Technical University of Crete

School of Electrical and Computer Engineering

INF 424 – Functional Programming, Analytics and Applications – 2025

Spring Semester 2025 **Movie Preference Analyzer** **Due: 16.04.2025**

Spyridon Stamou, 2021030090

Introduction

In this project we develop a Movie Perfomance Analyzer, an analytic application by using Functional Programming over Apache Spark.

We utilize the largest version of the MovieLens dataset, which cointains information about :

- **movies** (movieId,title,genres)
- **ratings** (userId,movieId,rating)
- **tags** (userId,movieId,tag)
- **genome-scores** (movieId,tagId,relevance)
- **genome-tags** (tagId,tag)

The scope of the project is to implement a set of 10 queries in order to extract information about both users and movies. In these queries, we perform aggregation operations over the files of the dataset to uncover the analytical value of each query.

The project is devived into two parts. In the first part, we leverage Spark RDDs operations which are categorized in **Transformations** and **Actions**.

Tranformation are lazy operations that define a new RDD. We have two categories :

- **Narrow:** Each partition of the parent RDD is used by at most one partition of the child RDD. Narrow operations used in the project are : map, flatmap, filter, keyby.
- **Wide:** Multiple child RDD partitions may depend on a single parent RDD partition. The Wide operations used in the project are : join, groupby.

Actions are operations that trigger the execution of all previously defined transformations on an RDD and return a result to the driver program or write data to external storage.We have two categories :

- **Distributed:** These operations occur across the cluster.The Distributed operations used in project are: saveAsTextFile.
- **Driver:** The result of operations must fit in driver JVM. Driver operations used in project are: max, sum.

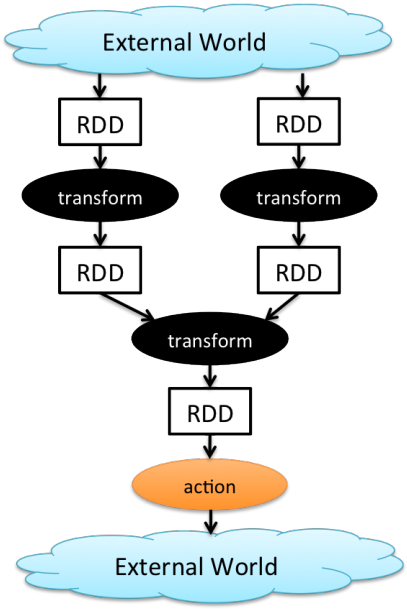


Figure 1: Transformations and Actions

In the second part, we leverage Spark Dataframe (DFs) operations to implement the corresponding queries. The Dataframe is similar to an RDD, but consists of rows, columns, and a schema.

Finally, we use the **Hadoop Distributed File System (HDFS)** to read the dataset as input. Also, after the necessary transformations and aggregations, we store a sample of the output for each query in a separate file.

Query's Implementation, Jobs and Stages

Part A - Using Spark RDDs

Query 2: Tag Dominance per Genre — Most Used Tags with Ratings

The goal of this query is to find, for each movie genre, the most frequently used tag, and then calculate the average rating for the movies with this tag.

Next, we commit a `join()` between movieTag and movieGenres, using the movieId as the key.Then, with the `.map()` and the `reduceByKey()` we count the occurrences for each (genre, tag).We continue by creating pairs of ((genre)(tag,counter), group them by key using `groupByKey()` and sort them to find the most used tag in descending order with `.mapValues()`.

Wide operations uses:

- `join()`
- `reduceByKey()`
- `groupByKey()`

Narrow operations uses:

- `map()`
- `mapValues()`
- `sortBy()`

```
val genretag = movieTag.join(movieGenres).map(x => ((x._2._2, x._2._1), 1)) // (genre,tag) , 1)
  .reduceByKey(_ + _) //  (genre,tag) counter)
  .map(x => (x._1._1, (x._1._2, x._2)))  //  (genre)(tag,counter).
  .groupByKey()
  .mapValues(_.toList.sortBy(_._2).reverse.head) // find the max counter aka most used tag
  .sortBy(_._2._2, ascending = false)            //return genres based on the most frequent tag
                                                 //(genre)(tag, maxcount)
```

Figure 2: Code

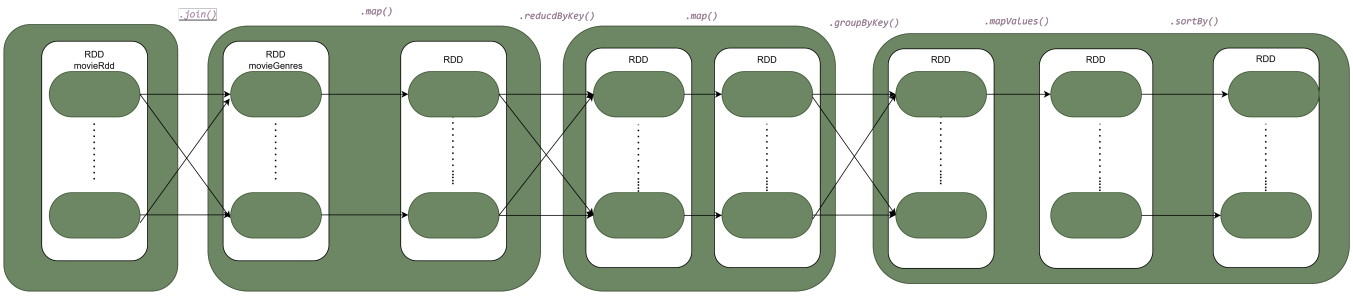


Figure 3: Dag

Next we make the same `join` as previous in order to keep pairs of (genre, id) to use it for the `join` in the final aggregation.

Then we `join` the two previous Rdds in order to keep pairs of (genre, tag) with the most frequently used tag per genre.In the next step, we perform another `join()` using ratings RDD and with the `map()` operation we keep pairs of the form: ((genre, tag), (rating, 1)).With the `reduceByKey()` we create (genre,tag) (sum, count) pairs to use it in `mapValues()` in order to compute the average rating. Finally, with the `mapValues()` operation we obtain pairs (genre, tag, avg) and we sort the result by the average rating value in ascending order using `sortBy()`.

Wide operations uses:

- `join`
- `reduceByKey()`

Narrow operations uses:

- `map()`
- `mapValues()`
- `sortBy()`

```
val avgTopTagMoviesRating = genretag.join(tagGenreId).map(x => (x._2._2, x._2._1._1)) //
  .join(movieRating).map{
    x => ((x._1, x._2._1), (x._2._2,1))}  // (movieid, tag) (rating,1)
  .reduceByKey((x,y) => ((x._1 + y._1), (x._2 + y._2))) //) (movieid,tag) (sum, count)
  .mapValues(x => x._1 / x._2)
  .map( x => (x._1._1, x._1._2, x._2)) // (id, tag, avg)
  .sortBy(_._3, ascending = false)
```

Figure 4: Code

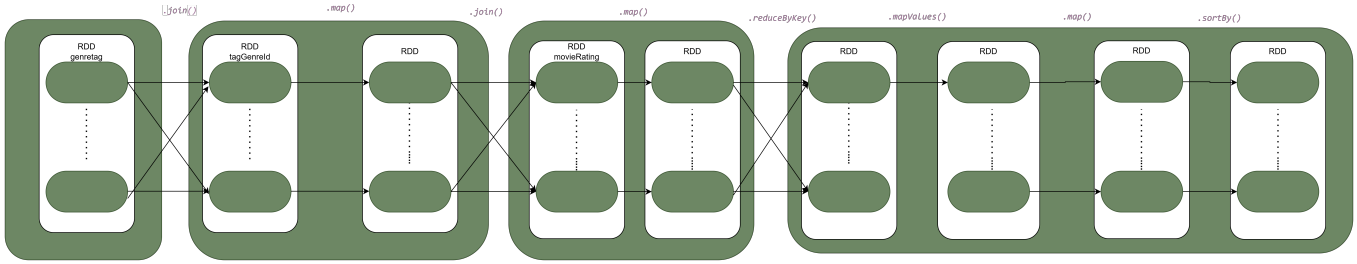


Figure 5: Final Dag

At the end of this query we store a sample of the output in the HDFS with the name of the file to be Query2SampleOutput.

Then we extract the JAR file and submitted it to Spark. From the History Server we can detect 7 jobs for the Query 2.

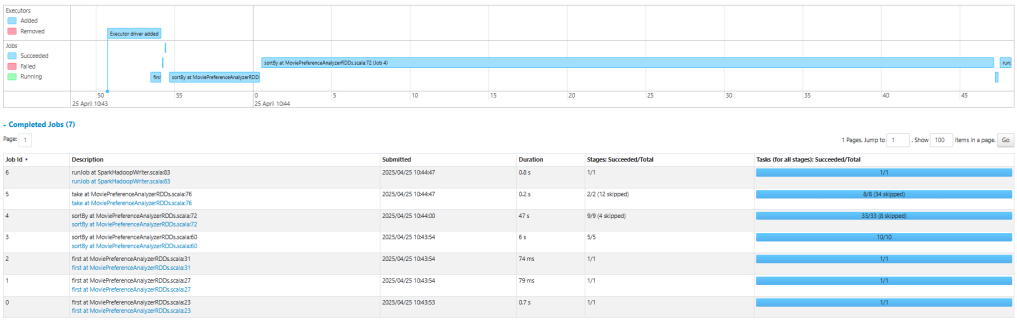


Figure 6: Jobs of Query2

The first three jobs are for the `.first()` action that was applied to each of the input files in order to keep the header.

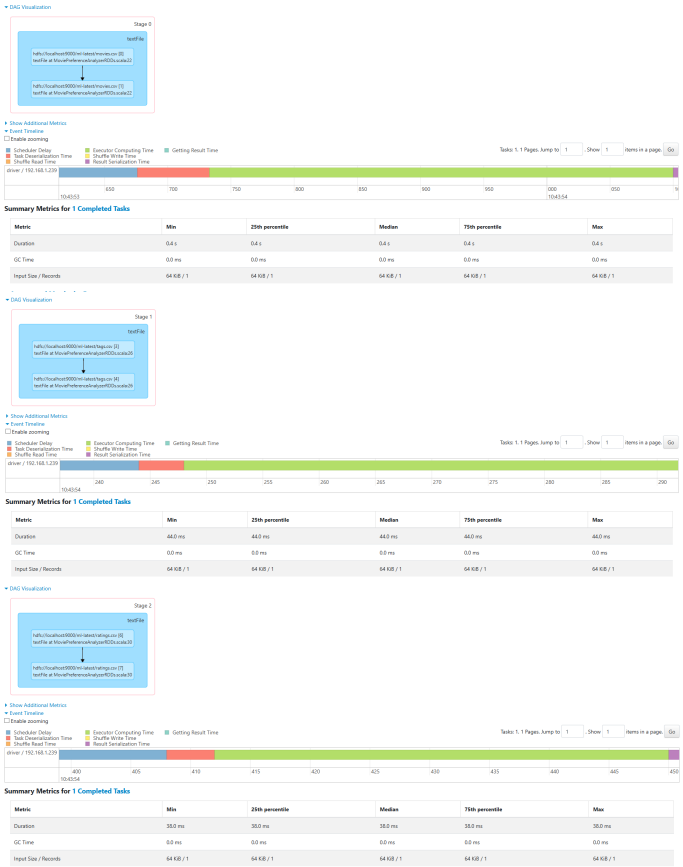


Figure 7: Jobs 0, 1, and 2

Each of these jobs have only one task.

We continue wiht the job 3, that created by `.sortBy()` in the genretag Rdd. This job is made up of 5 stages.Each stage was created due to a shuffle

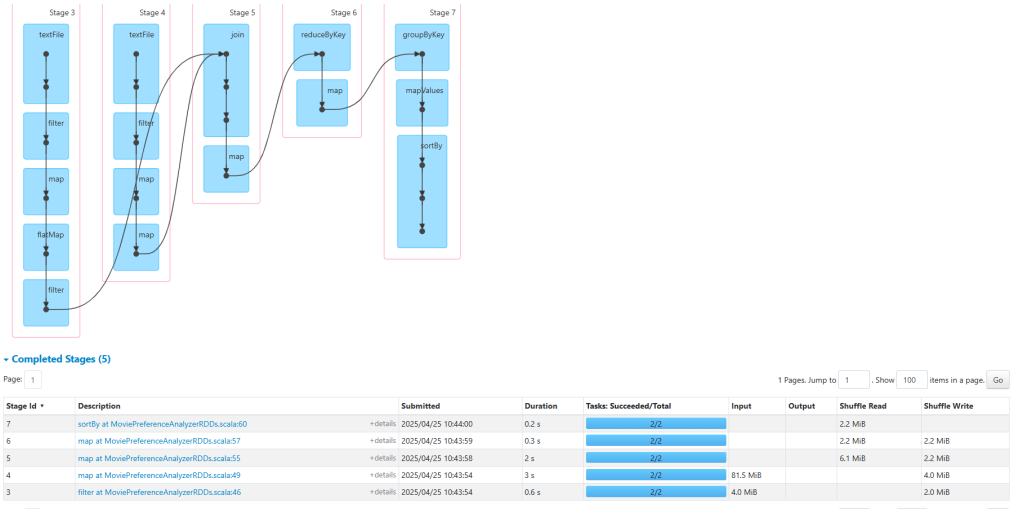
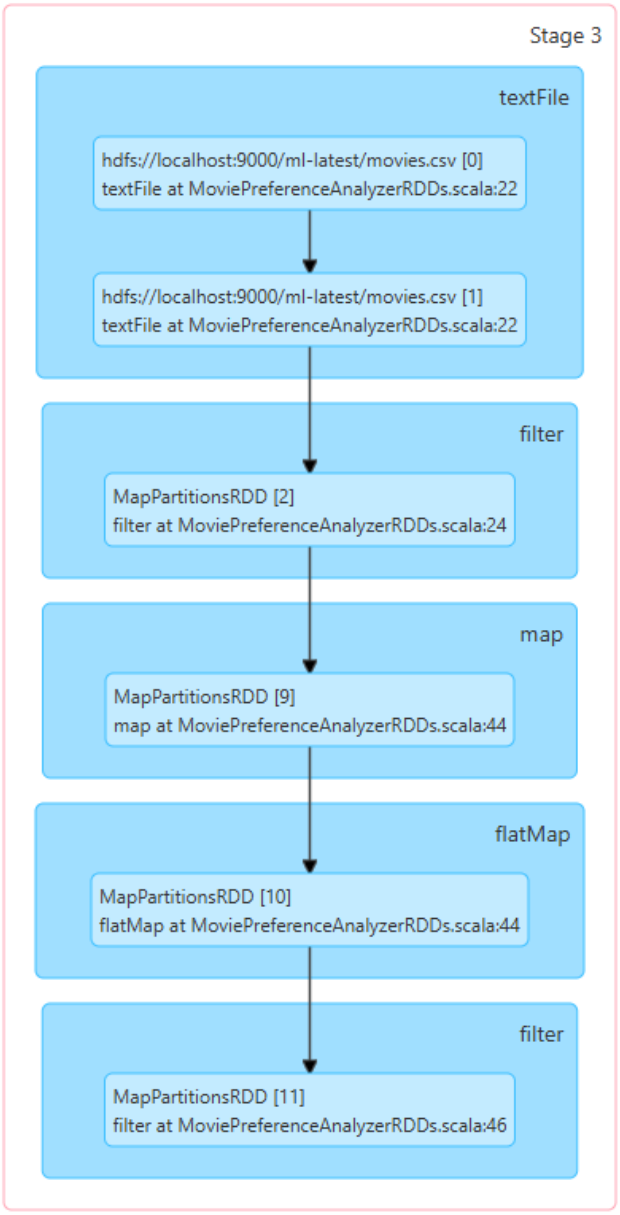


Figure 8: Stages of Job 3

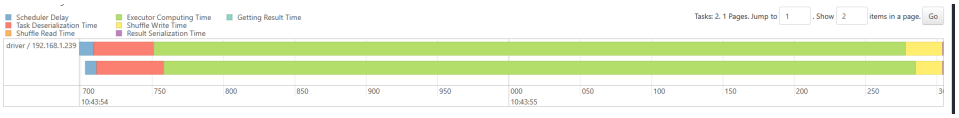
In the first stage of Job 3 (stage 3), we have narrow operations that was applied in the text file movies.csv in order to remove the header, seperate the fields of the file, split the genres by using the delimiter "|" and clean from entries such as (no genres listed).

Narrow operations were used:

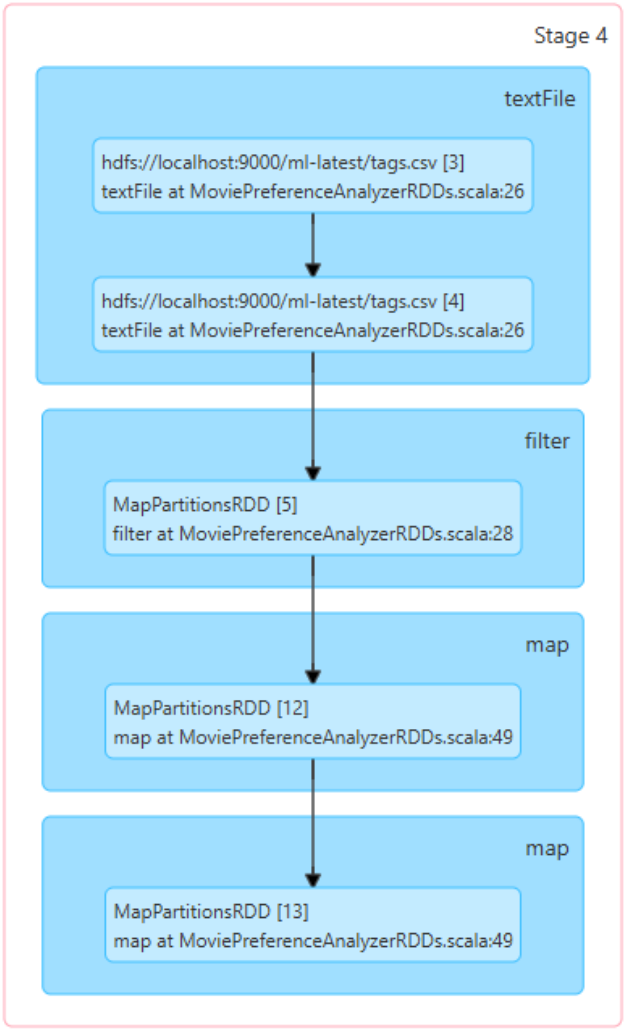
- `map()`
- `flatMap()`
- `filter()`



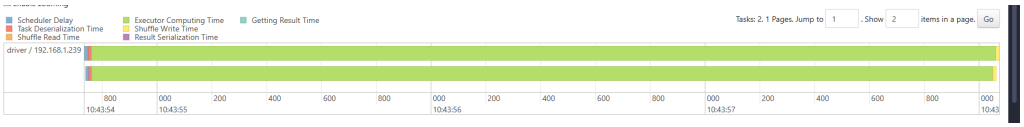
While all the transformations are part of Narrow operations, we have no shuffle here. Also because the input RDD is divided into two partitions,the stage 3 is executed with 2 tasks. Thus, the degree of parallelism achieved is 2.



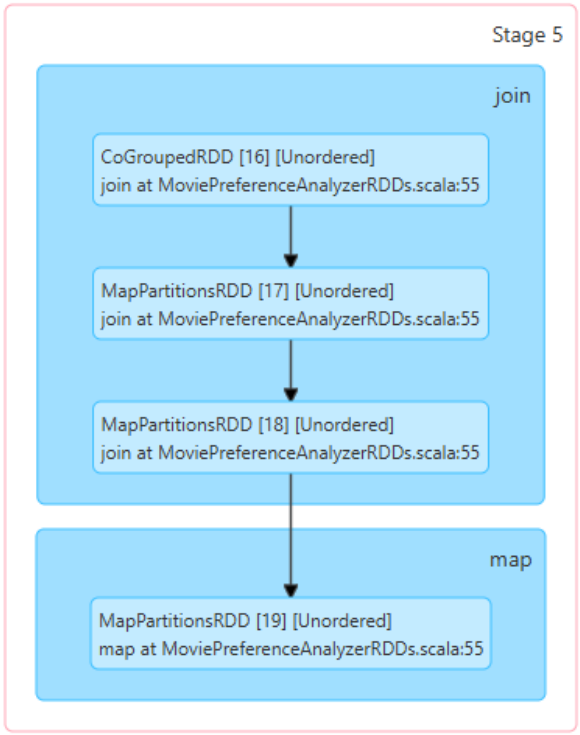
In the stage 4, we have the same narrow operations as previous `map()`, `filter()` for the file tags.csv and we make pairs of (movieId, Tag).



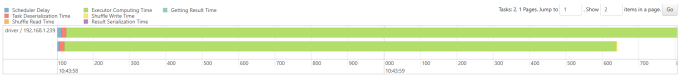
Due to narrow operations, no shuffle occurs in this stage. The input Rdd is divided into two partitions and so the stage 4 is executed with 2 tasks. Therefore, the degree of parallelism achieved is again 2.



