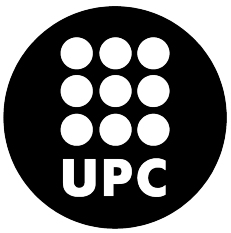
Universidad Politécnica de Barcelona

Facultad de Informática

Grado en Ciencia e Ingeniería de Datos



Proyecto Spark

Paralelismo y Sistemas Distribuidos

Pau González

48102077S

Rebeca Torrecilla

46189534Z

Profesora

Yolanda Becerra Fontal

Barcelona
Julio de 2025

1. Introducción

El objetivo del siguiente proyecto se centra en medir el rendimiento de tres programas al implementar una aplicación Spark que analiza una información descargada de la base de datos de IMDB. El propósito del cálculo radica en encontrar el número de películas en las que han coincidido un mismo director y actor junto con su calificación media, mínima y máxima para, finalmente, obtener la información en la terminal de las veinte colaboraciones más concurrentes.

El rendimiento se evaluará primeramente en una versión escrita en Python y ejecutada de forma secuencial. El segundo programa utilizará una versión distribuida escrita para Spark usando el interfaz basado en RDDs y ejecutada sobre cuatro máquinas virtuales de Amazon Web Services. El tercero y último será una versión también distribuida y ejecutada sobre cuatro máquinas virtuales del AWS con una interfaz basada en Dataframes.

2. Descripción del experimento

2.1. Configuración del entorno

A la hora de configurar el experimento comenzamos creando cuatro instancias en los servicios de AWS. Personalizamos todas ellas con las mismas opciones que, difiriendo de la selección por defecto, són las siguientes:

- Utilizamos un sistema operativo Linux con distribución Ubuntu.
- Tipo de instancia *m5.large* que dispone de 2 cores, 8GB de RAM y un coste de 0.096$ la hora, suficientes para contar con algo de paralelismo en cada nodo.
- Aumentamos el tamaño del disco a 30GB (en la sección de *Configuración de la cantidad de disco*).
- Configuramos la red para que el cluster de Spark nos funcione con las instancias de Amazon. Para ello, cambiamos las reglas que permiten el tráfico de entrada para las instancias. En específico, en el apartado de ajustes de red, añadimos una nueva regla de grupo de seguridad donde permitimos cualquier

tipo de origen y puerto cambiando la opción de *Source type* a *Anywhere* y *Portrange* a 0-65535.

Una vez realizados todos estos puntos, lanzamos las cuatro instancias (*launch instance*).

Como último paso para configurar el entorno, mandamos a ejecutar un script llamado `automatic_deploy.sh` mediante la comanda:

```
%sh automatic_deploy.sh  /.ssh/miclave.pem IP1 IP2 IP3 IP4
```

siendo `~/.ssh/miclave.pem` el path al fichero con la clave para acceder a las instancias de Amazon junto a las IPs de las cuatro instancias creadas.

El script se encarga de:

- Hacer la descarga e instalación de todo el software necesario para las instancias de Amazon.
- Ejecuta en toads las instancias el comando `ssh-keygen` y se encarga de que en todas ellas el fichero `.ssh/authorized_keys` tenga las claves de todas las instancias.

2.2. Desplegado de Spark

Seleccionamos una de las cuatro instancias como master y nos conectamos a ella. A continuación, creamos el fichero `workers` en el directorio `$SPARK_HOME/conf` del master donde escribimos las IPs públicas de las instancias. Ejecutamos el comando siguiente para poner en marcha el clúster de Spark:

```
$SPARK_HOME/sbin/start-all.sh
```

Si queremos asegurarnos de que los workers se hayan registrado sin problemas, podemos comprobarlo en el directorio `$SPARK_HOME/logs`.

Por último, en el fichero del log del master buscamos la frase *Starting Spark master at* pues, seguida de este, se encuentra su *url*, necesaria para poder ejecutar en distribuido.

