as palavras que não estão presentes na lista de palavras indesejadas.

Contamos as ocorrências das palavras filtradas e imprimimos as palavras únicas encontradas e suas respectivas contagens.

2.4 PipelinedRDD e DataFrame

Quando falamos de **Pipeline** rapidamente associamos ao **Jenkins** e **DataFrame** ao **Pandas**, porém devemos esquecer essa associação aqui, o **PipelinedRDD** e uma subcoleção de um **RDD** e o **DataFrame** é o modo como Spark realiza uma associação de linhas e colunas.

Vamos imaginar que temos uma lista de 1.000 pacientes no qual colhemos os dados de Pressão Arterial, tanto a sistólica (PAS) quanto a diastólica (PAD). Para simularmos esses dados vamos importar a classe Random:

```
import random
```

E criamos um método de modo a retornar os valores como uma lista:

```
def valorPressao(x):
    idt = x
    pas = random.randint(11, 20)
```

```
    pad = random.randint(70, 121)
    return (x, pas, pad)
```

Definimos para PAS um valor aleatório porém devemos nos manter entre 11 e 19 e para PAD entre 70 e 121 de modo a criarmos dados viáveis. Agora criamos um objeto PipelinedRDD:

```
rdd_pipe = sc.parallelize(range(1000)).map(valorPressao)
print(type(rdd_pipe))
```

E como resposta temos:
`<class 'pyspark.rdd.PipelinedRDD'>`

Podemos visualizar os 10 primeiros registros com:

```
rdd_pipe.take(10)
```

E transformá-lo em um DataFrame com:

```
df = rdd_pipe.toDF(("Id", "PAS", "PAD"))
type(df)
```

E como resposta temos:
`pyspark.sql.dataframe.DataFrame`

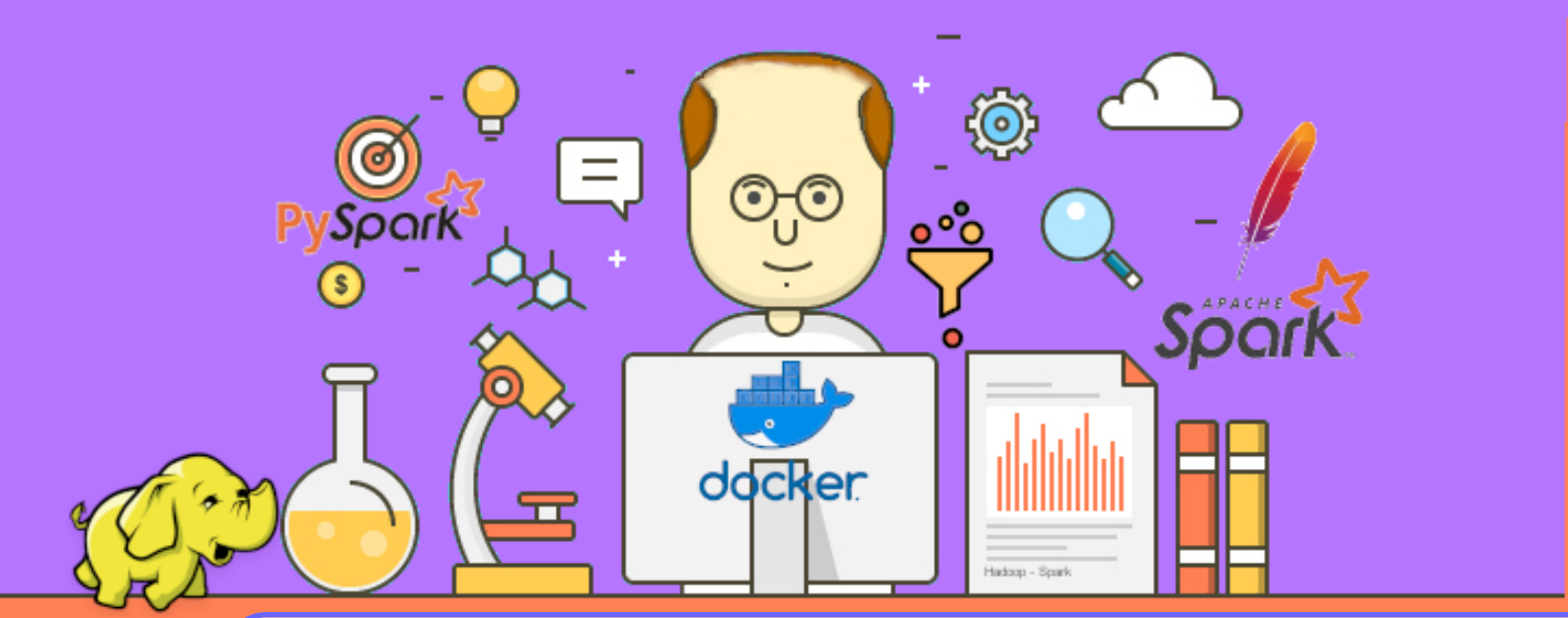
E visualizarmos os registros com:

```
df.show()
```

Para encerramos o trabalho, não devemos esquecer de sempre fechar a seção, antes mesmo de parar a composição Docker.

```
spark.sparkContext.stop()
```

E notamos na página de status do **Spark Master** que os **Spark Workers** alocados foram para a seção **Completed Applications**.



