Práctica 10

Análisis distribuido de

mantenimiento en una flota robótica

industrial

Jordi Blasco Lozano

Computación de alto rendimiento

Grado en Inteligencia Artificial

## Índice:

[Índice: 2](#_Toc196492306)

[1. Introducción 2](#_Toc196492307)

[2. Desarrollo 3](#_Toc196492308)

[Código 5](#_Toc196492309)

[Ejecución 6](#_Toc196492310)

## Introducción

Hemos montado un mini laboratorio para procesar los logs de varios brazos robóticos de forma rápida y paralela. La idea clave es:

1. Leer muchos ficheros de texto que contienen sucesos técnicos (sobrecalentamientos, reinicios, calibraciones, etc.).
2. Filtrar solo los eventos que nos interesan.
3. Contar cuántas veces aparece cada tipo de evento.
4. Mostrar un pequeño informe con las cuentas.

Para hacerlo usamos Apache Spark con Scala ya que:

* Spark divide los datos en particiones y ejecuta operaciones filter, map, reduceByKey al mismo tiempo en cada trozo, aprovechando todos los núcleos de la máquina.
* Hasta que no pedimos el resultado final con collect(), Spark solo construye el plan de trabajo; al llamar a collect() se dispara todo el procesamiento, se unen los resultados y se devuelven al programa principal.
* Scala encaja muy bien con Spark: su sintaxis funcional y tipada nos permite escribir las cadenas de transformación de forma muy clara y compacta.

## Desarrollo

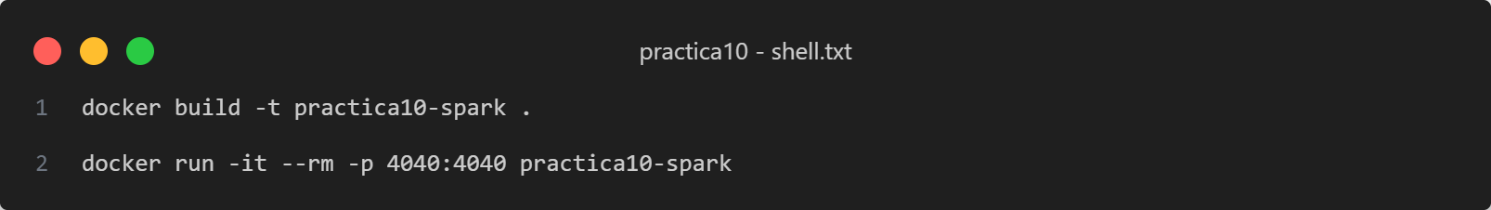
Para simplificar la puesta en marcha y evitar tener que invocar manualmente spark-shell con múltiples flags, hemos definido un Dockerfile que contiene la imagen oficial de Bitnami Spark y copia en /app todo nuestro proyecto (scripts Scala y carpeta registros).



Este Dockerfile:

* Establece un HOME válido y crea ~/.ivy2/local para que Ivy no falle al gestionar dependencias.
* Copia el código y los datos a /app.
* Fija como ENTRYPOINT el comando: spark-shell --master local[\*] --conf spark.ui.port=4040

De este modo, basta con ejecutar:



Automáticamente tendremos primero los logs de la ejecución de spark y finalmente una shell con scala para poder cargar archivos .scala que tengamos en nuestro contenedor o directamente ejecutar los comandos en terminal. Para esta práctica hemos optado por la estructura sugerida. Tendremos un archivo spark-shell.scala que se cargará en la terminal con el comando :load /app/spark-shell.scala.

