Project Documentation

Movie Preference Analyzer

Course: Functional Programming, Analytics and Applications - INF 424
Instructor: Nikos Giatrakos
ECE TUC

Students
Karalis Asterinos ID: 2020030107
Niaropetros Emmanouil ID: 2019030168

23 May 2025

Contents

1 Introduction **3**
1.1 About the project . . . 3
1.2 About the data . . . 3
1.3 The Queries . . . 4

2 Code Implementation **6**
2.1 Preprocessing . . . 6
2.2 PartA - RDDs . . . 8
2.2.1 Query 1: Iceberg Query . . . 8
2.2.2 Query 2: Tag Dominance per Genre . . . 10
2.2.3 Query 3: Iceberg . . . 11
2.2.4 Query 4: Sentiment Estimation . . . 13
2.2.5 Query 5: Multi-Iceberg Skyline Over Genre-Tag-User Triads . . . 13
2.3 PartB - DataFrames . . . 15
2.3.1 Query 6: Skyline Query . . . 15
2.3.2 Query 7: Correlation Between Tag Relevance and Average Ratings . . . 16
2.3.3 Query 8: Reverse Nearest Neighbor . . . 16
2.3.4 Query 9: Tag-Relevance Anomaly . . . 18
2.3.5 Query 10: Reverse Top-K Neighborhood . . . 19

3 Cluster Results **20**
3.1 Part A - RDDs . . . 20
3.1.1 PartA-Job4 . . . 21
3.1.2 PartA-Job5 . . . 24
3.1.3 PartA-Job6 . . . 28
3.1.4 PartA-Job7 . . . 30
3.1.5 PartA-Job8 . . . 33
3.2 Part B - DataFrames . . . 38
3.2.1 PartB-Job4 . . . 39
3.2.2 PartB-Job5 . . . 40
3.2.3 PartB-Job6 . . . 43
3.2.4 PartB-Job7 . . . 47
3.2.5 PartB-Job8 . . . 48
3.2.6 PartB-Job9 . . . 52
3.2.7 PartB-Job10 . . . 53
3.2.8 PartB-Job11 . . . 56
3.2.9 PartB-Job12 . . . 64

4 Sample Output **67**
4.1 Output Files . . . 67
4.2 Output File Partitioning . . . 67
4.3 Part A . . . 69
4.3.1 Query 1: Iceberg Query . . . 69
4.3.2 Query 2: Tag Dominance per Genre . . . 70
4.3.3 Query 3: Iceberg . . . 71
4.3.4 Query 4: Sentiment Estimation . . . 72

4.3.5 Query 5: Multi-Iceberg Skyline 74
4.4 Part B 75
4.4.1 Query 6: Skyline Query 75
4.4.2 Query 7: Correlation Between Tag Relevance and Average Ratings 76
4.4.3 Query 8: Reverse Nearest Neighbor 78
4.4.4 Query 9: Tag-Relevance Anomaly 79
4.4.5 Query 10: Reverse Top-K Neighborhood Users 80

5 References **82**

1 Introduction

1.1 About the project

This project was developed as part of the INF424 course, *"Functional Programming, Analytics and Applications"*, during the Spring Semester of the 2024–2025 academic year. The objective was to perform data analytics on the MovieLens dataset using functional programming in Scala, taking advantage of the distributed data platform Apache Spark. The project was divided into two main parts: the first involved implementing analytics using Spark RDDs, while the second focused on using Spark DataFrames.

Both parts involved the implementation of advanced analytics queries, such as Iceberg queries, tag-based sentiment estimation, multi-dimensional skyline computations, and cosine similarity-based user-movie matching. All data preprocessing, transformation, and analysis were executed using Scala's higher-order functions within the Spark framework. The solutions were tested and executed both locally and on the SoftNet cluster, ensuring compliance with scalability and correctness requirements.

This report documents the design decisions, the logic and structure of the implemented code, the Spark job structures, and performance insights obtained during the development of each analytics query.

