**Resumo expandido da aula - Hadoop**

**1. Sistemas de Arquivos Distribuídos**

**1.1 Google File System (GFS)**

* **O que é:** Sistema de arquivos distribuído desenvolvido pelo Google para atender às suas necessidades de armazenamento em larga escala; o GFS foi pioneiro em muitos conceitos adotados posteriormente por outros sistemas.
* **Características:**
  + Projetado para manipular arquivos muito grandes.
  + Otimizado para leituras e escritas sequenciais.
  + Usa replicação para tolerância a falhas.

**Arquitetura Simplificada do GFS:**

* **Master Node:** Gerencia metadata (nome dos arquivos, localização dos chunks, etc.).
* **Chunk Servers:** Armazenam os dados reais em pedaços (chunks).
* **Clientes:** Interagem com o Master para obter metadata e com os Chunk Servers para ler/escrever dados.

**1.2 Hadoop Distributed File System (HDFS)**

* **O que é:** Sistema de arquivos distribuído inspirado no GFS, parte do projeto Apache Hadoop; inspirado no GFS, o HDFS é amplamente utilizado em ambientes de big data, pois assegura o armazenamento de petabytes de dados e é otimizado para altas taxas de transferência.
* **Características:**
  + Armazena dados de forma redundante para garantir confiabilidade.
  + Otimizado para streaming de dados de alta largura de banda.
  + Estrutura mestre/escravo (NameNode e DataNodes).

**Arquitetura Simplificada do HDFS:**

* **NameNode:** Gerencia a metadata do sistema de arquivos.
* **DataNodes:** Armazenam e recuperam blocos quando solicitados pelos clientes ou pelo NameNode.
* **Clientes:** Interagem com o NameNode para operações de metadata e com os DataNodes para operações de leitura/escrita.

**2. Modelo de Processamento de Dados**

**MapReduce**

* **O que é:** Modelo de programação para processamento de grandes conjuntos de dados com um algoritmo paralelo distribuído em um cluster.
* **Fases Principais:**
  + **Map:** Processa e converte dados de entrada em pares chave-valor intermediários.
  + **Shuffle and Sort:** Organiza os pares intermediários para serem consumidos na fase reduce.
  + **Reduce:** Agrega os valores intermediários associados a uma mesma chave.

**Exemplo de Código MapReduce (Contagem de Palavras):**

**Mapper:**

public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] tokens = value.toString().split("\\s+");

for (String token : tokens) {

word.set(token);

context.write(word, one);

}

}

}

**Reducer:**

public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

context.write(key, new IntWritable(sum));

}

}

**3. Projeto Hadoop**

* **O que é:** Framework de software de código aberto para armazenamento distribuído e processamento de conjuntos de dados muito grandes usando o modelo MapReduce.
* **Componentes Principais:**
  + **HDFS:** Sistema de arquivos distribuído.
  + **MapReduce:** Modelo de processamento de dados.
* **Características:**
  + Escalabilidade horizontal.
  + Tolerância a falhas.
  + Comunidade ativa e ecossistema rico.

**Fluxo de Trabalho no Hadoop:**

1. **Dados são armazenados no HDFS.**
2. **Jobs MapReduce são submetidos ao cluster.**
3. **O JobTracker coordena a execução das tarefas nos TaskTrackers.**

**4. Apache Spark e Spark Stack**

**4.1 Apache Spark**

* **O que é:** Motor de análise unificado para processamento de dados em grande escala, mais rápido que o MapReduce tradicional.
* **Características:**
  + Processamento em memória para velocidade.
  + Suporte a várias linguagens (Java, Scala, Python, R).
  + APIs para diferentes cargas de trabalho (batch, streaming, machine learning).

**4.2 Spark Stack**

* **Componentes Principais:**
  + **Spark Core:** Funcionalidades básicas de processamento.
  + **Spark SQL:** Processamento de dados estruturados.
  + **Spark Streaming:** Processamento de dados em tempo real.
  + **MLlib:** Biblioteca de machine learning.
  + **GraphX:** Processamento de grafos.

