Ajustes de SO, Java, PySpark e dados (shell + Python)

```
# ========================== CÉLULA 1 ==========================
# Linhas que começam com "!" são COMANDOS DE SHELL executados pelo Jupyter
# diretamente no sistema operacional (Ubuntu/Debian no Colab/WSL/etc.). Esses
# comandos NÃO rodam no interpretador Python — eles chamam binários do SO.

# Atualiza o índice de pacotes do apt (lista local de pacotes/versões disponíveis nos
# Não instala nada; apenas sincroniza metadados para que instalações futuras encontre
!apt-get update

# Instala o JDK 8 (headless = sem componentes gráficos).
# O Apache Spark exige uma JVM. Versões do Spark são compatíveis com Java 8/11/17 (ve
# O " -qq > /dev/null " suprime saídas detalhadas, deixando o output mais limpo.
!apt-get install openjdk-8-jdk-headless -qq > /dev/null

# ---------- A partir daqui é Python ----------
import os  # Módulo padrão do Python para interagir com o Sistema Operacional (variáv

# Variável de ambiente JAVA_HOME: aponta para o diretório raiz do JDK.
# Diversas ferramentas (incluindo o Spark) usam isso para localizar 'java' e 'javac'.
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"

# update-alternatives permite escolher qual versão do 'java' será o padrão do sistema
# quando existem múltiplas instalações. Aqui, fixamos a versão instalada acima.
!update-alternatives --set java /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java

# Checa a versão ativa do Java para confirmar que a configuração está OK.
!java -version

# Instala o cliente PySpark (biblioteca Python que conversa com o engine do Spark).
# Sem isso, "from pyspark import ..." não funcionaria.
!pip install pyspark

# Baixa (clona) o repositório com os arquivos de exemplo e os datasets (logs da NASA)
# "git clone <url>" cria uma pasta local "aulapython/" com o conteúdo remoto.
!git clone https://github.com/leonardoamorim/aulapython.git

# Lista os arquivos dentro de "aulapython/" para verificar o clone com sucesso.
! ls aulapython/

# Descompacta os arquivos de log (.gz) para .txt (sem compressão), assim o Spark e o
# conseguem ler normalmente. 'gunzip' substitui o arquivo .gz pelo extraído.
! gunzip aulapython/NASA_access_log_Jul95.gz
! gunzip aulapython/NASA_access_log_Aug95.gz

# Mostra o tamanho (uso de disco) de cada arquivo (opção -h = "human readable": KB/MB
! du -h aulapython/NASA_access_log_Jul95
! du -h aulapython/NASA_access_log_Aug95

# Conta quantas linhas existem (cada linha = 1 requisição registrada).
! wc -l aulapython/NASA_access_log_Jul95
! wc -l aulapython/NASA_access_log_Aug95

# Mostra as 10 primeiras linhas do arquivo de Julho para inspecionar o formato:
# Formato comum (Common Log Format estendido):
# host ident authuser [date:time zone] "request" status bytes
! head -n10 aulapython/NASA_access_log_Jul95
```

```
update-alternatives: using /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java to provide /usr
openjdk version "1.8.0_462"
OpenJDK Runtime Environment (build 1.8.0_462-8u462-ga~us1-0ubuntu2~22.04.2-b08)
OpenJDK 64-Bit Server VM (build 25.462-b08, mixed mode)
Requirement already satisfied: pyspark in /usr/local/lib/python3.12/dist-packages (3.5.1)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.12/dist-packages
Cloning into 'aulapython'...
remote: Enumerating objects: 51, done.
remote: Total 51 (delta 0), reused 0 (delta 0), pack-reused 51 (from 1)
Receiving objects: 100% (51/51), 90.32 MiB | 35.09 MiB/s, done.
Resolving deltas: 100% (17/17), done.
 airlinedelaycauses_DelayedFlights.csv.part01.rar
 airlinedelaycauses_DelayedFlights.csv.part02.rar
 airlinedelaycauses_DelayedFlights.csv.part03.rar
 class_summary_statistics_asyncio.py
 class_summary_statistics_numba.py
 class_summary_statistics.py
 colaboradores.csv
 colaboradores_data_missing.csv
 exercicio_01.py
 funcionarios.json
'HPC_com_Python_para_Big_Data_lab_vecAdd_PyCUDA (1).ipynb'
 IRIS.csv
 iris.data
 KNN-RegressaoLogistica.ipynb
 NASA_access_log_Aug95.gz
 NASA_access_log_Jul95.gz
'Operações avançadas com DataFrame usando PySpark.ipynb'
'Operacoes com dados faltantes.ipynb'
'Operacoes com Datas e Timestamps.ipynb'
'Pandas - Intro.ipynb'
 programa1.py
 programa2.py
 programa3.py
'Pyspark - Operações Básicas com DF.ipynb'
' Python para Big Data - Introdução a Programação Funcional com Python Funções Lambda em
'RAPIDS(1).ipynb'
 Regressão_linear_com_Python.ipynb
 Regressão_logística_com_Python.ipynb
 titanic_test.csv
 titanic_train.csv
 Tutorial_Cuda_no_Colab.ipynb
 USA_Housing.csv
 Visualizacao-de-Dados-Dataset-Iris.ipynb
196M    aulapython/NASA_access_log_Jul95
161M    aulapython/NASA_access_log_Aug95
1891714 aulapython/NASA_access_log_Jul95
1569898 aulapython/NASA_access_log_Aug95
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0"
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html
199.120.110.21 - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/missions/sts-73/sts-73-patc
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /images/NASA-logosmall.gif HTTP/
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/video/livevic
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/countdown.html H
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
129.94.144.152 - - [01/Jul/1995:00:00:13 -0400] "GET / HTTP/1.0" 200 7074
```