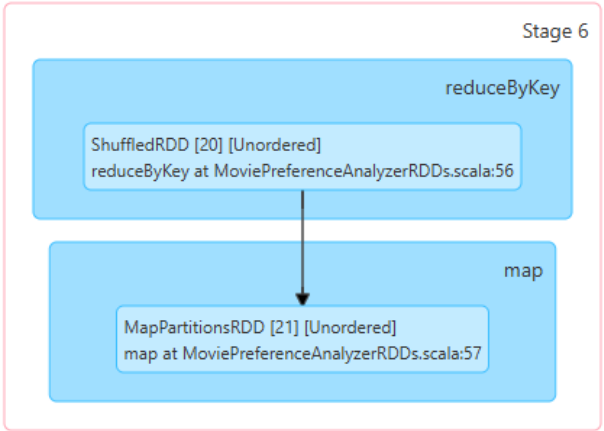
In the stage 5, we have a wide operation `join()` between the two previous files. Also we use the narrow operation `map()` to restructure the output.



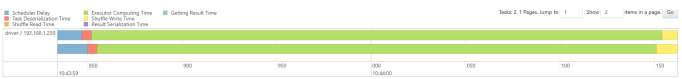
Because this is a wide opperation, a shuffle occurs. The stage is divided in 2 tasks and the parallelism is stable at 2.



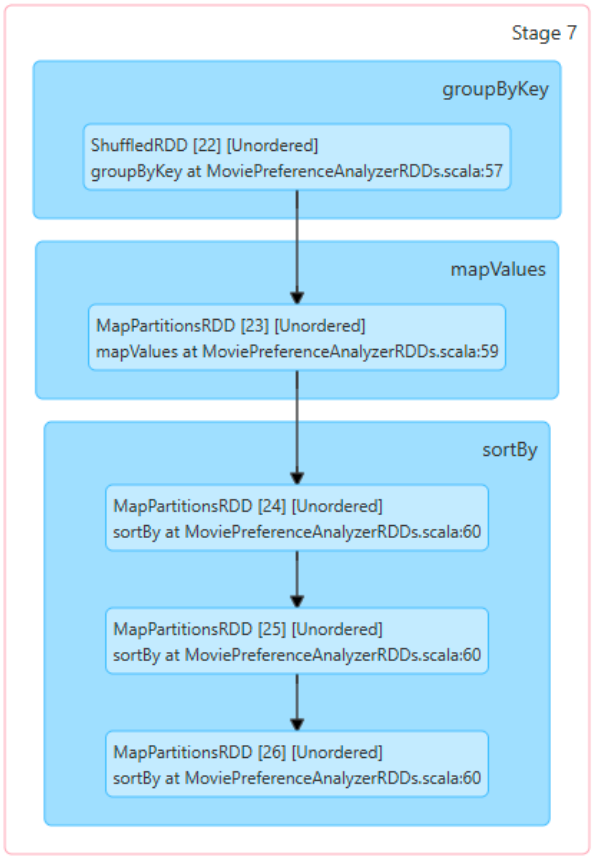
In the stage 6, we have a wide operation `reduceByKey()` in order to get the counter of the pairs. Also we use the narrow operation `map()` to restructure the output.



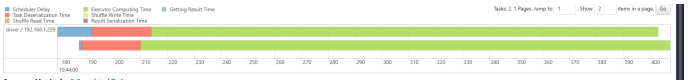
Because this is a wide opperation, a shuffle occurs. The stage is divided in 2 tasks and the parallelism is stable again at 2.



In final stage 7, we have wide operation `groupByKey()`, to group by the genre and the narrow operations `mapValues()` and `sortBy()` in order to sort the output by frequency of the tag.



Because of the wide operation, in this stage we observe a shuffling in the data.The stage 7 is executed in 2 tasks. Thus, the degree of parallelism achieved is again 2.



After, we have the job 4 with 9 completed stages and 4 skipped ones.This job is triggered by the `sortBy()` operation.

The skipped tasks are due to the Spark's optimizer, because they had been computed in previous jobs.

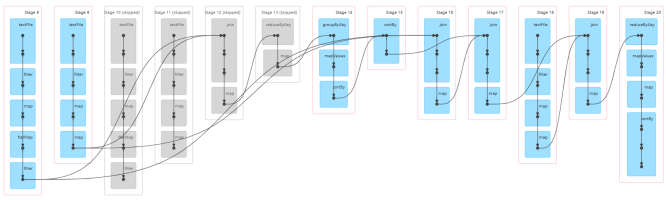


Figure 9: Dag for Job 4

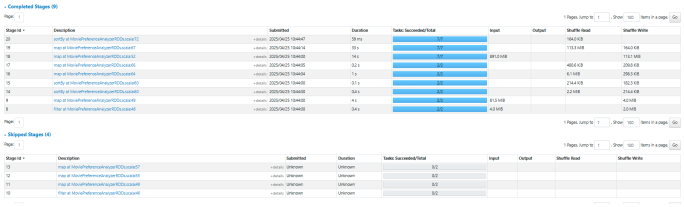


Figure 10: Tasks of Job 4

The first two stages (stage 8 and stage 9) use narrow operations on text files movies.csv and tags.csv to creat the pairs that we need to continue.

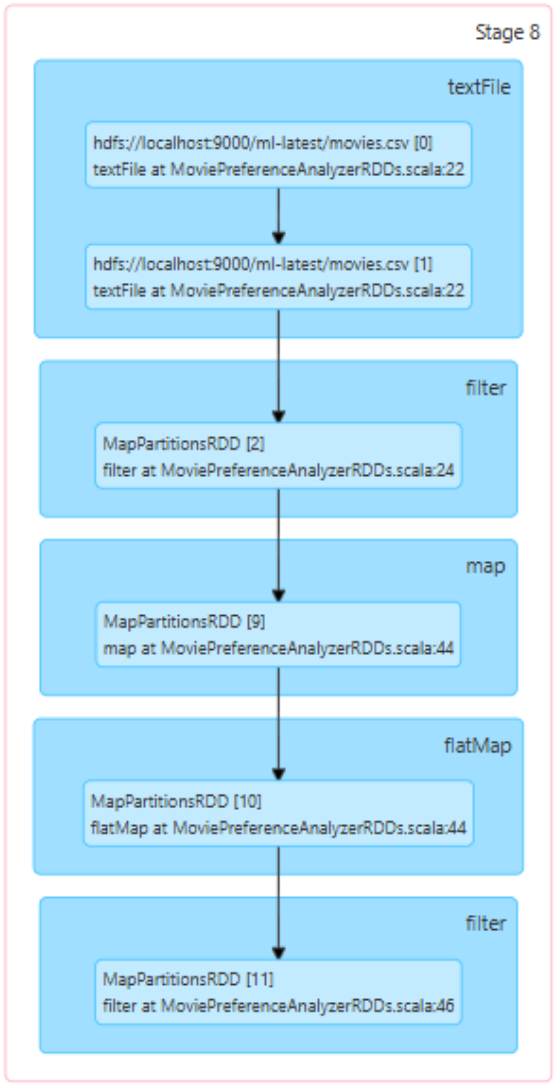


Figure 11: Stage 8

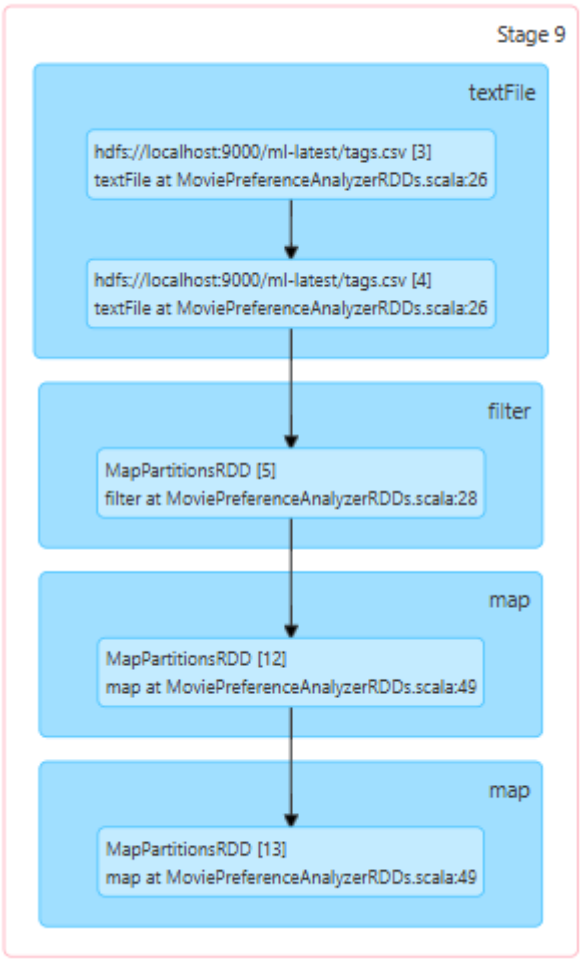


Figure 12: Stage 9:

In the others stages include wide operations, such as in the stage 14 we use `grouByKey()` which group the pairs using the genre as the key.

In stage 15, the `sortBy()` is used to sort the result by the frequency of the tag.

In stage 16 and 17, the `join()` operation is used to combine the previous result using the genre as key.

In stage 19, the result from the last join is joined with the ratings.csv.

Finally in stage 20, we count the number of ratings per movieId by using the wide operation `reduceByKey()` to leverage it for computing the average.The `sortBy()` is used to sort the result in descending order.

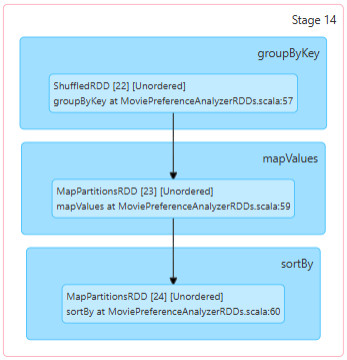


Figure 13: Stage 14

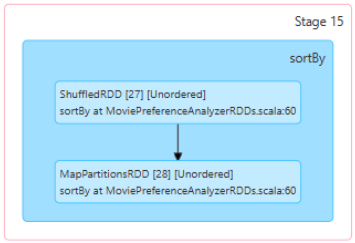


Figure 14: Stage 15

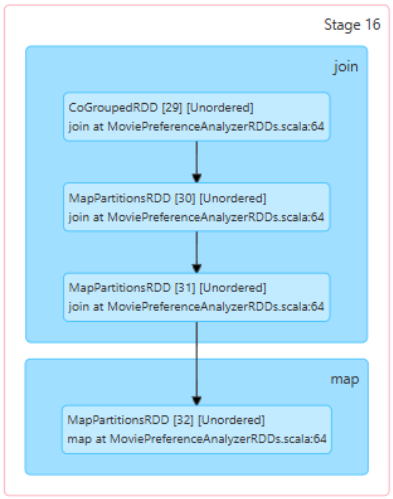


Figure 15: Stage 16

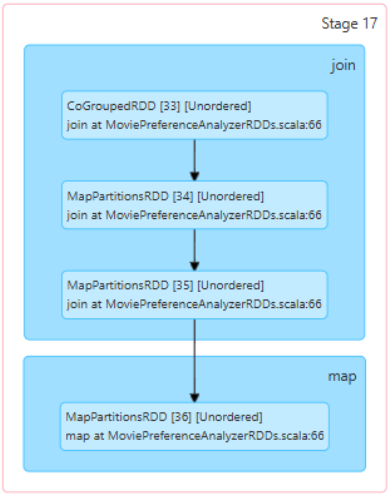


Figure 16: Stage 17

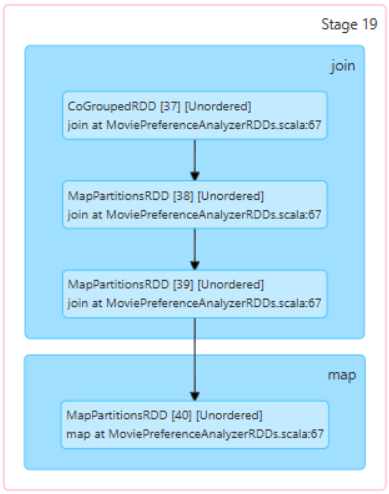


Figure 17: Stage 19

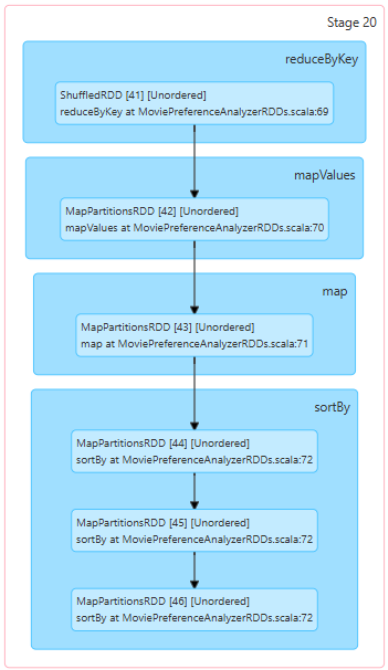


Figure 18: Stage 20

All wide operations that mentioned above cause shuffles in the data which is the reason that leading to the creation of new stages.

In addition, in all these stages there are also and narrow operations.The `map()` is used to reformat the key values pairs to prepare for joining, `mapValues()` to keep a counter for the pairs and `sortBy()` to sort the result.

In the stage 18, we use narrow operations like `filter()` and `map()` on the file ratings.csv to creat pairs of (movieId, rating).

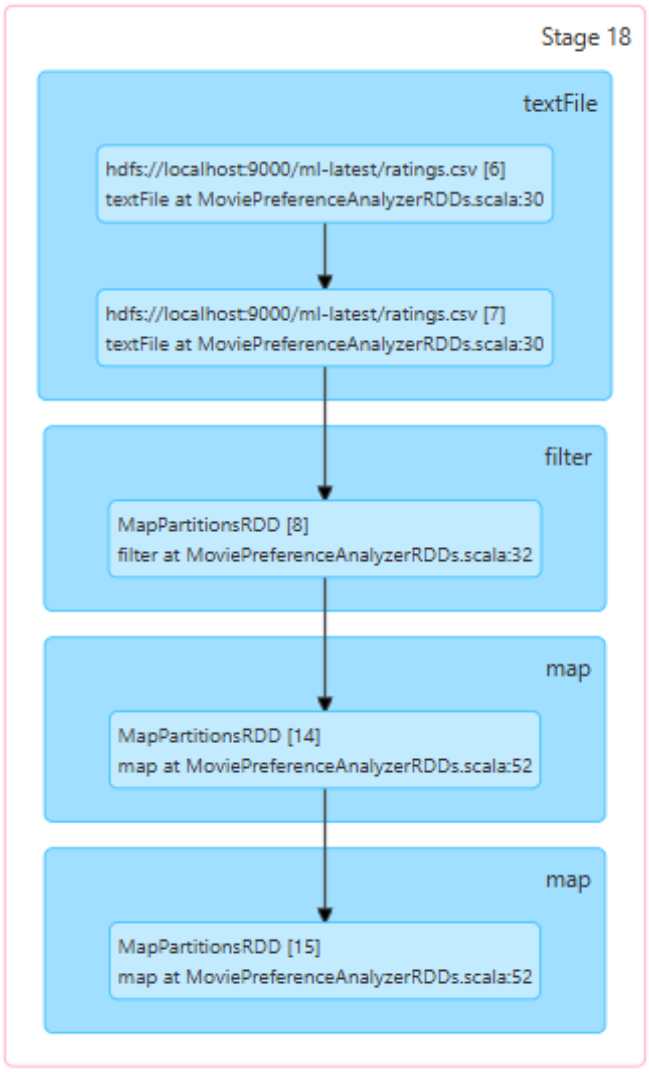


Figure 19: Stage 18

Until stage 17, where the input files are movies.csv and tags.csv, the number of partition is 2 and so the tasks for each stage are 2. Thus, the degree of parallelism achieved is 2

Although, in the next stages up to 20, where the data isvshuffled with the input file ratings.csv the partition increases in 7 and each stage is executed in 7 tasks. So the degree of parallelism is now 7.

Figure 20: Stages 8–16

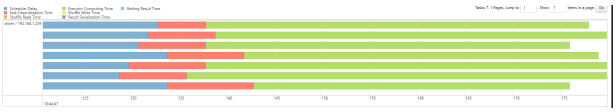


Figure 21: Stages 17–20

The job 5 is created by the action `take()` with 2 completed stages and 12 skipped ones.

The skipped tasks are due to the Spark's optimizer, because they had been computed in previous jobs.

Figure 22: Job 5

In the first stage with Id 33, we have the wide operation `reduceByKey` to count the number of ratings per movieId. Then, with narrow operations such as `mapValues()` and `map()` compute the average rating.In the end, we use the `sortBy()` to sort the result in ascending order.
In the final stage 34 the wide operation `sortBy()` is used to sort the result by the average value in ascending order.

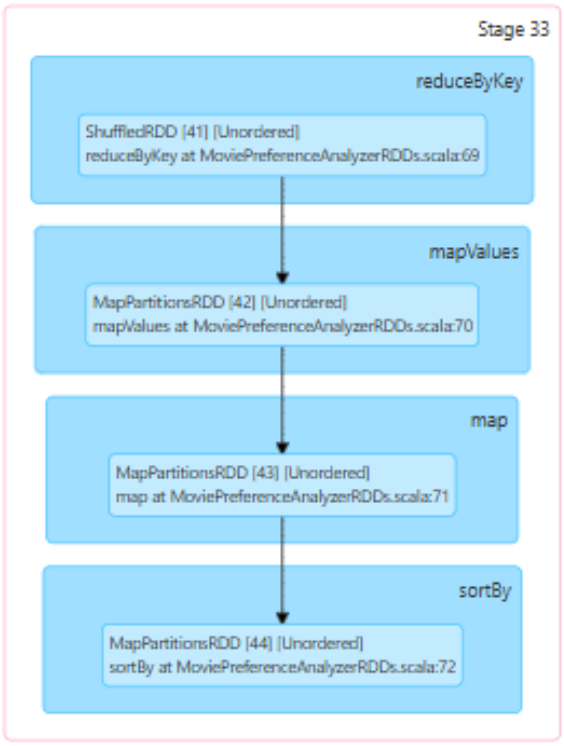


Figure 23: Stage 33

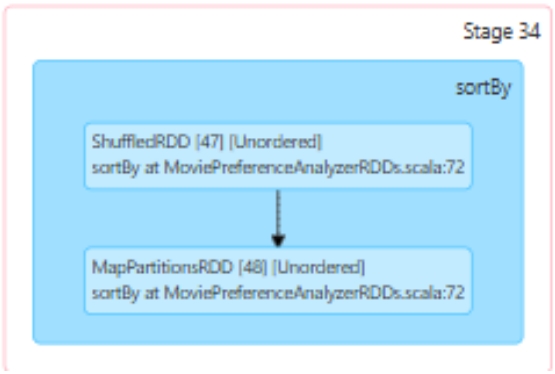


Figure 24: Stage 34

The wide operations `reduceByKey()` and `sortBy()` cause shuffles in the data leading to the creation of these two stages.

In the stage 33, the partition is 7 and so the stages is executed in 7 tasks.On the other hand in the stage 34, the partition drops to 2 and the stages is executed in 2 tasks. Therefore, the degree of parallelism is 7 and then reduced to 2.

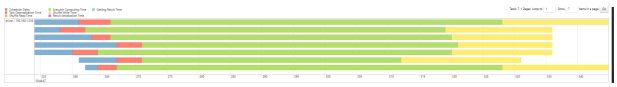


Figure 25: Stage 33

Figure 26: Stage 34

Finally, the job 6 is created by `saveAsTextFile()` which is used to save a sample of the result in HDFS and have one stage 35.

We use `parallelize()` to create RDD, `coalesce()` to reduce the partitions to 1 and `saveAsTextFile()` to store the result in the HDFS.

This stage is executed in a single task and then we don't have any parallelism.

Query 4: Sentiment Estimation — Tags Associated with Ratings

The goal of this query is to compute the average user rating for each user-assigned tag from tags.csv .