**Exemplo de Código Spark (Contagem de Palavras):**

from pyspark import SparkContext

sc = SparkContext("local", "Word Count")

text\_file = sc.textFile("hdfs://path/to/file.txt")

counts = text\_file.flatMap(lambda line: line.split(" ")) \

.map(lambda word: (word, 1)) \

.reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://path/to/output")

**5. Apache Flink e Outros Mecanismos de Processamento de Dados**

**5.1 Apache Flink**

* **O que é:** Framework e motor de processamento de fluxo de dados para computação em cluster.
* **Características:**
  + Processamento de fluxo (streaming) e batch.
  + Baixa latência e alto throughput.
  + Gerenciamento avançado de estado e tolerância a falhas.

**Exemplo de Código Flink (Contagem de Palavras Streaming):**

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> text = env.socketTextStream("localhost", 9999);

DataStream<Tuple2<String, Integer>> counts = text

.flatMap(new LineSplitter())

.keyBy(0)

.sum(1);

counts.print();

env.execute("Socket Window WordCount");

**LineSplitter Function:**

public static final class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {

public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {

for (String word : value.split("\\s")) {

out.collect(new Tuple2<>(word, 1));

}

}

}

**5.2 Outros Mecanismos**

* **Apache Beam:**
  + Modelo unificado para definir pipelines de processamento de dados batch e streaming.
  + Suporta execução em diferentes backends (runners), como Flink, Spark, Google Cloud Dataflow.
* **Apache Storm:**
  + Sistema distribuído de computação em tempo real.
  + Processa dados em streams contínuos com baixa latência.
* **Dask:**
  + Biblioteca em Python para computação paralela.
  + Permite estender estruturas de dados familiares (arrays, dataframes) para clusters.

Tabela de Comparação entre Sistemas de Processamento de Dados em Grande Escala

|  |  |  |  |
| --- | --- | --- | --- |
| Característica | Hadoop MapReduce | Apache Spark | Apache Flink |
| Processamento | Batch | Batch e Streaming | Batch e Streaming |
| Velocidade | Baixa (opera em disco) | Alta (processamento em memória) | Muito alta (otimizado para fluxo contínuo) |
| APIs | Java | Java, Scala, Python, R | Java, Scala, Python, SQL |
| Arquitetura | Baseada em MapReduce com tarefas Map e Reduce | Motor unificado com módulos para diversas cargas de trabalho | Motor de processamento de fluxo de dados com suporte nativo a streaming |
| Modelo de Programação | Baseado em Map e Reduce | RDDs (Resilient Distributed Datasets), DataFrames, Datasets | DataStream API, DataSet API |
| Gerenciamento de Recursos | Gerenciado pelo YARN (Yet Another Resource Negotiator) | Integra-se com YARN, Mesos, Kubernetes, ou modo standalone | Suporta YARN, Mesos, Kubernetes, ou modo standalone |
| Armazenamento de Dados | HDFS (Hadoop Distributed File System) | Pode usar HDFS, HBase, Cassandra, Amazon S3 e outros | Pode usar HDFS, HBase, Amazon S3 e outros |
| Tolerância a Falhas | Alta, através de replicação de dados e recomputação de tarefas | Alta, com recomputação de tarefas perdidas e DAGs resilientes | Muito alta, com gerenciamento avançado de estado e checkpoints |
| Latência | Alta latência, adequado para batch | Baixa latência para batch e micro-batching em streaming | Latência muito baixa para processamento de streaming em tempo real |
| Suporte a SQL | Possível com ferramentas como Hive | Spark SQL assegura consultas SQL sobre dados estruturados | Flink SQL suporta consultas SQL em streams e dados em batch |
| Bibliotecas de ML | Apache Mahout (menos usado atualmente) | MLlib (biblioteca integrada e madura) | Flink ML (em desenvolvimento, menos madura) |
| Casos de Uso | Processamento em lote de grandes volumes de dados | Machine Learning, ETL, processamento interativo, streaming | Streaming em tempo real, processamento de eventos, aplicações com baixa latência |
| Comunidade e Maturidade | Muito madura, amplamente utilizada | Muito madura, ampla adoção | Relativamente nova, mas crescendo rapidamente |
| Facilidade de Uso | Curva de aprendizado íngreme, programação complexa | APIs amigáveis, suporte a múltiplas linguagens | APIs poderosas, mas requer conhecimento aprofundado |
| Escalabilidade | Altamente escalável | Altamente escalável | Altamente escalável |
| Integração com Ecossistema | Parte do ecossistema Hadoop (Hive, Pig, HBase, etc.) | Integra-se bem com várias ferramentas e sistemas (Hadoop, Cassandra, Kafka, etc.) | Integra-se com sistemas como Kafka, Elasticsearch e outros |
| Gerenciamento de Memória | Baseado em disco, menor uso de memória | Processamento em memória, requer mais memória | Processamento otimizado, uso eficiente da memória |
| Latência de Inicialização | Alta, devido ao overhead de inicialização de tarefas | Baixa, permite operações interativas | Muito baixa, adequado para aplicações sensíveis a latência |
| Aprendizado | Exige conhecimento profundo de Java e MapReduce | APIs de alto nível facilitam o desenvolvimento | APIs avançadas, documentação menos extensa que Spark |
| Suporte a Streaming | Limitado, não é nativo | Suporte a streaming através de micro-batching (Spark Streaming) | Suporte a streaming nativo com processamento evento a evento |
| Comunidade | Grande, mas foco está migrando para novas tecnologias | Muito ativa e grande, com contribuições constantes | Crescente, com foco em streaming e aplicações em tempo real |