1.2 About the data

The `ml-latest` version of the MovieLens dataset used for this project consists of the following CSV files:

- **genome-scores.csv** — `(movieId, tagId, relevance)`: Contains tag relevance scores for each movie.
- **genome-tags.csv** — `(tagId, tag)`: Maps tag identifiers to their textual tag descriptions, used in combination with `genome-scores.csv`.
- **links.csv** — `(movieId, imdbId, tmdbId)`: Provides mappings from MovieLens movie IDs to corresponding identifiers in IMDb and TMDb (Not used in any query).
- **movies.csv** — `(movieId, title, genres)`: Contains movie metadata including title and genre labels.
- **ratings.csv** — `(userId, movieId, rating, timestamp)`: Records user ratings on a 5-star scale, with associated timestamps.
- **tags.csv** — `(userId, movieId, tag, timestamp)`: Stores free-text tags applied by users to movies, along with timestamps of when tags were assigned.

You can find the data here: `https://files.grouplens.org/datasets/movielens/ml-latest.zip`

1.3 The Queries

Below is a summary of the queries implemented in the project, including the files used, the objective of each query, and their analytics value.

- **Query 1: Iceberg Query — Top Tags by Genre with High Ratings**
  Files Used: `movies.csv`, `ratings.csv`, `tags.csv`
  Objective: Find genre-tag pairs that appear in more than 100 movies and have an average user rating greater than 4.0.
  Analytics Value: Such tags represent strong user interest and satisfaction for specific genres, valuable for enhancing tag-aware recommendations.

- **Query 2: Tag Dominance per Genre — Most Used Tags with Ratings**
  Files Used: `movies.csv`, `ratings.csv`, `tags.csv`
  Objective: For each movie genre, find the most commonly used tag, and return the average rating of the movies it was used on.
  Analytics Value: This reveals genre-specific user vocabulary and can be used for intelligent tagging, filtering, or explanation generation in recommender systems.

- **Query 3: Iceberg — Tags That Are Both Popular and Relevant**
  Files Used: `genome-scores.csv`, `genome-tags.csv`
  Objective: Find tags that appear in over 100 unique movies and have an average relevance greater than 0.8.
  Analytics Value: This helps identify globally meaningful and interpretable tags that can enhance tag-driven recommendation models.

- **Query 4: Sentiment Estimation — Tags Associated with Ratings**
  Files Used: `tags.csv`, `ratings.csv`
  Objective: Compute the average user rating for each user-assigned tag from `tags.csv`.
  Analytics Value: Identifies tags that imply user satisfaction (or dissatisfaction), useful for explainable AI and recommender trust.

- **Query 5: Multi-Iceberg Skyline Over Genre-Tag-User Triads**
  Files Used: `movies.csv`, `tags.csv`, `ratings.csv`
  Objective: For each (genre, tag) pair that appears in more than 200 movies, calculate the average rating and the number of unique users who applied the tag. Then, compute a skyline over the remaining (genre, tag) pairs, keeping only those not dominated in both rating and user count.
  Analytics Value: Detects the most relevant and popular tags per genre from both a qualitative (rating) and engagement (user count) perspective.

- **Query 6: Skyline Query — Non-Dominated Movies in Multiple Dimensions**
  Files Used: `ratings.csv`, `genome-scores.csv`
  Objective: Identify movies that are not dominated in average rating, rating count, and average tag relevance.
  Analytics Value: Skyline queries highlight movies that are top performers in balanced ways, useful for curated recommendation.

- **Query 7: Correlation Between Tag Relevance and Average Ratings**
  Files Used: `ratings.csv`, `genome-scores.csv`
  Objective: Estimate the Pearson correlation between movies' average genome tag relevance

and their average user rating.
Analytics Value: Helps validate whether tag-based features are predictive of user satisfaction.

- **Query 8: Reverse Nearest Neighbor — Match Users to a Movie's Tag Vector**
  Files Used: `ratings.csv`, `genome-scores.csv`
  Objective: Match users to a movie of your choice by comparing the average tag preferences of their liked movies (rated > 4.0) with the tag profile of the target movie using Cosine Similarity.
  Analytics Value: Enables personalized targeting (e.g., who might like a new movie), supporting content-driven recommendations.

- **Query 9: Tag-Relevance Anomaly — Overhyped Low-Rated Movies**
  Files Used: `genome-scores.csv`, `ratings.csv`, `genome-tags.csv`
  Objective: Identify movies that are highly relevant (higher or equal to 0.8) to popular tags such as "action", "classic", and "thriller", but have very low average user ratings (less than 2.5).
  Analytics Value: Helps detect "overhyped" content with misleading tag associations, improving trust in recommendations.