We start by transformating the files.First, with the narrow operation `split()` we divide each line of the Rdd in 3 parts. Then using the narrow operation `map()` we create pairs of the form ((userId, movieId), tag).

```
val tagUser = tagRdd.map(_.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)")).map(x =>
  ((x(0),x(1)), x(2))) // ((userId, movieId), tag)
```

Figure 27: Code

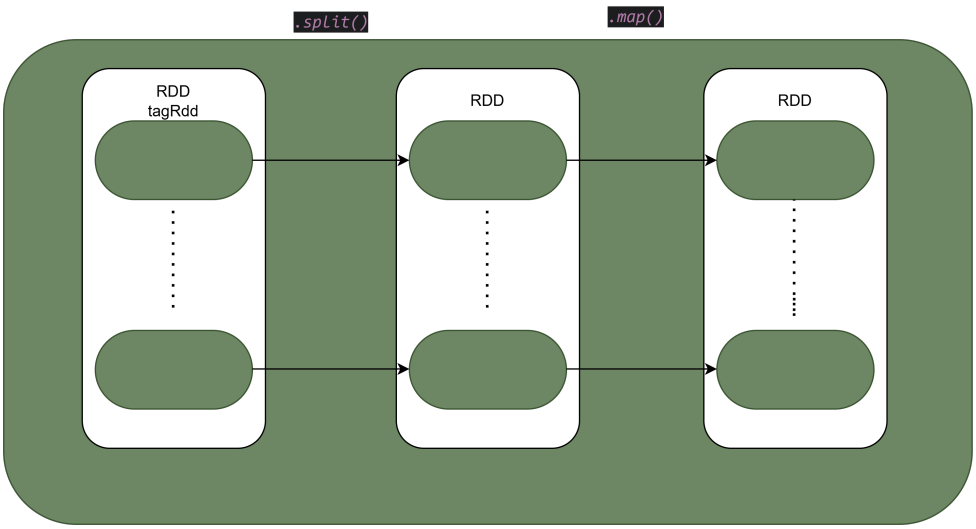


Figure 28: tagUser

Secondly, we work on the ratingsRdd in the same way by using the `split()` operation to separate the file in 3 parts and continue by making key value pairs such as ((userId, movieId), rating) with `map()` operation.

```
val ratingUser = ratingsRdd.map(_.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)")).map( x =>
  ((x(0), x(1)), x(2).toDouble)) //((userId, movieId), rating)
```

Figure 29: Code

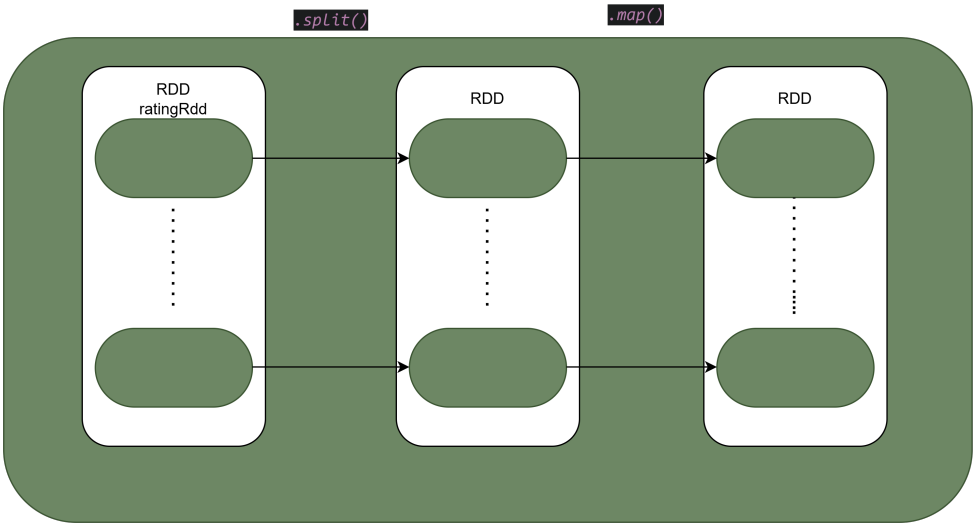


Figure 30: ratingUser

Next, we commit a `join()` between the two previously RDDs using (userId, movieId) as the key.Then using `map()` we set the tag as the key and define as value the (rating, 1). We doing this in order to compute both the sum of the total ratings and the counter using `reduceByKey()`.We continue by determining the average rating for each tag using `mapValues()` operation.In the end of this code, we sort the result by average rating in ascending order with the `sortBy()`.

Wide operations uses:

- `join()`
- `reduceByKey()`

Narrow operations uses:

- `map()`
- `mapValues()`
- `sortBy()`

```
val avgTagRating = tagUser.join(ratingUser).map(x => ((x._2._1), (x._2._2, 1)))
  .reduceByKey((x,y) => ((x._1 + y._1), (x._2 + y._2))) // (tag) (sum, count)
  .mapValues(x => x._1 / x._2)
  .sortBy(_._2, ascending = false)
```

Figure 31: Code

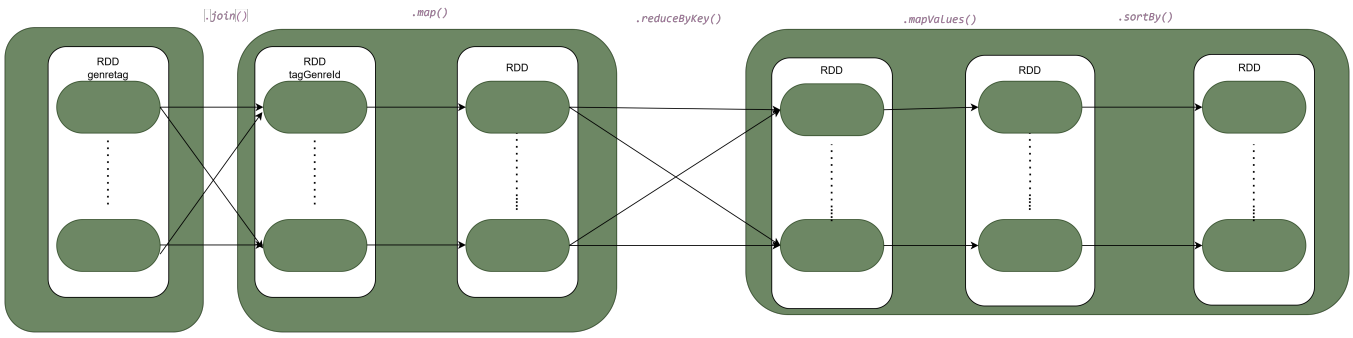


Figure 32: Dag (avgTagRating)

At the end of this query we store a sample of the output in the HDFS with the name of the file to be Query4SampleOutput.
Then we extract the JAR file and we submitted it to Spark. From the HistoryServer, we observe 6 jobs for the Query 4.

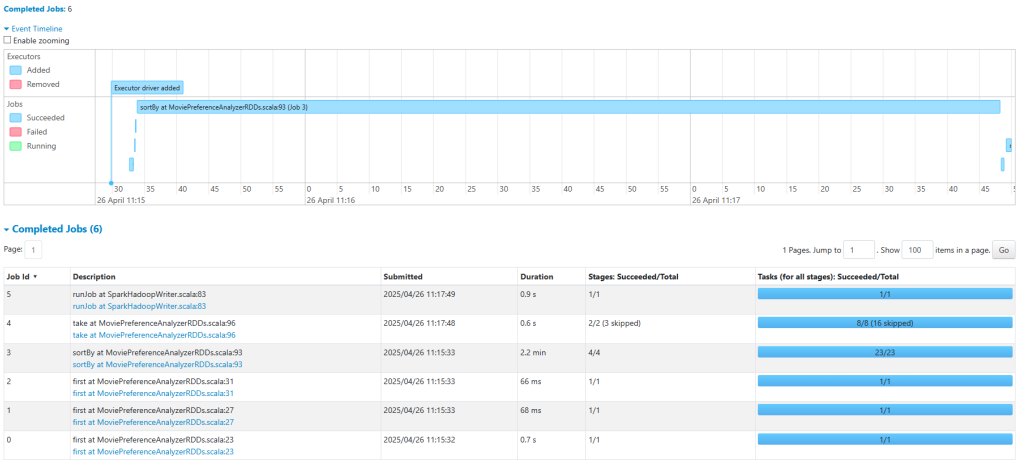


Figure 33: Jobs of Query 4

Again the stages from 0 to 3 is for the `first()` action that was applied to each of the input files in order to keep the header.

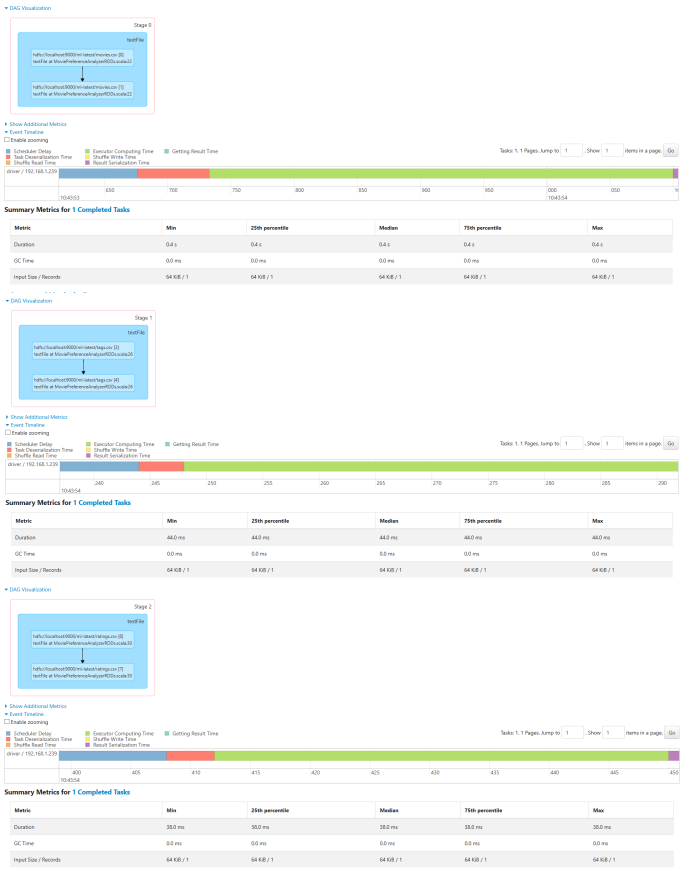


Figure 34: Jobs 0, 1, and 2

Each of these stages have only a single task, thus, no parallelism is achieved

We continue with the job 3, which triggered by the `sortBy()`.This job consist of 4 stages.

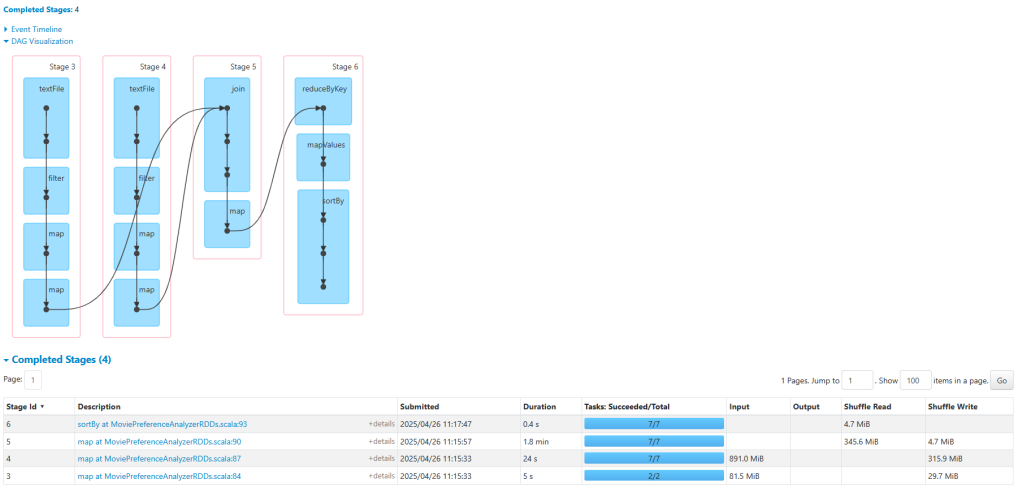


Figure 35: Job 3

In the first two stages (Stage 3 and Stage 4) are used the narrow operations such as `map()` and `filter()` on the tagRdd and the ratingRdd to create the pairs with (userId, movieId) as the key, which are required to continue.

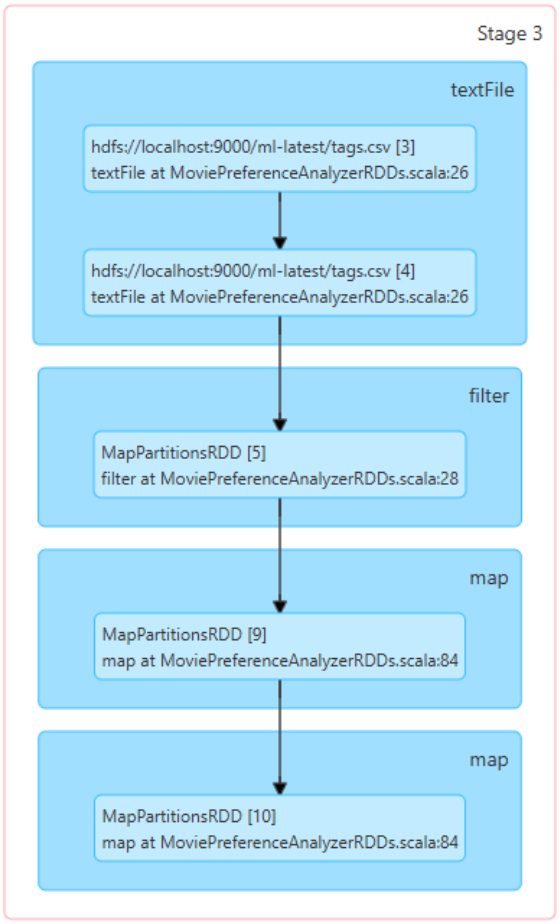


Figure 36: Stage 3

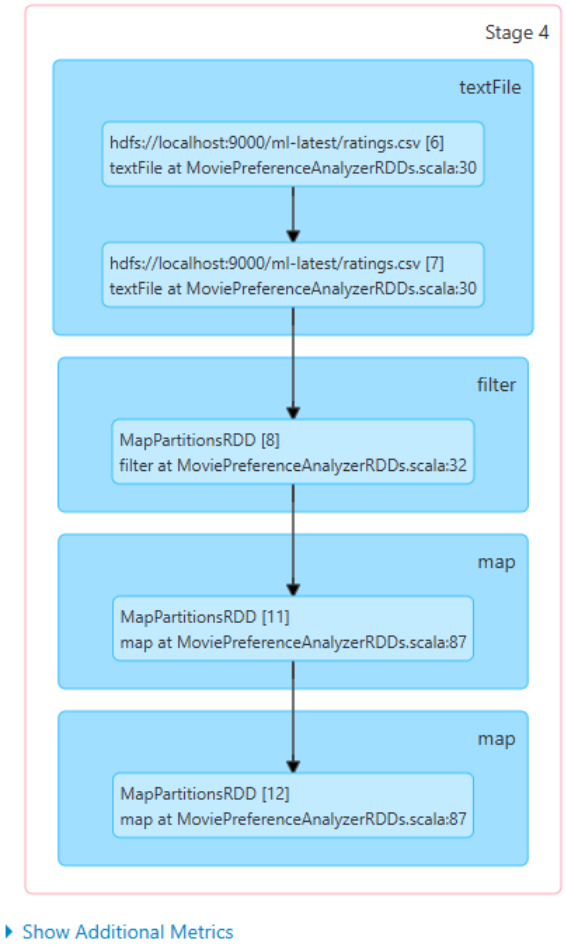


Figure 37: Stage 4

In the stage 5, `join()` is used to compine the output of the two previous stages using (userId, movieId) as the key.

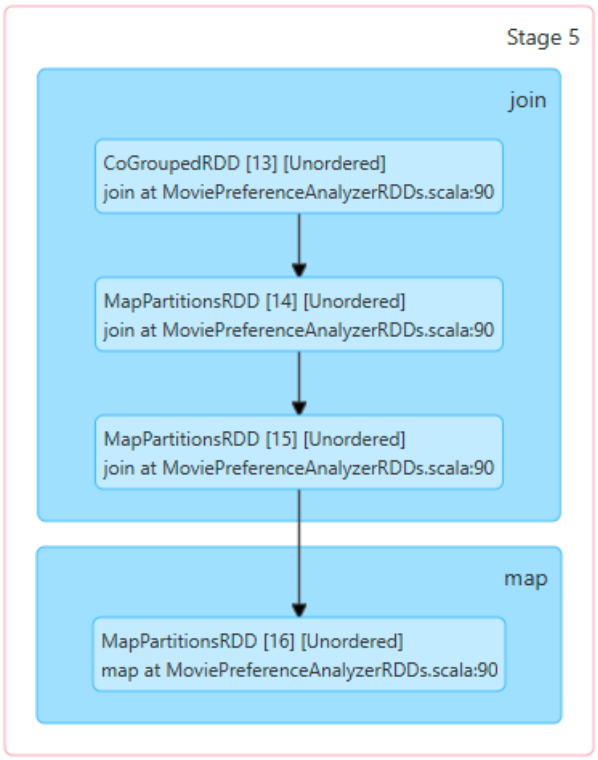


Figure 38: Satge 5

In the final stage 6, we count the number of ratings per movieId by using the wide operation `reduceByKey()` to leverage it for computing the average rating using the `mapValues()`.The `sortBy()` is used to sort the result in descending order.

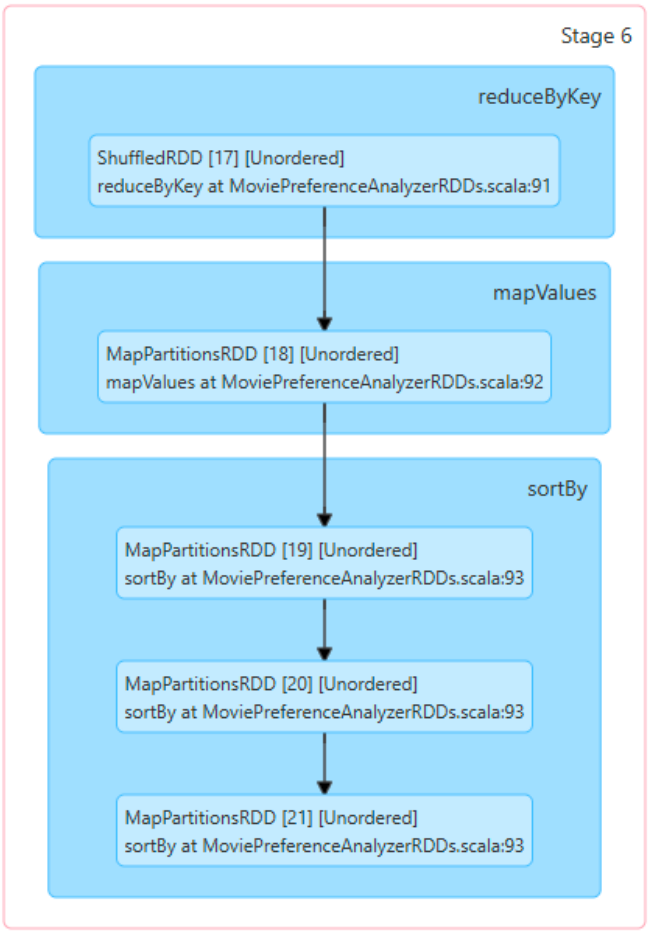


Figure 39: Stage 6

The wide operations `join()` and `reduceByKey()` cause shuffles in the data which is the reason that leading to the creation of new stages.

The first stage is executed with two tasks, leading to a degree of parallelism which is 2.In the following stages, the partition is 7 and so the stages is executed in 7 tasks. Therefore, the degree of parallelism is 7

Figure 40: Stage 3

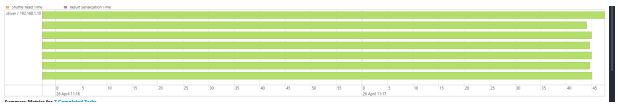


Figure 41: Stages 4 to 6

The job 5 is created by the action take() with 2 completed stages and 3 skipped ones.
The skipped tasks are due to the Spark's optimizer, because they had been computed in previous jobs.

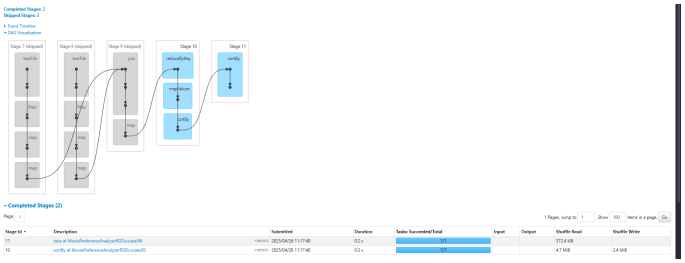


Figure 42: Enter Caption

In the stage 10, we have the wide operation `reduceByKey()`, which is the cause of the creation of the stage. The others narrow operations ,*mapValues()* and `sortBy()`, execute the same process that previously describe in the stage 6 of job 3.
In the stage 11, the wide operation `sortBy()` is executed due to the operation `take()` in order to return the sample.

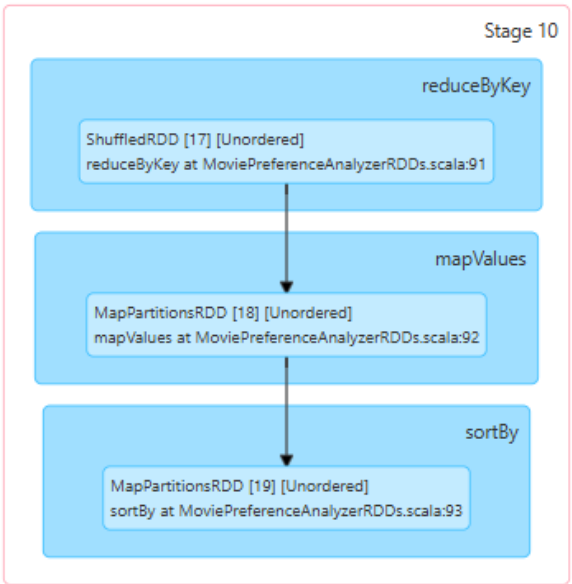


Figure 43: Stage 10

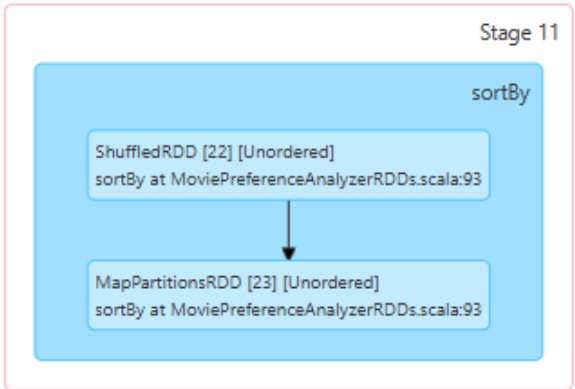


Figure 44: Stage 11

he first stage 10 is executed in 7 tasks so we achieve 7 as a degree of parallelism. In the last stage, the number of partition drops to 2 and the stage is executed in 2 tasks. Thus, the degree of parallelism is now 2.

