Tema 3. Spark I

3.1. Introducción y objetivos

Introducción:
Apache Spark es un motor de procesamiento de datos distribuido, de código abierto y de uso general que se utiliza para el procesamiento de grandes volúmenes de datos. Su principal característica es la capacidad de realizar cómputos en memoria, lo que le permite ser hasta 100 veces más rápido que Hadoop MapReduce para ciertas aplicaciones. Spark fue diseñado para ser rápido, fácil de usar y para cubrir una amplia gama de cargas de trabajo que MapReduce no manejaba eficientemente, como las consultas interactivas y el procesamiento de flujos de datos (streaming).

Objetivos de aprendizaje:

- Comprender qué es Apache Spark y por qué es una herramienta fundamental en el ecosistema Big Data.
- Identificar los diferentes módulos que componen Spark y sus casos de uso.
- Reconocer la arquitectura de Spark y el funcionamiento interno de un job.
- Entender el concepto de RDD, la abstracción de datos principal de Spark.
- Diferenciar entre transformaciones y acciones, y aprender a utilizar las más comunes.
- Ejecutar un ejemplo completo de procesamiento de datos utilizando RDDs.

3.2. Apache Spark

Apache Spark es un framework que proporciona un conjunto de APIs en Java, Scala, Python y R. Nació en 2009 en el AMPLab de la Universidad de California, Berkeley, como una solución a las limitaciones de MapReduce. Mientras que MapReduce está fuertemente ligado al disco (lee datos de HDFS, procesa, escribe resultados intermedios a HDFS, lee resultados intermedios, etc.), Spark realiza los cálculos intermedios en la memoria RAM de los nodos del clúster, reduciendo drásticamente la latencia.

Características principales:

- **Velocidad:** Gracias al procesamiento en memoria y a su optimizador de consultas.
- **Facilidad de uso:** Ofrece APIs de alto nivel que ocultan gran parte de la complejidad del procesamiento distribuido.
- **Unificación:** Proporciona una única plataforma para diversas tareas: procesamiento por lotes (batch), consultas SQL, procesamiento de datos en tiempo real (streaming), machine learning y procesamiento de grafos.
- **Compatibilidad:** Puede ejecutarse sobre Hadoop YARN, Apache Mesos,

Kubernetes, o en modo "standalone" (independiente). Puede leer datos de múltiples fuentes como HDFS, Cassandra, HBase, S3, etc.

3.3. Componentes de Spark

Spark no es una herramienta monolítica, sino un ecosistema de componentes construidos sobre su núcleo (Spark Core).

- **Spark Core:** Es el corazón de Spark. Proporciona la funcionalidad básica de E/S, la planificación de tareas (scheduling) y la abstracción principal: los RDDs (Resilient Distributed Datasets). Todos los demás componentes se construyen sobre esta base.
- **Spark SQL:** Permite realizar consultas sobre datos estructurados utilizando SQL o una API similar a la de los DataFrames de pandas/R. Proporciona un potente optimizador (Catalyst) y es la base para las APIs más modernas (Datasets y DataFrames).
- **Spark Streaming:** Facilita el procesamiento de flujos de datos en tiempo real (o casi en tiempo real). Ingiere datos en micro-lotes (mini-batches) y los procesa utilizando las APIs del Spark Core.
- **MLlib (Machine Learning Library):** Es la librería de aprendizaje automático de Spark. Contiene una amplia gama de algoritmos de clasificación, regresión, clustering y herramientas de álgebra lineal y estadística.
- **GraphX:** Es la API para el procesamiento de grafos y la computación paralela de grafos.

3.4. Arquitectura de Spark

Una aplicación Spark se ejecuta como un conjunto de procesos independientes en un clúster, coordinados por el objeto SparkContext en tu programa principal (llamado **driver program**).

