Apache Spark

Autor: José Luis Cendán Guzmán || Correo:jcendguz@myuax.com || Fecha: 08/01/2026

Configuración del entorno y ejecución del cluster

1. **Problema inicial: Virtualización Deshabilitada**

   Tras instalar y intentar iniciar el Docker Desktop, me daba el siguiente error:
   Server: failed to connect to the docker API at npipe:////./pipe/dockerDesktopLinuxEngine; check if the path is correct and if the daemon is running: open //./pipe/dockerDesktopLinuxEngine: The system cannot find the file specified.
   Para solucionarlo, tuve que activar en la BIOS (en mi caso al tener AMD) el SVM Mode, que se trata de la virtualización por hardware.

2. **Configuración del cluster spark**

   Una vez arreglado el error anterior, ejecuté docker compose up, dando lo siguiente(solo un worker):

```
60107\spark> docker compose up -d
[+] Running 3/3
 ✓ Network spark_default      Created                                    0.1s
 ✓ Container spark-master-1   Started                                    1.1s
 ✓ Container spark-worker-1   Started                                    1.3s
```

Name	Container ID	Image	Port(s)	CPU (%)
spark	-	-	-	0.37%
master-1	91a948af760c	apache/spark:4.1.0-scala2.13-java21-python	4040:4040 Show all ports (4)	0.24%
worker-1	e5755820692e	apache/spark:4.1.0-scala2.13-java21-python		0.13%

   Ahora, realizo la comprobación que el cluster de spark está ejecutándose en el docker

```
PS D:\Github\luiscendan-private\luiscendan-private\estudios\master\1º_cua
os_20260107\spark> docker compose exec master /bin/bash
spark@master:/opt/spark/work-dir$ /opt/spark/bin/spark-submit --version
WARNING: Using incubator modules: jdk.incubator.vector
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 4.1.0
      /_/

Using Scala version 2.13.17, OpenJDK 64-Bit Server VM, 21.0.9
Branch HEAD
Compiled by user runner on 2025-12-11T14:03:23Z
Revision e221b56be7b6d9e48e107fc4d1cf0c15f02700f8
Url https://github.com/apache/spark
Type --help for more information.
spark@master:/opt/spark/work-dir$
```

   Ahora lo escalé a 3 workers y lo compruebo en el Spark UI

```
PS D:\Github\luiscendan-private\luiscendan-private\estudios\master\1º_cuatrimestre\SM141500_Analisis_Informacion\Entregas\Entrega3\dat
os_20260107\spark> docker compose up --scale worker=3 -d
[+] Running 4/4
 ✓ Container spark-master-1   Running                                    0.0s
 ✓ Container spark-worker-1   Running                                    0.0s
 ✓ Container spark-worker-3   Started                                    1.5s
 ✓ Container spark-worker-2   Started                                    1.1s
```

Name	Container ID	Image	Port(s)	CPU (%)
spark	-	-	-	0.4%
master-1	91a948af760c	apache/spark:4.1.0-scala2.13-java21-python	4040:4040 Show all ports (4)	0.11%
worker-1	e5755820692e	apache/spark:4.1.0-scala2.13-java21-python		0.1%
worker-2	2a75ca32349c	apache/spark:4.1.0-scala2.13-java21-python		0.08%
worker-3	8f918f6a00fc	apache/spark:4.1.0-scala2.13-java21-python		0.11%

Desarrollo del Script Principal con PySpark

Objetivo del Análisis

El objetivo es implementar una adaptación del **One Billion Row Challenge** para procesar el archivo *measurements.txt* que contiene mediciones de temperatura de 413 estaciones meteorológicas. El script debe calcular el mínimo, promedio y máximo de temperatura por estación, presentando los resultados ordenados alfabéticamente en el formato estación=min/mean/max, con valores redondeados hacia abajo a un decimal.

Enfoque de Desarrollo

El script se implementó utilizando RDDs de Apache Spark, siguiendo un flujo de procesamiento distribuido en cuatro etapas:

1. **Carga y parseo de datos.** El archivo se lee mediante *sc.textFile()*, que distribuye automáticamente las líneas entre los workers del clúster. Cada línea se parsea con map() para extraer la estación y temperatura, generando un RDD de tuplas (estación, temperatura).
2. **Agrupación por estación**. Con *groupByKey*() se reúnen todas las temperaturas de cada estación en un solo lugar. Esta operación provoca un *shuffle* que redistribuye los datos entre los workers, pero es necesaria porque necesitamos acceder a todas las mediciones de forma conjunta para calcular el mínimo, promedio y máximo.
3. **Cálculo de estadísticas.** Con *mapValues*() se procesan en paralelo los grupos de temperaturas, calculando mínimo, promedio y máximo para cada estación. El redondeo hacia abajo se implementa con *math.floor*(valor * 10) / 10 para cumplir exactamente con la especificación del desafío.
4. **Ordenamiento y presentación** Finalmente, *sortByKey*() ordena alfabéticamente las estaciones aprovechando el particionamiento distribuido de Spark, antes de recopilar y mostrar los resultados.

