Capítulo 2

Introdução ao Spark com Pyspark

Leonardo Afonso Amorim

Abstract

Apache Spark is a unified analytics engine for large-scale data processing. PySpark is a frontend for Apache Spark in Python. It allows you to write Spark applications using Python APIs to analyze your data in a distributed environment. PySpark supports most Spark features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning), and Spark Core. In this workshop, we will understand how Spark works and perform data analysis using RDD and Dataframes in Google Colab.

Resumo

Apache Spark é um mecanismo de análise unificado para processamento de dados em grande escala. E o PySpark é uma interface para Apache Spark em Python. Ele permite que você escreva aplicativos Spark usando APIs Python para analisar interativamente seus dados em um ambiente distribuído. O PySpark é compatível com a maioria dos recursos do Spark, como Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) e Spark Core. Neste minicurso, vamos entender como o Spark funciona e como fazer análise de dados usando RDD e Dataframes no Google Colab.

1. Introdução ao Spark

O Spark é um framework para processamento de Big Data construído com foco em velocidade, facilidade de uso e análises sofisticadas. Está sendo desenvolvido desde de 2009 pela Universidade de Califórnia em Berkeley. O Spark tem muitas vantagens se comparado as outras tecnologias de Big Data e do paradigma MapReduce, como o Hadoop. Inicialmente, o Spark oferece um framework unificado e de fácil compreensão para gerenciar e processar Big Data com uma variedade de conjuntos de dados de diversas naturezas (por exemplo: texto, grafos, etc), bem como de diferentes origens (batch ou streaming de dados em tempo real).

O Spark permite que aplicações em clusters Hadoop executem até 100 vezes mais rápido em memória e até 10 vezes mais rápido em disco, desenvolver rapidamente aplicações em Java, Scala ou Python. Além disso, vem com um conjunto integrado de mais de 80 operadores de alto nível e pode ser usado de forma interativa para consultar dados diretamente do console. Além das operações de Map/Reduce, suporta consultas SQL, streaming de dados, aprendizado de máquina e processamento de grafos. Características do Spark:

- Framework para processamento de Big Data;
- Hadoop e Spark são tecnologias complementares;
- É possível usar Spark sem depender do Hadoop e vice-versa;
- O Spark suporta as linguagens: Scala, Python, Java, R e SQL;
- Com o Spark é possível:
  - Executar operações de ETL;
  - Trabalhar com Machine Learning;
  - Fazer acesso de dados com SQL;
  - Trabalhar com Mineração de Dados;
  - Trabalhar com Sistemas de Recomendação;

Quase todos os dados são processados usando Conjuntos de dados distribuídos resilientes (RDDs). Os RDDs são a principal abstração de programação no Spark. Em memória, mas resiliente. O Spark é composto de alguns componentes diferentes:

- Spark Core: A funcionalidade básica do Spark - RDDs. É apenas um mecanismo de computação e, portanto, necessita de 2 componentes adicionais:
  - Um sistema de armazenamento para persistir os dados a serem processados (Local ou HDFS);
  - Um Gerenciador de Cluster para ajudar o Spark a executar tarefas em um cluster de máquinas (Built-in Cluster Manager, YARN);
  - Ambos são componentes plug and play! Plug and Play torna mais fácil integrar com o Hadoop!

O Spark pode ser usado como uma alternativa em relação ao MapReduce aproveitando o Yarn e HDFS do Hadoop. Pré-requisitos para se trabalhar com Spark: Java 7 ou acima, Scala, Anaconda.

2. Conjunto de dados resilientes e distribuídos

O conjunto de dados resilientes e distribuídos ou RDD (Resilient Distributed Datasets) é o conceito central do framework Spark. Imagine o RDD como uma tabela do banco de dados que pode guardar qualquer tipo de dado. O Spark armazena os dados do RDD em diferentes partições. Isso ajuda a reorganização computacional e a otimização no processamento dos dados. As características dos RDDs (decompondo o nome):

- Resiliente, ou seja, tolerante a falhas;
- Distribuído com dados que residem em vários nós em um cluster;
- O conjunto de dados é uma coleção de dados particionados com valores primitivos ou tuplas ou outros objetos (que representam registros dos dados com os quais você trabalha).