- **Query 10: Reverse Top-K Neighborhood Users Using Semantic Tag Profiles**
  Files Used: `ratings.csv`, `genome-scores.csv`
  Objective: Given a target movie, find the top-K users whose semantic tag profiles (inferred from high-rated movies) are closest to the movie's tag profile using Cosine Similarity.
  Analytics Value: This query reverses the traditional recommendation flow and suggests users to content, useful for early screenings or targeted marketing.

2 Code Implementation

2.1 Preprocessing

Since the project was originally developed as a single program, the preprocessing stage was shared across both parts (RDD and DataFrame implementations). Because transformations in Spark are not executed until an action is called, defining all preprocessing steps upfront did not incur any performance penalty. Thus, we were able to compute and store all foundational structures (schemas, DataFrames, RDDs) without concern for redundancy or runtime cost, as only the relevant parts would be materialized during execution of each query.

The preprocessing part consists of 3 different parts: `Schema definitions`, `DataFrame declarations` and `RDDs Conversions.` .

1. Schema Definitions:
   Each file is structured differently, so we define custom schemas to parse them correctly into DataFrames(especially for `ratings.csv`,`tags.csv` and `genome-scores.csv`). These schemas specify column names and data types, ensuring Spark interprets the data properly.

   - ratingsSchema: Schema for the `ratings.csv` file. It is important to explicitly define this schema so that Spark can interpret the rating column as Double Type.
   - genomeScoresSchema: Schema for the `genome-Scores.csv` file. It is important to explicitly define this schema so that Spark can interpret the Relevance column as Double Type.
   - tagSchema: Schema for the `tags.csv` file. It is important to explicitly define this schema because we dont want spark to load excess, unusable data in memory(the column `timestamp`).

```
val ratingsSchema = StructType(Seq(
  StructField("UserId", StringType, nullable = false),
  StructField("MovieId", StringType, nullable = false),
  StructField("Rating", DoubleType, nullable = false)
))

val tagSchema = StructType(Seq(
  StructField("UserId", StringType, nullable = false),
  StructField("MovieId", StringType, nullable = false),
  StructField("Tag", StringType, nullable = false)
))

val genomeScoresSchema = StructType(Seq(
  StructField("MovieId", StringType, nullable = false),
  StructField("TagId", StringType, nullable = false),
  StructField("Relevance", DoubleType, nullable = false)
))
```

2. DataFrame Initialization:
   We load the CSV files into Spark DataFrames using the paths provided. The use of .option() ensures headers are parsed and special characters (like quotes and commas inside strings) are handled correctly.

```
//Defining Dataframes
val moviesFileDF = spark.read
  .option("header", "true")
```

```
  .option("quote", "\"")
  .option("escape", "\"")
  .csv(moviesPathHDFS)

val ratingsFileDF = spark.read
  .option("header", "true")
  .schema(ratingsSchema)
  .csv(ratingsPathHDFS)

val tagsFileDF = spark.read
  .option("header", "true")
  .option("quote", "\"")
  .schema(tagSchema)
  .csv(tagsPathHDFS)

val genomeScoresDF = spark.read
  .option("header", "true")
  .schema(genomeScoresSchema)
  .csv(genomeScoresPathHDFS)

val genomeTagsDF = spark.read
  .option("header","true")
  .csv(genomeTagsPathHDFS)
```

3. RDD Initialization and Conversions:
   Since some queries require the usage of RDDs, we are also creating the RDD equivelants of the files. On the data files that have unique schemas and peculiarities on the cluster, we need to make the rdds by using their respective dataframes and using the utility `.rdd`. Other than that, we create RDDs with the use of `spark.context.textFile()`:

```scala
//movies.csv
//We need to make the rdd this way for this file because some movies have commas
val moviesRDD = moviesFileDF.select(
    col("movieId").alias("MovieId"),
    col("title").alias("Title"),
    col("genres").alias("Genre")
  ).rdd
  .map(row => (row.getAs[String]("MovieId"), row.getAs[String]("Title"), row.getAs[String]("Genre").split("\\|")))

//ratings.csv
val ratingsRDD = ratingsFileDF
  .rdd
  .map(row => (row.getString(0), row.getString(1), row.getDouble(2)))


//tags.csv
//We need to make the rdd this way for this file because some tags have commas
val tagsRDD = tagsFileDF
  .select("UserId", "MovieId", "Tag")
  .rdd
  .map(row => (row.getAs[String]("UserId"), row.getAs[String]("MovieId"), row.getAs[String]("Tag"))) //(userId, movieId, tag)

//genome-scores.csv
val rawGenomeScoresRDD = sc.textFile(genomeScoresPathHDFS)
val genomeScoresHeader = rawGenomeScoresRDD.first()

val genomeScoresRDD = genomeScoresDF
  .rdd
  .map(row => (row.getString(0), row.getString(1), row.getDouble(2)))

//genome-tags.csv
val rawGenomeTagsRDD = sc.textFile(genomeTagsPathHDFS)
val genomeTagsHeader = rawGenomeTagsRDD.first()

val genomeTagsRDD = rawGenomeTagsRDD
  .filter(row => row != genomeTagsHeader)          //skip the csv headers.
  .map(line => line.split(",",2))                  //break the string into its components
  .map(fields => (fields(0), fields(1)))           //(tagId,tag)
```