Texto explicativo sobre RDD (docstring)

```
# ========================== CÉLULA 2 ==========================
"""
# Conjunto de dados distribuído resiliente (RDD)
```

```
4
5 O RDD é a API de baixo nível do Spark: uma coleção distribuída e imutável,
6 particionada e processável em paralelo. Você aplica TRANSFORMAÇÕES (lazy),
7 como `map`, `filter`, `reduceByKey`, e executa com AÇÕES como `count`,
8 `collect`, `take`.
9
10 Quando usar RDDs?
11 - Quando precisa de controle de baixo nível.
12 - Dados não estruturados (logs, texto).
13 - Preferência por programação funcional ao invés de expressões SQL.
14 - Não precisa impor esquema (colunas).
15 - Aceita abrir mão de otimizações automáticas de DataFrames (Catalyst/Tungsten).
16 """
```

```
'\n# Conjunto de dados distribuído resiliente (RDD)\n\nO RDD é a API de baixo nível do Sp
ark: uma coleção distribuída e imutável,\nparticionada e processável em paralelo. Você ap
lica TRANSFORMAÇÕES (lazy),\ncomo `map`, `filter`, `reduceByKey`, e executa com AÇÕES com
o `count`,\n`collect`, `take`.\n\nQuando usar RDDs?\n- Quando precisa de controle de baix
o nível.\n- Dados não estruturados (logs, texto).\n- Preferência por programação funciona
```

Configuração e criação do SparkContext

```
1 # ========================== CÉLULA 3 ==========================
2 # "from pyspark import SparkConf, SparkContext"
3 # - SparkConf: objeto de configuração (master, appName, memória, etc).
4 # - SparkContext: ponto de entrada da API de RDD (baixo nível).
5 from pyspark import SparkConf, SparkContext
6
7 # 'add' é a função de soma do módulo 'operator', usada em reduções (reduceByKey(add))
8 # para somar contagens por chave de forma legível e eficiente.
9 from operator import add
10
11 # Cria uma configuração de Spark:
12 # - SparkConf(): construtor do objeto de configuração do Spark.
13 # - .setMaster("local"): executa localmente com 1 thread (útil p/ testes). DICA: "loc
14 # - .setAppName("Exercicio Nasa Logs"): nome que aparece na UI do Spark e nos logs.
15 # - .set("spark.executor.memory", "5g"): memória destinada a cada executor (~5 GiB).
16 #   Em modo "local", você terá 1 executor; em cluster, isso vale por executor.
17 configuracao = (SparkConf()
18                 .setMaster("local")
19                 .setAppName("Exercicio Nasa Logs")
20                 .set("spark.executor.memory", "5g"))
21
22 # 'type(obj)': função nativa do Python que retorna o tipo do objeto, útil para checag
23 type(configuracao)
24
25 # Cria o SparkContext com a configuração acima.
26 # IMPORTANTE: deve haver apenas UM SparkContext por aplicação.
27 sc = SparkContext(conf = configuracao)
28
29 # Confirma que 'sc' é um SparkContext.
30 type(sc)
```

```
pyspark.context.SparkContext
def __init__(master: Optional[str]=None, appName: Optional[str]=None, sparkHome:
Optional[str]=None, pyFiles: Optional[List[str]]=None, environment: Optional[Dict[str,
Any]]=None, batchSize: int=0, serializer: 'Serializer'=CPickleSerializer(), conf:
Optional[SparkConf]=None, gateway: Optional[JavaGateway]=None, jsc:
Optional[JavaObject]=None, profiler_cls: Type[BasicProfiler]=BasicProfiler,
udf_profiler_cls: Type[UDFBasicProfiler]=UDFBasicProfiler, memory_profiler_cls:
Type[MemoryProfiler]=MemoryProfiler)
```

```
/usr/local/lib/python3.12/dist-packages/pyspark/context.py
Main entry point for Spark functionality. A SparkContext represents the
connection to a Spark cluster, and can be used to create :class:`RDD` and
broadcast variables on that cluster.

When you create a new SparkContext, at least the master and app name should
```