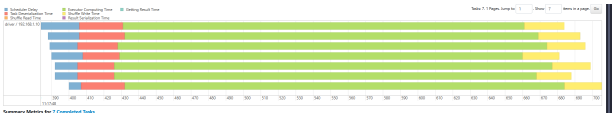


Figure 45: Stage 10

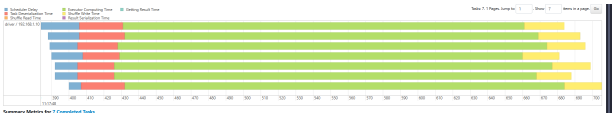


Figure 46: Stage 11

Finally, the job 5 is created by `saveAsTextFile()` which is used to save a sample of the result in HDFS and have one stage 12.

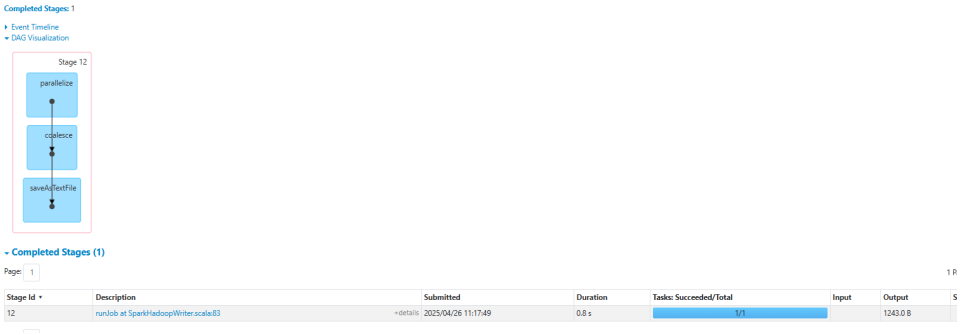


Figure 47: Job 5

We use `parallelize()` to create RDD, `coalesce()` to reduce the partitions to 1 and `saveAsTextFile()` to store the result in the HDFS.

This stage is executed in a single task and then we don't have any parallelism.

Part B - Using Spark DFs

Query 6: Skyline Query — Non-Dominated Movies in Multiple Dimensions

The goal of this query is to find movies that are not dominated in :

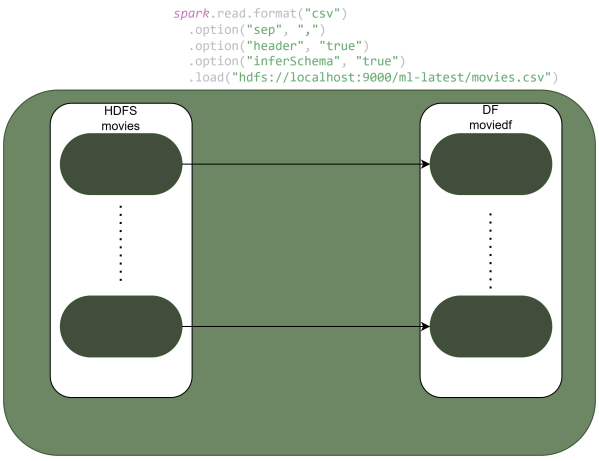
- average rating
- rating count
- average tag relevance

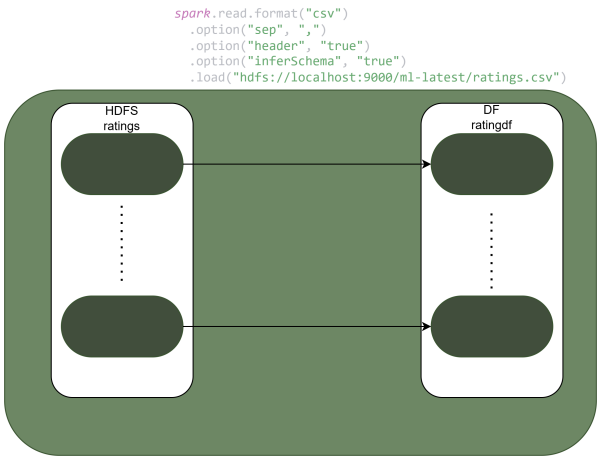
We begin by loading the files from the **HDFS**.The format of the files are set to CSV, the seperator is set to ",", the header is enabled and inferSchema is activated to detect the type of each column.

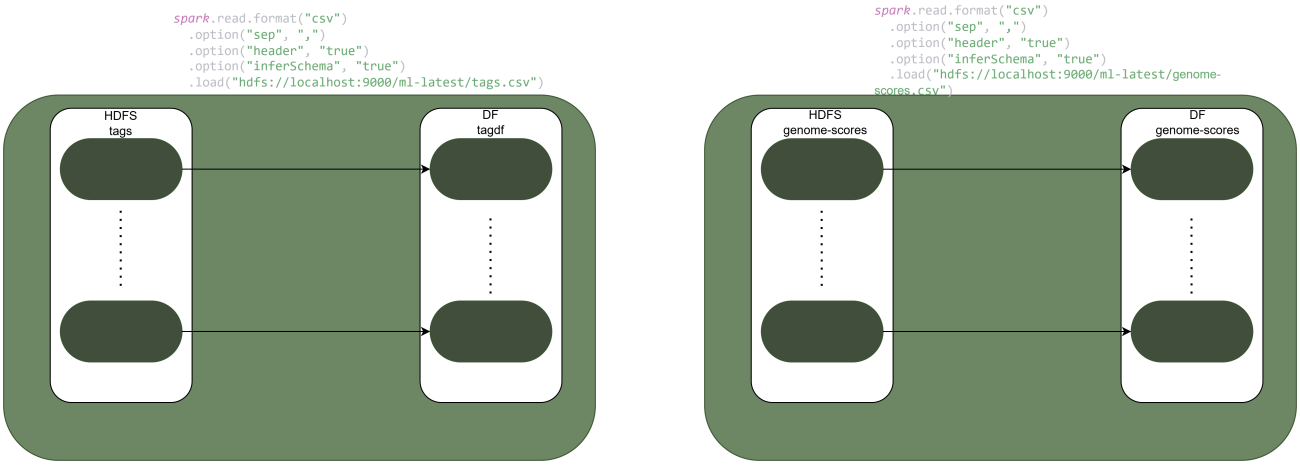
```scala
val moviedf = spark.read.format("csv")
  .option("sep", ",")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("hdfs://localhost:9000/ml-latest/movies.csv")


val ratingsdf = spark.read.format("csv")
  .option("sep", ",")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("hdfs://localhost:9000/ml-latest/ratings.csv")


val tagsdf = spark.read.format("csv")
  .option("sep", ",")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("hdfs://localhost:9000/ml-latest/tags.csv")

val genscoresdf = spark.read.format("csv")
  .option("sep", ",")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("hdfs://localhost:9000/ml-latest/genome-scores.csv")
```

Figure 48: Loading the Input Files

The logical plan for this code section is :







Next, we group the data from the ratingdf by the movieId column using the wide operation `groupBy()`.Then, we compute two aggregations functions, the average rating and the count of ratings for each movie.

```
val ratingStats = ratingsdf.groupBy(col("movieId"))
  .agg(avg("rating").alias("avgRating"),
    count("rating").alias("ratingCount")
  )
```

Figure 49: code

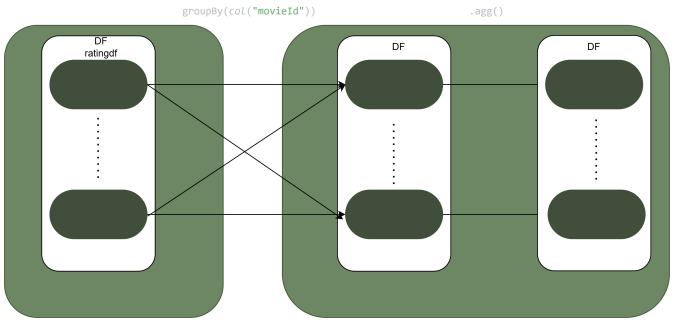


Figure 50: ratingStats

Similarly, we apply the same operation on the genscoresdf.

```
val avgTagRel = genscoresdf.groupBy(col("movieId"))
  .agg(avg("relevance").alias("avgTagRelevance"))
```

Figure 51: code

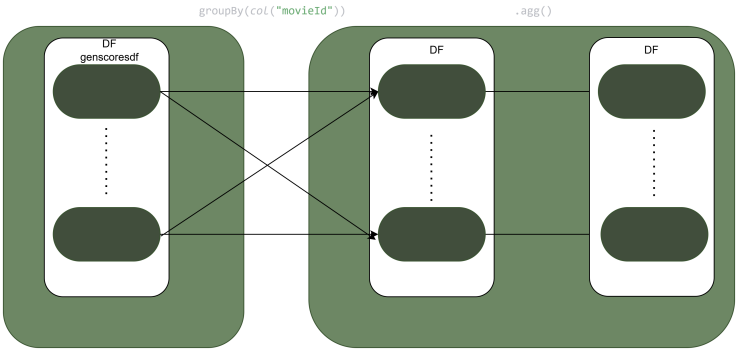


Figure 52: genscoresdf

Afterwards, we combine the two previous DataFrames using the wide operations `join()` with the movieId as the key.o avoid duplicates moviedId columns, we use the narrow operation `drop()`.

```
val Q6data = ratingStats.join(avgTagRel, ratingStats("movieId") === avgTagRel("movieId")).drop(avgTagRel("movieId"))
```

Figure 53: code

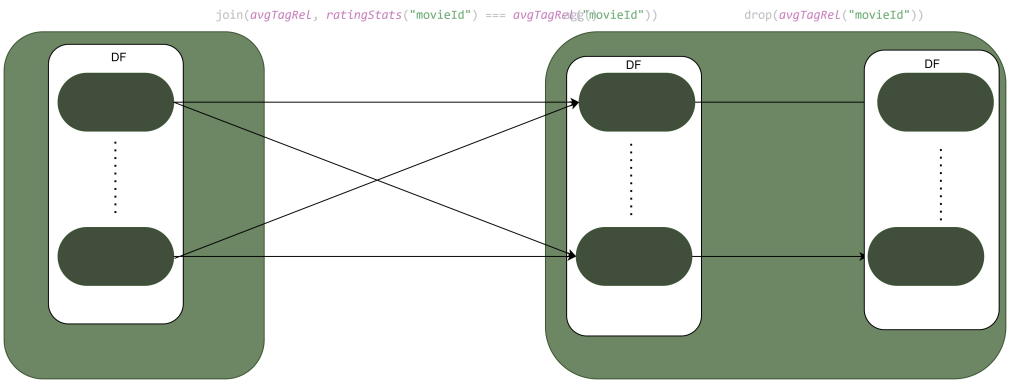


Figure 54: Q5Data

In this code section, we complete the skyline query. We perform a self-join between the data, in order to keep 2 copies of the data. By comparing them, we find which movies are not dominated by others in :

- average rating
- rating count
- average tag relevance

We use the `left_andi` join to keep only the movies of the left DataFrame that are not dominated by any movie from the right DataFrame.

```
val skylineRatGenSc = Q6data.alias("t1").join(Q6data.alias("t2"),
  col("t1.avgRating") >= col("t2.avgRating") &&
    col("t1.ratingCount") >= col("t2.ratingCount") &&
    col("t1.avgTagRelevance") >= col("t2.avgTagRelevance") &&
    (
      col("t2.avgRating") > col("t1.avgRating") ||
        col("t2.ratingCount") > col("t1.ratingCount") ||
        col("t2.avgTagRelevance") > col("t1.avgTagRelevance")
      )
  , "left_anti"
```

Figure 55: Code

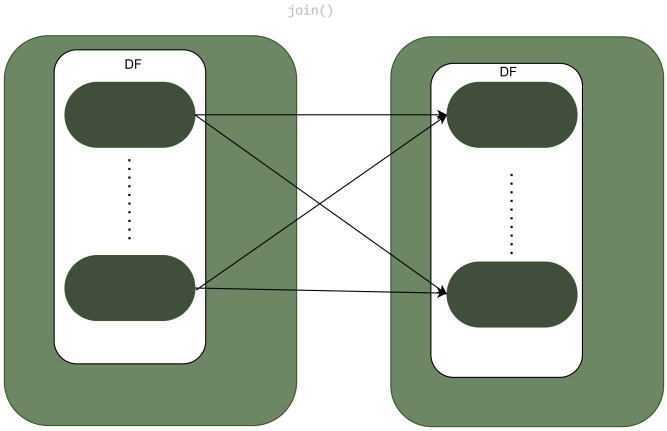


Figure 56: skylineRatGenSc

At the end of this query we store a sample of the output in the HDFS with the name of the file to be Query6SampleOutput.

Then, we extract the JAR file and submitted it to Spark. From the History Server, we can detect 14 jobs for the query 6.

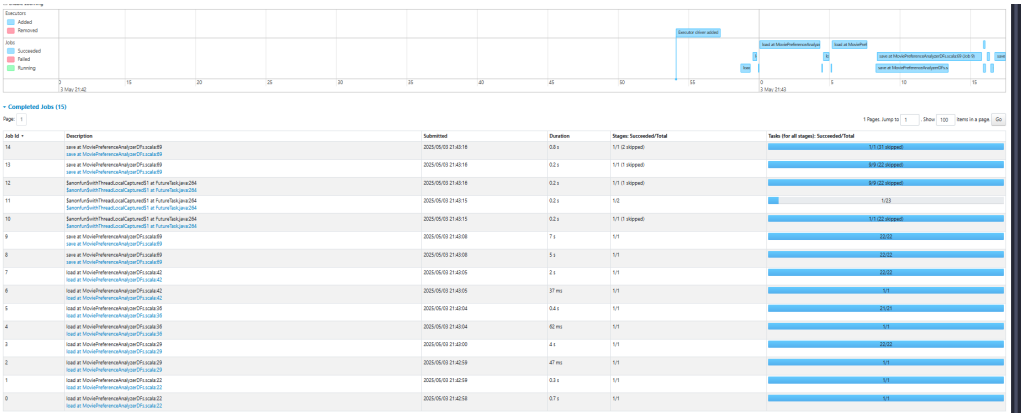


Figure 57: Jobs of Query 6

The job ids from 0-7 correspond to the loading of the input files. For each file, we have 2 jobs.The first job is executed to detect the scheme of the columns and the second parses the data.

Each job is executed in one stage.We are going to analyze the physical jobs 0 and 1.

For the first job, is created due to the `load()` in the movies.csv file and contains only 1 stage.

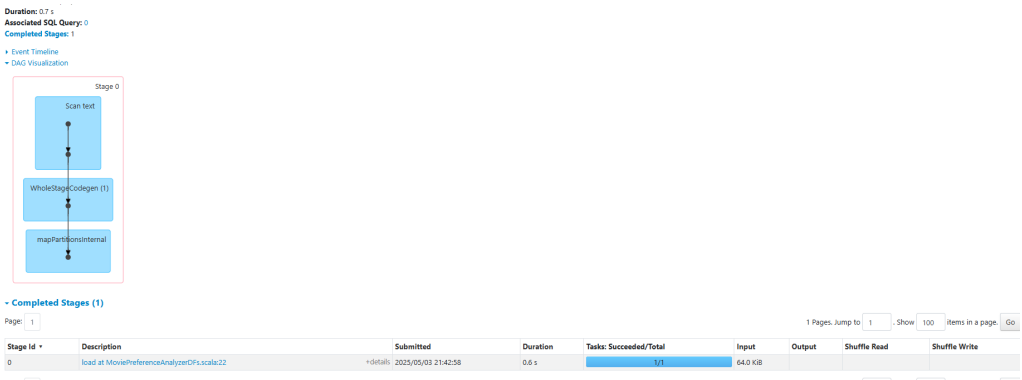
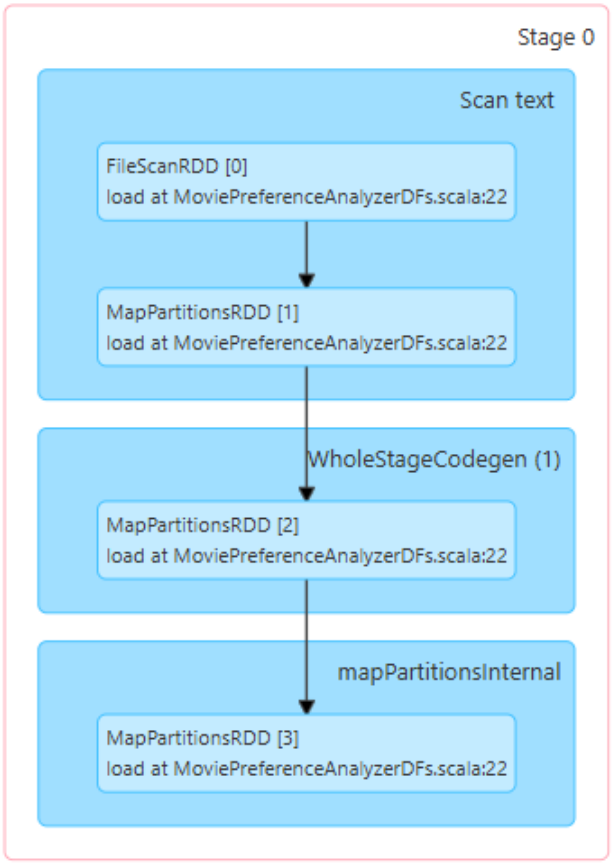


Figure 58: Job 0

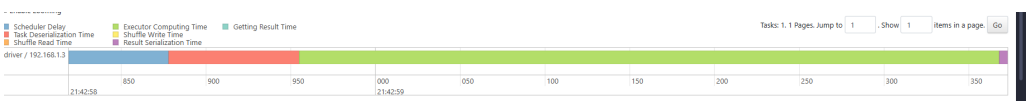
In this stage we have narrow operations because the inferSchema option is enabled. So Spark scans a sample of the file to determine the data type of each column.

Narrow operations were used:

- `FileScanRdd`
- `MapPartionsRdd`



We have no shuffle here, and thus, the stage 0 is executed with 1 task and we do not achieve any degree of parallelism.



The Job 1 is triggered by the `load()` operation in movies.csv and is executed in one stage. The physical job is the following :

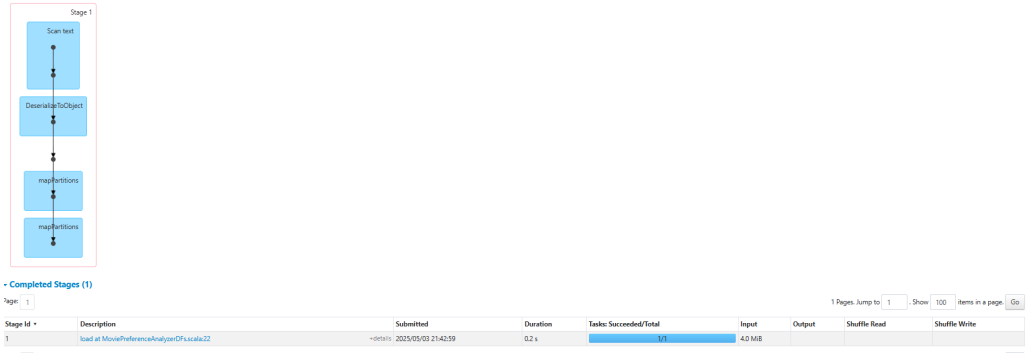


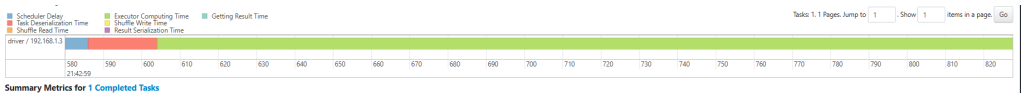
Figure 59: Job 1

In this stage, the loading of the file into a DataFrame is completed.

Narrow operations were used:

- `FileScanRDD`
- `MapPartitionsRDD`
- `DeserializeToObject`
- `SQLExecutionRDD`

The stage is executed with 1 task and thus, we do not have any degree of parallelism.



As mentioned, the next jobs (2 to 7) follow the same logic, with similar dags for the jobs and with the same operations executed.

