Apache Spark: o que é e como utilizá-lo com o Google Colab

COMPARTILHE

Olá pessoas! Bem vindos(as) à nossa série de posts sobre *Machine Learning e Spark*. O objetivo desses textos é dar uma introdução básica ao conceito de Apache Spark e treinar alguns modelos usando o Google Colab para facilitar a execução dos códigos e maximizar o aprendizado.

Como ninguém faz nada sozinho nesse mundo, Há links de referências durante o texto e você pode encontrar mais referências utilizadas ao final do artigo. Acessem, mas fiquem a vontade para mandar perguntas também!

O que é Apache Spark?

Apache Spark é uma ferramenta de uso geral para **processamento de dados** e **computação paralela**. A *engine* é compatível com diversos tipos de dados distribuídos, por exemplo: HDFS, HBase, Cassandra, Hive e qualquer *input* do Hadoop.

O Spark pode ser configurado para rodar em um *cluster* Hadoop em conjunto com um gerenciador, como Yarnou Mesos, ou pode ser instalado utilizando o *manager* do próprio *core* do Spark, chamado de instalação *standalone*.

Além da parte Core do Spark, que é o responsável pelo gerenciamento dos processamentos e das bases de dados (caso *standalone*), o Spark possui mais alguns componentes. A figura a seguir apresenta o componentes do Spark:

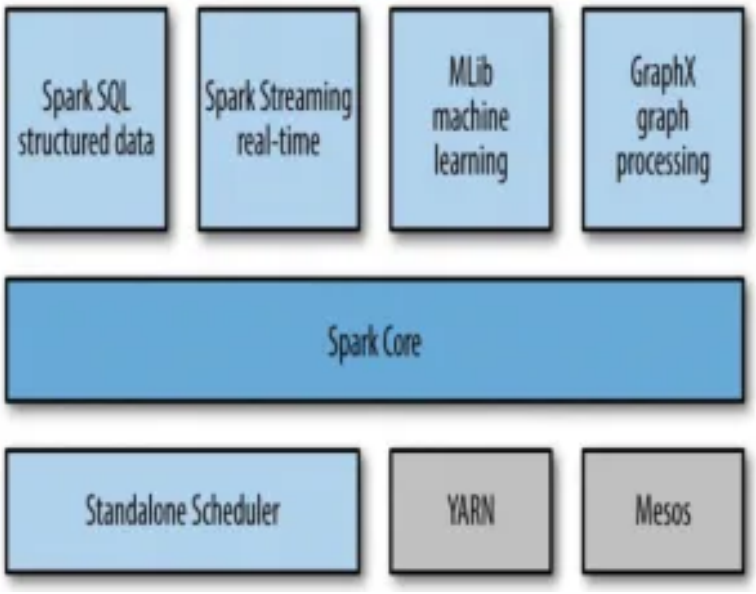


Figura 1 – Componentes do Spark. Extraído de [2].

O componente de Spark SQL permite ao desenvolvedor realizar consultas no banco de dados distribuído.

Essas consultas são feitas usando uma linguagem bastante similar ao SQL, que pode ser utilizado por meio de sua interface em Python com o *pyspark.sql*.

O objetivo desse componente é a **manipulação e processamento dos dados distribuído**s. Focaremos no Spark SQL nesse breve texto.

Outros componentes interessantes são:

- **Spark Streaming**: permite ao desenvolvedor manipular e processar dados oriundos de algum *streaming*.
- **Spark MLLib**: uma biblioteca de algoritmos de *machine learning* distribuídos.
- **Spark GraphX**: uma biblioteca capaz de processar grafos utilizando a arquitetura distribuída do Spark.

Como funciona o Spark?

Para entender o funcionamento do Spark, é necessário entender primeiro alguns conceitos. Primeiro, o conceito de Driver Node e Worker Node.

Worker Node é um computador dentro do *cluster*, que executa as *tasks* geradas. Esse computador ficará ouvindo o agendador de tarefas do Spark, recebendo parte dos dados e do código para ser aplicado sobre esses dados, realizando o processo e devolvendo o resultado.