Exemplo:

Vamos construir um exemplo que opera com a contagem de palavras em um conjunto de dados de texto usando PySpark, que é a interface Python para o Apache Spark. Assim, demonstraremos como usar o modelo MapReduce no Spark para processar dados em paralelo. Primeiramente, precisamos instalar o Apache Spark e o Java Development Kit (JDK) no ambiente do Google Colab.

# Instalação do Java Development Kit (JDK)

!apt-get install openjdk-8-jdk-headless -qq > /dev/null

# Download do Apache Spark

!wget -q http://mirror.nbtelecom.com.br/apache/spark/spark-3.1.2/spark-3.1.2-bin-hadoop2.7.tgz

# Descompactar o Apache Spark

!tar xf spark-3.1.2-bin-hadoop2.7.tgz

# Instalação do findspark para facilitar a configuração

!pip install -q findspark

**Explicações:**

* !apt-get install openjdk-8-jdk-headless: Instala o JDK necessário para executar o Spark.
* !wget: Faz o download do Apache Spark.
* !tar xf: Descompacta o arquivo baixado.
* !pip install findspark: Instala a biblioteca findspark que ajuda a localizar o Spark no sistema.

Em seguida, precisamos configurar as variáveis de ambiente necessárias para o Spark:

import os

os.environ["JAVA\_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64" # Define a variável de ambiente JAVA\_HOME

os.environ["SPARK\_HOME"] = "/content/spark-3.1.2-bin-hadoop2.7" # Define a variável de ambiente SPARK\_HOME

**Explicações:**

* os.environ["JAVA\_HOME"]: Especifica o caminho para o JDK instalado.
* os.environ["SPARK\_HOME"]: Especifica o caminho para o Spark descompactado.

Agora, iniciamos uma sessão Spark para começar a utilizar o PySpark.

import findspark

findspark.init()

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").appName("WordCount").getOrCreate() # Cria uma sessão Spark

**Explicações:**

* findspark.init(): Inicializa o findspark para localizar o Spark.
* from pyspark.sql import SparkSession: Importa a classe SparkSession do PySpark.
* SparkSession.builder...: Cria uma sessão Spark com o nome "WordCount".

Para este exemplo, usaremos um texto um pouco mais simples. Em uma aplicação real, você poderia ler de um arquivo armazenado no HDFS ou outro sistema de arquivos distribuído.

# Texto de exemplo

texto = ["Apache Spark é um mecanismo de análise unificado para processamento de dados em grande escala.",

"Ele fornece APIs em Python, Java, Scala e R.",

"O Spark é mais rápido que o Hadoop MapReduce."]

rdd = spark.sparkContext.parallelize(texto) # Cria um RDD a partir da lista de textos

**Explicações:**

* texto: Uma lista de strings que representa nossos dados.
* spark.sparkContext.parallelize(texto): Cria um RDD (Resilient Distributed Dataset) a partir da lista.