La carga de este archivo ejecuta las siguientes operaciones definidas en él:

1. **Importación**: Se importa org.apache.spark.SparkContext.\_ para habilitar las transformaciones y acciones sobre RDDs, como reduceByKey.
2. **Lectura de datos**: Se utiliza sc.textFile("/app/registros/\*.txt") para leer todos los archivos .txt presentes en la carpeta /app/registros (montada desde la carpeta local registros/) y crear un RDD de strings llamado registros, donde cada elemento es una línea de un archivo de log.
3. **Definición de eventos clave**: Se crea un Set[String] llamado eventosClave que contiene los identificadores de los eventos que se desean rastrear.
4. **Filtrado y Mapeo flatMap**: Se aplica una transformación flatMap sobre el RDD registros. Esta operación:

* Divide cada línea usando " - " como separador.
* Comprueba si la división produjo al menos dos partes (parts.length >= 2) y si la segunda parte (parts(1)) está contenida en el conjunto eventosClave.
* Si ambas condiciones son ciertas, emite una tupla (evento, 1).
* Si alguna condición falla (línea mal formada o evento no clave), no emite nada (None), descartando esa línea de forma segura y evitando errores ArrayIndexOutOfBoundsException. El resultado es un nuevo RDD llamado pares de tipo RDD[(String, Int)].

1. **Reducción (reduceByKey):** Se aplica reduceByKey(\_ + \_) sobre el RDD pares. Esta operación agrupa todas las tuplas por su clave (el nombre del evento) y suma sus valores, generando un RDD llamado conteos que contiene pares (evento, total\_ocurrencias).
2. **Recolección (collect):** Se ejecuta la acción conteos.collect(). Esta acción desencadena la ejecución de todas las transformaciones anteriores (lectura, flatMap, reduceByKey) en el clúster Spark (en este caso, localmente). El resultado final, que es un Array[(String, Int)] con el conteo total de cada evento clave, se envía de vuelta al programa driver (el spark-shell) y se almacena en la variable resultado.
3. **Impresión:** Finalmente, se itera sobre el array resultado usando foreach y se imprime cada par (evento, conteo) formateado en la consola.

Una vez que el comando :load /app/spark-shell.scala ha terminado de ejecutarse en el REPL, la variable resultado ya contiene el array con los conteos finales y está disponible en el scope de la sesión de spark-shell. Para volver a ver o trabajar con estos resultados, simplemente se puede escribir resultado en el shell y presionar Enter:

## Código

import org.apache.spark.SparkContext.\_

// 1) Leer todos los logs

val registros = sc.textFile("/app/registros/\*.txt")

// 2) Definir eventos clave

val eventosClave = Set(

"ALERTA\_SOBRECALENTAMIENTO",

"REINICIO\_SISTEMA",

"CALIBRACION\_AUTOMATICA",

"CARGA\_EXCESIVA",

"PAUSA\_NO\_PROGRAMADA"

)

// 3) Filtra, parsea y mapea a pares (String,Int)

val pares = registros.flatMap { line =>

val parts = line.split(" - ")

if (parts.length >= 2 && eventosClave.contains(parts(1)))

Some((parts(1), 1))

else

None

}

// 4) Reduce por clave

val conteos = pares.reduceByKey(\_ + \_)

// 5) Traer al driver y mostrar

val resultado = conteos.collect()