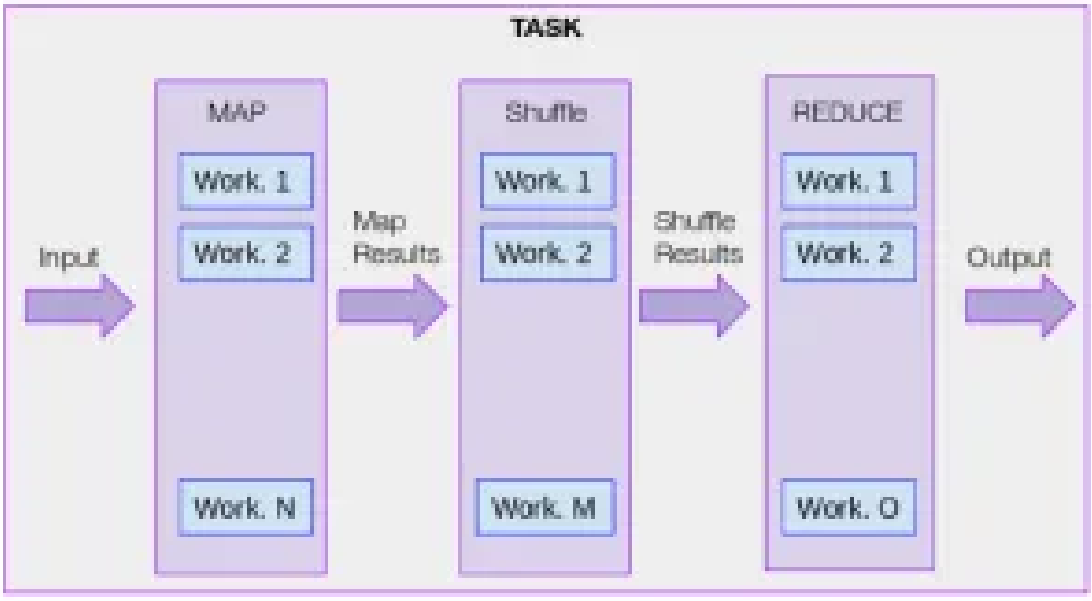
Driver Node é o computador no qual você desenvolve o código. Ele é responsável por fazer toda a parte de codificação e de repassar as *tasks* para os nós que efetivamente vão executá-los (os *workers*). O Driver Node é responsável por realizar a interface entre o desenvolvedor e o *cluster* Spark.

De forma geral, o Spark foi construído em cima da estrutura de MapReduce. Então, sempre que uma *task* é iniciada, o agendador do Spark distribui partes dos dados para N computadores do *cluster*, que individualmente

processam suas partes.

Depois do processamento, chamado MAP, é feita a etapa de REDUCE, que consiste em reunir todos os dados processados e apresentar o resultado final de cada *task*.

Um exemplo de task pode ser visto na imagem a seguir. Ressaltando que um comando em Spark normalmente desencadeia uma série de *tasks*.



Esse é um **exemplo de task usando MapReduce no Spark**.

Dessa forma, o processamento é feito distribuído um pouco em cada Worker – o que possibilita a exploração de bases gigantescas.

É importante ressaltar que tudo isso é realizado usando as bibliotecas do Spark, ou seja, se em algum momento os dados saem do Spark (com um .toPandas(), por exemplo), eles sobem para a memória do Driver, o que derruba o computador. Portanto, **cuidado com as bibliotecas** que estão sendo usadas.

Outro conceito importante do Spark, que possibilita o processamento distribuído, é o conceito de ***Lazy Load***. Esse conceito permite que os dados não sejam carregados na memória quando são lidos, no lugar, é criado um ponteiro para onde esses dados estão.

Com isso, o usuário pode criar uma sequência de processamentos nos dados muito rapidamente, pois o Spark somente processará esses dados quando houver uma ação (um .count(), por exemplo).

Configurando o Spart no Google Colab

Dito isso, estamos ansiosos para trabalhar com Spark, não é mesmo? Então vamos configurar um ambiente no **Google Colab**!

Para quem não conhece, essa é a plataforma do Google para codificação online compartilhada.