1. **Driver Program (Programa Controlador):** Es el proceso que ejecuta la función main() de tu aplicación y donde se crea el SparkContext. Es el cerebro de la operación. Reside en un nodo del clúster y es responsable de:
    - Convertir el código del usuario en Jobs.
    - Dividir los Jobs en Stages y luego en Tasks.
    - Coordinarse con el Cluster Manager para planificar y asignar las Tasks a los Executors.
2. **Cluster Manager (Gestor del Clúster):** Es el responsable de adquirir recursos en el clúster para que la aplicación Spark se ejecute. Los más comunes son:
    - **Standalone:** Un gestor simple incluido con Spark.
    - **Apache Mesos:** Un gestor de clúster de propósito general.
    - **Hadoop YARN:** El gestor de recursos de Hadoop, muy común en entornos

productivos.
    - **Kubernetes:** Un sistema de orquestación de contenedores.
3. **Executors (Ejecutores):** Son los procesos de trabajo que se lanzan en los nodos del clúster (worker nodes) al inicio de una aplicación Spark. Son responsables de:
    - Ejecutar las Tasks que les asigna el Driver.
    - Almacenar los datos en memoria (cache) o en disco.
    - Devolver los resultados al Driver.

3.5. Resilient Distributed Datasets (RDD)

El RDD es la abstracción fundamental de Spark. Es una **colección de elementos inmutable y distribuida, tolerante a fallos, que puede ser operada en paralelo.**

- **Resilient (Tolerante a fallos):** Spark mantiene el "linaje" (lineage) de cada RDD, es decir, sabe exactamente cómo se creó a partir de otros RDDs. Si una partición de un RDD se pierde (por ejemplo, si un nodo se cae), Spark puede reconstruirla automáticamente a partir de los datos originales aplicando las mismas transformaciones.
- **Distributed (Distribuido):** Los datos de un RDD se dividen en particiones, y estas particiones se distribuyen a través de los nodos Executor del clúster.
- **Dataset (Conjunto de datos):** Representa una colección de datos sobre la que se pueden realizar operaciones.

Los RDDs se pueden crear de dos maneras:

1. Paralelizando una colección existente en tu programa Driver.
2. Leyendo un conjunto de datos de una fuente externa (HDFS, S3, etc.).

3.6. Transformaciones y Acciones

Las operaciones sobre RDDs se dividen en dos categorías:

Transformaciones (Transformations):
Son operaciones que crean un nuevo RDD a partir de uno existente. Son "perezosas" (lazy), lo que significa que Spark no las ejecuta en el momento en que se declaran. En su lugar, construye un grafo de dependencias (el linaje) que se ejecutará más tarde.

- **Transformaciones Estrechas (Narrow):** Cada partición del RDD padre es utilizada por, como máximo, una partición del RDD hijo. No requieren mover datos entre nodos (no hay "shuffle").
    - map(func): Aplica una función a cada elemento del RDD y devuelve un nuevo RDD con los resultados.
    - filter(func): Devuelve un nuevo RDD que contiene solo los elementos que satisfacen una condición.
    - flatMap(func): Similar a map, pero cada elemento de entrada puede ser

mapeado a 0 o más elementos de salida.

- **Transformaciones Anchas (Wide):** Múltiples particiones hijas pueden depender de una partición padre. Requieren un "shuffle", que es el proceso de redistribuir los datos a través de los nodos del clúster. Son operaciones costosas.
  - groupByKey(): Agrupa todos los valores para cada clave en un único RDD.
  - reduceByKey(func): Agrupa los valores para cada clave y los reduce usando una función asociativa. Es más eficiente que groupByKey seguido de una reducción.
  - sortByKey(): Ordena el RDD por clave.

Acciones (Actions):
Son operaciones que desencadenan la computación y devuelven un valor al programa Driver o escriben datos en un sistema de almacenamiento externo. Cuando se llama a una acción, Spark ejecuta el grafo de transformaciones (el linaje) que termina en ese RDD.

- count(): Devuelve el número de elementos en el RDD.
- collect(): Devuelve todos los elementos del RDD como un array al programa Driver. ¡Cuidado! Usar solo en RDDs pequeños.
- take(n): Devuelve los primeros n elementos del RDD.
- first(): Devuelve el primer elemento del RDD (similar a take(1)).
- reduce(func): Agrega los elementos del RDD usando una función.
- saveAsTextFile(path): Guarda el contenido del RDD en un archivo de texto.

3.7. Jobs, Stages y Tasks

La ejecución de una aplicación Spark sigue una jerarquía:

1. **Job (Trabajo):** Se crea un Job por cada **acción** que se invoca en el código. Es la unidad de ejecución de más alto nivel.
2. **Stage (Etapa):** Cada Job se divide en un conjunto de Stages. El límite entre Stages está definido por las transformaciones anchas (las que requieren un "shuffle"). Spark ejecuta los Stages en orden, y los Stages dentro de un Job forman un Grafo Acíclico Dirigido (DAG).
3. **Task (Tarea):** Cada Stage se compone de un conjunto de Tasks, que son la unidad de trabajo más pequeña. Una Task se ejecuta sobre una partición de datos en un Executor. Por ejemplo, un Stage que aplica un map sobre un RDD con 100 particiones se dividirá en 100 Tasks.

