UA

Universidad

de Alicante

ESCUELA POLITÉCNICA SUPERIOR

Ejercicio de Middleware

Curso académico 2024 / 2025

Semestre 2

Grado en Ingeniería en Inteligencia Artificial

Grupo 1

Profesora: Ricardo Moreno Rodríguez

Alumno: Santiago Álvarez Geanta

Universitat d'Alacant
Universidad de Alicante

Abril 2025

ÍNDICE

Introducción

Durante el desarrollo de esta práctica de Computación de Alto Rendimiento, mi objetivo fue construir un flujo de análisis distribuido de eventos técnicos registrados por múltiples brazos robóticos. Para ello, empleé Apache Spark ejecutándose sobre un contenedor Docker, siguiendo los principios de procesamiento distribuido de datos.

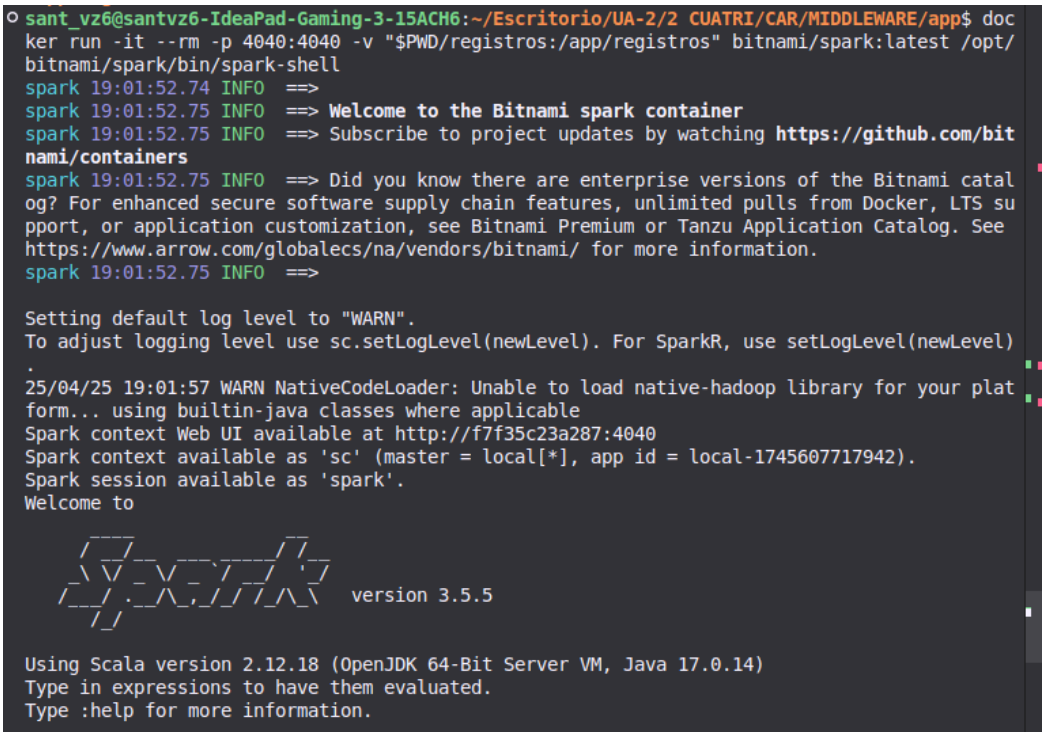
A continuación, detallo las fases realizadas, las decisiones técnicas tomadas y los resultados obtenidos.

Desarrollo

1. Pasos seguidos para construir la solución

En primer lugar, diseñé cuatro archivos de registros simulados (eventos_robot1.txt, eventos_robot2.txt, eventos_robot3.txt, eventos_robot4.txt) dentro de una carpeta llamada registros/. Estos archivos contenían eventos realistas tales como **ALERTA_SOBRECALENTAMIENTO, REINICIO_SISTEMA, PAUSA_NO_PROGRAMADA,** entre otros. Cada archivo representa los sucesos de un brazo robótico distinto.

Después, preparé el entorno de ejecución de Spark utilizando Docker. Para montar correctamente la carpeta registros dentro del contenedor, ejecuté el contenedor desde la ruta superior, asegurándome de no ubicarme dentro de registros/. El comando empleado fue:



Dentro del spark-shell, procedí a realizar la lectura de todos los archivos .txt mediante textFile, filtré los eventos relevantes con filter, transformé cada evento en un par clave-valor (evento, 1) usando map, y finalmente apliqué una reducción distribuida con reduceByKey para obtener la frecuencia de cada evento.