Leitura dos arquivos com RDD e cache

```
# ========================== CÉLULA 4 ==========================
# Lê os arquivos como RDDs de strings (cada elemento = 1 linha do arquivo).
# 'sc.textFile(path)' é LAZY: a leitura real ocorre apenas quando executamos uma ACTI
julho = sc.textFile('aulapython/NASA_access_log_Jul95')
agosto = sc.textFile('aulapython/NASA_access_log_Aug95')

# 'cache()': mantém o RDD em memória após a primeira computação.
# Isso evita recomputar a cadeia de transformações em AÇÕES subsequentes,
# acelerando o workflow.
julho = julho.cache()
agosto = agosto.cache()

# Inspeção via shell das primeiras linhas do arquivo de julho (não é Spark).
! head -n3 aulapython/NASA_access_log_Jul95
```

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 2
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts-
```

Explorando a estrutura de uma linha de log

```
# ========================= CÉLULA 5 =========================
# Exemplo de linha de log (string). Isso facilita explicar parsing e tokens.
exemplo = '199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1

# Checa o tipo do objeto (deve ser 'str').
type(exemplo)

# 'split()' (sem argumentos) separa por QUALQUER espaço em branco.
operacao = exemplo.split()

print(operacao)   # Lista de tokens da linha
print(operacao[0])# 'operacao[0]' usa INDEXAÇÃO por '[]' para obter o 1º item (host/I

# Mapeia as primeiras 5 linhas do RDD 'julho' para o 1º token (host) e coleta para in
julho.map(lambda line: line.split()[0]).take(5)

# Variante com separador explícito: split(' ') (equivalente neste caso).
# 'distinct()' remove duplicatas via shuffle.
# 'count()' é ACTION: dispara a execução e retorna a contagem de elementos (hosts úni
julho.map(lambda line: line.split(' ')[0]).distinct().count()
```

```
['199.72.81.55', '-', '-', '[01/Jul/1995:00:00:01', '-0400]', '"GET', '/history/apollo/',
199.72.81.55
81983
```

Sobre lambda e []:

lambda line: ... cria uma função anônima que recebe line (cada linha do RDD) e retorna algo.

line.split(' ')[0] usa indexação com colchetes para pegar o primeiro elemento da lista de tokens.

Função utilitária para hosts distintos

```
# ========================= CÉLULA 6 =========================
def obterQtdHosts(rdd):
  """
  Retorna a quantidade de hosts distintos em um RDD de linhas de log.
  Etapas:
    1) line.split(' ')[0] -> extrai o host/IP (1º token).
    2) distinct() -> remove host repetido.
    3) count() -> ACTION que retorna a quantidade de elementos únicos.
  """
  qtdHosts = rdd.map(lambda line: line.split(' ')[0]).distinct().count()
  return qtdHosts

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

# Usando a função
obterQtdHosts(julho)

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

# Usando a função
obterQtdHosts(agosto)
```

```
Numero de hosts distintos no mes de Julho: 81983
Numero de hosts distintos no mes de Agosto: 75060
75060
```

Função para detectar linhas com código HTTP 404

```
# ========================== CÉLULA 7 ==========================
def codigo404(linha):
    """
    Retorna True se o penúltimo token da linha for '404' (código HTTP de "Not Found")
    Estrutura típica de linha:
      ... "REQUEST" STATUS BYTES
      penúltimo token = STATUS
      último token    = BYTES (pode ser '-')
    """
    try:
        # 'linha.split()[-2]' pega o penúltimo token usando índice negativo.
        # Em Python, índices negativos contam a partir do fim: [-1] último, [-2] penú
        codigohttp = linha.split()[-2]
        if codigohttp == '404':
            return True
    except:
        # Linhas malformadas (sem tokens suficientes) ou erros de parsing caem aqui.
        pass
    return False

