Aluna:Maria Clara Fernandes

Primeira Célula — Preparação do Ambiente

Nesta célula ocorre a preparação do ambiente para processar os logs da NASA. As principais ações são:

- Atualiza os pacotes do sistema (apt-get update).
- Instala o Java Development Kit (JDK 8), necessário para o Apache Spark.
- Define a variável de ambiente JAVA_HOME.
- Instala a biblioteca PySpark via pip.
- Clona o repositório com os arquivos de log.
- Descompacta os arquivos .gz dos logs de julho e agosto de 1995.
- Mostra informações iniciais:
  - Tamanho dos arquivos
  - Quantidade de linhas
  - Exemplo das primeiras linhas do log

Segunda Célula — Comentário Explicativo: Conceito de RDD

Esta célula contém somente explicações teóricas (comentário multilinha). Ela descreve o conceito de RDD (Resilient Distributed Dataset):

O que é um RDD?

- É uma estrutura de dados distribuída (dividida entre os nós do cluster).
- Imutável: uma vez criado, não pode ser alterado — apenas transformado em outro RDD.
- Projetado para processar grandes volumes de dados com eficiência.
- Permite operações funcionais em paralelo.

Operações comuns com RDDs:

- map(): transforma cada item.
- filter(): filtra os itens com base em uma condição.
- reduceByKey(): agrega os valores com a mesma chave.
- take(n): retorna os n primeiros elementos.
- count(): conta quantos elementos existem.

É recomendado usar RDDs quando:

- Os dados são não estruturados (como logs de texto).
- Não é necessário impor esquema (colunas/tipos) como em DataFrames.
- Você prefere programação funcional ao invés de SQL.
- Precisa de maior controle manual, mesmo sem otimizações automáticas como Catalyst/Tungsten.

Terceira Célula — Configuração do SparkContext

Esta célula configura o Spark, criando o SparkContext, ponto de entrada para trabalhar com RDDs no PySpark.

Principais etapas:

- SparkConf(): cria o objeto de configuração do Spark:
  - .setMaster("local"): executa localmente.
  - .setAppName("Exercicio Nasa Logs"): define o nome da aplicação.

o .set("spark.executor.memory", "5g"): define a memória do executor.
- SparkContext(conf=...): inicializa o Spark com as configurações definidas.
- from operator import add: importa a função add, usada para somas em agregações (reduceByKey(add)).
- type(sc): verifica o tipo do objeto sc, que deve ser SparkContext.

Quarta Célula — Leitura dos Arquivos de Log
Nesta célula, os arquivos de log são carregados como RDDs:

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

- Cada linha do arquivo vira um elemento do RDD (string).
- Os RDDs são cacheados em memória (.cache()) para melhorar o desempenho em análises futuras.
- Também é exibido o conteúdo das 3 primeiras linhas do arquivo de julho, para inspecionar o formato.

Quinta Célula — Análise Inicial dos Dados (Extração de Hosts)
Aqui começa a análise dos dados dos logs da NASA:
Etapas:
1. Cria uma string de exemplo com uma linha de log para facilitar o entendimento do formato.
2. Utiliza split() para dividir a linha em campos.
3. Extrai o primeiro campo (host/IP) com split()[0].
4. Aplica isso no RDD de julho:
o .map(lambda line: line.split()[0]): extrai os hosts reais.
o .distinct().count(): conta quantos hosts únicos acessaram o site em julho.

Esse é o primeiro insight analítico do projeto: quantos visitantes únicos acessaram o site da NASA naquele mês.

Sexta Célula — Função para Contar Hosts Únicos
Nesta célula, é criada uma função genérica que conta o número de hosts distintos em qualquer RDD de logs:

```
def obterQtdHosts(rdd):
    return rdd.map(lambda line: line.split(' ')[0]).distinct().count()
```

Aplicações:
- Usa a função obterQtdHosts(julho) e também a lógica direta para contar os hosts únicos de julho.
- Repete o mesmo para o RDD de agosto, usando tanto a função quanto o código direto.
- Isso melhora a organização e reutilização do código.

Célula 7

Executa-se um def executando o token, e caso a linha seja 404, mas há tem um caso a parte em que linhas malformadas ou erros.

Célula 8

Filtrando as linhas de julho e agosto com o status 404 e mostra a quantidade.

Célula 9

Mapeia e faz a extração da URL das linhas 4040 em julho, realiza uma pipeline completa fazendo uma contagem porchave com reduceByKey.
-Célula 10:
Esta célula implementa uma função para identificar as 5 URLs que mais geraram erros HTTP 404 (recurso não encontrado) nos arquivos de log da NASA dos meses de julho e agosto de 1995. A análise é feita sobre RDDs filtrados previamente para conter apenas linhas com erros 404.

-Célula11:
Esta célula tem como objetivo extrair a data (dia/mês/ano) das linhas dos logs que contêm erro 404 e contar quantas ocorrências de erro 404 aconteceram em cada dia. Onde a função contador_dias_404 realiza as seguintes etapas: extração do dia(extrai do trecho entre colchetes a data no formato dd/Mon/yyyy, ignorando o horário e o fuso), mapeamento para tuplas (para cada ocorrência, cria uma tupla com a data e o valor 1), agregação com reduceByKey(soma as ocorrências para cada dia), coleta dos resultados (transforma o RDD distribuído em uma lista Python para posterior manipulação).

-Célula12:
Esta célula demonstra como ordenar os dados coletados da análise anterior, usando funcionalidades nativas do Python (sem Spark).
Relatório do Código
Célula 13. a gente criou uma função chamada quantidade_bytes_acumulados, que serve pra calcular quantos bytes foram transferidos no total dentro do log.
A ideia é a seguinte: cada linha do log tem vários pedaços de informação separados por espaço, e o último pedaço token é a quantidade de bytes.
Só que às vezes esse campo vem com um traço (-) ou algum valor zoado. Nesses casos, a função trata como se fosse zero.
Também não deixa passar valores negativos, porque faz zero sentido ter bytes negativos.
Depois, a função roda assim:
1. map(contador) → pega cada linha e transforma em um número de bytes (ou 0).
2. reduce(add) → soma todos os números de forma distribuída no cluster do Spark.
No final, a gente imprime o total de bytes consumidos em Julho e em Agosto.

Célula 14. É bem simples: a gente só chama o sc.stop().

Isso é basicamente pra encerrar o Spark e liberar a memória e os recursos que estavam sendo usados, tanto na máquina local quanto se fosse o caso no cluster.
7