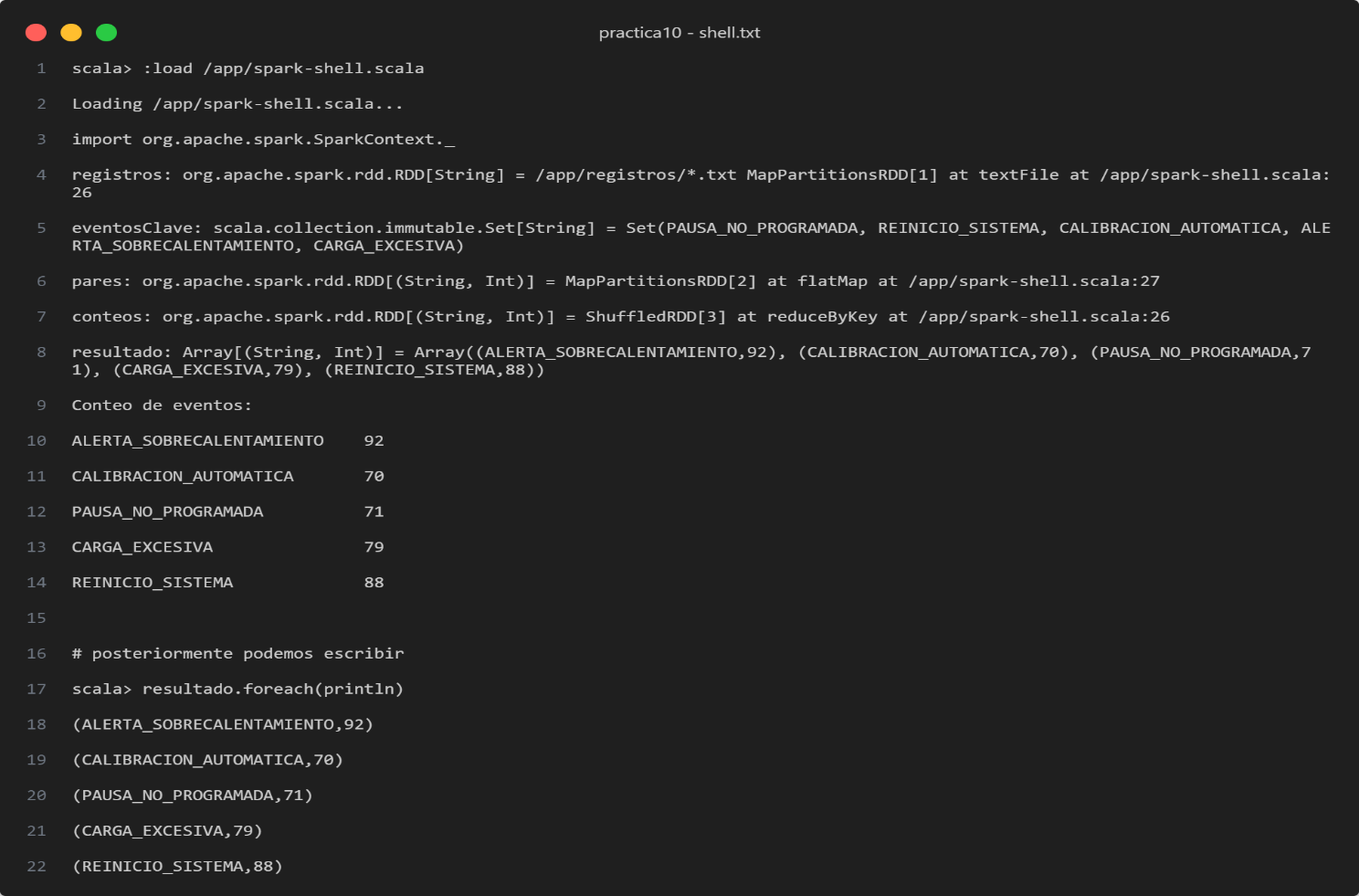
println("Conteo de eventos:")