The only differnce that we should mention is that in jobs 3, 5 and 7 the stages are executed with 22, 21 and 22 tasks respectively. Therefore, in these cases the degree of parallelism is 22, 21 and 22. This happens due to the size of the input file. Spark splits the files into multiple partitions in order to leverage parallelism and reduce the execution time.
The dag of the physical jobs (2 to 7):

And the stages of the jobs :

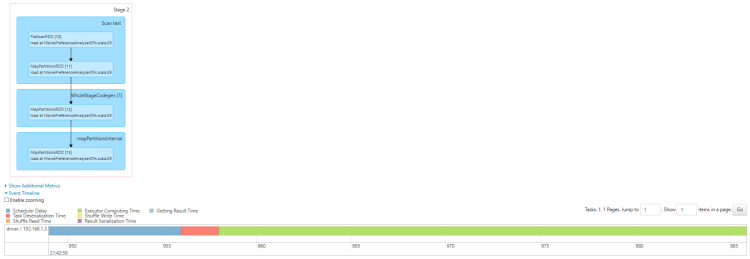


Figure 60: stage 3

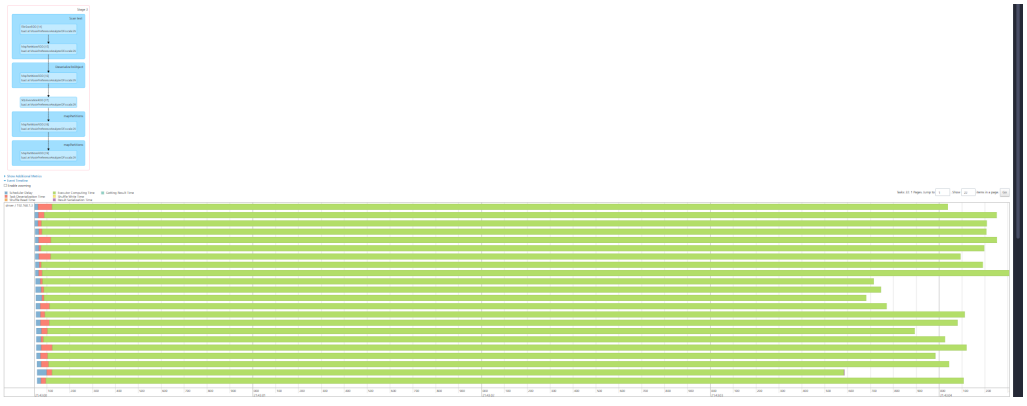


Figure 61: stage 3

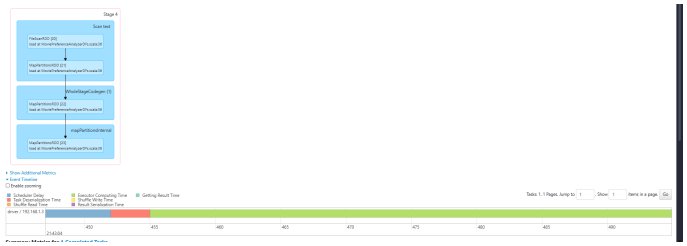


Figure 62: stage 4

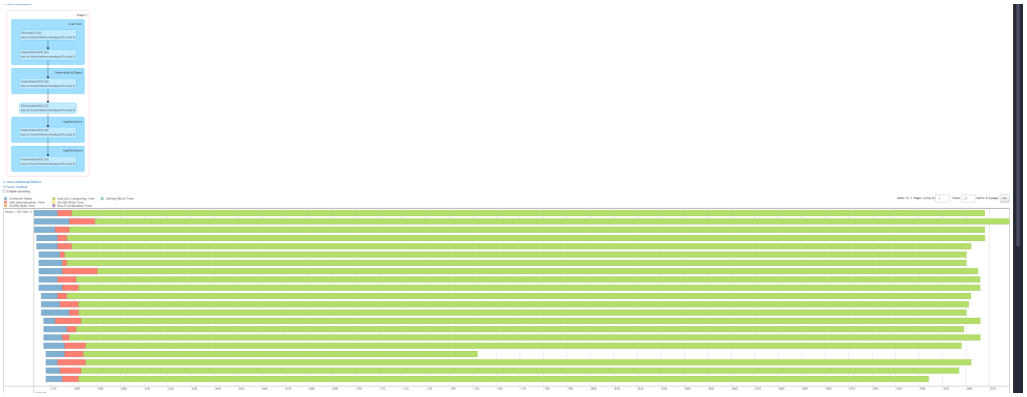


Figure 63: stage 5

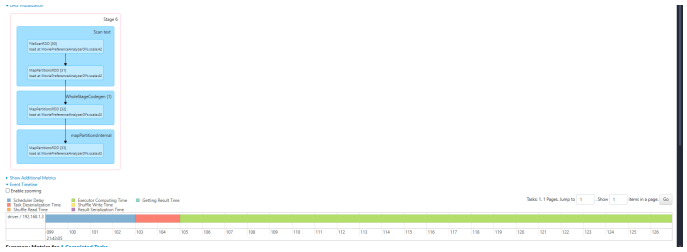


Figure 64: stage 6

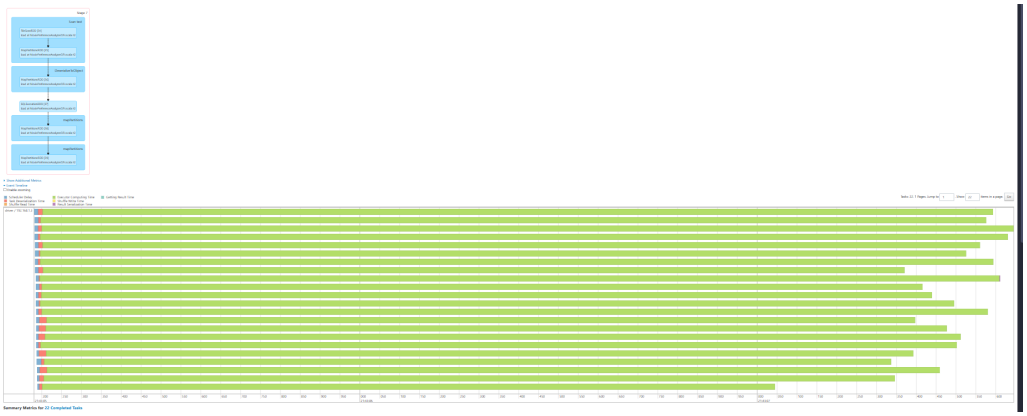


Figure 65: stage 7

Then, the jobs 8 to 14 refers to the computations and the store of the result for this query.

Next, the Job 8 is triggered by `save()` operation which stores a sample of the output in the HDFS. The job is executed in a single stage.

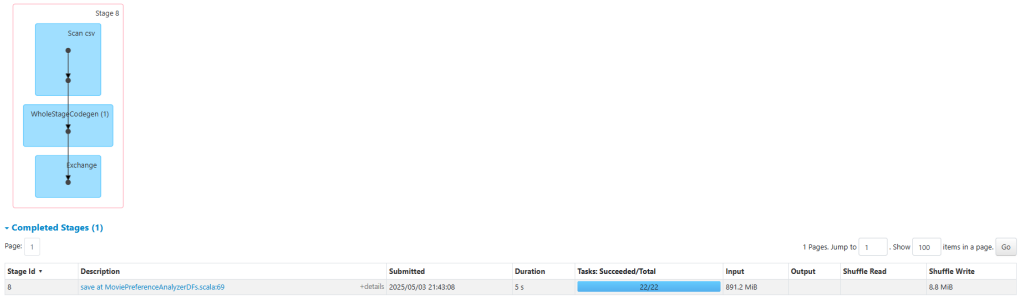


Figure 66: Physical job 8

In stage 8, we observe that the data is read, the skyline computations are performed, and finally the output is saved. From the Dag we can see that Spark Catalyst applies operations such as the Exchange which leads to a shuffle in the data.

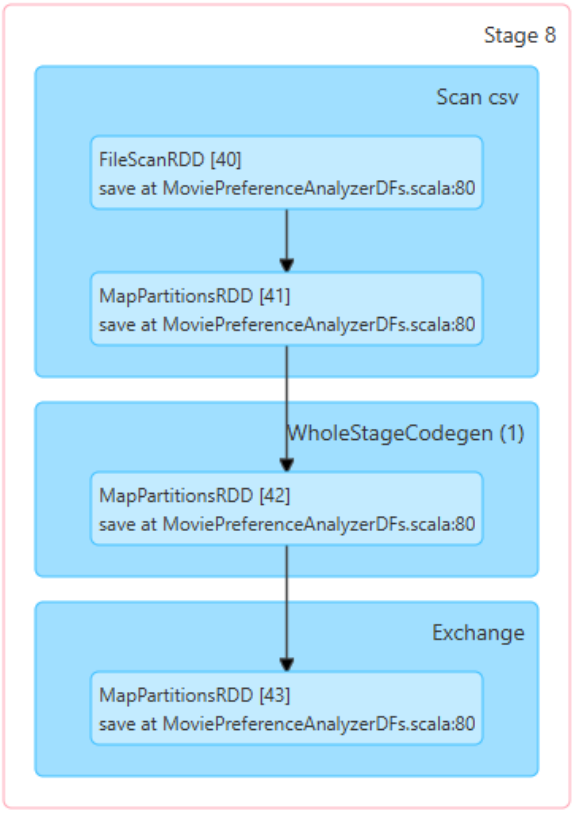
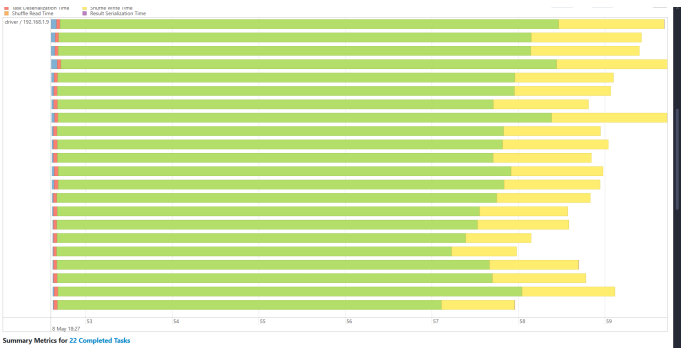


Figure 67: DAG

Stage 8 is completed in 22 tasks which reflecting a 22 degree of parallelism.



As we can see from the Spark UI, the jobs 9, 13 and 14 are also triggered by the operation `save()`

This happens due to Spark's lazy evaluation, where different operations like , `limit()`, `coalesce()` and `write()` can lead to the compute the same dag.In this recomputation, the Spark Catalyst can optimize the execution and reduce the number of computation. This the reason why the degree of parallelism decreases as we approach the final `save()`.
Jobs 9, 13 and 14 are executed in a single stage.

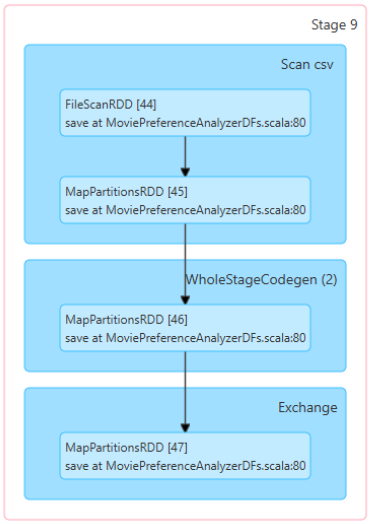


Figure 68: DAG for Stage of Job 9

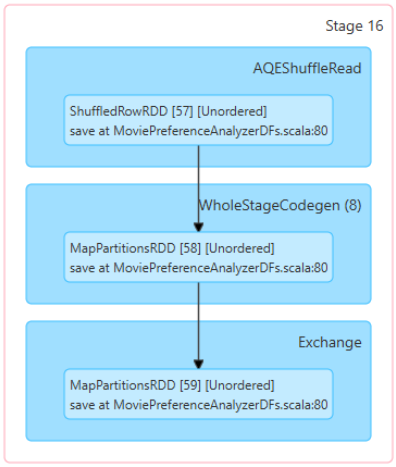


Figure 69: DAG for Stage of Job 13

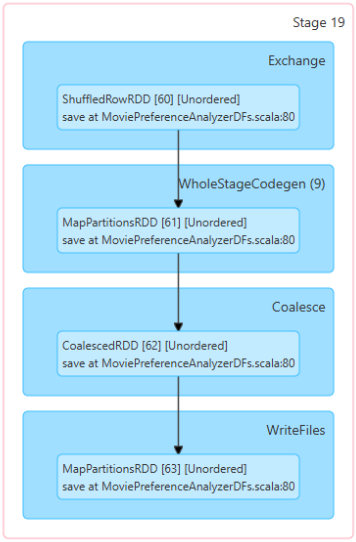
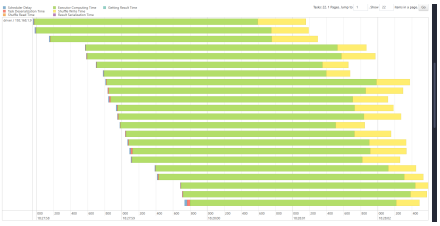
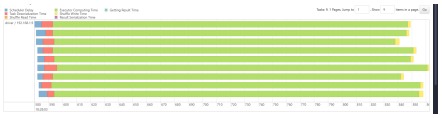


Figure 70: DAG for Stage of Job 14

The degree of parallelism in the dags above is 22 for Job 9, 9 for Job 13, and 1 for Job 14 respectively.

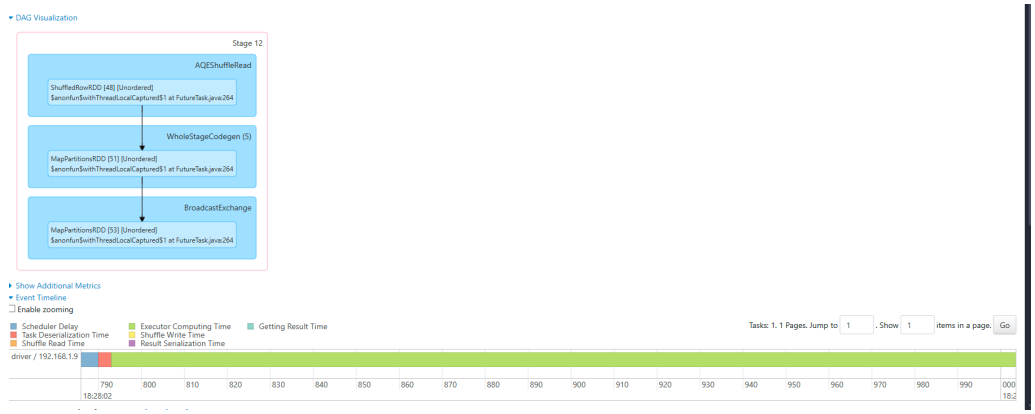




Next Jobs 10, 11 and 12 are triggered due to the optimizations that applied from the Spark Catalyst.These jobs accompany the jobs (9, 13 and 14) that triggered by Spark's lazy evaluation in the `save()` operation. All jobs are executed in a single task.

The stage 11 of Job 10 , is executed in a short time showing that minor optimizations are being performed on the dag including `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The stage is completed in a single task, and thus no parallelism is achieved.

The stage 12 of Job 11, is executed in a single task when 22 are skipped. This happens because the Spark Optimizer do not compute the same data of a previous job. So, the Spark avoids unnecessary calculations. The operations that included in this stage are `AQEShuffleRead()`, which choose most efficient query execution plan, and `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The stage is completed in a single task, and thus no parallelism is achieved.



The Job 12 is executed in 2 stages. In stage 13 we have 22 skipped tasks due to optimizations of Spark Catalyst. The stage 14 is executed in 9 tasks.The operations that included in this stage are `AQEShuffleRead()`, which choose most efficient query execution plan, and `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The number of tasks increases in this stage compared to the previous stages that were executed in a single task. This happens because this Job is the final that triggered for internal optimization purposes. So, Spark Optimizer shuffles a bigger number of partitions to prepare the most efficient execution of the dag for `save()` operation. As a result, the degree of parallelism achieved is 9.

Query 8: Reverse Nearest Neighbor — Match Users to a Movie's Tag Vector

The goal of this query is to match users to a movie by comparing the average tag preferences of their liked movies, with the tag profile of the target movie, using Cosine Similarity. The chosen movie has movieId = 4011 and the title is Snatch

Cosine similarity is a metric that is used to find the similarity between two non-zero vectors defined in an inner product space. In our query, the first vector is the tag profile vector of the Snatch and the second is the average tag preference vector of each user. The formula of cosine similarity is shown below.

$$\text{Cosine Similarity} = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2} \cdot \sqrt{\sum_{i=1}^{n} {B_i}^2}}$$

where :

- A_i is the tag profile of Snatch.
- B_i is the average tag preference vector for each user.
- n is the total number of tags.

To achieve this, we begin by defining the movie vector. We use the narrow operation `filter()` over the genscoresdf in order to extract only the rows that are related to our movie. Then, with the narrow operation `select()` we keep the columns that we are interested in, tagId and relevance.

```
val movieVector = genscoresdf.filter(col("movieId") === "4011").select(
  col("tagId").as("Tag"),
  col("relevance").as("RelevanceOfMyMovie")
)
```

Figure 71: Code

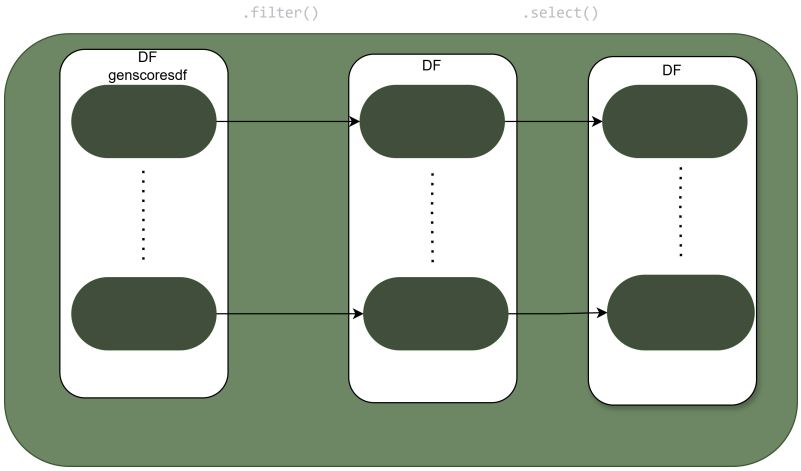


Figure 72: Logical Job

Then, we set the user vector. From the ratingsdf we use the narrow operation `filter()` to keep only data with rating score greater than 4.0. Afterwards, we perform a wide operation `join()` with the genscoresdf in order to associate each user with their tag and their relevance.Also, we group the data by the userId column using the wide operation `groupBy()`. In the end, we compute the average relevance using the `agg()`.

```
val usersvector = ratingsdf.filter(col("rating") > 4.0).select(
    col("userId"),
    col("movieId")
  ).join(genscoresdf, ratingsdf("movieId") === genscoresdf("movieId")).select(
    col("userId"),
    col("tagId"),
    col("relevance")
  ).groupBy("userId","tagId")
  .agg(avg("relevance").alias("usersRelevance"))
```

Figure 73: Code

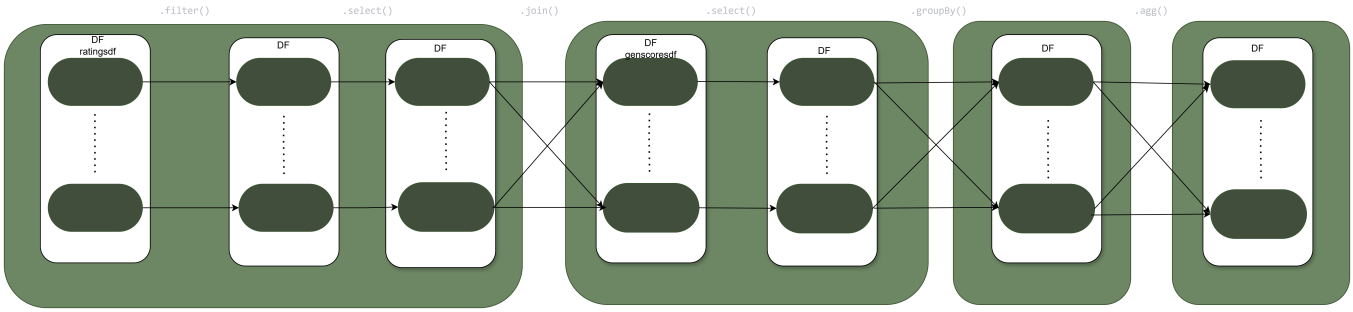


Figure 74: Logical Job

Thereafter, we perform a wide operation `join()` to combine the two vectors using tagId as the key. After that, we group the data by userId column using the wide operation `gorupBy()`.

Then, we apply the wide operation `agg()` to compute the numerator and the denominator of the cosine similarity. Specifically, for the numerator, we calculate the dot product of the vectors and using the `sum` we compute the total. For the denominator, we calculate the sum of squares for each vector , and then apply the `pow(... , 0.5)` to extract the square root.