Decisiones Técnicas Clave

- **Uso de RDDs según los requisitos del ejercicio**
  La actividad requería trabajar con RDDs (Resilient Distributed Datasets), que es la API fundamental de Spark. Aunque hoy en día DataFrames son más comunes, los RDDs me permitieron entender cómo funciona Spark a bajo nivel. Esto me dio control total sobre cómo se distribuyen y procesan los datos en el clúster.
- **groupByKey() para calcular múltiples estadísticas**
  Necesitaba calcular tres métricas (mínimo, promedio y máximo) para cada estación. Aunque *reduceByKey*() es más eficiente en términos de red, groupByKey() me permitió hacer todo en una sola función (funcion calcular y despues agruparlo en stats)
  ¿Por qué esta decisión?
    - Con groupByKey() tengo todas las temperaturas juntas y puedo calcular las tres estadísticas en una pasada.
    - Si usara reduceByKey(), tendría que hacer tres operaciones separadas o crear estructuras más complejas.

```python
#Cargar y parsear datos
lineas = sc.textFile(archivo_datos)

def parsear(linea):
    partes = linea.split(';')
    return (partes[0], float(partes[1]))

mediciones = lineas.map(parsear)

#Agrupar por estación y calcular estadísticas
def calcular(temps):
    lista = list(temps)
    return (min(lista), sum(lista)/len(lista), max(lista))

stats = mediciones.groupByKey().mapValues(calcular)
```

  Comparación con Pandas
  Implementé la misma lógica con Pandas para poder comparar y lo que esperaba encontrar:
    - Pandas sería más rápido para este dataset pequeño (y lo fue)
    - Spark tiene overhead de inicialización del clúster
    - Pero Spark escalaría mucho mejor con datasets grandes (varios GB)

```python
#Versión con Pandas para comparar rendimiento
import pandas as pd

df = pd.read_csv(archivo_datos, sep=";",
                 names=["estacion", "temperatura"])

stats = df.groupby("estacion")["temperatura"].agg(['min', 'mean', 'max'])
```

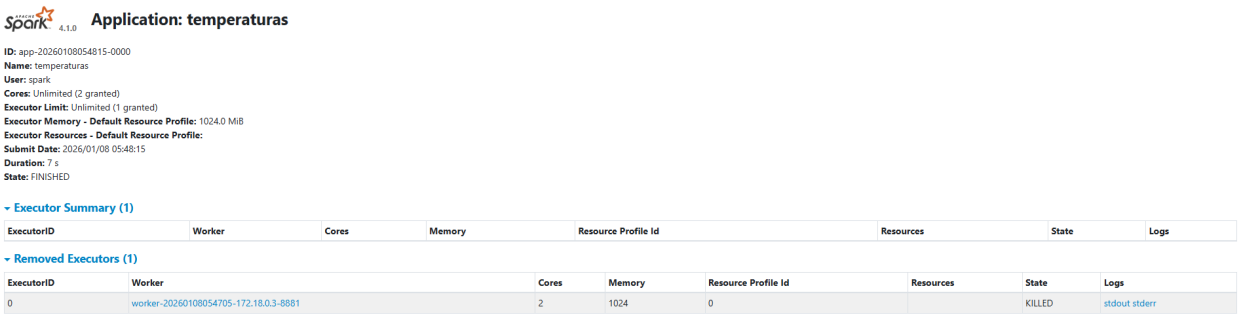
  Esto demuestra que la herramienta correcta depende del problema: Pandas para análisis rápido local, Spark para Big Data distribuido
- **Formato consistente de resultados**
  Usé math.floor() para redondear a 1 decimal de forma consistente (como se pedía en el enunciado)
- **Archivos de resultados diferenciados y detección automática del entorno.**
  Por un lado, guardamos los resultados mediante de si son versión pandas o si tiene 1 o 2 workers, para el caso de spark. Y, por el tema del entorno,si detectamos que está dentro del docker (os.path.exists("/tmp/data/measurements.txt"))o no.