Os RDDs são imutáveis. Ainda que aparentemente seja possível modificar um RDD com uma transformação, na verdade o resultado dessa transformação é um novo RDD, sendo que o original permanece intocável. O RDD suporta dois tipos de operações:

- Transformação: Não retornam um único valor, mas um novo RDD. Nada é avaliado quando a função de transformação é chamada, ela apenas recebe um RDD e retorna um novo RDD. Algumas das funções de transformação são map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe e coalesce;
- Ação: Esta operação avalia e retorna um novo valor. Quando uma função de ação é chamado em um objeto RDD, todas as consultas de processamento de dados são computadas e o valor é retornado. Algumas das operações de ação são reduce, collect, count, first, take, countByKey e foreach;

3. Pyspark

O Pyspark possui um shell semelhante ao shell do Python. O Pyspark permite o uso de funções Python, dicionários, listas etc. É possível importar e usar quaisquer módulos do Python instalados. Inicia por padrão em um modo local não distribuído. Quando o shell é iniciado, inicializa um SparkContext. O SparkContext:

- Representa um conexão com o Spark Cluster;
- É usado para carregar dados na memória de uma fonte específica;
- Os dados são carregados em um RDD.
- RDDs - Resilient Distributed Datasets:
  - Partições (Partitions). Observe como pode ser abstraído o particionamento de uma tabela na Figura 2.1;
  - Apenas Leitura (Read-only);

Figura 2.1. Particionamento de dados com RDD

  - Linha do tempo (Lineage);

- RDDs representam dados na memória

- Os dados são divididos em partições e distribuídos entre várias máquinas chamadas de nós;

- Os nós processam dados em paralelo;

- RDDs são imutáveis;

- Apenas dois tipos de operações:
  - Transformação: Transforme-se em outro RDD como pode ser observado na Figura 2.2;
  - Ação: Solicite um resultado.

- O usuário pode definir uma cadeia de transformações no conjunto de dados:
  - Carregar dados
  - Escolha apenas a terceira coluna
  - Classifique os valores
- Espere até que um resultado seja solicitado antes de executar qualquer uma dessas transformações;
- Ação: Solicitação de resultado.
  - As primeiras 10 linhas
  - Uma contagem
  - Uma soma
- Os dados são processados somente quando o usuário solicita um resultado. A cadeia de transformações definidas anteriormente é executada.
- Lazy Evaluation (Avaliação Preguiçosa):
  - Spark mantém um registro da série de transformações solicitadas pelo usuário;
  - Agrupa as transformações de maneira eficiente quando uma ação é solicitada;
  - Linhagem (Lineage):
  - Quando criado, um RDD apenas contém metadados:
  - Uma transformação
  - É pai RDD
  - Cada RDD sabe de onde veio
- A linhagem pode ser rastreada até a origem;
- Quando uma ação é solicitada em um RDD, todos os seus RDDs pai são materializados ;
- Resiliência:
  - Tolerância a falhas embutida;
  - Se algo der errado, reconstrua da fonte;
- Avaliação preguiçosa:
- Materializar apenas quando necessário;

Desenvolvedores podem usar esses recursos no modo *standalone* ou combiná-los em um único *pipeline*. O Spark estende o MapReduce evitando mover os dados durante seu processamento, através de recursos como armazenamento de dados em memoria e processamento próximo ao tempo real, o desempenho pode ser várias vezes mais rápido

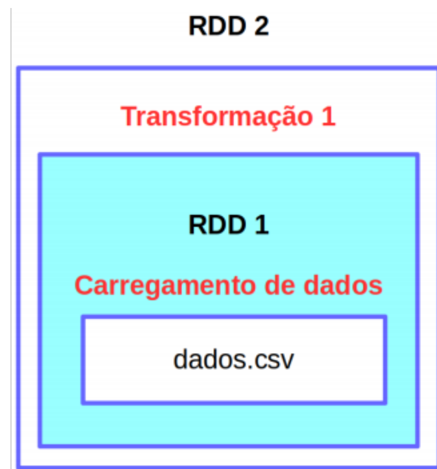

Figura 2.2. Ocorrendo uma transformação de dados em um RDD, um novo RDD será criado.

do que outras tecnologias de Big Data. O Spark armazena resultados intermediários na memória, em vez de escrevê-los no disco, o que é muito útil quando se precisa processar o mesmo conjuntos de dados muitas vezes. Seu projeto teve por objetivo torná-lo um mecanismo de execução que funciona tanto na memória como em disco e, por isso, o Spark executa operações em disco quando os dados não cabem mais na memória. Assim, é possível usá-lo para o processamento de conjuntos de dados maiores que a memória agregada em um cluster.