After that, we use the `select()` to define the final DataFrame, keeping a column for the userId and computing the cosine similarity in a different column. Finally, we sort the output in descending order using the wide operation `orderBy()`.

```
val cosSim = movieVector.join(usersvector, movieVector("Tag") === usersvector("tagId")).groupBy("userId")
  .agg(
    sum(col("RelevanceOfMyMovie") * col("usersRelevance")).alias("numerator"),
    pow(sum((col("RelevanceOfMyMovie")*col("RelevanceOfMyMovie"))),0.5).alias("denominator1"),
    pow(sum((col("usersRelevance")*col("usersRelevance"))),0.5).alias("denominator2")
  )
  .select(
    col("userId"),
    (col("numerator") / (col("denominator1") * col("denominator2"))).alias("Cosine Similarity")
  )
  .orderBy(col("Cosine Similarity").desc)
```

Figure 75: Code

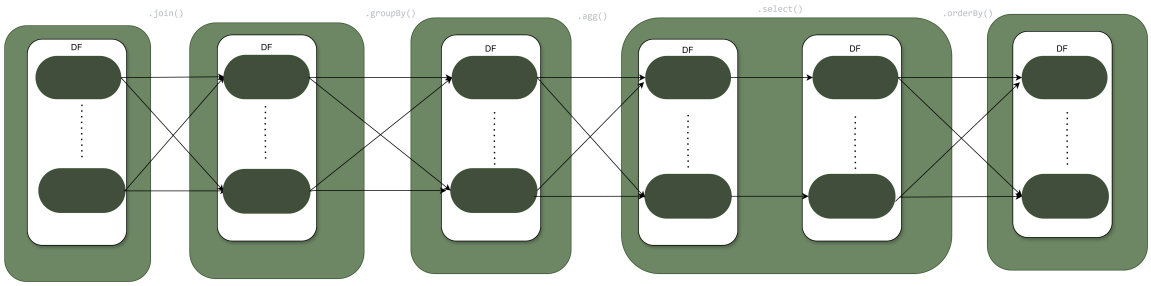


Figure 76: Logical Job

At the end of this query we store a sample of the output in the HDFS with the name of the file to be Query8SampleOutput.

Unfortunately, due to long execution time on the cluster, we had to make a limit on the size of the input files for this query.

Then, we extract the JAR file and submitted it to Spark. From the History Server, we can detect 14 jobs for the query 8.

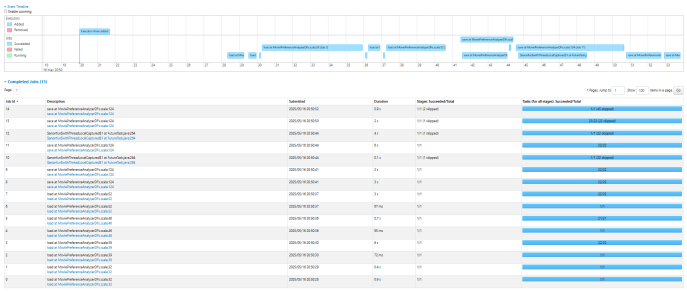


Figure 77: Jobs of Query 8

The first 7 jobs are the same as previous in the Query 6 and correspond to the loading of the input files.Again, for each file we have 2 jobs. The first job is executed to detect the scheme of the columns and the second parses the data.

Each job is executed in one stage.We are going to analyze the physical jobs 0 and 1.

For the first job, is created due to the `load()` in the movies.csv file and contains only 1 stage.

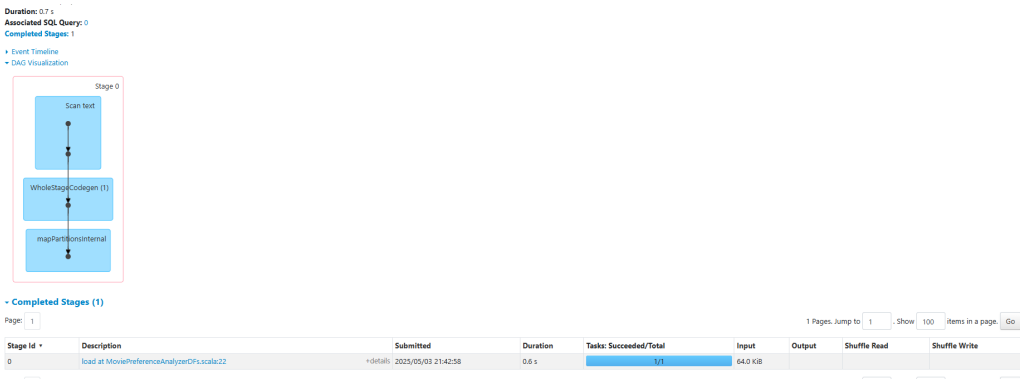
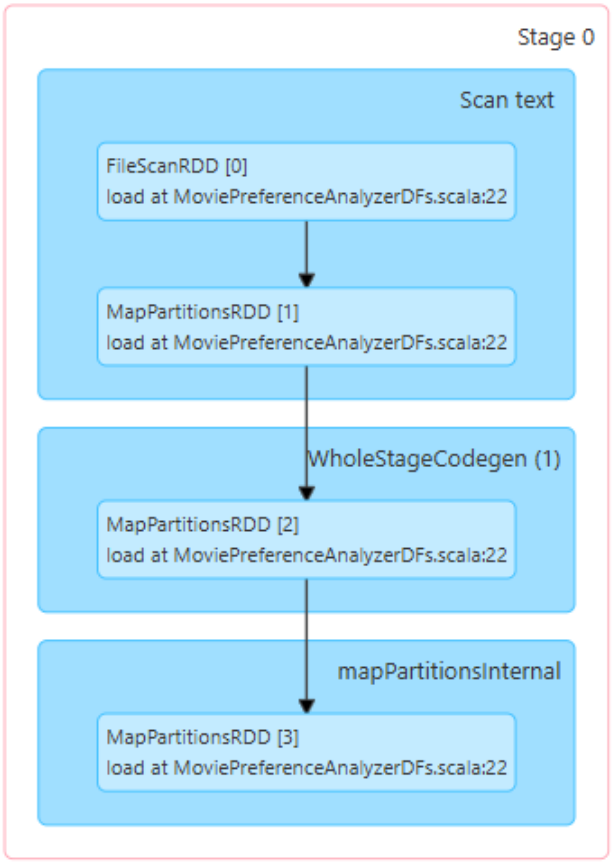


Figure 78: Job 0

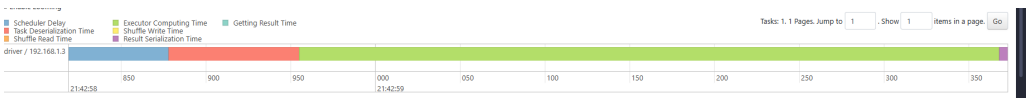
In this stage we have narrow operations because the inferSchema option is enabled. So Spark scans a sample of the file to determine the data type of each column.

Narrow operations were used:

- `FileScanRdd`
- `MapPartionsRdd`



We have no shuffle here, and thus, the stage 0 is executed with 1 task and we do not achieve any degree of parallelism.



The Job 1 is triggered by the `load()` operation in movies.csv and is executed in one stage. The physical job is the following :

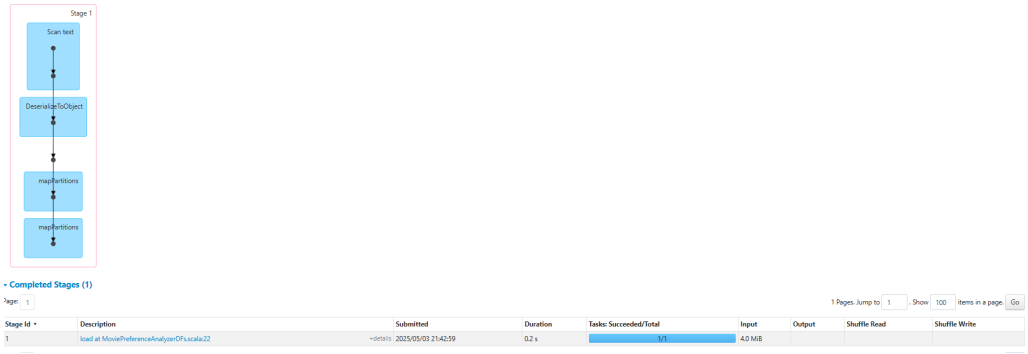


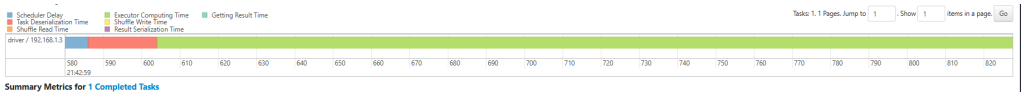
Figure 79: Job 1

In this stage, the loading of the file into a DataFrame is completed.

Narrow operations were used:

- `FileScanRDD`
- `MapPartitionsRDD`
- `DeserializeToObject`
- `SQLExecutionRDD`

The stage is executed with 1 task and thus, we do not have any degree of parallelism.



As mentioned, the next jobs (2 to 7) follow the same logic, with similar dags for the jobs and with the same operations executed.

The only differnce that we should mention is that in jobs 3, 5 and 7 the stages are executed with 22, 21 and 22 tasks respectively. Therefore, in these cases the degree of parallelism is 22, 21 and 22. This happens due to the size of the input file. Spark splits the files into multiple partitions in order to leverage parallelism and reduce the execution time.
The dag of the physical jobs (2 to 7):

And the stages of the jobs :

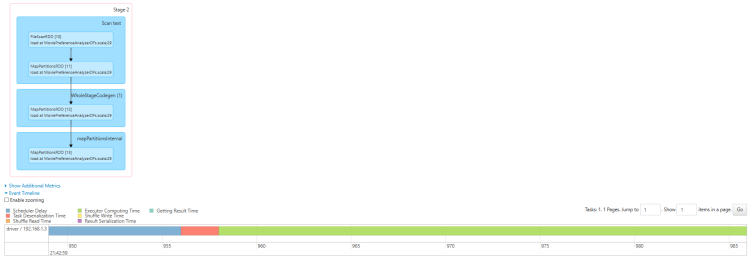


Figure 80: stage 3

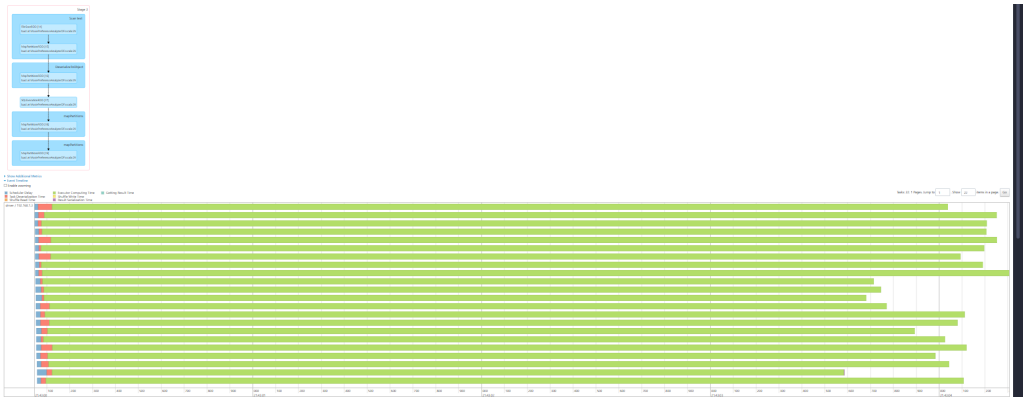


Figure 81: stage 3

Figure 82: stage 4

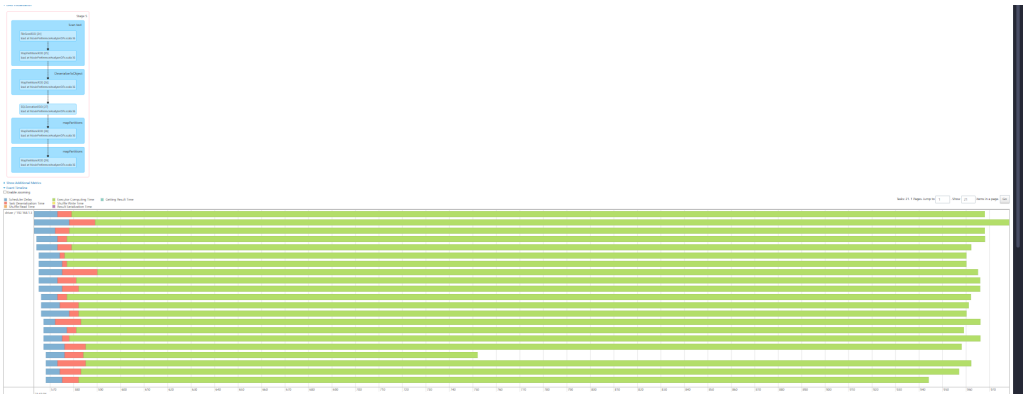


Figure 83: stage 5

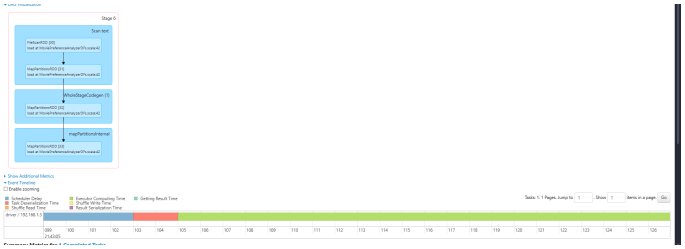


Figure 84: stage 6

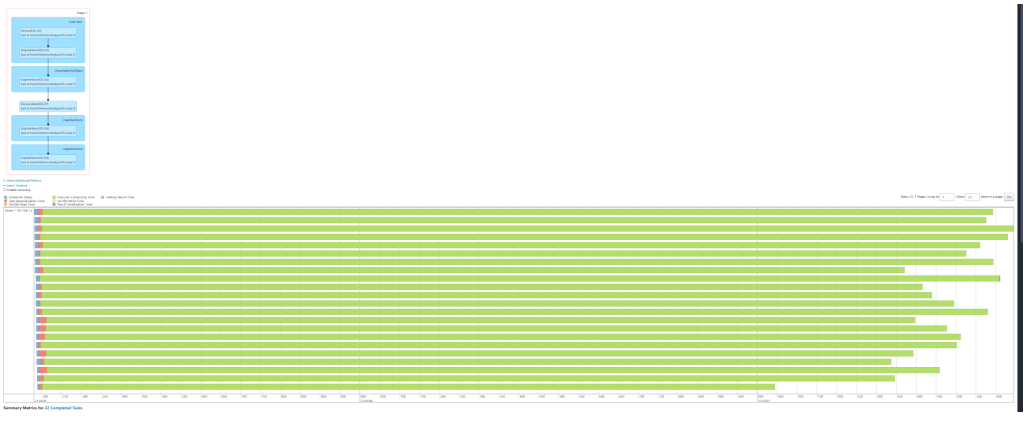
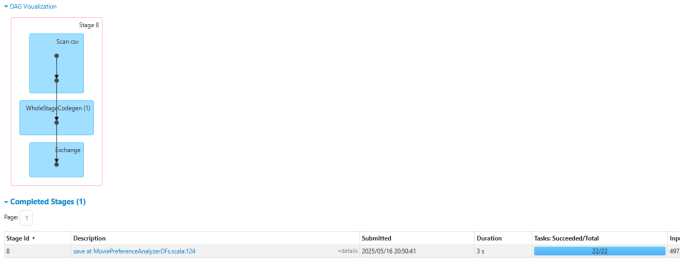
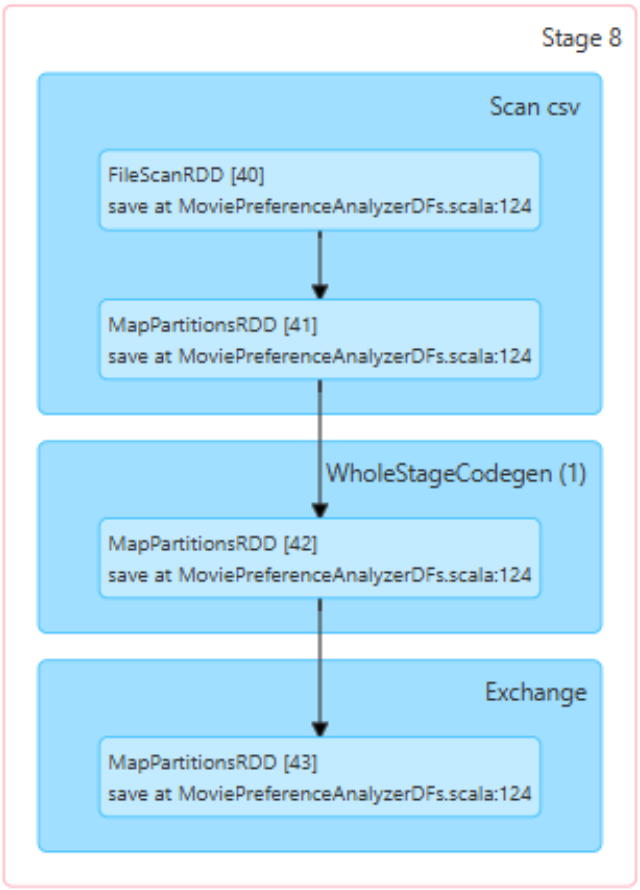


Figure 85: stage 7

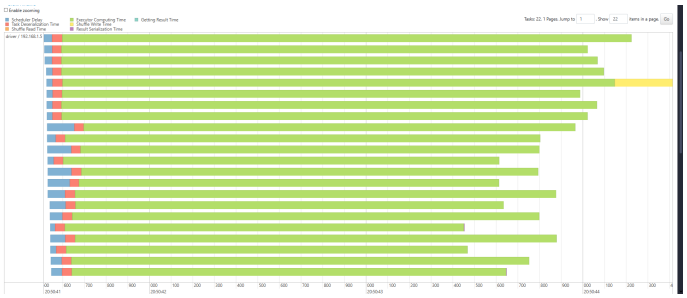
Stage 8 is triggered by `save()` operation.The job is executed in a single stage.



In this stage the Spark Catalyst applies operations such as Exchange which leads to a shuffle in the data

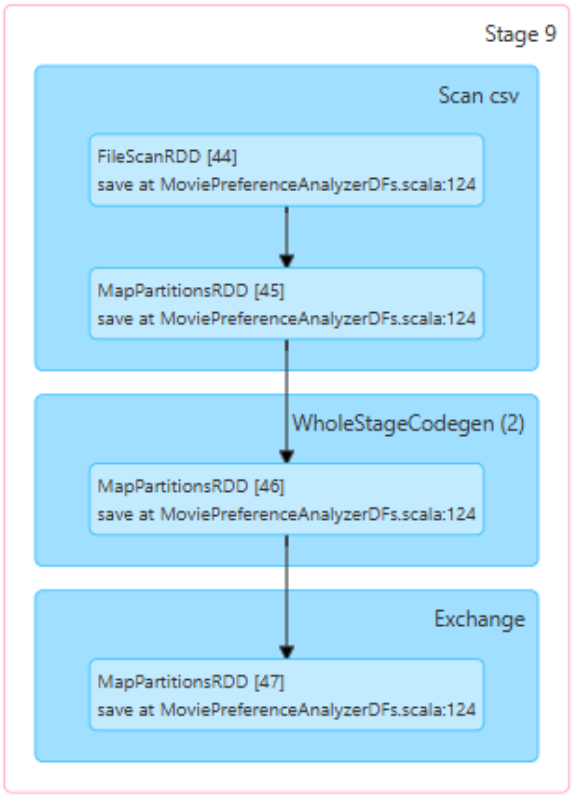


The stage is executed in 22 tasks. So the degree of parallelism achieved is 22.

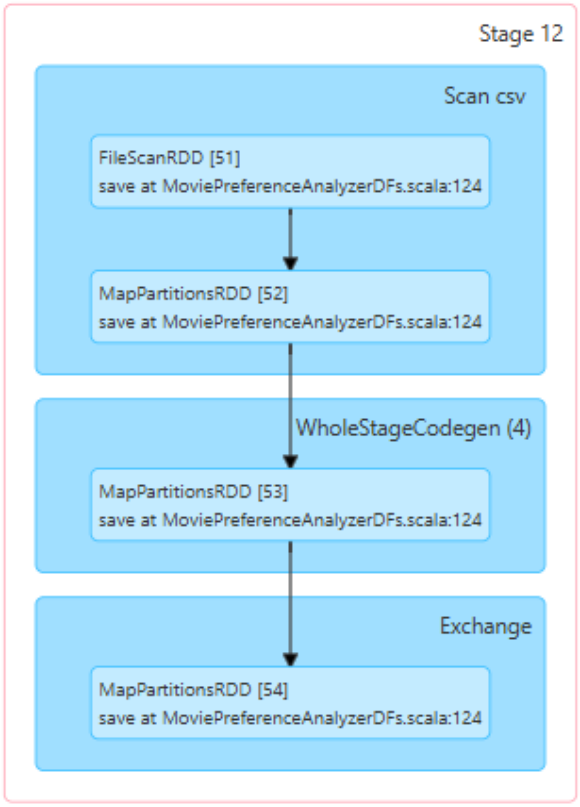


As we can see from the Spark UI, the jobs 9, 11, 13 and 14 are also triggered by the operation `save()`

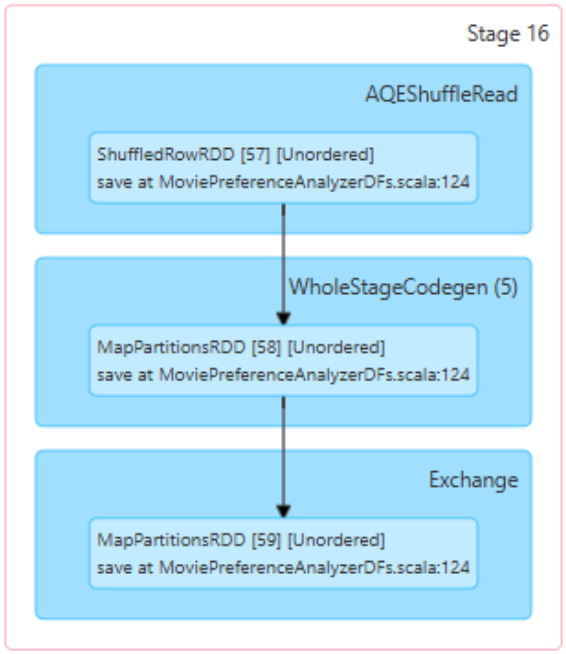
This happens due to Spark's lazy evaluation, where different operations like , `limit()`, `coalesce()` and `write()` can lead to the compute the same dag.In this recomputation, the Spark Catalyst can optimize the execution and reduce the number of computation. This the reason why the degree of parallelism decreases as we approach the final `save()`.
Jobs 9, 11, 13 and 14 are executed in a single stage.