2.2 PartA - RDDs

2.2.1 Query 1: Iceberg Query

For the first query, we took the ratingsRDD, we transformed it so that the movieId is the key. Then, we use the transformations `.reduceByKey()` and then `.mapValues`, resulting in the `avgRatingPerMovie` RDD that has all the movieIds and their respective average user rating(`(movieId, avg(Rating))`)

After calculating the average rating per movie, we transform both tagsRDD and moviesRDD and make movieId their key, so that we can join them together, resulting in the `tagGenreJoinedWithMovies` RDD where every tadId-genre pair is mapped to every movie it appeared to. We then created the `movieByTagGenreRating` RDD by joining `tagGenreJoinedWithMovies` and `avgRatingPerMovie`, so

now we have an RDD that looks like this `(movieId ,((Tag,Genre),avgMovieRating))`. We do the necessary transformations so that we have an RDD in the form of `((Tag, Genre), avgMovieRating)`, we then use `.groupByKey()` so now we have every tag, genre pair mapped to a list with the ratings of all the movies that are associated with those pairs. Moreover, we need to filter out the genre,tag pairs that don't associate with more than 100 movies(or their ratings).
Finally, for each (genre, tag) pair, we calculate the average of the ratings contained in the corresponding list, yielding the final result stored in `icebergResults`.

```
1  //Finding the average Rating per movie
2  val avgRatingPerMovie = ratingsRDD                                    //
       avgRatingPerMovie -> (movieId, avg(Rating))
3    .map { case (userId, movieId, rating) => (movieId, (rating, 1)) }   // (movieId
       ,(rating, 1))
4    .reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2))                  // sum all
       the ratings and all the counts respectively
5    .mapValues { case (sum, count) => sum / count }                     // compute
       the average (movieId, average)
6
7  // Getting movieId - tag pairs (dropping the userId)
8  val movieTagPairs = tagsRDD                                  //movieTagPairs -> (
       movieId,tag)
9    .map { case (userId, movieId, tag) => (movieId, tag) }
10
11 // Getting movieId - Genre pairs
12 val movieGenrePairs = moviesRDD                               //movieGenrePairs ->
       (movieId,genre)
13   .flatMap { case (movieId, title, genres) =>
14     genres
15       .filter(genre => genre != "(no genres listed)")
16       .map( genre => (movieId, genre))
17   }
18
19 //Join movieId - tag pairs with movieId - Genre pairs on movieId
20 val tagGenreJoinedWithMovies = movieTagPairs       //tagGenreJoinedWithMovies ->
       (movieId, (Tag,Genre))
21   .join(movieGenrePairs)
22   //.distinct()
23
24 val movieByTagGenreRating = tagGenreJoinedWithMovies     //movieByTagGenreRating
       -> (movieId ,((Tag,Genre),avgMovieRating))
25   .join(avgRatingPerMovie)
26
27 val tagGenreAndRatings = movieByTagGenreRating
       //tagGenreAndRatings -> (Tag,Genre),List(avg_rating_of_each_movie_associated)
28   .map{ case(movieId, ((tag,genre),avgRating)) => ((tag,genre), avgRating)}
       //((Tag,Genre),avgMovieRating)
29   .groupByKey()
       //group by tag-genre pair
30
31 val filteredGroupByTagGenrePair = tagGenreAndRatings             //
       filteredGroupByTagGenrePair -> filtered((Tag,Genre),List(
       avg_rating_of_each_movie_associated))
32   .filter(_._2/*.toSet*/.size > 100)                          //filter out the
       tag-genre pairs that have less than 100 average movie ratings associated
       with them(and thus movies)
33
34
35 val icebergResults = filteredGroupByTagGenrePair
```

```
  .mapValues(list => {
    list
      .map( rating => (rating,1))
      .reduce((x,y) => (x._1 + y._1, x._2 + y._2))        //Compute the sum and
         count of the rating associated with each pair
  })
  .mapValues(tuple => tuple._1 / tuple._2)                 //compute average
  .filter(tuple => tuple._2 > 4.0)                         //filter out the pairs
     that  have an average lower than 4.0
```

