Cuestionario / Actividad Práctica: Procesamiento en Tiempo Real con Spark y Kafka

Parte 1: Conceptos básicos – Responde a las siguientes preguntas:

Docker

- ¿Qué es un contenedor y cómo se diferencia de una máquina virtual?

Un contenedor es una unidad ligera que empaqueta una aplicación con todo lo que requiere para correr (su código, librerías, herramientas y configuraciones). Todo se ejecuta aislado pero compartiendo el sistema operativo del host. Se diferencia de una máquina virtual en que no incluye un sistema operativo, su arranque es más rápido, consume menos recursos, tiene un menor tamaño y es más rápido para escalar.

- ¿Qué son los volúmenes en Docker y para qué sirven?

Son un mecanismo de persistencia de datos para guardar datos fuera del ciclo de vida del contenedor, compartir datos entre contenedores y evitar la pérdida de información al borrar o recrear contenedores.

- Explica la diferencia entre docker run y docker compose.

docker run es un comando para crear y ejecutar un contenedor a partir de una imagen. Docker compose es una herramienta de orquestación local que usa un archivo .yml, en el cual se decriben contenedores y cómo se relacionan.

Kafka

- ¿Qué es un topic en Kafka?

Es un canal lógico donde se publican y se consumen mensajes (eventos). Se dividen en particiones, que son secuencias ordenadas e inmutables de mensajes con un identificador único.

- ¿Cuál es la función de Zookeeper dentro de un clúster Kafka?

Coordina el clúster Kafka, monitoreando el estado de los brokers del clúster y detecta fallos para permitir la reconfiguración del clúster.

- Explica qué significan “particiones” y “replication factor” en un topic.

Un topic se divide en particiones. Una partición es un log ordenado inmutable almacenado en un broker. El replication factor indica cuántas copias de cada partición existen en el clúster.

Spark

- ¿Qué es Spark y cuál es su propósito?

Es un motor de procesamiento distribuido de datos, diseñado para procesar grandes volúmenes de datos de forma rápida, paralela y escalable.

- Diferencia entre RDD, DataFrame y Dataset.
RDD es la estructura de base de datos de Spark. Es una colección inmutable y distribuida tolerante a fallos. Un Dataframe es una colección distribuida de datos organizada en columnas. Un Dataset es una extensión de los Dataframes que combina la seguridad de los RDD con las optimizaciones de los Dataframes. RDD es la abstracción básica de Spark mientras que los Dataframes y Datasets son abstracciones de alto nivel que permiten optimización automática y mejor rendimiento. A diferencia de los RDD y Dataframes, los Dataset solamente están disponibles en Scala y Java.

- ¿Qué es Spark Streaming y para qué se utiliza?
es un componente de la plataforma Apache Spark diseñado para procesar flujos de datos en tiempo real de manera escalable y tolerante a fallos. A diferencia de otros sistemas que procesan registro por registro, Spark Streaming utiliza una arquitectura de micro-lotes. Divide el flujo de datos continuo en pequeños fragmentos (lotes de pocos segundos o milisegundos) y los procesa utilizando el motor central de Spark.

Parte 2: Actividad práctica con Docker + Kafka + Spark

Introducción

En este taller se desarrolló una actividad práctica orientada a comprender el funcionamiento básico de una arquitectura de procesamiento de datos en tiempo real utilizando Docker, Apache Kafka y Apache Spark. El objetivo principal fue levantar los servicios necesarios mediante contenedores, crear y gestionar un tópico en Kafka, producir y consumir mensajes, y finalmente observar en consola el procesamiento de eventos.

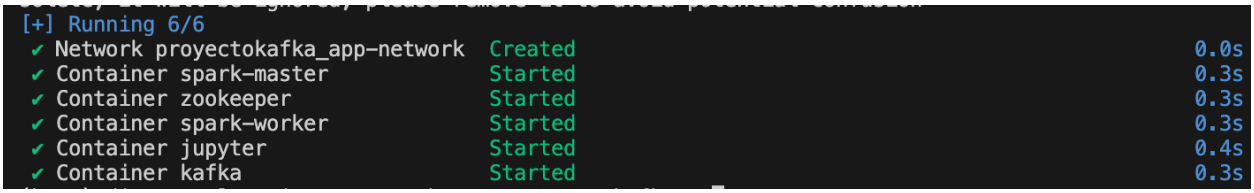
La práctica se realizó utilizando Docker Compose para la orquestación de los servicios y Jupyter Notebook como entorno interactivo para el desarrollo del productor y el consumidor. Los datos que se procesan se obtienen de la API http://api.open-notify.org/iss-now.json la cual devuelve datos de timestamp y ubicación en coordenadas de longitud y latitud.