4. Práctica 1 Uso entorno Spark

Apache Spark es un motor de procesamiento unificado para el análisis de datos a gran escala

- Spark proporciona APIs de alto nivel en Java Scala Python y R
- Spark también soporta un conjunto enriquecido de herramientas de alto nivel
- incluyendo Spark SQL para SQL y procesamiento de datos estructurados
- MLlib para machine learning GraphX para el procesamiento de grafos
- Spark Streaming para el procesamiento de flujos

4.1 Instrucciones para el Laboratorio Práctico

Sigue estos pasos para ejecutar tu primer Job de Spark.

1. Prepara tu entorno:
- Crea una carpeta principal para tu proyecto (ej. `mi_lab_spark`).
- Dentro, crea el archivo `docker-compose.yml` con el contenido proporcionado.
- Crea dos subcarpetas: `data` y `scripts`.
- Dentro de `scripts`, crea el archivo `word_count.py`.
- Dentro de `data`, crea el archivo `libro.txt`.

- docker-compose.yml

```
version: '3.8'


services:
  spark-master:
    image: bde2020/spark-master:3.3.0-hadoop3.3
    container_name: spark-master
    ports:
      - "8080:8080"
      - "7077:7077"
```

```
    environment:
      - INIT_DAEMON_STEP=setup_spark
    volumes:
      - ./data:/data
      - ./scripts:/scripts
    networks:
      - spark-network


  spark-worker-1:
    image: bde2020/spark-worker:3.3.0-hadoop3.3
    container_name: spark-worker-1
    depends_on:
      - spark-master
    ports:
      - "8081:8081"
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
    volumes:
      - ./data:/data
      - ./scripts:/scripts
    networks:
      - spark-network


  jupyter-lab:
    image: jupyter/pyspark-notebook:spark-3.3.0
    container_name: jupyter-lab
    depends_on:
```

```
      - spark-master
    ports:
      - "8888:8888"
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
      - "JUPYTER_ENABLE_LAB=yes"
      - "PYSPARK_SUBMIT_ARGS=--master spark://spark-master:7077
pyspark-shell"
    volumes:
      - ./data:/data
      - ./scripts:/scripts
      - ./notebooks:/home/jovyan/work
    networks:
      - spark-network
    command: start-notebook.sh --NotebookApp.token=''
--NotebookApp.password=''


networks:
  spark-network:
    driver: bridge
```

- word_count.py

```
from pyspark import SparkContext
import sys

print("=== Iniciando WordCount ===")

# Crear contexto Spark
```

```
sc = SparkContext("local[2]", "WordCount")

try:
    # Leer archivo
    input_file = sys.argv[1] if len(sys.argv) > 1 else "/input/libro.txt"
    print(f"Procesando archivo: {input_file}")

    # Verificar que el archivo existe
    import os
    if not os.path.exists(input_file):
        print(f"ERROR: Archivo {input_file} no encontrado")
        sc.stop()
        exit(1)

    # Leer líneas del archivo
    lines = sc.textFile(input_file)
    print(f"Líneas leídas: {lines.count()}")

    # Contar palabras
    words = lines.flatMap(lambda line: line.split())
    word_counts = words.map(lambda word: (word.lower().strip(),
1)).reduceByKey(lambda a, b: a + b)

    # Ordenar por frecuencia
    sorted_counts = word_counts.map(lambda x: (x[1],
x[0])).sortByKey(False)

    # Limpiar directorio de salida
    output_dir = "/output/output_wordcount"
    os.system(f"rm -rf {output_dir}")

    # Guardar resultados
```

```
    sorted_counts.saveAsTextFile(output_dir)

    # Mostrar primeros 10 resultados
    print("\n=== TOP 10 PALABRAS ===")
    top_10 = sorted_counts.take(10)
    for count, word in top_10:
        print(f"{word}: {count}")

    print(f"\nResultados guardados en: {output_dir}")

except Exception as e:
    print(f"Error: {e}")
    import traceback
    traceback.print_exc()
finally:
    sc.stop()
    print("=== Job terminado ===")
```

2. Inicia el Clúster de Spark:

- Abre una terminal en la carpeta principal (`mi_lab_spark`) y ejecuta:

```
docker-compose up -d
```

- Esto descargará las imágenes y levantará los contenedores.
- Puedes ver la interfaz web del Master de Spark en tu navegador, accede a los servicios:
    - Spark Master UI: http://localhost:8080
    - Spark Worker UI: http://localhost:8081
    - Jupyter Lab: http://localhost:8888
- Verás que tienes un Worker registrado.