2.3. Implementación de la aplicación

Antes de comenzar con las aplicaciones, descargamos de la base de datos de IMDB un fichero conteniendo, para cada película, una línea con cada actor que ha participado en ella. La cabecera es de la forma:

`tconst,actor,actorName,director,directorName,averageRating,numVotes`

Ahora sí, implementamos los códigos que calculen en cuántas películas han coincidido actor con director, la calificación media de todas estas películas, calificación mínima y calificación máxima. Todo esto deberá aparecer en la terminal al final de la ejecución, mostrando las veinte primeras películas resultantes en orden descendente por concurrencia en las colaboraciones.

Nosotros utilizamos tres scripts diferentes con el objetivo de evaluar más tarde el rendimiento de éstos. Cada uno se basa en una interfaz diferente con el objetivo de comprobar la eficiencia de cada uno y compararlos. Los códigos son los siguientes:

Listing 1: Aplicación secuencial basada en Python.

```
1  import csv
2  import sys
3
4  d={}
5  #header: tconst,actor,actorName,director,directorName,
       averageRating,numVotes
6  with open('infoActors.csv', newline='') as csvfile:
7      reader = csv.DictReader(csvfile,escapechar='\\')
8      for row in reader:
9          rating=float((row['averageRating']))
10         t=d.get((row['actor'],row['director']),(row['
               actorName'], row['directorName'],0,0,rating,
               rating))
11         d[(row['actor'],row['director'])] = (row['
               actorName'], row['directorName'],t[2]+1, t
               [3]+rating, min(rating,t[4]), max(rating,t
               [5]))
12
13 d\_filtered = [(v[0],v[1],v[2],v[3]/v[2],v[4],v[5]) for
       k,v in d.items() if v[2]>=2]
14 d\_ordered=sorted(d\_filtered,key=lambda l: -l[2])
15
16 print(d\_ordered[:20])
```

Listing 2: Aplicación distribuida basada en RDDs.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, avg, min, max
from pyspark.sql.types import StructType, StructField, StringType, DoubleType

# Guardamos los datos en una estructura
schema = StructType([
    StructField("tconst", StringType(), True),
    StructField("actor", StringType(), True),
    StructField("actorName", StringType(), True),
    StructField("director", StringType(), True),
    StructField("directorName", StringType(), True),
    StructField("averageRating", DoubleType(), True),
])

spark = SparkSession.builder \
    .appName("ActorDirectorDF") \
    .getOrCreate()

df = spark.read.csv("infoActorsx10.csv", header=True, schema=schema, escape="\\") \
    .select("actor", "actorName", "director", "directorName", "averageRating") \
    .repartition(200, "actor", "director") \
    .cache()

# Agrupamos y filtramos los datos
grouped = df.groupBy("actor", "actorName", "director", "directorName") \
    .agg(
        count("*").alias("num_collaborations"),
        avg("averageRating").alias("avg_rating"),
        min("averageRating").alias("min_rating"),
        max("averageRating").alias("max_rating")
    ) \
    .filter(col("num_collaborations") >= 2) \
    .orderBy(col("num_collaborations").desc()) \
    .limit(20) \  # Mostramos solo las 20 primeras
```

```
            peliculas
        .select("actorName", "directorName", "
            num_collaborations", "avg_rating", "min_rating",
            "max_rating")

    grouped.show(truncate=False)

    spark.stop()
```

Listing 3: Aplicación distribuida basada en DataFrames.