O Spark armazenará a maior quantidade possível de dados na memória e, em seguida, irá persisti-los em disco. Cabe ao arquiteto do sistema olhar para os seus dados e casos de uso para avaliar os requisitos de memória. Com esse mecanismo de armazenamento de dados em memória, o uso do Spark traz vantagens de desempenho. O Spark é escrito na linguagem Scala e executa em uma máquina virtual Java. Atualmente, suporta as seguintes linguagens para o desenvolvimento de aplicativos: Scala, Java, Python, Clojure, R.

Normalmente o Spark é usado com o HDFS, mas é possível usar outros sistemas de arquivos ou de armazenamento:

- Network FileSystem (NFS);
- Amazon S3;
- RDBMS;
- NoSQL (Hbase, Casassandra, MongoDB);
- Sistemas de mensagens (Kafka);

O Spark pode ser até 100 vezes mais rápido que o Hadoop como pode ser observado na Figura 2.3:

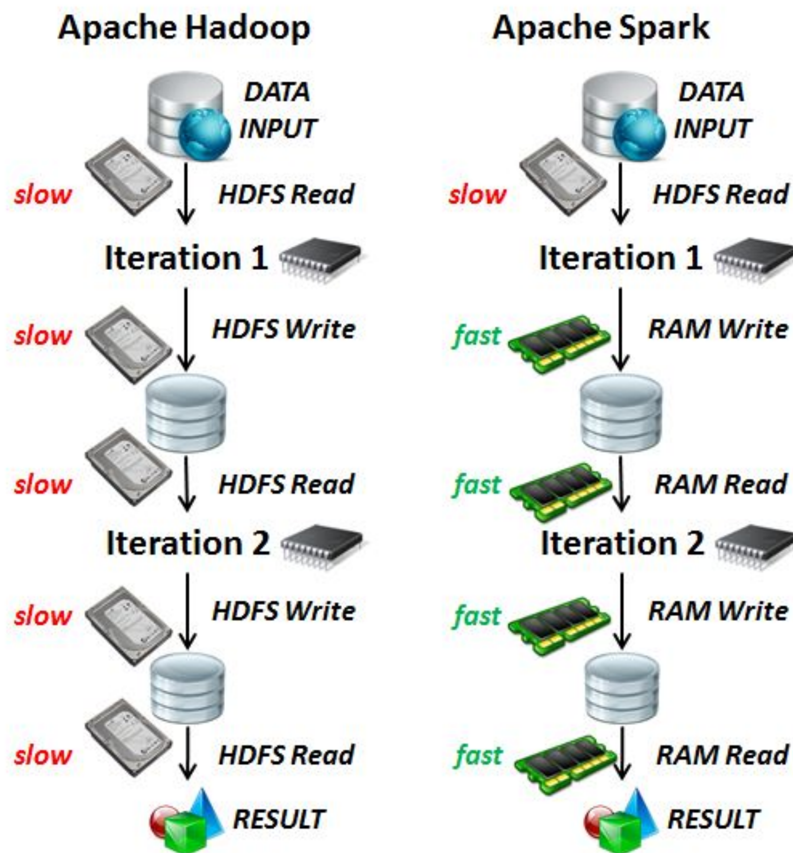

Figura 2.3. Comparação entre Hadoop e Spark em relação ao processamento de dados. Fonte: https://spark.apache.org/docs/latest/cluster-overview.html

A justificativa principal para maior desempenho do Spark em relação ao Hadoop é que o Spark executa a maioria de suas operações em memória RAM. Já o Hadoop executa suas operações em disco, que possui maior latência em comparação à memória RAM. A seguir temos a descrição dos Componentes de um programa Spark:

- SparkContext é um processo independente através do qual a aplicação de criptografia é executada em um cluster. Ele fornece o identificador ao mecanismo/cluster distribuído para que você possa usar os recursos das máquinas distribuídas em seu trabalho;

- O seu programa de aplicação que utilizará o objeto SparkContext seria conhecido como programa de driver como pode ser observado na Figura 2.4. Especificamente, para ser executado em um cluster, o SparkContext se conecta a vários tipos de gerenciadores de cluster (como o próprio gerenciador de cluster autônomo do Spark, o Apache Mesos ou o YARN do Hadoop), que alocam recursos em aplicativos;

- Uma vez conectado, a Spark assume os executores em nós distribuídos no cluster, que são processos nos nós distribuídos que executam cálculos e armazenam dados para sua aplicação. Em seguida, ele envia seu código de aplicativo para os executores através do SparkContext. Finalmente, as tarefas são enviadas aos executores para executá-lo e completá-lo.