resultado.foreach{ case (ev, cnt) =>

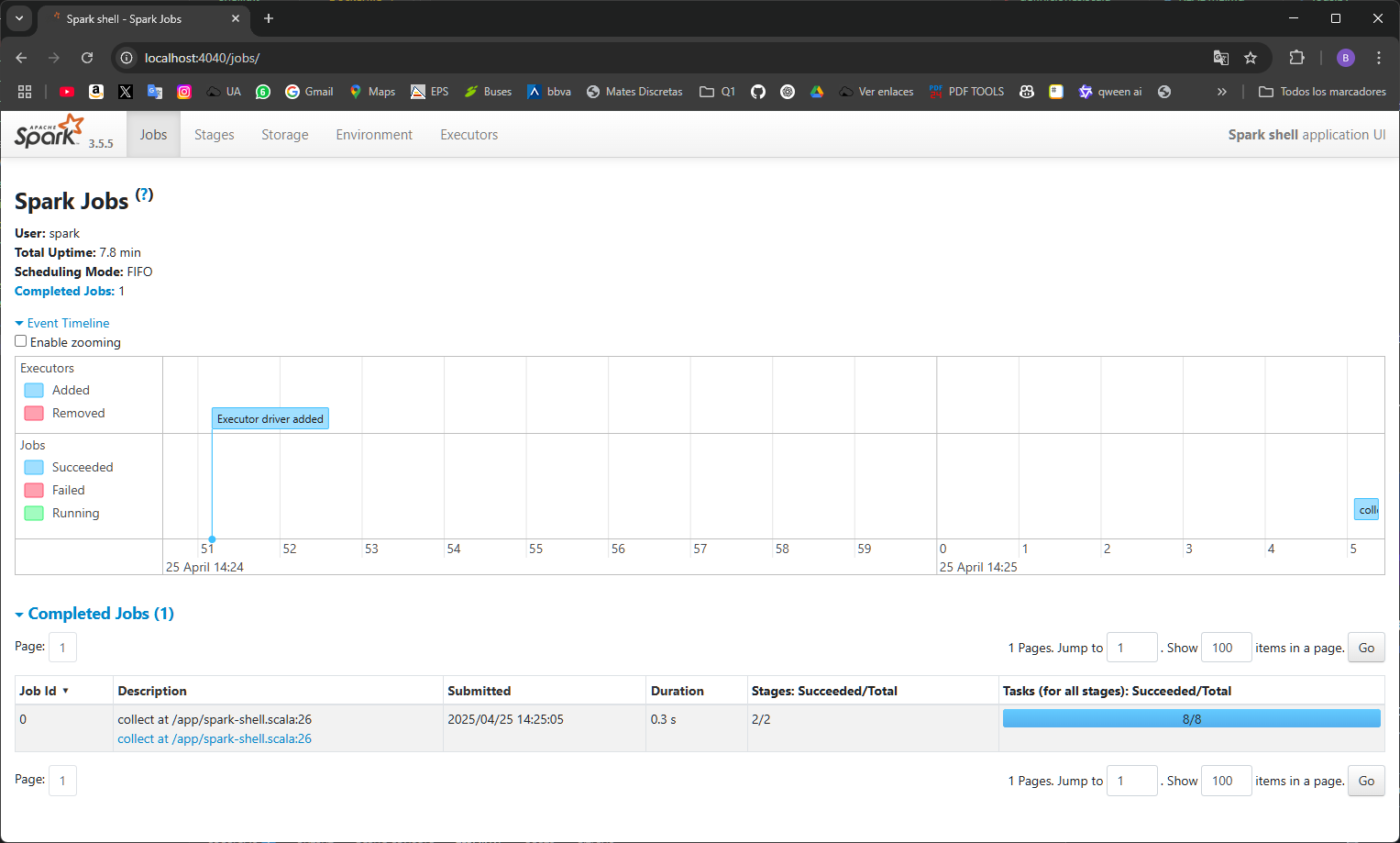
println(f"$ev%-25s $cnt%5d")