2.2.2 Query 2: Tag Dominance per Genre

For this Query, we first need to find every tag-genre pair associated with each movie. Fortunately, this mapping was already defined in Query 1 via the tagGenreJoinedWithMovies value. Although the tagGenreJoinedWithMovies value was originally defined in Query 1, it is reused here to avoid code duplication. However, due to Spark's lazy evaluation model, this value will be recomputed when accessed again because it has not been explicitly cached or persisted using methods like .cache() or .persist(). Starting from the tuple `(movieId, (Tag, Genre))` obtained from `tagGenreJoinedWithMovies`, we transform it into the form `((Genre, Tag), 1)`. This intermediate representation allows us to count the occurrences of each unique genre-tag pair, resulting in the value `genreTagCounts`, which reflects how frequently each tag appears within each genre. After that, we reshape the tuple so the key is now the genre `((Genre),(Tag, count))`, and we apply the `reduceByKey()` transformation so that we keep only the tag with the highest count per genre. This results in the value `mostUsedTagPerGenre`.

```
val genreTagCounts = tagGenreJoinedWithMovies              //We get the (movieId,(
   Tag,Genre))
  .map { case(movieId,(tag,genre)) =>
    ((genre,tag), 1)                                       //we transform them to ((
       Genre,Tag), 1)
  }
  .reduceByKey(_ + _)                                      //we count the instances
     of each pair


val mostUsedTagPerGenre = genreTagCounts                   //mostUsedTagPerGenre ->
   (genre, (tag, count)), count = max, tag = most popular
  .map{ case((genre,tag),count) =>
    (genre, (tag, count))                                  // Make genre the key
  }
  .reduceByKey { (a, b) => if (a._2 >= b._2) a else b } // Pick tag with the
     highest count for each genre
```

We then reshape both `tagGenreJoinedWithMovies` and `mostUsedTagPerGenre` so that they both have as key, the tuple `(Genre, Tag)`. We are now ready to join them together so that we can find the movies where each dominant `(Genre, Tag)` appears. After some more reshaping(throwing out the `counts` element and making the element `movieId` the key), we join the resulting RDD (`filteredMovieTagGenre`) with the `avgRatingPerMovie` RDD. This gives us a new mapping in which each `movieId` is now associated with both its genre-tag pair and its average rating.Lastly, we reshape the `movieGenreRatings` RDD—which has the structure `(movieId, ((genre, tag), avgRating))`—into the form `((genre, tag), (rating, 1))`. This format prepares the data for aggregation, allowing us to compute average ratings per genre-tag pair. This aggregation is performed using the `reduceByKey()` and `mapValues()` transformations, resulting in the `avgRatingPerGenreTag`

RDD.

```
val tagGenrePairsByMovie  = tagGenreJoinedWithMovies
  .map { case (movieId, (tag, genre)) =>                //we transform from (movieId, (tag, genre))
    ((genre,tag), movieId)                              //to ((genre,tag), movieId)
  }

val dominantTagGenre = mostUsedTagPerGenre
  .map {
    case (genre, (tag, count)) => ((genre, tag), count) //we make the key to be (genre,tag) so we can join with tagGenrePairsByMovie
  }

val filteredMovieTagGenre = tagGenrePairsByMovie
  .join(dominantTagGenre)                                                   // ((genre, tag), (movieId, count))
  .map { case ((genre, tag), (movieId, _)) => (movieId, (genre, tag)) } //we now have the movies of each pair

val movieGenreRatings = filteredMovieTagGenre
  .join(avgRatingPerMovie)  // (movieId, ((genre, tag), avgRating))     //get the avg rating of each movie

val genreRatingPairs = movieGenreRatings
  .map { case ( _, ( (tag,genre), rating)) =>
    ((genre,tag), (rating, 1))                                  // Convert to ((genre,tag), (rating, 1)) for averaging
  }

val avgRatingPerGenreTag = genreRatingPairs
  .reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))            // Sum ratings and counts
  .mapValues(x => x._1 / x._2)                                  // Compute average rating per genre
```