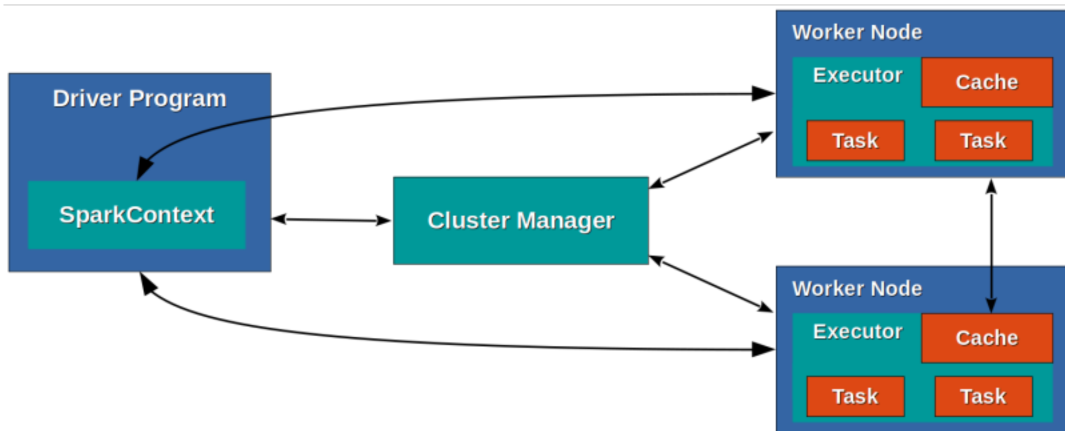

Figura 2.4. SparkContext envia tarefas para os executores executarem. Fonte: https://spark.apache.org/docs/latest/cluster-overview.html

4. O ecossistema do Spark

Além da API do Spark, existem bibliotecas adicionais que fazem parte do seu ecossistema e fornecem capacidades adicionais para as áreas de análise de Big Data e aprendizado de máquina como pode ser observado na Figura 2.5.

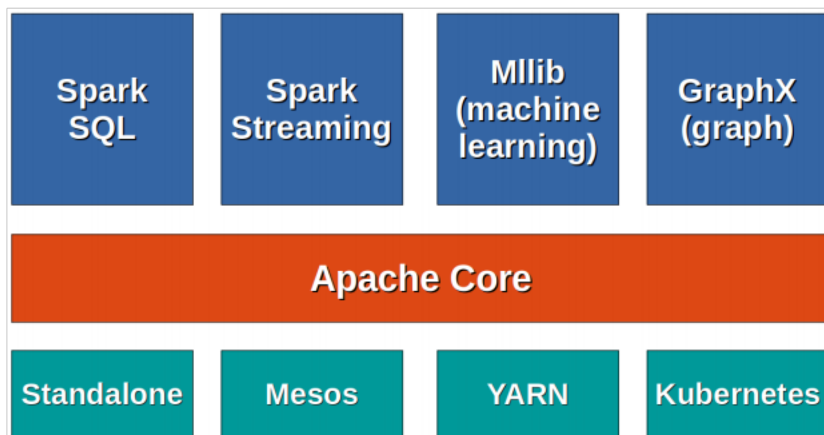

Figura 2.5. Ecossistema Spark

Estas bibliotecas incluem:

- Spark Streaming: O Spark Streaming pode ser usado para processar dados de streaming em tempo real baseado na computação de microbatch. Para isso é utilizado o DStream que é basicamente uma série de RDD para processar os dados em tempo real;