```
    from pyspark.sql import SparkSession
    from pyspark.sql.functions import col, count, avg, min,
        max
    from pyspark.sql.types import StructType, StructField,
        StringType, DoubleType

    schema = StructType([
        StructField("tconst", StringType(), True),
        StructField("actor", StringType(), True),
        StructField("actorName", StringType(), True),
        StructField("director", StringType(), True),
        StructField("directorName", StringType(), True),
        StructField("averageRating", DoubleType(), True),
    ])

    spark = SparkSession.builder \
        .appName("ActorDirectorDF") \
        .getOrCreate()

    df = spark.read.csv("infoActorsx10.csv", header=True,
        schema=schema, escape="\\") \
        .select("actor", "actorName", "director", "
            directorName", "averageRating") \
        .repartition(200, "actor", "director") \
        .cache()

    grouped = df.groupBy("actor", "actorName", "director", "
        directorName") \
        .agg(
            count("*").alias("num_collaborations"),
            avg("averageRating").alias("avg_rating"),
```

```
            min("averageRating").alias("min_rating"),
            max("averageRating").alias("max_rating")
        ) \
        .filter(col("num_collaborations") >= 2) \
        .orderBy(col("num_collaborations").desc()) \
        .limit(20) \
        .select("actorName", "directorName", "
           num_collaborations", "avg_rating", "min_rating",
           "max_rating")

    grouped.show(truncate=False)

    spark.stop()
```

Para poder ejecutar los códigos basados en RDDs y DataFrames en distribuido, los mandamos a ejecutar con las siguientes comandos:

```
$SPARK_HOME/bin/spark-submit --master="URLMASTER"
                RDD_actorDirector.py