El código principal fue el siguiente:

```
scala> val registros = spark.sparkContext.textFile("/app/registros/*.txt")
registros: org.apache.spark.rdd.RDD[String] = /app/registros/*.txt MapPartitionsRDD[1] at textFile at <console>:22

scala>

scala> val eventosClave = List(
     |   "ALERTA_SOBRECALENTAMIENTO",
     |   "REINICIO_SISTEMA",
     |   "CALIBRACION_AUTOMATICA",
     |   "CARGA_EXCESIVA",
     |   "PAUSA_NO_PROGRAMADA"
     | )
eventosClave: List[String] = List(ALERTA_SOBRECALENTAMIENTO, REINICIO_SISTEMA, CALIBRACION_AUTOMATICA, CARGA_EXCESIVA, PAUSA_NO_PROGRAMADA)

scala>

scala> val eventosFiltrados = registros
eventosFiltrados: org.apache.spark.rdd.RDD[String] = /app/registros/*.txt MapPartitionsRDD[1] at textFile at <console>:22

scala>   .filter(linea => eventosClave.exists(evento => linea.contains(evento)))
res0: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at <console>:25

scala>   .map(evento => (evento, 1))
res1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:24

scala>   .reduceByKey(_ + _)
res2: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:24
```

Este enfoque permitió procesar de manera distribuida los datos provenientes de los registros y obtener un resumen de la frecuencia de los eventos críticos.

2. Dificultades encontradas y resolución

Durante el montaje del contenedor Docker, inicialmente cometí un error al situarme dentro de la carpeta registros y ejecutar el comando de arranque. Esto provocó que Docker no montase los archivos correctamente, resultando en el error **Input Pattern file:/app/registros/*.txt matches 0 files.**

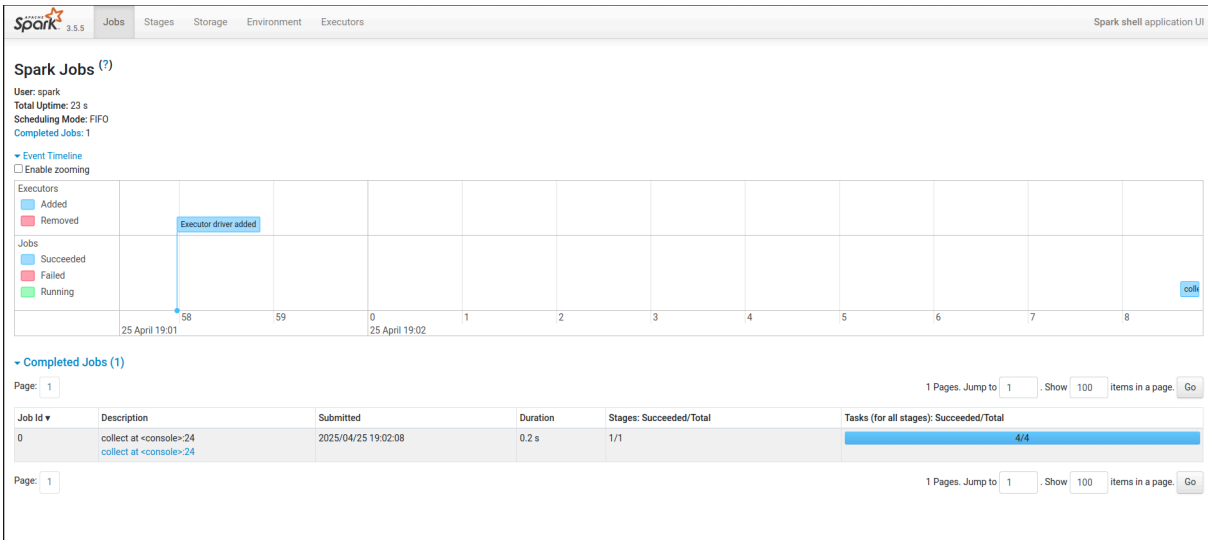
La solución fue volver a la carpeta app/ (el directorio padre) y lanzar nuevamente el contenedor, corrigiendo así la ruta del volumen. Además, verifiqué dentro del contenedor que los archivos estuvieran correctamente montados ejecutando un pequeño script en Scala que listaba el contenido de /app/registros.

Esta experiencia me ayudó a comprender la importancia de la correcta gestión de rutas en entornos virtualizados y la necesidad de validar que los volúmenes estén accesibles.

