

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
1. Introducción	2
2. Desarrollo	3
Código	5
Ejecución	6

1. 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.

2.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).

```
practica10 - DOCKERFILE
1 FROM bitnami/spark:latest
2
3 # 1) Nos aseguramos de usar el usuario root (para poder crear ~/.ivy2)
4 USER root
5
6 # 2) Creamos el HOME y el dir de ivy
7 ENV HOME=/root
8 RUN mkdir -p /root/.ivy2/local
9
10 # 3) Copiamos todo el proyecto a /app
11 WORKDIR /app
12 COPY . /app
13
14 # 4) Arrancamos spark-shell en modo local[*] con UI en 4040
15 ENTRYPOINT ["spark-shell", \
16             "--master", "local[*]", \
17             "--conf", "spark.ui.port=4040" \
18             ]
```

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:

```
practica10 - shell.txt
1 docker build -t practica10-spark .
2 docker run -it --rm -p 4040:4040 practica10-spark
```

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)]`.
5. **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`).
6. **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`.
7. **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

```
practica10 - shell.txt

1  scala> :load /app/spark-shell.scala
2  Loading /app/spark-shell.scala...
3  import org.apache.spark.SparkContext._
4  registros: org.apache.spark.rdd.RDD[String] = /app/registros/*.txt MapPartitionsRDD[1] at textFile at /app/spark-shell.scala:26
5  eventosClave: scala.collection.immutable.Set[String] = Set(PAUSA_NO_PROGRAMADA, REINICIO_SISTEMA, CALIBRACION_AUTOMATICA, ALERTA_SOBRECALENTAMIENTO, CARGA_EXCESIVA)
6  pares: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[2] at flatMap at /app/spark-shell.scala:27
7  conteos: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3] at reduceByKey at /app/spark-shell.scala:26
8  resultado: Array[(String, Int)] = Array((ALERTA_SOBRECALENTAMIENTO,92), (CALIBRACION_AUTOMATICA,70), (PAUSA_NO_PROGRAMADA,71), (CARGA_EXCESIVA,79), (REINICIO_SISTEMA,88))
9  Conteo de eventos:
10 ALERTA_SOBRECALENTAMIENTO    92
11 CALIBRACION_AUTOMATICA      70
12 PAUSA_NO_PROGRAMADA        71
13 CARGA_EXCESIVA              79
14 REINICIO_SISTEMA            88
15
16 # posteriormente podemos escribir
17 scala> resultado.foreach(println)
18 (ALERTA_SOBRECALENTAMIENTO,92)
19 (CALIBRACION_AUTOMATICA,70)
20 (PAUSA_NO_PROGRAMADA,71)
21 (CARGA_EXCESIVA,79)
22 (REINICIO_SISTEMA,88)
```

Pestaña Jobs

Spark shell - Spark Jobs

localhost:4040/jobs/

spark 3.5.5

Jobs

Stages

Storage

Environment

Executors

Spark shell application UI

Spark Jobs (?)

User: spark
Total Uptime: 7.8 min
Scheduling Mode: FIFO
Completed Jobs: 1

Event Timeline

Enable zooming

Executors

Added

Removed

Jobs

Succeeded

Failed

Running

Completed Jobs (1)

Page: 1

1 Pages. Jump to 1. Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at /app/spark-shell.scala:26 collect at /app/spark-shell.scala:26	2025/04/25 14:25:05	0.3 s	2/2	8/8

Page: 1

1 Pages. Jump to 1. Show 100 items in a page. Go

Details 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.

Spark shell - Details for Job 0

localhost:4040/jobs/job?id=0

Spark 3.5.5

JobsStagesStorageEnvironmentExecutors

Spark shell application UI

Details for Job 0

Status: SUCCEEDED

Submitted: 2025/04/25 14:25:05

Duration: 0.3 s

Completed Stages: 2

Event Timeline

DAG Visualization

Completed Stages (2)

Page: 1

1 Pages. Jump to 1. Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at /app/spark-shell.scala:26	+details 2025/04/25 14:25:05	45 ms	4/4			1028.0 B	
0	flatMap at /app/spark-shell.scala:27	+details 2025/04/25 14:25:05	0.2 s	4/4	16.9 KiB			1028.0 B

Page: 1

1 Pages. Jump to 1. Show 100 items in a page. Go

Pestaña Executors

Spark shell - Executors

localhost:4040/executors/

Spark 3.5.5

JobsStagesStorageEnvironmentExecutors

Spark shell application UI

Executors

Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	0	44.4 KiB / 434.4 MiB	0.0 B	16	0	0	8	8	2.6 min (89.0 ms)	16.9 KiB	1 KiB	1 KiB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	0	44.4 KiB / 434.4 MiB	0.0 B	16	0	0	8	8	2.6 min (89.0 ms)	16.9 KiB	1 KiB	1 KiB	0

Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump	Heap Histogram	Add Time	Remove Time
driver	11f1cb72635a:39299	Active	0	44.4 KiB / 434.4 MiB	0.0 B	16	0	0	8	8	2.6 min (89.0 ms)	16.9 KiB	1 KiB	1 KiB	Thread Dump	Heap Histogram	2025-04-25 16:24:51	-

Showing 1 to 1 of 1 entries

Previous1Next

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[*]`.