É excelente para fazermos alguns testes e nos aventurarmos em coisas novas, sem todo o problema e a limitação das nossas máquinas físicas. Para acessar, basta clicar aqui!

Ao acessar o notebook pela primeira vez, é necessário instalar os pacotes, configurar as variáveis de ambiente e iniciar o Spark! Para tanto, basta rodar os 3 conjuntos de código abaixo:

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-eu.apache.org/dist/spark/spark-2.4.1/spark-2.4.1-bin-hadoop2.7.tgz
!tar xf spark-2.4.1-bin-hadoop2.7.tgz
!pip install -q findspark
!pip install -q pyspark
```

view raw

Code1
hosted with ♡ by GitHub

Essa primeira parte do código é responsável por instalar o Java, baixar e instalar o Spark com o Hadoop, instalar o findSpark, que é utilizado para encontrar a instalação do Spark na máquina, e o PySpark, que é a API de Spark em Python.

IMPORTANTE: De tempos em tempos, eles removem as distribuições antigas do Spark do diretório da Apache. Se o código não funcionar para você, provavelmente você precisará atualizar a versão do Spark que está baixando.

```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.1-bin-hadoop2.7"
```

view raw

Code2
hosted with ♡ by GitHub

Nessa segunda parte do código, estamos definindo as variáveis de ambiente e mostrando para o sistema onde buscar os arquivos necessários.

Aqui é importante ressaltar que, caso você altere a versão do Spark que vai utilizar, é necessário também alterar o arquivo que está sendo enviado para a variável SPARK_HOME.

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
```

view raw

Code3
hosted with ♡ by GitHub

Nessa última etapa, encontramos o Spark e iniciamos uma sessão. Utilizamos “local[*]” pois, dessa forma, nosso processamento será distribuído em todos os *cores* da máquina que estamos utilizando. Se sua variável de ambiente estiver errada no comando anterior, o comando findspark.init() irá falhar.

Pronto! Temos um ambiente configurado e rodando Spark! Agora é só sair codificando. Na próxima e última seção deste artigo, iremos apresentar alguns **comandos** que você pode fazer usando o pyspark.sql.

Executando alguns comandos

Depois de tudo instalado, finalmente está na hora da diversão! Irei executar alguns comandos no Google Colab.

Para entender todas as possibilidades que você tem, basta checar a documentação da API do PySpark em [6]. Primeiro, é necessário importar a API que será utilizada. Para esse exemplo, irei importar tudo da API pyspark.sql.

```
from pyspark.sql import *
```

Com esse simples comando, todas as funções do pyspark.sql estão disponíveis para uso. Iremos utilizar uma base de exemplos disponíveis dentro do próprio Google Colab.

Para ler a base, basta executar o comando abaixo. Para mais informações referente a base, clique aqui.

```
dataset = spark.read.format("json").option("multiLine",True).load("sample_data/anscombe.json")
dataset.columns
```

view raw

Code4
hosted with ♡ by GitHub

Após a leitura, executei também o comando dataset.columns, que irá apresentar as colunas disponíveis no dataframe. É possível você usar o comando show(), para verificar os valores. Conforme abaixo:

```
dataset.show(10)
```

view raw

Code5
hosted with ♡ by GitHub

```
+------+----+-----+
|Series|   X|    Y|
+------+----+-----+
|     I|10.0| 8.04|
|     I| 8.0| 6.95|
|     I|13.0| 7.58|
|     I| 9.0| 8.81|
|     I|11.0| 8.33|
|     I|14.0| 9.96|
|     I| 6.0| 7.24|
|     I| 4.0| 4.26|
|     I|12.0|10.84|
|     I| 7.0| 4.81|
+------+----+-----+
```

Agora que carregamos nossa base, verificamos que X e Y são possivelmente valores contínuos. *Series* é um campo categórico. Para isso, iremos usar o comando dtypes, para verificar as colunas.

```
dataset.printSchema()
```

view raw

Code6
hosted with ♡ by GitHub

```
root
 |-- Series: string (nullable = true)
 |-- X: double (nullable = true)
 |-- Y: double (nullable = true)
```

Ótimo, agora temos certeza que X e Y são valores contínuos e que Series é categórico. Por fim, vamos realizar um agrupamento na coluna Series e fazer a média dos valores de X e Y.