Ejecuciones

Se han realizado las pruebas con spark (1 worker y 3 workers) y con la librería de Pandas. A continuación las sentencias y donde se depositan los resultados y evidencias del Spark U

1. **Ejecutar el script con un worker.**
   Docker compose up -d
   docker compose exec master /bin/bash

/opt/spark/bin/spark-submit --master spark://master:7077 /opt/spark/scripts/measurements_analysis.py
–Los resultados se encuentran en /tmp/data/results/resultados_spark_1w.txt–
Me dirijo a localhost:8080 y cojo la ejecución que acabo de realizar, cuyo nombre es temperaturas (lo definimos en el codigo de la siguiente forma (conf1 = SparkConf().setAppName("temperaturas")) y donde se muestra que solo se uso un worker


Application: temperaturas

ID: app-20260108054815-0000
Name: temperaturas
User: spark
Cores: Unlimited (2 granted)
Executor Limit: Unlimited (1 granted)
Executor Memory - Default Resource Profile: 1024.0 MiB
Executor Resources - Default Resource Profile:
Submit Date: 2026/01/08 05:48:15
Duration: 7 s
State: FINISHED

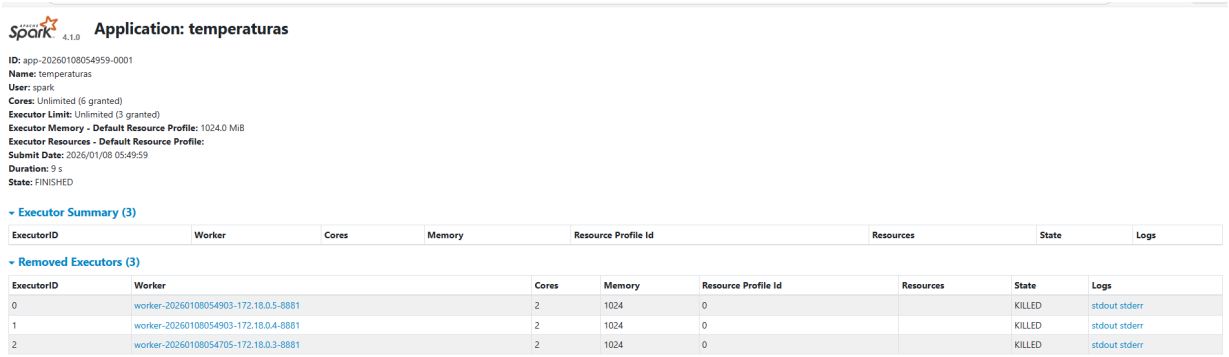
Executor Summary (1)

ExecutorID	Worker	Cores	Memory	Resource Profile Id	Resources	State	Logs

Removed Executors (1)

ExecutorID	Worker	Cores	Memory	Resource Profile Id	Resources	State	Logs
0	worker-20260108054705-172.18.0.3-8881	2	1024	0		KILLED	stdout stderr

2. **Ejecución con Pandas**
   Para la comparación con Pandas, ejecuté el script **en local** (fuera de Docker), ya que Pandas no requiere el clúster Spark y puede ejecutarse directamente en cualquier entorno Python:
   python scripts\measurements_analysis.py pandas
   –Los resultados se encuentran en data/results/resultados_pandas.txt–
3. **Ejecución con 3 workers**
   docker compose up --scale worker=3 -d
   docker compose exec master /bin/bash
   /opt/spark/bin/spark-submit --master spark://master:7077 /opt/spark/scripts/measurements_analysis.py spark3
   –Los resultados se encuentran en /tmp/data/results/resultados_spark_3w.txt–


Application: temperaturas

ID: app-20260108054959-0001
Name: temperaturas
User: spark
Cores: Unlimited (6 granted)
Executor Limit: Unlimited (3 granted)
Executor Memory - Default Resource Profile: 1024.0 MiB
Executor Resources - Default Resource Profile:
Submit Date: 2026/01/08 05:49:59
Duration: 9 s
State: FINISHED

Executor Summary (3)

ExecutorID	Worker	Cores	Memory	Resource Profile Id	Resources	State	Logs

Removed Executors (3)

ExecutorID	Worker	Cores	Memory	Resource Profile Id	Resources	State	Logs
0	worker-20260108054903-172.18.0.5-8881	2	1024	0		KILLED	stdout stderr
1	worker-20260108054903-172.18.0.4-8881	2	1024	0		KILLED	stdout stderr
2	worker-20260108054705-172.18.0.3-8881	2	1024	0		KILLED	stdout stderr

Evaluación del Rendimiento

Los resultados muestran que Pandas (0.92s) superó ampliamente a Spark con 1 worker (7.00s) y 3 workers (9.00s). El overhead de Spark (inicialización, serialización, shuffle) no se compensa con un dataset de solo 13.7 MB. El speedup negativo de 0.78x al escalar a 3 workers confirma que la paralelización es contraproducente en datasets pequeños. Esto demuestra que Pandas es la herramienta correcta para datos en memoria, mientras que Spark brilla con datasets masivos que requieren distribución.

Conclusiones

Estos resultados demuestran que la herramienta correcta depende del tamaño de los datos. Para datasets pequeños que caben en memoria, Pandas es mucho más eficiente. Spark solo muestra ventajas con datasets masivos que no pueden procesarse en un solo nodo. Lecciones clave:

- El overhead de Spark solo se justifica con Big Data
- Más workers no siempre mejoran el rendimiento (Ley de Amdahl)
- Los RDDs permiten entender cómo funciona la distribución de datos en Spark