# Teste da função (esta linha tem status 200, então deve retornar False)
codigo404(exemplo)

# Apenas para relembrar o conteúdo
exemplo

print(exemplo)

# Linha sintética com status 404 (para validar True)
exemplo2 = '199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/
codigo404(exemplo2)  # Esperado: True
```

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
True
```

Filtrando os 404 por mês e contando

```
# ========================== CÉLULA 8 ==========================
# Filtra linhas de julho que têm status 404 usando a função robusta 'codigo404'.
# 'cache()' para reutilizar os resultados sem recomputar o filtro.
erros404_julho = julho.filter(codigo404).cache()

# Em agosto usamos uma lambda equivalente, mas MENOS robusta (assume que sempre exist
# Preferir a versão com try/except para tolerar linhas malformadas.
erros404_agosto = agosto.filter(lambda linha: linha.split(' ')[-2] == '404').cache()

# 'count()' é ACTION que dispara a execução e retorna a quantidade de elementos no RD
print('Erros 404 em Julho: %s' % erros404_julho.count())
print('Erros 404 em Agosto: %s' % erros404_agosto.count())
```

```
Erros 404 em Julho: 10845
Erros 404 em Agosto: 10056
```

Extração de URL dentro do bloco "REQUEST"

```
# ========================== CÉLULA 9 ==========================
print(exemplo)
print(exemplo.split('"'))                # Quebra a linha nas aspas -> ["antes", 'GE
print(exemplo.split('"')[1].split(' ')[1])# Pega o bloco do meio (índice 1) e, dele,

# Extrai apenas a URL das linhas 404 de julho (amostra de 5)
erros404_julho.map(lambda linha: linha.split('"')[1].split(' ')[1]).take(5)

# Mapeia cada URL para o par (url, 1), ou seja, 1 ocorrência por linha/URL. Amostra.
erros404_julho.map(lambda linha: linha.split('"')[1].split(' ')[1]) \
              .map(lambda urls: (urls, 1)) \
              .take(5)

# Pipeline completo: URLs -> (url, 1) -> contagem por chave com reduceByKey(add)
urls = erros404_julho.map(lambda linha: linha.split('"')[1].split(' ')[1])
counts = urls.map(lambda urls: (urls, 1)).reduceByKey(add)

counts.take(10)  # amostra de 10 pares (url, contagem)
type(counts)     # RDD[Tuple[str, int]]
```

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
['199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] ', 'GET /history/apollo/ HTTP/1.0', ' 200
/history/apollo/
```

pyspark.rdd.PipelinedRDD
def __init__(prev: RDD[T], func: Callable[[int, Iterable[T]], Iterable[U]], preservesPartitioning: bool=False, isFromBarrier: bool=False)

/usr/local/lib/python3.12/dist-packages/pyspark/rdd.py
Examples

Pipelined maps:

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
```

Por que reduceByKey(add)?

Ele faz combiner local (agregação por partição) ANTES de embaralhar os dados (shuffle), sendo muito mais eficiente que groupByKey().

Top 5 URLs que mais geram 404

```
# ========================== CÉLULA 10 ==========================
def top5_hosts404(rdd):
    """
    Calcula as 5 URLs com mais erros 404 em um RDD de linhas de log 404.
    Passos:
      1) extrair URL: linha.split('"')[1].split(' ')[1]
      2) map para (url, 1)
      3) reduceByKey(add) para somar
      4) sortBy(lambda par: -par[1]) para ordenar por contagem desc
      5) take(5)
    """
    urls = rdd.map(lambda linha: linha.split('"')[1].split(' ')[1])   # '/history/apo
    counts = urls.map(lambda urls: (urls, 1)).reduceByKey(add)        # (url, soma)
    top5 = counts.sortBy(lambda par: -par[1]).take(5)                 # ordena por -c
```

```
    return top5

# Retorna lista com 5 tuplas (url, contagem), ordenadas por maior contagem
top5_hosts404(erros404_julho)
top5_hosts404(erros404_agosto)
```

```
[('/pub/winvn/readme.txt', 1337),
 ('/pub/winvn/release.txt', 1185),
 ('/shuttle/missions/STS-69/mission-STS-69.html', 683),
 ('/images/nasa-logo.gif', 319),
 ('/shuttle/missions/sts-68/ksc-upclose.gif', 253)]
```