Conforme descrição da base, ela contém **4 diferentes datasets** com basicamente os mesmos valores de estatística descritiva. Será que conseguimos encontrar isso?

```
from pyspark.sql import functions as F
dataset_agrupado = dataset.groupBy("Series") \
                          .agg(F.avg("X").alias("X_agrupado"), F.avg("Y").alias("Y_agrupado"))  \
                          .orderBy("Series")


dataset_agrupado.show()
```

view raw

Code7
hosted with ♡ by GitHub

```
+------+----------+----------------+
|Series|X_agrupado|       Y_agrupado|
+------+----------+----------------+
|     I|       9.0|             7.5|
|    II|       9.0|7.500909090909091|
|   III|       9.0|7.500000000000001|
|    IV|       9.0| 7.50090909090909|
+------+----------+----------------+
```

Excelente! O resultado é exatamente o esperado. Agora, vamos dar uma olhada nesse código mais a fundo.

Na **primeira linha** eu importo novamente a API de funções com o alias F.

Essa é uma das formas de se realizar o procedimento. Eu poderia simplesmente ter escrito “functions.avg()” no lugar de “F.avg()”, porém, prefiro usar o alias “F” nos meus códigos.

Na **segunda linha** é definido todo o processamento em Spark para como se obter o valor agrupado de X e Y.

Primeiro eu realizo um “groupBy”, que é uma função agregadora. Basicamente o que essa função faz é pegar os valores distintos da coluna “Séries” e usar como agrupador para os próximos comandos. Na segunda linha eu acho o comando .agg() é utilizado para realizar operação de agregação. F.avg é a função utilizada para calcular a média de um determinado campo e, por fim, .alias() é utilizado para definir o nome do campo.

Se você executar esse comando sem o .show(), verá que ele é instantâneo. Isso não significa que o Spark já processou tudo e tem tudo calculado na memória, na verdade, isso apenas significa que ele montou o *pipeline*de execução para obter o valor que você deseja visualizar. Para que ele efetivamente processe o comando, você precisa usar uma ação, no nosso caso, um .show().

Para dar uma pequena espiada em como o Spark funciona, execute o comando abaixo:

```
dataset_agrupado.explain()
```

view raw

Code8
hosted with ♡ by GitHub

Conclusões

Nesse artigo podemos ter uma **visão de alto nível** de como o Spark funciona, quais são as APIs disponíveis, o que cada uma delas se propõe a fazer e como configurá-lo e programar usando PySpark no Google Colab.

Sugiro que você dê uma lida na documentação do **PySpark.sql** e tente criar algumas análises diferentes da base de dados.

Tente entender melhor como a API funciona e como o processamento é feito, só não vale usar Pandas, ok? Salvei o notebook com todos os comandos. Basta clicar aqui, fazer o download, subir para o Google Colab e brincar com os comandos.

Fique ligado, nas próximas semanas iremos conhecer a biblioteca de machine learning e treinar nosso primeiro modelo.

Um grande abraço!

Referências e materiais extras

1. https://towardsdatascience.com/pyspark-in-google-colab-6821c2faf41c
2. Mark Hamstra, Matei Zaharia, Holden Karau. Learning Spark: Lightning-Fast Big Data Analysis. 2016.
3. https://spark.apache.org/docs/2.4.1/api/python/pyspark.html
4. https://github.com/renanpadua/codeExamples/blob/master/PySpark_%2B_Google_Colab_Movile_Blog.ipynb

Grupo Movile marca presença na Qcon com a ferramenta Apache Spark
06/06/2018
Em "Backend"

Outliers: Achar os Pontos Fora da Curva
18/05/2020
Em "Banco de Dados"

O Impacto de Outliers em Algoritmos de Machine Learning
24/06/2020
Em "Banco de Dados"

Renan Padua

Formado em Ciências da Computação na Unesp Rio Claro, com mestrado e finalizando doutorado na USP - São Carlos, Renan é Cientista de Dados Sênior no iFood. É apaixonado por tech e por comida japonesa