Trabajo Final: Procesamiento de Grandes Volúmenes de Datos

Maestría de Ciencia de Datos - UNAJ

Raúl Burgos Mauro Cejas Marcovecchio

2026-02-15

Introducción

En el contexto del análisis de grandes volúmenes de datos, las arquitecturas distribuidas juegan un rol fundamental para el procesamiento eficiente y escalable de información. Tecnologías como Apache Spark y Apache Kafka se han convertido en estándares para el procesamiento batch, streaming y la ingestión de datos en tiempo real.

El objetivo de este trabajo es diseñar e implementar un clúster virtualizado que permita simular una infraestructura orientada al análisis de datos del mercado financiero. Para ello, se utilizó Docker como tecnología de virtualización liviana, desplegando un clúster compuesto por tres nodos que integran Apache Spark, Apache Kafka y Apache Zookeeper.

El caso de uso se inspira en un concurso académico realizado en 2022, donde se propuso el análisis en tiempo real de datos de mercado. Dado que la API original ya no se encuentra disponible, se optó por simular la ingestión de datos utilizando Kafka como sistema de mensajería distribuido.

Desarrollo

El sistema diseñado consta de tres niveles distintos y cinco nodos:

Arquitectura de infraestructura

La implementación se basó en la tecnología de contenedores Docker, utilizando Docker Compose como orquestador para definir una red virtual aislada denominada *bigdata-network*. Esta red permite la comunicación entre nodos, garantizando un entorno reproducible y escalable.

- **Zookeeper**: servicio de coordinación para gestionar el estado del broker, las cuotas y, lo más importante, la elección de líderes para las particiones de los temas (topics).

```yaml
zookeeper:
  image: confluentinc/cp-zookeeper:7.0.1
  container_name: zookeeper
  ports:
    - "2181:2181"
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 2000
  networks:
    - bigdata-network
```

- **Kafka**: encargado de la ingestión y distribución de eventos que simulan datos del mercado financiero, que describiremos más adelante. Fue configurado con múltiples *listeners* para diferenciar el tráfico interno del clúster y el tráfico externo de los productores de datos.

```yaml
kafka:
  image: confluentinc/cp-kafka:7.0.1
  container_name: kafka
  depends_on:
    - zookeeper
  ports:
    - "9092:9092"
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
    KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka:29092,EXTERNAL://localhost:9092
    KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  networks:
    - bigdata-network
```

- **Spark**: Tendremos un nodo maestro (*Master*), que no procesa datos directamente sino que su función es asignarle tareas a los nodos trabajadores (*Workers*) y gestionar el ciclo de vida de las aplicaciones Spark. Estos últimos recibirán las divisiones del trabajo realizadas por el master, llamadas *tasks*, para ejecutarlas y devolver los resultados.

```yaml
  spark-master:
    image: bde2020/spark-master:3.3.0-hadoop3.3
    container_name: spark-master
    ports:
      - "8080:8080"
      - "7077:7077"
    environment:
      - PYSPARK_PYTHON=python3
    deploy:
      resources:
        limits:
          cpus: '0.30'
          memory: 512M
    networks:
      - bigdata-network

  # Dos spark-workers idénticos con la misma estructura
  spark-worker:
    image: bde2020/spark-worker:3.3.0-hadoop3.3
    container_name: spark-worker-1
    depends_on:
      - spark-master
    environment:
      - SPARK_MASTER_URL=spark://spark-master:7077
      - SPARK_WORKER_CORES=2
      - SPARK_WORKER_MEMORY=1024m
    deploy:
      resources:
        limits:
          cpus: '1.0'
    networks:
      - bigdata-network
```

La virtualización mediante Docker permitió cumplir con el requerimiento sin necesidad de máquinas virtuales completas, reduciendo el consumo de recursos y simplificando el despliegue. De todas formas, la separación de roles y la limitación estricta de recursos mediante contenedores posibilitaron la estabilidad del sistema, evitando la sobrecarga.