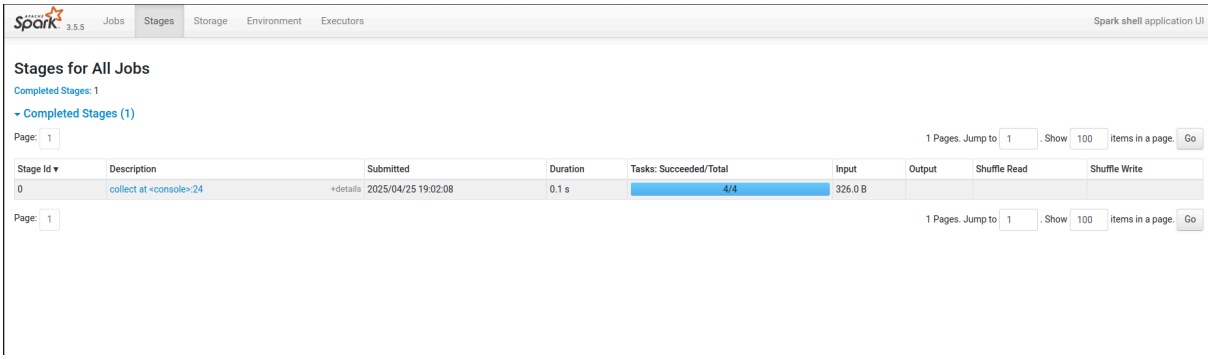
3. Análisis de la ejecución en Spark UI

Accedí a la interfaz web de Spark en **http://localhost:4040** para observar cómo se distribuyó el procesamiento.

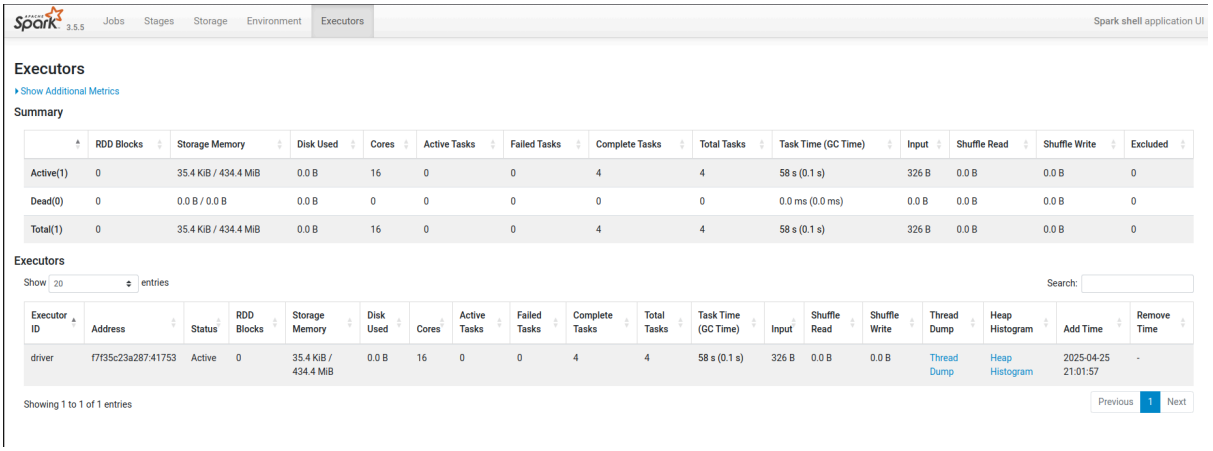
En la pestaña **Jobs**, pude identificar claramente el Job correspondiente al procesamiento de los registros. El job incluía las tareas de lectura, filtrado, mapeo y reducción, agrupadas en una única ejecución debido al encadenamiento de operaciones transformadoras.



En la pestaña **Stages**, se evidenciaron las etapas generadas por Spark. Cada stage correspondía a una transformación importante en el flujo de datos. Observé que se produjo un **shuffle** durante la operación reduceByKey, lo cual es lógico ya que esta operación requiere re-agrupamiento de datos por clave entre particiones.



Opcionalmente, también revisé la pestaña **Executors**, donde se mostraba el uso de recursos. Aunque en mi caso sólo había un ejecutor principal (ya que la ejecución fue local), me permitió visualizar estadísticas como número de tareas completadas y cantidad de memoria usada.



La interfaz gráfica de Spark fue esencial para entender visualmente la secuencia de operaciones y cómo Spark gestiona los recursos en segundo plano.

Conclusión

Realizar esta práctica me permitió consolidar mis conocimientos sobre procesamiento distribuido y el middleware Spark.

Aprendí cómo Spark divide los datos en particiones y ejecuta operaciones de transformación de forma perezosa, lanzándolas efectivamente sólo tras una acción como collect(). También comprendí mejor la diferencia entre transformaciones que requieren shuffle y aquellas que no lo requieren, lo cual tiene un impacto directo en el rendimiento del sistema.

Además, la experiencia me permitió familiarizarme con aspectos prácticos de Docker y el manejo de volúmenes, algo que en contextos reales de computación distribuida es crucial para garantizar que los datos estén correctamente accesibles por los nodos de procesamiento.

En conclusión, esta práctica no sólo reforzó los conceptos de alto rendimiento y procesamiento paralelo, sino que también me dio herramientas prácticas para desplegar y analizar aplicaciones distribuidas en entornos controlados y reproducibles.