Paso 1: Levantar los servicios con Docker Compose

El primer paso del taller consistió en levantar los servicios necesarios para la arquitectura: Zookeeper, Kafka, Spark Master, Spark Worker y Jupyter Notebook. Para ello, se utilizó un archivo `docker-compose.yml` que define cada uno de estos servicios y los conecta a través de una red Docker compartida.

Desde Visual Studio Code se ejecutó el comando `docker-compose up -d`

Este comando permitió iniciar todos los contenedores. En la salida del terminal se pudo verificar que los servicios se encontraban correctamente en ejecución, lo cual confirmó que la infraestructura base estaba lista para continuar con el taller.



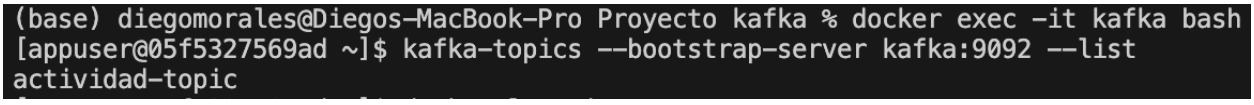
Paso 2: Creación de un nuevo tópico en Kafka

Con los servicios activos, el siguiente paso fue crear un tópico en Kafka que serviría como canal de comunicación para los mensajes del ejercicio. El tópico se llamó `actividad-topic` y se configuró con una sola partición y una sola réplica.

La creación del tópico se realizó ejecutando el siguiente comando desde el host:

```
docker compose exec kafka kafka-topics \
--create \
--topic actividad-topic \
--bootstrap-server kafka:9092 \
--partitions 1 \
--replication-factor 1
```

Posteriormente, se verificó que el tópico había sido creado correctamente ingresando al contenedor de Kafka y listando los tópicos existentes.



Paso 3: Envío y lectura de mensajes

Para este ejercicio se crearon dos notebooks en Jupyter:

- **Notebook Producer**: encargado de enviar mensajes al tópico de Kafka.
- **Notebook Consumer**: encargado de leer los mensajes publicados.

En el notebook del productor se desarrolló un flujo sencillo para simular la generación y envío de eventos hacia Kafka. Desde el notebook del productor se crearon las siguientes funciones:

- *sourceClient*: realiza una solicitud a una API externa, captura la respuesta y la almacena en formato JSON, el cual representa la estructura base de los datos a enviar.
- *binary*: transforma el JSON obtenido en una cadena de texto, dejándolo preparado para su posterior conversión a formato binario.
- *writer*: establece la conexión con el broker de Kafka y publica los mensajes en el tópico `actividad-topic`.

Finalmente, se creó un bucle que ejecuta estas tres funciones de manera secuencial para realizar cinco inserciones consecutivas en el tópico, permitiendo así generar múltiples eventos de prueba y validar el funcionamiento del productor dentro de la arquitectura implementada.