}

## Ejecución

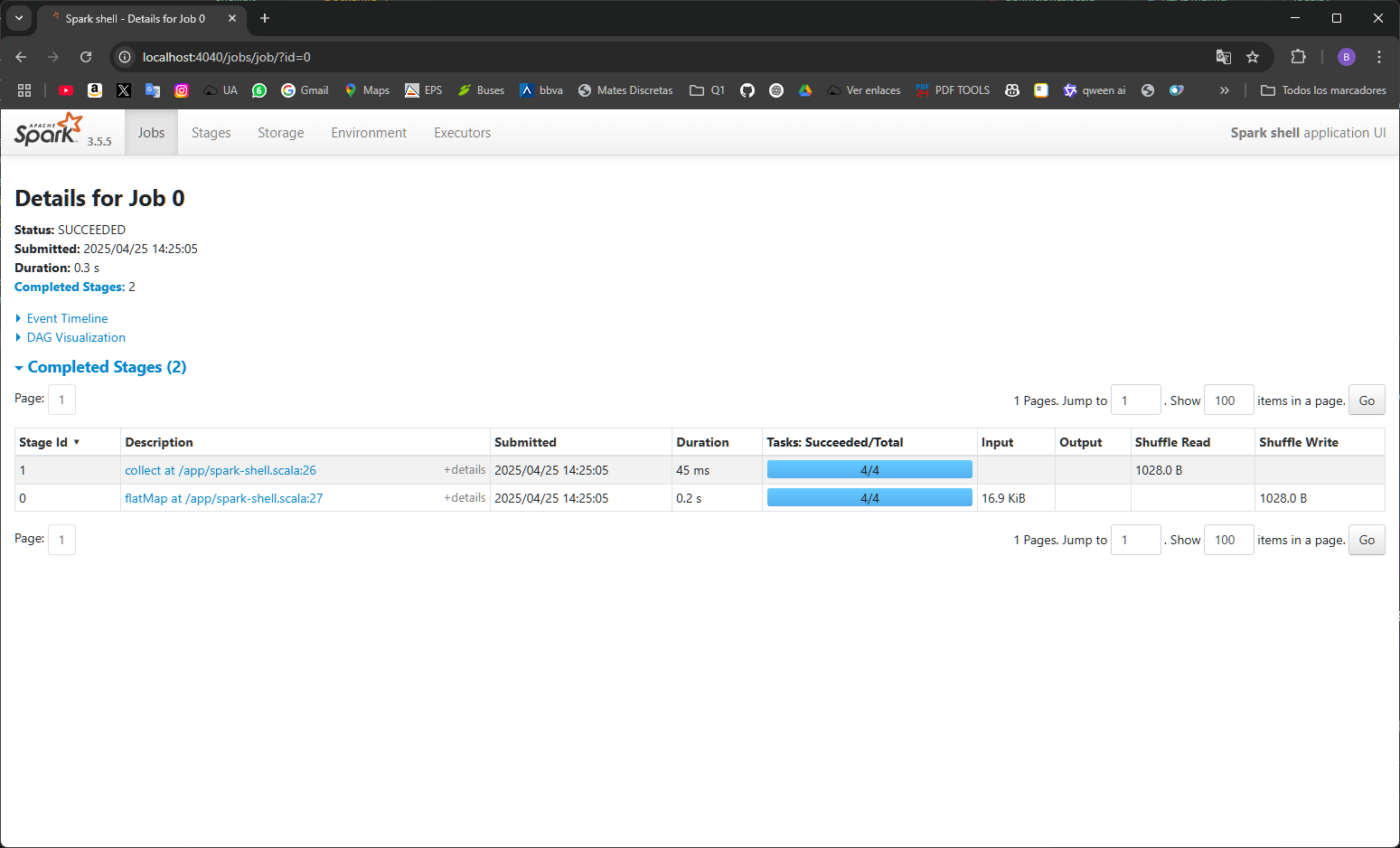
**Terminal**

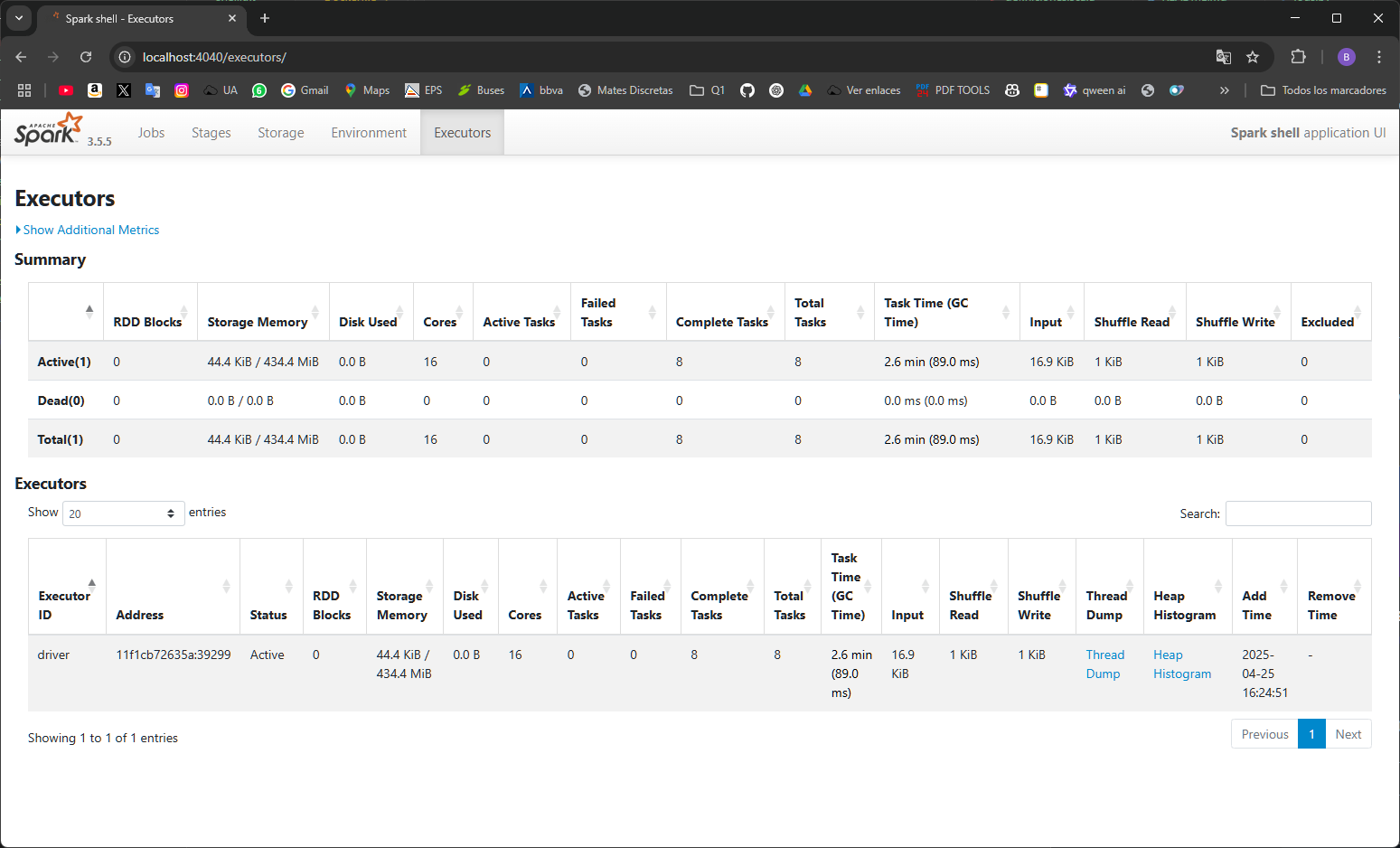
**Pestaña Jobs**



**Datails del Job ejecutado (incluyendo stages)**

Es la misma pestaña que `Stages` pero con más datos sobre el Job, contiene las mismas Stages que la pestaña `Stages` al solo haber ejecutado un único Job.



**Pestaña Executors**

**Conclusiones sobre la ejecución.**

Al ejecutar :load /app/spark-shell.scala, la acción final conteos.collect() dispara un Job en Spark.

La Pestaña Jobs (y Details del Job) muestra que Spark dividió este Job en 2 Stages debido a la operación reduceByKey, que requiere un shuffle (redistribución de datos):

* Stage 0: Corresponde a la lectura de los archivos (textFile) y la transformación flatMap. Spark lee los datos de entrada, filtra las líneas y mapea los eventos clave a pares (evento, 1). El resultado de esta etapa son datos intermedios que se escriben para el shuffle.
* Stage 1: Corresponde a la operación reduceByKey y la acción collect. Spark lee los datos intermedios de la etapa anterior, agrupa los pares por evento, suma los conteos, y finalmente envía el array resultante (resultado) al driver (la terminal).

La Pestaña Executors confirma que todo el trabajo se realizó en el driver al ejecutarse en modo local[\*].