3. Sube los datos a HDFS:

- Los contenedores `bde2020` vienen con HDFS. Primero, vamos a entrar al

master para usar los comandos de HDFS.

```
docker exec -it spark-master bash
```

- Una vez dentro, crea un directorio en HDFS y sube tu archivo `libro.txt`. El archivo ya está disponible dentro del contenedor en la ruta `/data/libro.txt` gracias a los volúmenes de Docker.

```
mkdir -p /input
mkdir -p /output
ls -la /data/  # Verificar que libro.txt está aquí
cp /data/libro.txt /input/  # Copiar el archivo al directorio input
ls -la /input/  # Verificar que se copió correctamente
```

4. Ejecuta el Job con `spark-submit`:
- Aún dentro del contenedor `spark-master`, usa `spark-submit` para enviar tu script de Python.

```
/spark/bin/spark-submit /scripts/word_count.py /input/libro.txt
```

- Importante: Si quieres volver a ejecutar el job, primero debes borrar el directorio de salida: rm -rf /output/output_wordcount

5. Verifica los Resultados:
- Una vez que el job termine, los resultados estarán en HDFS.

```
ls -la /output/output_wordcount/
```

- * Para ver el resultado, usa `cat` sobre el archivo `part-00000`:

```
cat /output/output_wordcount/part-00000
```

- También puedes ver todos los archivos de resultado:

```
cat /output/output_wordcount/part-*
```

Verás el conteo de cada palabra del archivo `libro.txt`.

6. Comandos útiles adicionales:

- Ver logs del job:

  ls -la /spark/logs/

- Verificar conexión con workers:

  /spark/bin/spark-shell --master spark://spark-master:7077

- Limpiar resultados anteriores:

  rm -rf /output/output_wordcount

- Copiar resultados al host (desde PowerShell en Windows):

  docker cp spark-master:/output/output_wordcount ./resultados/

7. Solución de problemas:

- Si el job falla, verifica:
  - Que el archivo /input/libro.txt existe
  - Que el directorio /output/ existe
  - Que el script /scripts/word_count.py está disponible
- Para ver logs detallados:

  /spark/bin/spark-submit --verbose /scripts/word_count.py /input/libro.txt
- Para ejecutar en modo local (sin cluster):

  /spark/bin/spark-submit --master local[2] /scripts/word_count.py /input/libro.txt

Alternativa: Práctica Interactiva con JupyterLab

* Abre tu navegador y ve a `http://localhost:8888`.

* Desde la interfaz de JupyterLab, puedes crear un nuevo Notebook.

* Dentro del notebook, puedes escribir y ejecutar código PySpark de forma

interactiva, lo cual es excelente para aprender y experimentar con las diferentes transformaciones y acciones. El `SparkSession` ya viene pre-configurado.

Análisis del ejemplo:

- sc.textFile(...) crea el primer RDD.
- flatMap, map y reduceByKey son **transformaciones**. No se ejecuta nada todavía. Spark solo construye el DAG.
- collect() es la **acción**. En este momento, Spark:
  - Crea un Job.
  - Analiza el DAG y lo divide en Stages (en este caso, reduceByKey causará un shuffle, creando un límite de Stage).
  - Lanza las Tasks a los Executors para leer el archivo, mapear las palabras y finalmente barajar y reducir los conteos.
  - Envía el resultado final al Driver.

5. Referencias bibliográficas

- **Documentación Oficial de Apache Spark:** El mejor y más actualizado recurso.
  - Guía de Programación de Spark
- **Libros:**
  - "Spark: The Definitive Guide" por Bill Chambers y Matei Zaharia.
  - "Learning Spark, 2nd Edition" por Jules S. Damji, Brooke Wenig, Tathagata Das, y Denny Lee.