Extraindo data e contando 404 por dia

```
# ========================== CÉLULA 11 ==========================
print(exemplo)                      # relembrando o formato geral
print(exemplo.split('[')[1])        # pega tudo após o primeiro '[': "01/Jul/1995:00:
print(exemplo.split('[')[1].split(':')[0])  # fica só a parte "01/Jul/1995"

# Função para contar erros 404 por DIA (retorna lista Python de tuplas (dia, contagem
def contador_dias_404(rdd):
    """
    1) Extrai o dia no formato 'dd/Mon/yyyy' do trecho entre colchetes.
    2) Mapeia para (dia, 1) e soma com reduceByKey(add).
    3) collect() traz o resultado (lista de tuplas) para o Driver.
    """
    dias = rdd.map(lambda linha: linha.split('[')[1].split(':')[0])  # '01/Jul/1995'
    counts = dias.map(lambda dia: (dia, 1)).reduceByKey(add).collect()
    return counts

print(contador_dias_404(erros404_julho))  # lista Python de ("01/Jul/1995", 123), ...
```

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
01/Jul/1995
[('01/Jul/1995', 316), ('09/Jul/1995', 348), ('21/Jul/1995', 334), ('26/Jul/1995', 336), (
```

Ordenando os resultados (Python puro)

```
# ========================== CÉLULA 12 ==========================
# Exemplo bobo de ordenação alfabética com Python puro (sem Spark).
a = ("b", "g", "a", "d", "f", "c", "h", "e")
x = sorted(a)  # 'sorted' retorna NOVA lista ordenada; não altera 'a'.
print(x)

# Agora, ordenando os dias por quantidade de 404 (descendente) com Python puro:
# 'contador_dias_404(...)' já devolve uma LISTA Python de tuplas (dia, contagem).
# 'sorted(lista, key=..., reverse=...)' ordena essa lista.
#
# key=lambda x: -x[1]
# - 'lambda x: ...' cria uma função anônima.
# - 'x' é cada tupla (dia, contagem).
# - 'x[1]' é a CONTAGEM (2º elemento da tupla) — indexação via '[]'.
# - '-x[1]' usa o NEGATIVO da contagem para transformar a ordenação padrão (asc) em
#
# Equivalente (mais legível): key=lambda x: x[1], reverse=True

sorted(contador_dias_404(erros404_julho), key=lambda x: -x[1])
```

```
20
21 # Ordenação por TEXTO (dia), desc: para strings, use reverse=True (não faz sentido r
22 sorted(contador_dias_404(erros404_julho), key=lambda x: x[0], reverse=True)
23
24 # Ordenação por TEXTO (dia), asc:
25 sorted(contador_dias_404(erros404_julho), key=lambda x: x[0])
26
27 # Reaproveita a função para agosto (lista não ordenada):
28 contador_dias_404(erros404_agosto)
29
30 # Conversão de string para inteiro (exemplo auxiliar):
31 int('10')  # retorna 10 (int)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
10
```

Soma total de bytes transferidos

```
1 # ========================= CÉLULA 13 =========================
2 def quantidade_bytes_acumulados(rdd):
3     """
4     Soma o campo de BYTES (último token) de cada linha.
5     O campo pode ser '-' (sem valor) -> tratamos como 0.
6     Também ignoramos valores inválidos via try/except.
7     """
8     def contador(linha):
9         try:
10            # 'linha.split(" ")[-1]' pega o ÚLTIMO token (índice -1) — BYTES.
11            # int(...) converte de string para inteiro; se for '-', cai no except e v
12            count = int(linha.split(" ")[-1])
13            if count < 0:
14                # bytes negativos não fazem sentido — dispara exceção para tratar com
15                raise ValueError()
16            return count
17        except:
18            # Retorna 0 para linhas malformadas ou com '-' em BYTES.
19            return 0
20
21    # 'map(contador)': aplica a função acima em cada linha, produzindo um RDD de inte
22    # 'reduce(add)': soma todos os inteiros de forma DISTRIBUÍDA (reduceByKey é para
23    count = rdd.map(contador).reduce(add)
24    return count
25
26 print('Quantidade de bytes total em Julho: %s' % quantidade_bytes_acumulados(julho))
27 print('Quantidade de bytes total em Agosto: %s' % quantidade_bytes_acumulados(agosto)
```

```
Quantidade de bytes total em Julho: 38695973491
Quantidade de bytes total em Agosto: 26828341424
```