Agora, aplicamos uma série de transformações para realizar a contagem de palavras.

words\_rdd = rdd.flatMap(lambda line: line.split(" ")) # Divide cada linha em palavras

word\_pairs = words\_rdd.map(lambda word: (word.lower().strip(",."), 1)) # Associa o valor 1 a cada palavra

word\_counts = word\_pairs.reduceByKey(lambda a, b: a + b) # Soma as ocorrências de cada palavra

**Explicações:**

* flatMap: Transforma cada linha em uma lista de palavras e achata o resultado em um único RDD de palavras.
* lambda line: line.split(" "): Função anônima que divide a linha em palavras.
* map: Transforma cada palavra em um par (palavra, 1).
* lambda word: (word.lower().strip(",."), 1): Normaliza a palavra para minúsculas e remove pontuações simples.
* reduceByKey: Agrupa as palavras iguais e soma seus valores.
* lambda a, b: a + b: Função que soma os valores dos pares com a mesma chave.

Finalmente, coletamos os resultados e os exibimos.

resultados = word\_counts.collect() # Coleta os resultados

# Exibe as contagens de palavras

for palavra, contagem in resultados:

print(f"'{palavra}': {contagem}")

**Explicações:**

* word\_counts.collect(): Recolhe os resultados do RDD para uma lista no driver.
* O loop for percorre cada par (palavra, contagem) e imprime o resultado.

Saída:

'apache': 1

'spark': 2

'é': 2

'um': 1

'mecanismo': 1

'de': 4

'análise': 1

'unificado': 1

'para': 1

'processamento': 1

'dados': 2

'em': 1

'grande': 1

'escala': 1

'ele': 1

'fornece': 1

'apis': 1

'em': 1

'python': 1

'java': 1

'scala': 1

'e': 1

'r': 1

'o': 1

'mais': 1

'rápido': 1

'que': 1

'o': 1

'hadoop': 1

'mapreduce': 1

Após a execução, é uma boa prática encerrar a sessão Spark:

spark.stop() # Encerra a sessão Spark

Assim, temos:

**Map (Transformações):**

* **flatMap**: Divide as linhas em palavras.
* **map**: Cria pares (palavra, 1).

**Shuffle and Sort (Interno ao Spark):**

* Os pares são redistribuídos para que todas as ocorrências da mesma palavra sejam enviadas para o mesmo executor.

**Reduce (Ação):**

* **reduceByKey**: Soma as contagens de cada palavra.

**Observações Adicionais**

* **RDD (Resilient Distributed Dataset):** É a estrutura de dados do Spark, representando uma coleção distribuída de elementos que podem ser operados em paralelo.
* **Transformações vs. Ações:**
  + **Transformações:** Operações que retornam um novo RDD (e.g., map, flatMap, reduceByKey). São preguiçosas, ou seja, não executam imediatamente.
  + **Ações:** Operações que retornam um valor ao driver ou escrevem em um sistema de armazenamento (e.g., collect, count). Executam as transformações acumuladas.
* **Lazy Evaluation:** O Spark acumula as transformações e só executa quando uma ação é chamada, otimizando o processamento e, consequentemente, reduzindo a movimentação de dados.
* **Processamento em Memória:** O Spark armazena os dados na memória sempre que possível, acelerando o processamento em comparação com o MapReduce tradicional, que lê e escreve no disco entre as etapas.

Observe, portanto, que este exemplo, embora simples, demonstra claramente como os conceitos de MapReduce são implementados no Spark usando PySpark:

* **Distribuição de Dados:** Os dados são distribuídos automaticamente entre os nós do cluster.
* **Paralelismo:** As operações são executadas em paralelo nos diferentes executores.
* **Tolerância a Falhas:** Se um executor falha, o Spark reexecuta as tarefas afetadas em outro executor.
* **Abstração de Baixo Nível:** O desenvolvedor não precisa se preocupar com detalhes de paralelização, sincronização ou comunicação entre nós.