Simulación de datos financieros

Ante la falta de una fuente de datos externa, se desarrolló un productor de datos en Python. Este código parte de los valores de las acciones al 12/02/2026, simulando la volatilidad del mercado financiero mediante *random walk*, generando registros en formato JSON que incluyen: la fecha, el símbolo de la acción correspondiente y su precio unitario. Sus variaciones porcentuales fueron ajustadas para garantizar cierto grado de realismo, estando contenidas en un rango entre el -0.1% y el 0.1%, con una distribución uniforme, por lo que no tendrán una tendencia intrínseca a bajar o subir. Además, se ajustó la frecuencia de publicación a intervalos de cinco segundos, permitiendo evaluar la capacidad de respuesta del sistema ante flujos continuos.

```
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

topic_nombre = "precios_mercado"
acciones = ['AAPL', 'TSLA', 'GOOG', 'AMZN', 'MSFT']
precios = {
    "AAPL": 267.0,
    "MSFT": 399.0,
    "GOOG": 310.0,
    "AMZN": 197.0,
    "TSLA": 423.0
}

print(f" Iniciando envío de datos al tópico: {topic_nombre}")

try:
    while True:
        for accion in acciones:
            variacion = precios[accion] * random.uniform(-0.001, 0.001)
            precios[accion] += variacion

            mensaje = {
                'timestamp': time.time(),
                'simbolo': accion,
                'precio': round(precios[accion], 2)            }

            producer.send(topic_nombre, value=mensaje)
            print(f"Enviado: {mensaje}")
```

```
        time.sleep(5)

except KeyboardInterrupt:
    print("\n Simulación detenida por el usuario.")
finally:
    producer.close()
```

Procesamiento en tiempo real

El procesamiento se realiza mediante Spark Structured Streaming, una API que permite tratar los flujos de datos como tablas. El motor de Spark consume los mensajes desde Kafka utilizando el conector especializado *spark-sql-kafka*. La lógica de procesamiento implementa una agregación contínua por activo financiero para calcular métricas en tiempo real, como el precio promedio. Para garantizar la integridad de los datos, se integró una política de marcas temporales de 60 segundos, lo que permite al motor de Spark gestionar el estado de los eventos y tolerar retrasos en la llegada de la información sin comprometer la estabilidad de la memoria del clúster.

Dada la naturaleza del entorno virtualizado y la restricción de recursos computacionales, se configuró el intercambio de datos a través de la red virtual (*shuffling*), reduciendo la fragmentación de tareas y alineando el número de particiones con la cantidad de núcleos asignados a los nodos trabajadores anteriormente.

Asimismo, se aplicó una estrategia de optimización de la capa de presentación para mejorar la legibilidad de los resultados en consola. Esto se logró mediante el modo de salida *Update*, que permite que el sistema solo emita aquellos registros que han sufrido cambios (reduciendo la redundancia visual), y una transformación de datos para redondear los valores y se eliminaron los metadatos temporales de las ventanas.

```
spark = SparkSession.builder \
    .appName("AnalisisFinancieroRealTime") \
    .config("spark.sql.shuffle.partitions", "2") \
    .config("spark.default.parallelism", "2") \
    .config("spark.executor.memory", "800m") \
    .getOrCreate()

spark.sparkContext.setLogLevel("WARN")

schema = StructType([
    StructField("timestamp", DoubleType()),
    StructField("simbolo", StringType()),
    StructField("precio", DoubleType())
```

```
])

df_crudo = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:29092") \
    .option("subscribe", "precios_mercado") \
    .option("startingOffsets", "latest") \
    .load()

df_json = df_crudo.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), schema).alias("data")) \
    .select("data.*")

df_formateado = df_json \
    .withColumn("event_time", col("timestamp").cast(TimestampType())) \
    .withWatermark("event_time", "1 minute")

df_simbolo = df_formateado \
    .groupBy("simbolo") \
    .agg(avg("precio").alias("precio_promedio")) \
    .select(
        col("simbolo"),
        col("precio_promedio").cast("decimal(10,2)")
    )

query = df_simbolo.writeStream \
    .outputMode("update") \
    .format("console") \
    .option("truncate", "false") \
    .start()

query.awaitTermination()
```