- Spark SQL: Spark SQL fornece a capacidade de expor os conjuntos de dados Spark através de uma API JDBC. Isso permite executar consultas no estilo SQL sobre esses dados usando ferramentas tradicionais de BI e de visualização. Além disso, também permite que os usuários usem ETL para extrair seus dados em diferentes formatos (como JSON, Parquet, ou um banco de dados), transformá-los e expô-los para consultas ad-hoc;

- Spark MLlib: MLlib é a biblioteca de aprendizado de máquina do Spark, que consiste em algoritmos de aprendizagem, incluindo a classificação, regressão, clustering, filtragem colaborativa e redução de dimensionalidade;

- Spark GraphX: é uma nova API do Spark para grafos e computação paralela. Em alto nível, o GraphX estende o Spark RDD para grafos. Para apoiar a computação de grafos, o GraphX expõe um conjunto de operadores fundamentais (por exemplo, subgrafos e vértices adjacentes), bem como uma variante optimizada do Pregel. Além disso, o GraphX inclui uma crescente coleção de algoritmos para simplificar tarefas de análise de grafos.

5. Tipos de Gerenciador de Cluster

O sistema atualmente oferece suporte a três gerenciadores de cluster:

- Standalone: Um gerenciador de cluster simples, significa que Spark ocupa o lugar em cima do HDFS (Hadoop Distributed File System) e o espaço é alocado para HDFS, explicitamente. Aqui, Spark e MapReduce serão executados lado a lado para cobrir todos os trabalhos do Spark no cluster;

- Apache Mesos: O Apache Mesos é um gerenciador de cluster centralizado e com tolerância a falhas, projetado para ambientes de computação distribuída. Mesos é um projeto de código aberto e foi desenvolvido na Universidade da Califórnia em Berkeley. Ele fornece gerenciamento de recursos e isolamento, agendamento de CPU e memória em todo o cluster. Mesos junta recursos físicos múltiplos em um único virtual. De certa forma, é o oposto da virtualização clássica, onde um único recurso físico é dividido em múltiplos recursos virtuais. Com o Apache Mesos, você pode criar / agendar frameworks como o Apache Spark;

- Hadoop YARN: A implantação do Yarn Hadoop significa que o Spark é executado no YARN. Isso ajuda a integrar o Spark no ecossistema Hadoop ou na pilha Hadoop. Ele permite que outros componentes sejam executados em cima da pilha;

- Kubernetes: Kubernetes é uma plataforma de código aberto para fornecer infraestrutura centrada em contêineres.

6. Prática no Google Colab

O objetivo deste laboratório é analisar dados de logs de acessos em um servidor web Apache, dados não estruturados, a fim de responder as seguintes perguntas:

- Número de hosts únicos;
- O total de erros 404;
- As 5 URLs que mais causaram erro 404;
- Quantidade de erros 404 por dia;
- O total de bytes retornados nas requisições web.

Dados:

- Jul 01 to Jul 31, ASCII format, 20.7 MB gzip compressed, 205.2 MB.
- Aug 04 to Aug 31, ASCII format, 21.8 MB gzip compressed, 167.8 MB.

Sobre o dataset: Esses dois conjuntos de dados possuem todas as requisições HTTP para o servidor da NASA Kennedy Space Center WWW na Flórida para um período específico. Os logs estão em arquivos ASCII com uma linha por requisição com as seguintes colunas:

- Host fazendo a requisição. Um hostname quando possível, caso contrário o endereço de internet se o nome não puder ser identificado.
- Timestamp no formato "DIA/MÊS/ANO:HH:MM:SS TIMEZONE"
- Requisição (entre aspas);
- Código do retorno HTTP;
- Total de bytes retornados.

O código abaixo será executado no Google Colab. Para executá-lo, você precisa ter uma conta Google. A linha que inicia com um símbolo de exclamação significa um comando que será executado na máquina virtual do Google Colab que usa o sistema operacional Linux, distribuição Ubuntu.

```
1 !apt-get update
2 !apt-get install openjdk-8-jdk-headless -qq > /dev/null
3 import os
4 os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
5 !update-alternatives --set java /usr/lib/jvm/java-8-openjdk-amd64/jre/
    bin/java
6 !java -version
```

Instale o Spark na máquina virtual do Google Colab:

```
!pip install pyspark
```

Obtenção das bases de dados (comando do Linux):

```
!git clone https://github.com/leonardoamorim/aulapython.git
!ls aulapython
```