$SPARK_HOME/bin/spark-submit --master="URLMASTER"
                DF_actorDirector.py
```

2.4. Evaluación de rendimiento

Ahora, de las tres aplicaciones utilizadas, vamos a medir su rendimiento en función del tamaño del fichero de entrada.

Para ello, usamos el comando `time` de Linux donde, de todas las métricas que dispone, nos quedaremos en especial con *elapsed time*, que es el tiempo transcurrido desde que nuestro programa comienza la ejecución hasta que acaba.

Finalmente, con los datos obtenidos, podemos observar y comentar los resultados.

3. Resultados

En la Figura 1 podemos ver la pendiente de las aplicaciones en función del nombre de datasets de entrada, correspondiendo las rectas azul, naranja y verde a las

aplicaciones de Python, RDDs y DataFrames respectivamente.

Cuadro 1
Tiempo de ejecución (en segundos) de las diferentes aplicaciones utilizadas en función del tamaño del fichero de entrada.

	Pyhton	RDD	DF
1	12,75	26,46	32,49
2	25,95	33,20	34,63
4	50,45	47,99	40,89
6	73,91	59,91	47,68
8	97,96	74,23	58,42
10	122,60	87,31	67,73
12	144,80	100,19	71,50
14	171,34	113,46	74,20

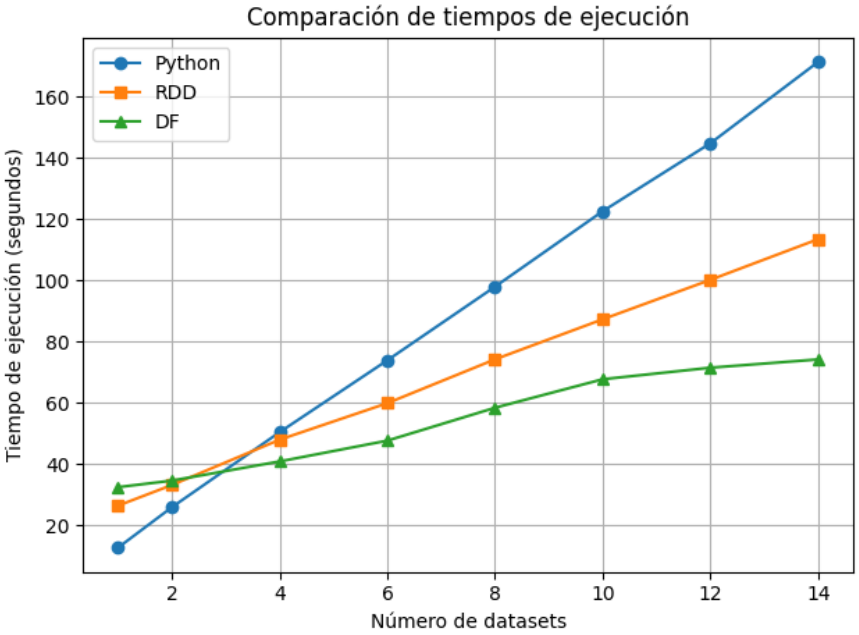


Figura 1
Gráfico comparativo del rendimiento de las aplicaciones.

Observamos la recta correspondiente a la aplicación Python. Vemos que, para datasets relativamente pequeños, el script secuencial obtiene mejores resultados al

compararlo con los casos distribuidos. Sin embargo, a medida que aumentamos el tamaño de la entrada, el tiempo de ejecución se incrementa con una pendiente lineal de, aproximadamente, once segundos por dataset añadido. De primeras puede no parecer un gran incremento aunque, si llegásemos a tener una base de datos importante, resolver el cálculo con este método no nos resultaría rentable. Por poner un ejemplo, si quisiésemos enviar una entrada proporcional a 200 datasets como el de nuestro experimento, obtendríamos el resultado después de 36,67 minutos.

Por el otro lado, si nos enfocamos en las aplicaciones que mandamos a ejecutar en distribuido, tanto RDD como DataFrame parecen presentar también una pendiente lineal mucho menos pronunciada que en el caso de la recta azul. Al inicio vemos que, para casos con uno o dos datasets, estos dos códigos llegan a tardar el doble de tiempo en ejecutar comparando con el secuencial, lo que nos da un resultado más pobre. Esto se debe a que, para conjuntos de datos pequeños, la sobrecarga de configuración, tiempo de inicio y la comunicación entre nodos son factores que hacen empeorar el rendimiento notablemente. Sin embargo, a medida que el número de líneas en el fichero de entrada aumenta, el tiempo de ejecución se incrementa más pausadamente, lo que indica más escalabilidad. Por poner un ejemplo, en el caso de 14 datasets, podemos ver una diferencia de casi 100 segundos de ejecución entre la aplicación secuencial y la distribuida más rápida (DataFrames).

Por último, vemos cómo la versión secuencial comienza a tener un mayor tiempo de ejecución (punto de cruce entre las rectas) alrededor de los tres datasets. Por tanto, podemos afirmar que a partir de 7.500.000 líneas aproximadamente, en este problema en particular nos saldría más rentable utilizar una aplicación distribuida antes que una secuencial, asegurándonos de ello a partir de los cuatro datasets.

Este punto de vista también se puede aplicar respecto al consumo de la CPU, una de las métricas en porcentaje disponibles al ejecutar nuestros programas. Inicialmente, al procesar los datos de forma secuencial, se observaba un consumo cercano al máximo de recursos disponibles, con un valor siempre superior al 90 % (especialmente en las últimas ejecuciones). Por otro lado, en el caso distribuido, hemos llegado a ver un resultado contrario, llegando a consumir el 50 % de la CPU para el caso de 10 datasets y casi un 30 % con 14 datasets.

4. Conclusiones

Finalmente, teniendo en consideración los resultados descritos en la sección anterior, podemos deducir que, para tamaños de entrada pequeños (1 o 2 datasets), la ejecución distribuida no compensa debido a que el costo añadido por la configuración y la comunicación de los nodos empeora bastante el rendimiento.

Sin embargo, a medida que aumentamos el tamaño del fichero de entrada, se observa una creciente diferencia entre los tiempos de ejecución y el consumo de la CPU. En conjuntos de datasets grandes, las aplicaciones distribuidas no solamente son capaces de acabar el trabajo mucho más rápido, sino que también aprovechan en gran medida el paralelismo, la distribución de carga y la capacidad de procesamiento agregada para reducir el uso de la CPU.