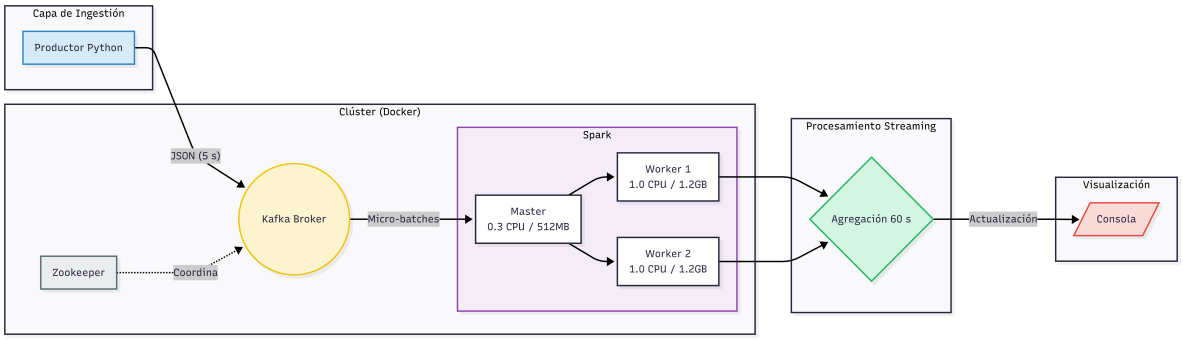

Figura 1: Gráfico 1. Flujo de información

Pruebas

En primer lugar se verificó el correcto funcionamiento de cada componente: accedimos exitosamente a la interfaz web de Spark, permitiendo confirmar que el Master y sus correspondientes Workers se encontraban activos. En cuanto al Broker de Kafka, verificamos la conectividad de los *listeners* mediante el listado de tópicos activos a través del puerto expuesto y confirmando que la red interna de Docker y el broker estaban operativos. En cuanto a Zookeeper, la verificación resulta indirecta, a través del correcto funcionamiento de Kafka.

	Name	Container ID	Image	Port(s)	CPU (%)
●	tp_bigdata	-	-	-	0.96%
●	zookeeper	51be4a300067	confluentinc/cp-zookeeper:7.(	2181:2181	0.15%
●	spark-master	c38e77b35a23	bde2020/spark-master:3.3.0-h	7077:7077 Show all ports (2)	0.09%
●	kafka	17f357328a44	confluentinc/cp-kafka:7.0.1	9092:9092	0.58%
●	spark-worker-1	11b63cdf2793	bde2020/spark-worker:3.3.0-h		0.07%
●	spark-worker-2	665986af407e	bde2020/spark-worker:3.3.0-h		0.07%

Figura 2: Gráfico 2. Evidencia de ejecución del clúster en Docker Desktop

Como prueba funcional del sistema se implementó un caso de uso simple de ingestión de datos. Se enviaron mensajes simulando datos de mercado financiero y, mediante un consumidor Kafka, se verificó la recepción exitosa de los mensajes producidos, confirmando el correcto funcionamiento del flujo productor–broker–consumidor. Este procedimiento valida que el clúster es capaz de manejar eventos en tiempo real, cumpliendo con el objetivo de simular una arquitectura de streaming de datos.

Durante las pruebas el sistema demostró una sincronía impecable. Validamos que el flujo end-to-end funcionara correctamente, desde la creación del mensaje en Python hasta la actualización del promedio de precios en la consola de Spark. Asimismo, al agrupar los datos, logramos una visión clara del mercado en ventanas específicas, obteniendo promedios de precios precisos para cada Ticker de forma casi instantánea.

Conclusiones

En este trabajo se logró implementar exitosamente un clúster Big Data compuesto por tres nodos virtualizados utilizando Docker. La infraestructura desplegada permite simular un entorno realista de ingestión y procesamiento de datos, integrando Apache Spark y Apache Kafka. Podríamos destacar como punto fuerte del proyecto:

- El despliegue de un clúster distribuido con tecnologías ampliamente utilizadas en la industria.
- La implementación de un caso de uso funcional de ingestión de datos en tiempo real.
- La resolución de problemas reales asociados a la configuración de Kafka en entornos Docker, demostrando comprensión del funcionamiento de la plataforma.

La solución desarrollada cumple con los requerimientos planteados y sienta las bases para la extensión hacia escenarios de procesamiento más complejos.

En síntesis, hemos logrado construir una base sólida y reproducible. El uso de Docker ha sido un acierto para la portabilidad, aunque aprendimos que la gestión de scripts mediante volúmenes es un punto clave para la persistencia.

Trabajo a futuro

Este pipeline es solo el comienzo. Para llevar esta solución al siguiente nivel, nuestras metas a corto plazo son:

- Ampliación: Crear otras métricas financieras que permitan un análisis más amplio para la toma de decisiones.
- Visualización: Conectar los resultados a un dashboard para ver las curvas de precios en tiempo real.
- Almacenamiento: Persistir las métricas en una base de datos No SQL (como Cassandra) para análisis históricos.

Referencias

Apache Software Foundation. (n.d.). *Kafka structured streaming programming guide*. Apache Spark Documentation. Retrieved https://spark.apache.org/docs/latest/streaming/index.html

Apache Software Foundation. (2026). *Apache kafka documentation*. https://kafka.apache.org/documentation.

Calavaro, Russo, Cardelini. (2022). *Realtime analysis of market data leveraging apache flink*. https://doi.org/10.1145/3524860.3539650.

Confluent Inc. (2026). *Confluent kafka configuration guide*. https://docs.confluent.io.

Docker Inc. (2026). *Docker compose documentation*. https://docs.docker.com/compose.

Mermaid Team. (2026). *Mermaid: Diagramming and charting tool*. https://mermaid.js.org/

Xie, Dervieux, Riederer. (2026). *R markdown cookbook*. https://yihui.org/rmarkdown-cookbook/.