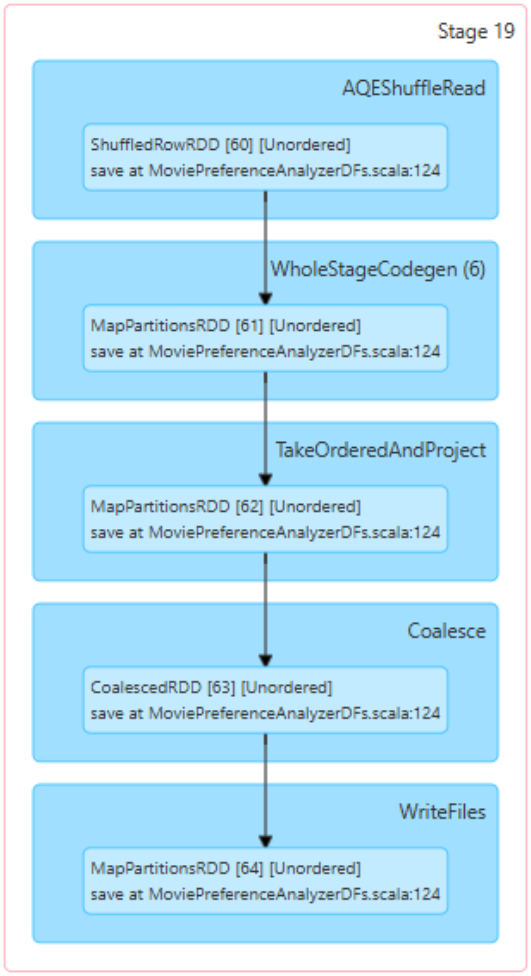
(a) Job 9



(b) Job 11



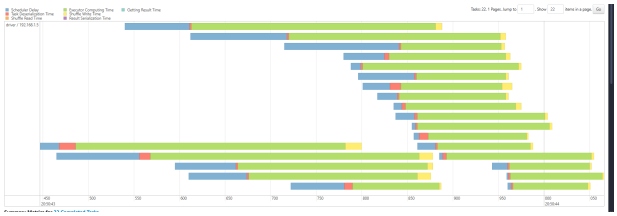
(c) Job 13



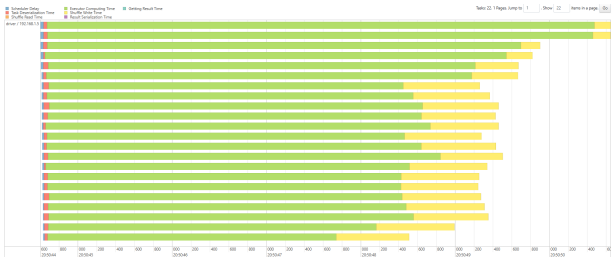
(d) Job 14

Figure 86: DAGs for Spark Jobs 9, 11, 13, and 14

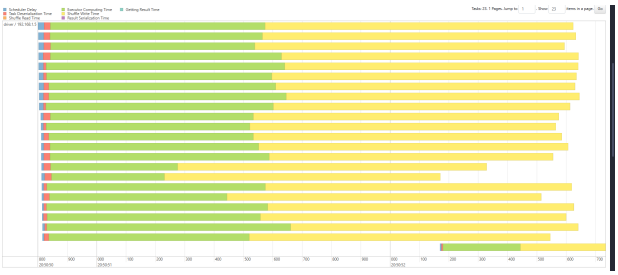
The degree of parallelism in the dags above is 22 for Job 9,22 for Job 11, 23 for Job 13, and 1 for Job 14 respectively.



(a) Job 9



(b) Job 11



(c) Job 13

(d) Job 14

Next Jobs 10 and 12 are triggered due to the optimizations that applied from the Spark Catalyst.These jobs accompany the jobs (9, 11, 13 and 14) that triggered by Spark's lazy evaluation in the `save()` operation. All jobs are executed in a single stage.

The stage 11 of Job 10 , is executed in a short time , with 1 task and 22 skipped, showing that minor optimizations are being performed on the dag including `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The stage is completed in a single task, and thus no parallelism is achieved.

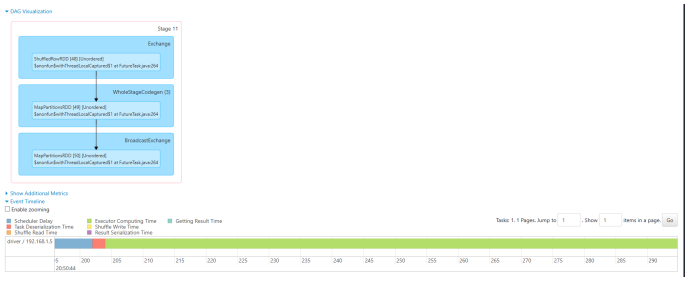


Figure 88: Stage 11

The stage 14 of Job 12, is executed in a single task when 22 are skipped. This happens because the Spark Optimizer do not compute the same data of a previous job. So, the Spark avoids unnecessary calculations. The operations that included in this stage are `AQEShuffleRead()`, which choose most efficient query execution plan, and `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The stage is completed in a single task, and thus no parallelism is achieved.

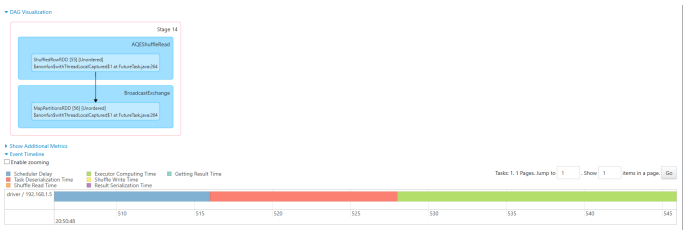


Figure 89: Stage 14

Query 10: Reverse Top-K Neighborhood Users Using Semantic Tag Profiles

The objective of this query is similar to that of Query 8. The difference here is that the result should include only the top-3 most similar users to the target movie.

The implementation steps are the same as in the query 8 but at the end we also include the `limit()` to keep only the top-K most similar users.

The logical plan is also the same, with the only difference being that in the last code block the final operation is the narrow `limit()`.

At the end of this query we store a sample of the output in the HDFS with the name of the file to be Query10SampleOutput.

Unfortunately, due to long execution time on the cluster, we had to make a limit on the size of the input files for this query.

Then, we extract the JAR file and submitted it to Spark. From the History Server, we can detect 14 jobs for the query 8.

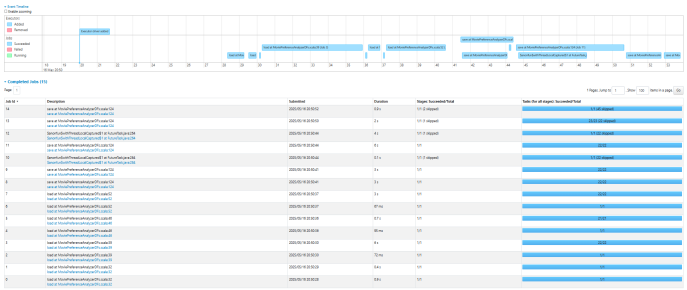


Figure 90: Jobs of Query 10

The first 7 jobs are the same as previous in the Query 6 and 8 and correspond to the loading of the input files.Again, for each file we have 2 jobs. The first job is executed to detect the scheme of the columns and the second parses the data.

Each job is executed in one stage.We are going to analyze the physical jobs 0 and 1.

For the first job, is created due to the `load()` in the movies.csv file and contains only 1 stage.

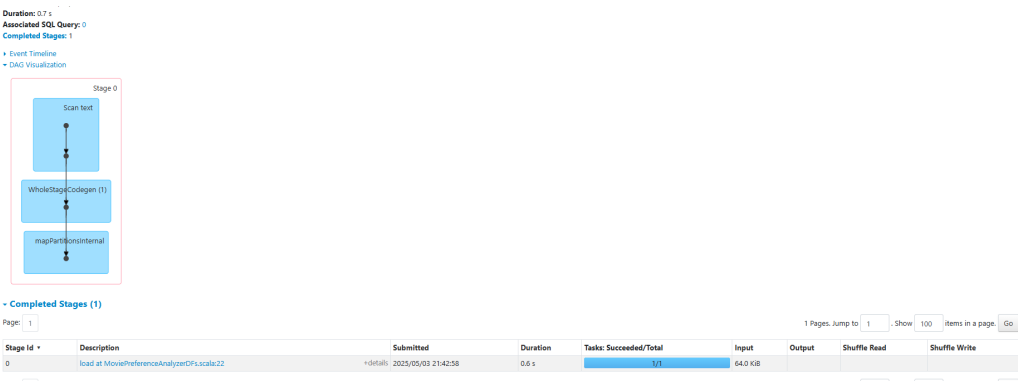
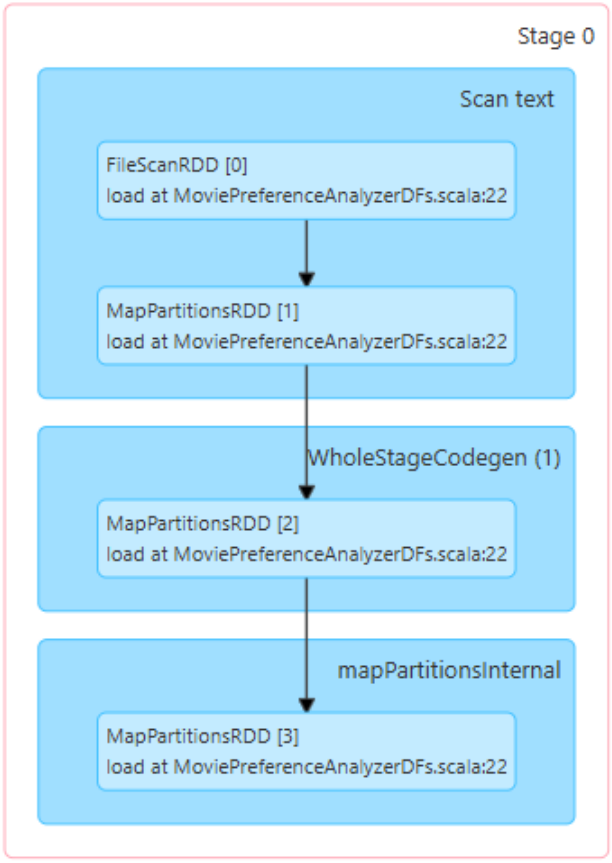


Figure 91: Job 0

In this stage we have narrow operations because the inferSchema option is enabled. So Spark scans a sample of the file to determine the data type of each column.



We have no shuffle here, and thus, the stage 0 is executed with 1 task and we do not achieve any degree of parallelism.

The Job 1 is triggered by the `load()` operation in movies.csv and is executed in one stage. The physical job is the following :

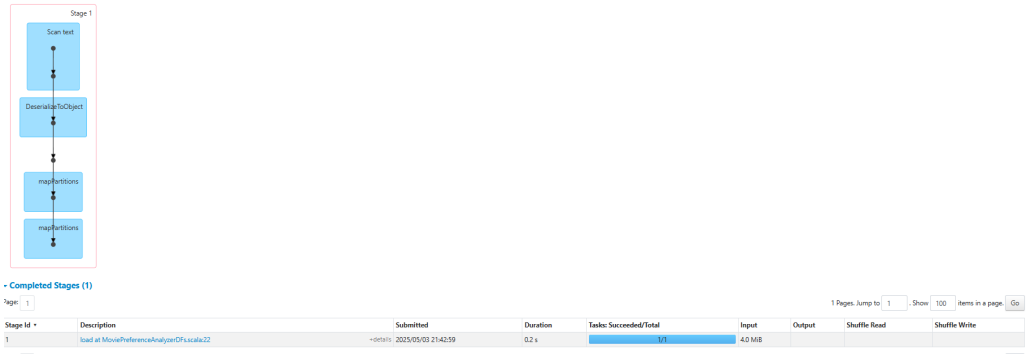
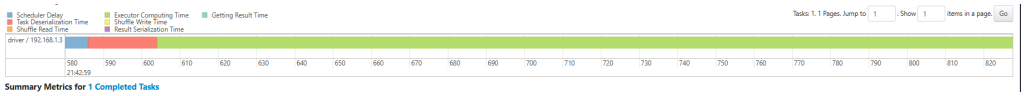


Figure 92: Job 1

In this stage, the loading of the file into a DataFrame is completed.
The stage of this Job is executed with 1 task and thus, we do not have any degree of parallelism.



As mentioned, the next jobs (2 to 7) follow the same logic, with similar dags for the jobs and with the same operations executed.

The only differnce that we should mention is that in jobs 3, 5 and 7 the stages are executed with 22, 21 and 22 tasks respectively. Therefore, in these cases the degree of parallelism is 22, 21 and 22. This happens due to the size of the input file. Spark splits the files into multiple partitions in order to leverage parallelism and reduce the execution time.
The dag of the physical jobs (2 to 7):

And the stages of the jobs :

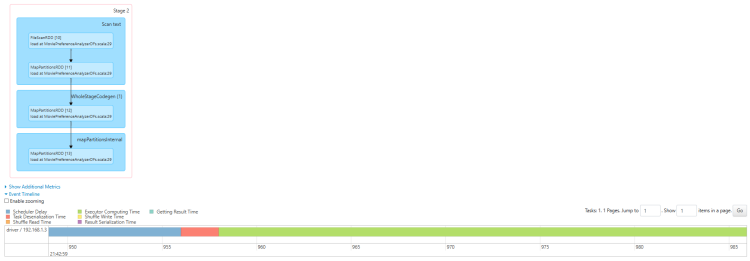


Figure 93: stage 3

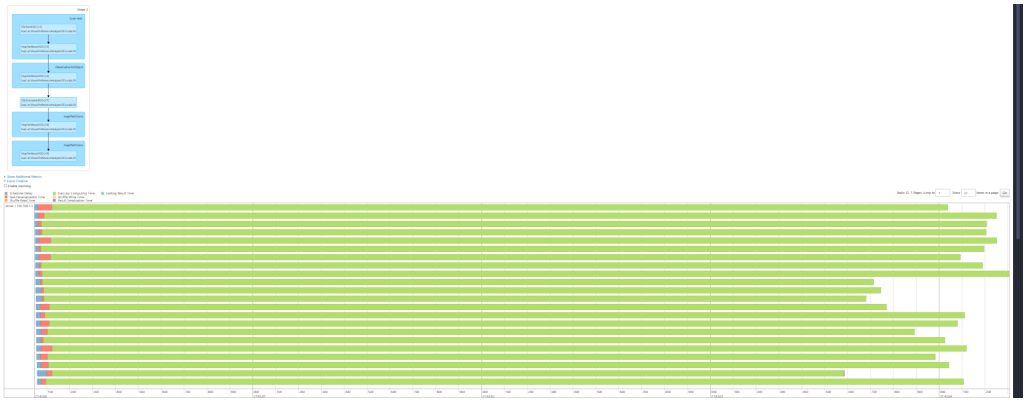


Figure 94: stage 3

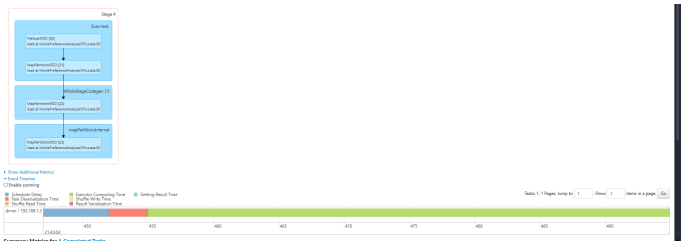


Figure 95: stage 4

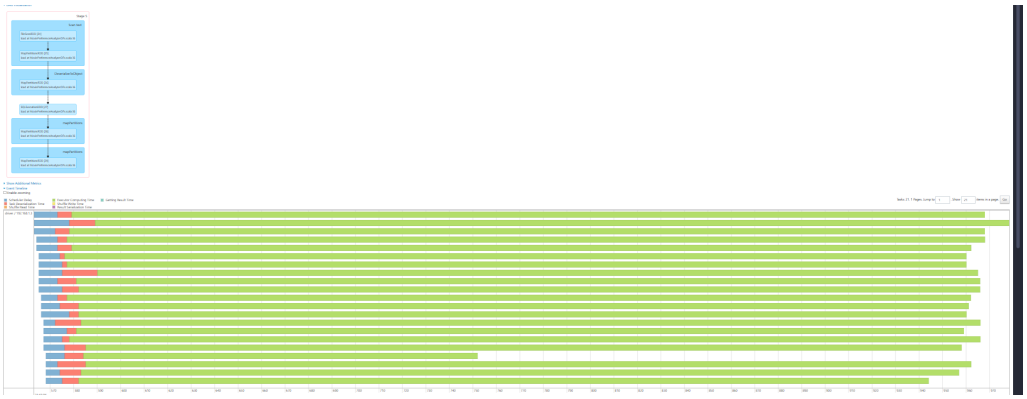


Figure 96: stage 5

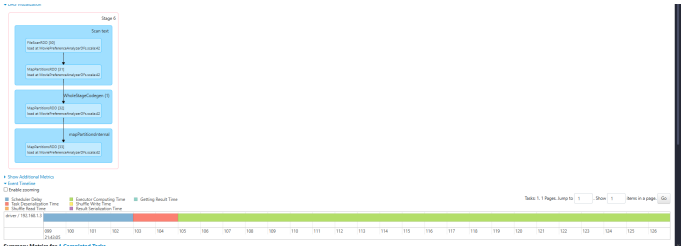


Figure 97: stage 6

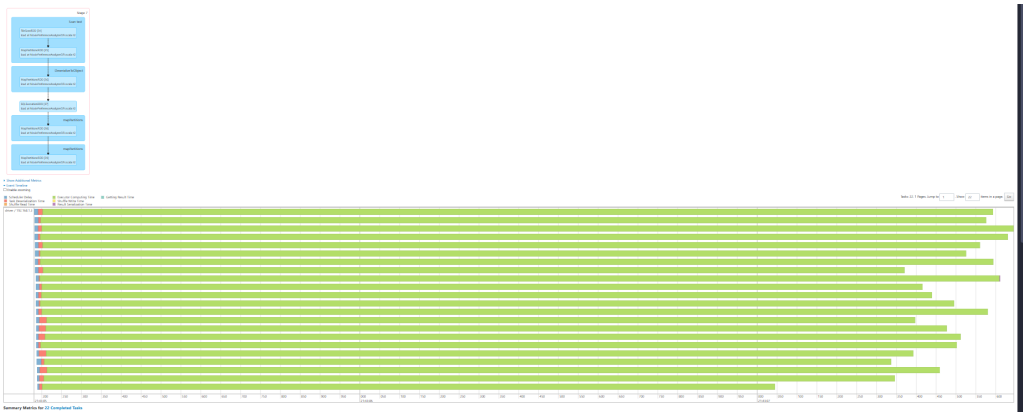
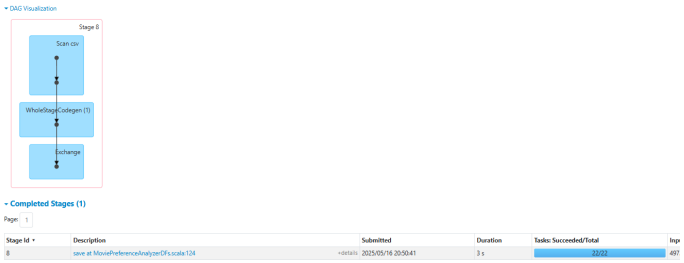
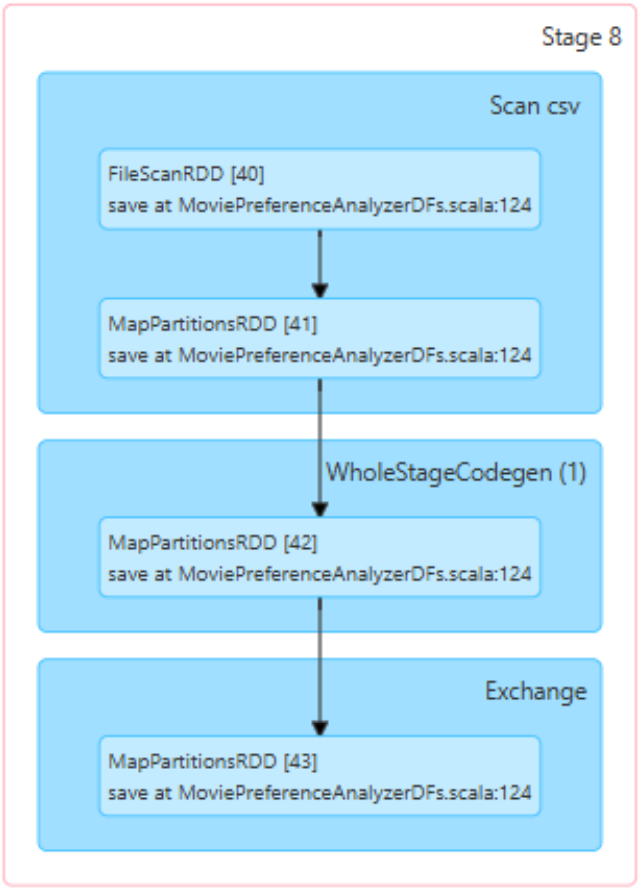


Figure 98: stage 7

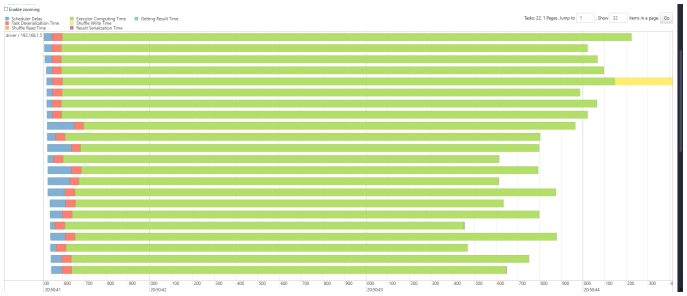
Job 8 is triggered by `save()` operation.The job is executed in a single stage.