Descompactando as bases de dados (comando do Linux):

```
! gunzip aulapython/NASA_access_log_Jul95.gz
! gunzip aulapython/NASA_access_log_Aug95.gz
```

Verificando o tamanho das bases de dados (comando do Linux):

```
! du -h aulapython/NASA_access_log_Jul95
! du -h aulapython/NASA_access_log_Aug95
```

Verificando a quantidade de linhas dos arquivos de texto das bases de dados (comando do Linux):

```
! wc -l aulapython/NASA_access_log_Jul95
! wc -l aulapython/NASA_access_log_Aug95
```

Importando as bibliotecas SparkConf e SparkContext para criação do RDD:

```
from pyspark import SparkConf, SparkContext
```

Importando a biblioteca **add** para realizar operações de agregação como somar dois elementos;

```
from operator import add
```

Definindo a configuração do RDD. O processo será executado localmente com o nome "Laboratorio" e usará até 5GB de memória RAM:

```
configuracao = (SparkConf().setMaster("local").setAppName("Laboratorio")
                .set("spark.executor.memory", "5g"))
```

```
sc = SparkContext(conf = configuracao)
```

Carregando as bases de dados em um RDD:

```
julho = sc.textFile('aulapython/NASA_access_log_Jul95')
agosto = sc.textFile('aulapython/NASA_access_log_Aug95')
```

Fazendo cache dos RDDs:

```
julho = julho.cache()
agosto = agosto.cache()
```

Chamamos a função **cache** para armazenar o RDD criado na etapa acima no cache, para que o Spark não precise calculá-lo toda vez que o usarmos para consultas de dados adicionais. Observe que **cache** é uma operação lenta. O Spark não armazena imediatamente os dados na memória quando chamamos **cache**. Na verdade, ocorre quando uma ação é chamada em um RDD.

Calculando o número de hosts distintos dos meses de julho e agosto:

```
contagem_julho = julho.map(lambda line: line.split(' ')[0]).distinct().
    count()
print("Numero de hosts distintos no mes de Julho:", contagem_julho)

contagem_agosto = agosto.map(lambda line: line.split(' ')[0]).distinct
    ().count()
print("Numero de hosts distintos no mes de Agosto:", contagem_agosto)
```

Função para verificar se em uma linha existe um **codigohttp** igual a 404:

```
def codigo404(linha):
    try:
        codigohttp = linha.split(' ')[-2]
        if codigohttp == '404':
            return True
    except:
        pass
    return False

erros404_julho = julho.filter(codigo404).cache()
erros404_agosto = agosto.filter(lambda linha: linha.split(' ')[-2] == '
    404').cache()

print('Erros 404 em Julho: %s' % erros404_julho.count())
print('Erros 404 em Agosto: %s' % erros404_agosto.count())
```

Função para retornar as top 5 urls com mais erros 404:

```
def top5_hosts404(rdd):
    urls = rdd.map(lambda linha: linha.split('"')[1].split(' ')[1])
    counts = urls.map(lambda urls: (urls, 1)).reduceByKey(add)
    top5 = counts.sortBy(lambda par: -par[1]).take(5)
        return top5

top5_hosts404(erros404_julho)
top5_hosts404(erros404_agosto)
```

Calculando a Quantidade de erros 404 por dia:

```
def contador_dias_404(rdd):
    dias = rdd.map(lambda linha: linha.split('[')[1].split(':')[0])
    counts = dias.map(lambda dia: (dia, 1)).reduceByKey(add).collect()
    return counts

contador_dias_404(erros404_julho)

contador_dias_404(erros404_agosto)
```

Calculando a quantidade de bytes acumulados:

```
def quantidade_bytes_acumulados(rdd):
    def contador(linha):
        try:
            count = int(linha.split(" ")[-1])
            if count < 0:
                raise ValueError()
            return count
```

```python
        except:
            return 0

    count = rdd.map(contador).reduce(add)
    return count

print('Quantidade de bytes total em Julho: %s' %
    quantidade_bytes_acumulados(julho))
print('Quantidade de bytes total em Agosto: %s' %
    quantidade_bytes_acumulados(agosto))
```

Finalizando o RDD:

```python
sc.stop()
```

Referências

[1] https://spark.apache.org/

[2] Fonte oficial do dateset: http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html