A. Considerações Finais

"A paixão pela tecnologia é a faísca que impulsiona o desejo de aprender, experimentar e criar soluções inovadoras."(Satya Nadella - CEO (*Chief Executive Officer*) da Microsoft Corporation)

Os artigos deste livro foram selecionados das diversas publicações que fiz no Linkedin e encontradas em outros sites que foram nesta obra explicitamente citadas. Acredito que apenas com a prática podemos almejar o cargo de Cientista de Dados, então segue uma relação de boas bases que podemos encontrar na Internet:

- 20BN-SS-V2: `https://20bn.com/datasets/something-something`
- Actualitix: `https://pt.actualitix.com/`
- Banco Central do Brasil: `https://www3.bcb.gov.br`
- Banco Mundial: `http://data.worldbank.org`
- Censo dos EUA (População americana e mundial): `http://www.census.gov`
- Cidades Americanas: `http://datasf.org`
- Cidade de Chicago: `https://data.cityofchicago.org/`
- CIFAR-10: `https://www.cs.toronto.edu/~kriz/cifar.html`
- Cityscapes: `https://www.cityscapes-dataset.com/`
- Criptomoedas: `https://pro.coinmarketcap.com/migrate/`
- Dados da União Europeia: `http://open-data.europa.eu/en/data`
- Data 360: `http://www.data360.org`
- Datahub: `http://datahub.io/dataset`
- DBpedia: `http://wiki.dbpedia.org/`
- Diversas áreas de negócio e finanças: `https://www.quandl.com`
- Diversos assuntos: `http://www.freebase.com`
- Diversos países (incluindo o Brasil): `http://knoema.com`
- Fashion-MNIST: `https://www.kaggle.com/zalando-research/fashionmnist`
- Gapminder: `http://www.gapminder.org/data`
- Google Finance: `https://www.google.com/finance`
- Google Trends: `https://www.google.com/trends`
- Governo do Brasil: `http://dados.gov.br`
- Governo do Canadá (em inglês e francês): `http://open.canada.ca`
- Governo dos EUA: `http://data.gov`
- Governo do Reino Unido: `https://data.gov.uk`
- ImageNET: `http://www.image-net.org/`

- IPEA: http://www.ipeadata.gov.br
- IMDB-Wiki: https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/
- Kinetics-700: https://deepmind.com/research/open-source/kinetics
- Machine Learning Databases: https://archive.ics.uci.edu/ml/machine-learning-databases/
- MEC Microdados INEP: http://inep.gov.br/microdados
- MS coco: http://cocodataset.org/#home
- MPII Human Pose: http://human-pose.mpi-inf.mpg.de/
- Músicas: https://aws.amazon.com/datasets/million-song-dataset
- NASA: https://data.nasa.gov
- Open Data Monitor: http://opendatamonitor.eu
- Open Data Network: http://www.opendatanetwork.com
- Open Images: https://github.com/openimages/dataset
- Portal de Estatística: http://www.statista.com
- Públicos da Amazon: http://aws.amazon.com/datasets
- R-Devel: https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html
- Reconhecimento de Faces: http://www.face-rec.org/databases
- Saúde: http://www.healthdata.gov
- Statsci: http://www.statsci.org/datasets.html
- Stats4stem: http://www.stats4stem.org/data-sets.html
- Stanford Large Network Dataset Collection: http://snap.stanford.edu/data
- Vincent Rdatasets: https://vincentarelbundock.github.io/Rdatasets/datasets.html
- Vitivinicultura Embrapa: http://vitibrasil.cnpuv.embrapa.br/

Esse não é o fim de uma jornada acredito ser apenas seu começo. Espero que este livro possa lhe servir para criar algo maravilhoso e fantástico que de onde estiver estarei torcendo por você.

A.1 Sobre o Autor

Fortes conhecimentos em linguagens de programação Java e Python. Especialista formado em Gestão da Tecnologia da Informação com forte experiência em Bancos Relacionais e não Relacionais. Possui habilidades analíticas necessárias para encontrar a providencial agulha no palheiro dos dados recolhidos pela empresa. Responsável pelo desenvolvimento de dashboards com a capacidade para analise de dados e detectar tendências, autor de 15 livros e diversos artigos em revistas especializadas, palestrante em seminários sobre tecnologia. Focado em aprender e trazer mudanças para a organização com conhecimento profundo do negócio.

- Perfil no Linkedin: http://www.linkedin.com/pub/fernando-anselmo/23/236/bb4
- Endereço do Git: https://github.com/fernandoans/machinelearning

Machine Learning na Prática

ESTE LIVRO PODE E DEVE SER DISTRIBUÍDO LIVREMENTE

Fernando Anselmo