In this stage the Spark Catalyst applies operations such as Exchange which leads to a shuffle in the data



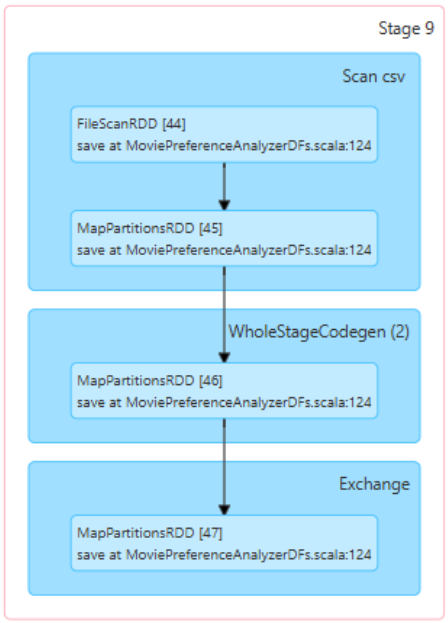
The stage 8 is executed in 22 tasks. So the degree of parallelism achieved is 22.



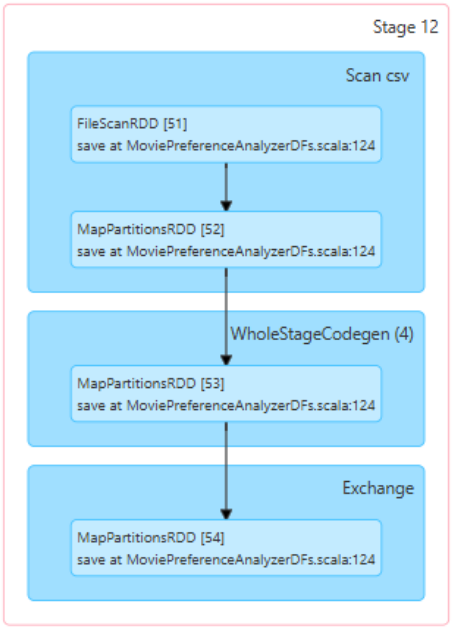
As we can see from the Spark UI, the jobs 9, 11, 13 and 14 are also triggered by the operation `save()`

This happens due to Spark's lazy evaluation, where different operations like , `limit()`, `coalesce()` and `write()` can lead to the compute the same dag.In this recomputation, the Spark Catalyst can optimize the execution and reduce the number of computation. This the reason why the degree of parallelism decreases as we approach the final `save()`.

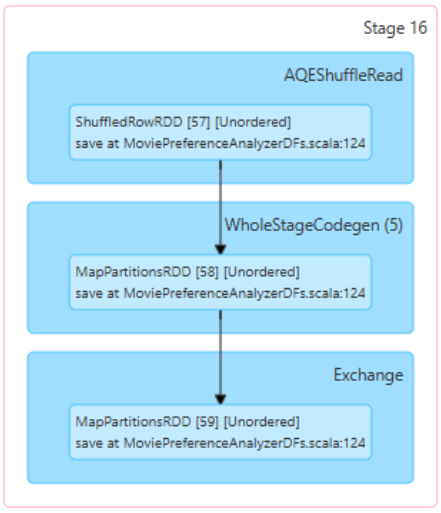
Jobs 9, 11, 13 and 14 are executed in a single stage (with 11, 13 and 14 having one skipped job).



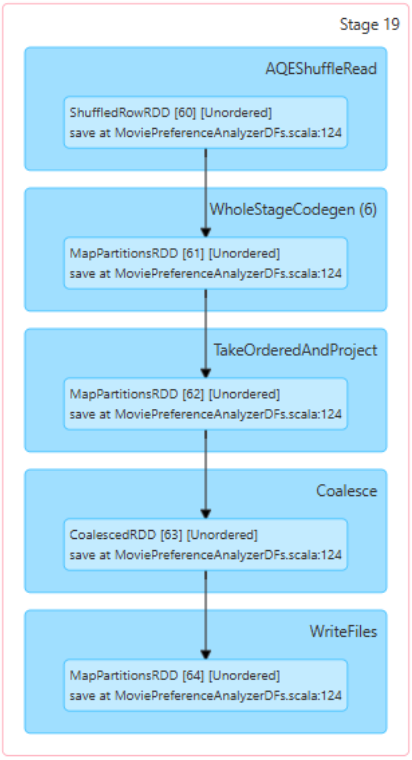
(a) Job 9



(b) Job 11



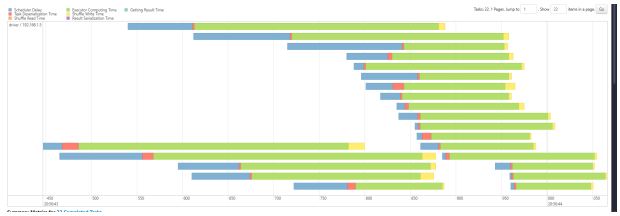
(c) Job 13



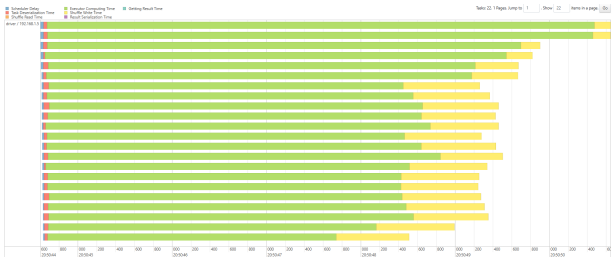
(d) Job 14

Figure 99: DAGs for Spark Jobs 9, 11, 13, and 14

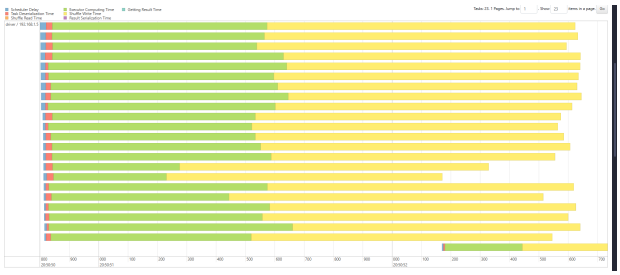
The degree of parallelism in the dags above is 22 for Job 9,22 for Job 11, 23 for Job 13, and 1 for Job 14 respectively.



(a) Job 9



(b) Job 11



(c) Job 13

(d) Job 14

Next Jobs 10 and 12 are triggered due to the optimizations that applied from the Spark Catalyst.These jobs accompany the jobs (9, 11, 13 and 14) that triggered by Spark's lazy evaluation in the `save()` operation. All jobs are executed in a single stage.

The stage 11 of Job 10 , is executed in a short time , with 1 task and 22 skipped, showing that minor optimizations are being performed on the dag including `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The stage is completed in a single task, and thus no parallelism is achieved.

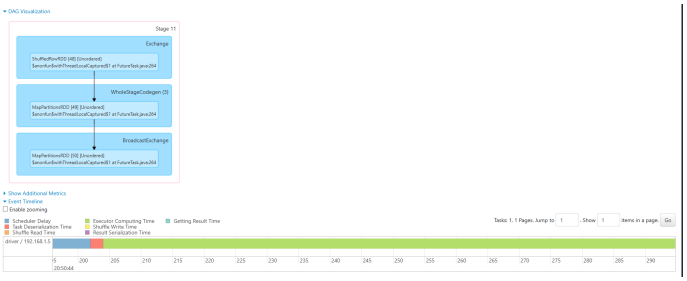


Figure 101: Stage 11

The stage 14 of Job 12, is executed in a single task when 22 are skipped. This happens because the Spark Optimizer do not compute the same data of a previous job. So, the Spark avoids unnecessary calculations. The operations that included in this stage are `AQEShuffleRead()`, which choose most efficient query execution plan, and `BroadcastExchange` operation which collects and broadcasts rows for use in the next Job. The stage is completed in a single task, and thus no parallelism is achieved.

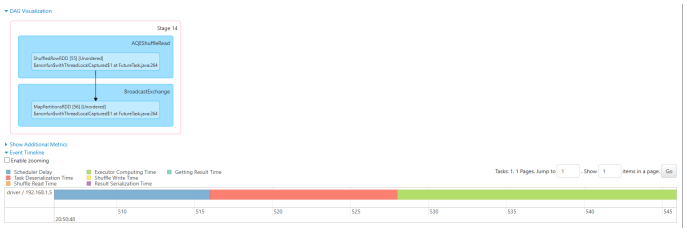


Figure 102: Stage 14

Final Cluster Submission

Following the official instructions provided by the SoftNet Lab, the necessary changes were made in both the 'build.sbt' and SparSession.builder.

In the build.sbt file we change the SBT version and the dependencies.More specifically :

- We downgrade the version of Scala from 2.13.11 to `2.11.8`
- We replaced the Spark's library dependencies:
  - From: `"org.apache.spark" %% "spark-core" % "3.5.5"` and `"org.apache.spark" %% "spark-sql" % "3.5.5"`
  - To: `"org.apache.spark" %% "spark-core" % "2.3.1"` and `"org.apache.spark" %% "spark-sql" % "2.3.1"`
- We added the Hadoop client dependency required by the cluster's HDFS interface:
  - `"org.apache.hadoop" % "hadoop-client" % "3.1.1"`

```
ThisBuild / version := "0.1.0-SNAPSHOT"

//local
//ThisBuild / scalaVersion := "2.13.11"

//cluster
ThisBuild / scalaVersion := "2.11.8"

lazy val root = (project in file("."))
  .settings(
    name := "ScalaProject1"
  )

//local
//libraryDependencies ++= Seq(
//  "org.apache.spark" %% "spark-core" % "3.5.5",
//  "org.apache.spark" %% "spark-sql" % "3.5.5" )

//config for the SoftNet Cluster
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "2.3.1",
  "org.apache.spark" %% "spark-sql" % "2.3.1",
  "org.apache.hadoop" % "hadoop-client" % "3.1.1")
```

Figure 103: build.sbt

Also we make the necessary changes to the SparkSession.builder :

- We changed the master:
  - From: `master("local[*]")`
  - To: `master("yarn")`
- We updated the default filesystem URI:
  - From: `"hdfs://localhost:9000"`
  - To: `"hdfs://clu01.softnet.tuc.gr:8020"`

- We added the TUC Cluster's HDFS addresses using:
  - `.config("spark.hadoop.yarn.resourcemanager.address", "http://clu01.softnet.tuc.gr`
  - `.config("spark.hadoop.yarn.application.classpath", "$HADOOP_CONF_DIR,...")`

Figure 104: SparkSession.builder

Finally we change the paths for the input files and the output files accroding to the TUC Cluster's HDFS.

- For input files:
  - From: `"hdfs://localhost:9000/ml-latest/filename.csv`
  - To: `"hdfs://clu01.softnet.tuc.gr:8020/user/chrisa/ml-latest/filename.csv"`
- For output files:
  - From: `"hdfs://localhost:9000/QueryXSampleOutput"`
  - To:`"hdfs://clu01.softnet.tuc.gr:8020/user/fp25_5/QueryXSampleOutput"`

After making these small changes, we were ready to submit our JAR to the Softnet Lab's cluster. From the Cluster's HistoryServer, we can see that our JAR executed in 19min to execute with 24 completed jobs.

Completed Jobs (24)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
23	save at INF424Project.scala:257 save at INF424Project.scala:257	2025/05/20 16:46:31	1.4 min	6/6	609/609
22	run at ThreadPoolExecutor.java:1142 run at ThreadPoolExecutor.java:1142	2025/05/20 16:45:58	32 s	2/2	8/8
21	save at INF424Project.scala:219 save at INF424Project.scala:219	2025/05/20 16:45:52	5 s	2/2 (4 skipped)	201/201 (408 skipped)
20	save at INF424Project.scala:219 save at INF424Project.scala:219	2025/05/20 16:44:18	1.6 min	5/5	608/608
19	run at ThreadPoolExecutor.java:1142 run at ThreadPoolExecutor.java:1142	2025/05/20 16:43:45	33 s	2/2	8/8
18	save at INF424Project.scala:175 save at INF424Project.scala:175	2025/05/20 16:43:38	6 s	2/2 (2 skipped)	201/201 (11 skipped)
17	run at ThreadPoolExecutor.java:1142 run at ThreadPoolExecutor.java:1142	2025/05/20 16:42:38	59 s	3/3	211/211
16	load at INF424Project.scala:146 load at INF424Project.scala:146	2025/05/20 16:41:51	46 s	1/1	8/8
15	load at INF424Project.scala:146 load at INF424Project.scala:146	2025/05/20 16:41:51	57 ms	1/1	1/1
14	load at INF424Project.scala:140 load at INF424Project.scala:140	2025/05/20 16:41:46	5 s	1/1	3/3
13	load at INF424Project.scala:140 load at INF424Project.scala:140	2025/05/20 16:41:46	77 ms	1/1	1/1
12	load at INF424Project.scala:133 load at INF424Project.scala:133	2025/05/20 16:40:40	1.1 min	1/1	14/14
11	load at INF424Project.scala:133 load at INF424Project.scala:133	2025/05/20 16:40:40	58 ms	1/1	1/1
10	load at INF424Project.scala:126 load at INF424Project.scala:126	2025/05/20 16:40:37	3 s	1/1	2/2
9	load at INF424Project.scala:126 load at INF424Project.scala:126	2025/05/20 16:40:36	0.7 s	1/1	1/1
8	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/20 16:40:32	0.2 s	1/1	1/1
7	take at INF424Project.scala:113 take at INF424Project.scala:113	2025/05/20 16:40:32	0.6 s	2/2 (3 skipped)	8/8 (16 skipped)
6	sortBy at INF424Project.scala:108 sortBy at INF424Project.scala:108	2025/05/20 16:34:02	6.5 min	4/4	23/23
5	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/20 16:34:01	0.3 s	1/1	1/1
4	take at INF424Project.scala:90 take at INF424Project.scala:90	2025/05/20 16:34:01	35 ms	1/1 (13 skipped)	2/2 (41 skipped)
3	take at INF424Project.scala:90 take at INF424Project.scala:90	2025/05/20 16:34:01	49 ms	1/1 (13 skipped)	4/4 (41 skipped)
2	take at INF424Project.scala:90 take at INF424Project.scala:90	2025/05/20 16:34:01	0.1 s	2/2 (12 skipped)	8/8 (34 skipped)
1	sortBy at INF424Project.scala:84 sortBy at INF424Project.scala:84	2025/05/20 16:29:26	4.6 min	9/9 (4 skipped)	33/33 (8 skipped)
0	sortBy at INF424Project.scala:72 sortBy at INF424Project.scala:72	2025/05/20 16:29:08	17 s	5/5	10/10

Some samples from each query's output:

```
fp25_5@clu04:~$ hdfs dfs -cat Query2SampleOutput/part-00000
(Musical,action,4.030195842754316)
(Film-Noir,dark,3.722885133569316)
(IMAX,action,3.6122399431439454)
(War,atmospheric,3.5922525582722113)
(Crime,sci-fi,3.5908394578638108)
(Sci-Fi,sci-fi,3.544723290523833)
(Comedy,sci-fi,3.3421978392129295)
(Thriller,atmospheric,3.2920118339921873)
(Children,nudity (topless),3.2833139786330303)
(Drama,based on a book,3.280301433587672)
(Western,superhero,3.227444931665948)
(Action,drama,3.1603041170895394)
(Horror,sci-fi,3.0918238789064336)
(Romance,action,3.0809108797479388)
(Animation,drama,3.062639993170651)
(Adventure,drama,3.043255636429757)
(Fantasy,drama,3.0308120629493875)
(Documentary,sci-fi,2.807798279264525 7)
```

Figure 105: Query 2 Sample Output

```
fp25_5@clu04:~$ hdfs dfs -cat Query4SampleOutput/part-00000
(This ,5.0)
(weltraum,5.0)
(Writer: Michael Petroni,5.0)
(stuffed in a locker,5.0)
(sci fantasy,5.0)
(Wonderfull story about what happened in the mid 80's,5.0)
(complicated friendships,5.0)
(basalt,5.0)
(Schumacher,5.0)
(lovestory over a span of time,5.0)
(thankfully uncorny,5.0)
(Writer: Lorenzo Semple Jr.,5.0)
(I think I should watch this again with subtitles...,5.0)
(psychedelic folk music,5.0)
(announcer,5.0)
(Heart wrenching,5.0)
(walking into a wall,5.0)
(Writer: James Fujii,5.0)
(smoking a cigarette in a bathtub,5.0)
(commin of age,5.0)
(radical theater,5.0)
(Writer: Waldo Salt,5.0)
(hollywood witchhunt,5.0)
(austin,5.0)
(Bard Pitt,5.0)
```

Figure 106: Query 4 Sample Output

```
fp25_5@clu04:~$ hdfs dfs -cat Query6SampleOutput/part-00000-2b44d0ec-2645-4de
31,3.235021132012928,12067,0.09621210106382969
516,3.0853457738748626,3644,0.08179432624113486
1143,3.4191176470588234,68,0.07246010638297873
1270,3.957795417324461,67777,0.20492907801418428
1303,4.003821499013807,4056,0.1726908244680851
1322,1.7920634920634921,315,0.087500664893617
1339,3.4482539682539683,12600,0.14688741134751776
1352,3.022235576923077,832,0.08989051418439707
1650,3.434549356223176,466,0.08889250886524816
1699,3.6149797570850204,1235,0.11969104609929089
1903,2.8857142857142857,70,0.08165979609929064
2393,3.193371212121212,10560,0.11670301418439713
2572,3.560852433542372,18019,0.10695722517730505
2711,3.515850144092219,347,0.08201573581560288
2776,3.5496688741721854,151,0.05990026595744669
2996,3.393103448275862,870,0.09534308510638301
3000,4.166026555470207,18226,0.19129233156028344
3213,3.5230801510700798,2383,0.13226684397163135
3488,2.1358024691358026,81,0.07910062056737571
3704,3.132420710219572,7378,0.11712721631205672
3761,3.6424418604651163,516,0.10052659574468097
4391,3.6049723756906076,181,0.11680452127659563
4489,3.5385329619312906,7539,0.11927460106382977
5071,3.621761658031088,193,0.10284951241134734
5117,2.9408396946564888,262,0.09433333333333323
5173,3.135593220338983,59,0.07658532801418447
5287,3.4925373134328357,201,0.10490070921985808
5345,3.290909090909091,55,0.07947163120567374
5984,3.080246913580247,162,0.09079277482269507
6266,3.130076838638858,1822,0.07702482269503541
```

Figure 107: Query 6 Sample Output

```
fp25_5@clu04:~$ hdfs dfs -cat Query8SampleOutput/part-0000
145736,0.9111192685390336
145968,0.9051797262197254
146000,0.8984246533152983
145789,0.8976348376667815
145748,0.8964248813411382
145835,0.8940172914823321
145862,0.8931740960787709
145761,0.8912261757021839
145756,0.8882426867539974
145896,0.8832734657359496
146040,0.878784595875563
145970,0.8780428376688523
145994,0.8765457918333716
145827,0.8764445417211165
145982,0.8760146422901134
146045,0.8743583190244194
145786,0.8718601827196304
145998,0.870799400966766
145941,0.8695163412915373
145729,0.8692953260478912
145948,0.8692876999467939
146029,0.8692021934969459
145989,0.8689080131385796
145731,0.8688010941075474
145752,0.8687177791740154
145816,0.8685561041538612
145790,0.8683850792821518
145889,0.868002893824857
```

Figure 108: Query 8 Sample Output

```
fp25_5@clu04:~$ hdfs dfs -cat Query10SampleOutput/part-00000-ab525dd
48931,0.8868139644976526
48769,0.8808453615004715
48870,0.8808015654996538
```

Figure 109: Query 10 Sample Output