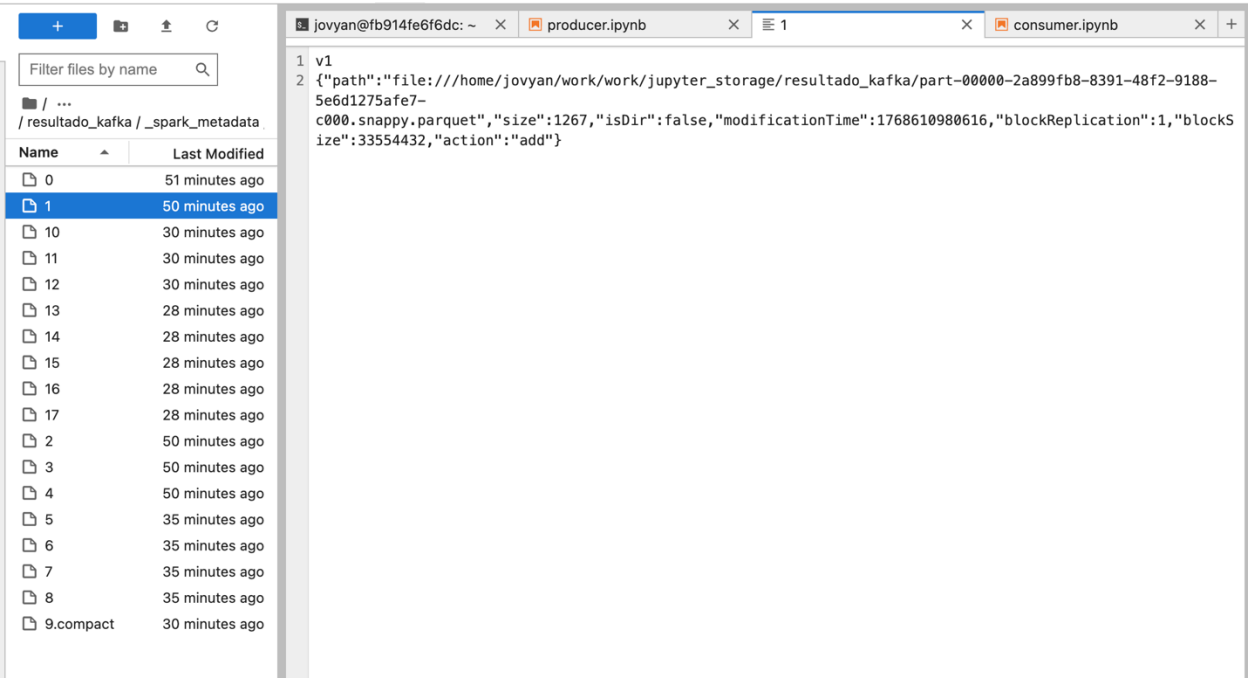
```
#Haremos un bucle de 5 inserciones hacia el tópico
for i in range(1, 6):
    #Mostramos el número de la iteración
    print("Iteración: "+str(i))

    #Leemos los datos desde la fuente de datos
    registro = sourceClient()

    #Dejamos el registro listo para la binarización
    registroBin = binary(registro)

    #Enviamos el registro al tópico
    writer(registroBin)

    #Dormimos el bucle de ingesta por 1 segundo
    time.sleep(1)
```

```
Iteración: 1
Iteración: 2
Iteración: 3
Iteración: 4
Iteración: 5
```

En el notebook del consumer se implementó un flujo de procesamiento de datos en tiempo real utilizando Spark Structured Streaming integrado con Apache Kafka. En primer lugar, se creó una sesión de Spark mediante `SparkSession.builder`, configurando la conexión con el broker de Kafka y habilitando el conector necesario para la lectura de mensajes en streaming. A partir de esta sesión, se definió un DataFrame de Spark Streaming que consume los mensajes publicados en el tópico `actividad-topic`, permitiendo la ingesta continua de datos conforme estos son producidos.

Los mensajes recibidos desde Kafka, inicialmente en formato binario, fueron transformados a String y posteriormente a formato JSON, con el fin de facilitar su

interpretación y procesamiento. Una vez transformados, los datos fueron almacenados de manera persistente en archivos Parquet.



Para la visualización de los datos en un entorno de Jupyter Notebook, se utilizó una estrategia basada en el memory sink de Spark. En lugar de imprimir directamente en consola, los datos del stream se almacenaron temporalmente en una tabla en memoria mediante `writeStream.format("memory")`. Este enfoque permitió consultar los datos del stream utilizando sentencias SQL, ejecutando una instrucción `SELECT` sobre la tabla en memoria y mostrando los resultados en consola. De esta manera, fue posible inspeccionar los mensajes procesados en tiempo real de forma controlada.

```
memoryQuery = dfResultado.writeStream \
    .format("memory") \
    .queryName("kafka_stream") \
    .outputMode("append") \
    .start()

print("Streaming query iniciado")


spark.sql("SELECT * FROM kafka_stream").show(truncate=False)
```

En la siguiente imagen se muestra la inspección de algunos de los mensajes.

message	longitude	latitude	timestamp
success	20.0481	-7.9763	1768612204
success	20.1765	-8.1525	1768612208
success	20.2132	-8.2029	1768612209
success	20.2683	-8.2784	1768612210
success	24.8665	-14.401	1768612333
success	24.9821	-14.5497	1768612336
success	25.04	-14.624	1768612337
success	25.098	-14.6984	1768612339
success	25.1946	-14.8222	1768612341