2.2.3 Query 3: Iceberg

Query 3 is slightly more nuanced, as it does not utilize `tags.csv`, but instead relies solely on `genome-scores.csv` and `genome-tags.csv` to extract tag relevance information.The `genome-scores.csv` file contains relevance scores for every `(movieId, tagId)` pair, regardless of how meaningful the association actually is. Therefore, it is necessary to filter out low-relevance tags in order to focus on semantically significant ones. A natural approach is to apply a relevance threshold. While 0.5 might seem intuitive as a cutoff, we needed to test that hypothesis. To avoid noisy or weak tag associations from the `genome-scores.csv` file, we experimented with different relevance thresholds and recorded two key metrics:

- Number of tags that appear in more than 100 movies — to ensure the tags are widely used.
- Number of tags whose average relevance across movies exceeds 0.8 — to confirm their overall semantic strength.

Relevance Threshold	**# Tags in** >100 **Movies**	**# Tags with avg(Relevance)** >0.8
0.80	448	448
0.75	536	532
0.70	615	507
0.65	678	346
0.60	733	232
0.55	791	123
0.50	840	59
0.45	882	30
0.40	926	15

Total Tags: 1128

From the table, we can see that as the threshold decreases, more tags are included — but many of these have low average relevance, meaning they may not be semantically meaningful. We chose a threshold of 0.6 because we felt that the amount of tags that were left after the filtering(232) was neither too low nor too high in respect to the total tags(1128). This suggests that our choice of threshold did not lead to either an overestimation or underestimation of the number of semantically relevant tags. Even though a relevant threshold of 0.5 might also seem good choice from the table, a slightly higher threshold helps ensure that only genuinely representative tags are considered.

Back in our query, we transformed the `genomeScoresRDD` so that the tagId is the key and then we used the transformation `groupByKey()` and our RDD `genomeScoresGrouped` now looks like this `(tagId,list(movieId,relevance))`. Then, we filtered the RDD by removing all `(movieId, relevance)` pairs inside the lists where the relevance score was below 0.6 and we kept only the tags that appear in more than 100 movies. After that, we computed the average relevance for each tagId and excluded any tag whose average relevance fell below 0.8. Lastly, we joined the `filteredTags` RDD with `genomeTagsRDD` in order to map each `tagId` to its corresponding tag name. After this join, we transformed the resulting RDD into the form `(tag, avg_relevance)`, effectively producing a list of meaningful and popular tags based on their average relevance(`finalFilteredTags` RDD).

Below is the code used for the above described query:

```
val genomeScoresGrouped = genomeScoresRDD
  .map(tuple => (tuple._2,(tuple._1,tuple._3))) //(tagId,(movieId,relevance))
  .groupByKey()                                 //(tagId,list(movieId,relevance))
    )

 val filteredTags = genomeScoresGrouped
  .mapValues{iterable => iterable               //go inside the list and for
     each (movieId,relevance) element,
    .filter(_._2 > 0.6)                         //filter out those with
      relevance < 0.6 (filtering around 0.4-0.6 is good)
  }
  .filter(_._2.map(_._1).toSet.size > 100)       //count the unique movies
     associated with each tag and filter out those that have < 100 movies
  .mapValues{list=> list                         //go inside the list again and
     for each (movieId,relevance) element,
    .map(tuple =>(tuple._2, 1))                  //keep only the tuple (
      relevance, 1)
    .reduce((x,y) => (x._1 + y._1, x._2 + y._2)) //sum the relevance and the
      counts
  }
  .mapValues{case (sum,count) => sum/count}      //compute the average
     relevance
```

```
15    .filter(_._2 > 0.8)                                    //filter out whole tuples
          where avg_relevance < 0.8. filteredTags -> (tagId, avg_relevance)
16
17
18  val finalFilteredTags = genomeTagsRDD                    //genomeTagsRDD -> (tagId,tag)
19    .join(filteredTags)                                    //filteredTags -> (tagId,
          avg_relevance), join by tagId
20    .map(tuple => tuple._2)                                //finalFilteredTags -> (tag,
          avg_relevance)
```

Looking back at the code, one small optimization we could have applied is to filter the records based on the relevance threshold *before* performing the `.groupByKey()` transformation. This would reduce the number of elements being shuffled and grouped, which is generally more efficient. However, in our case, this query already executes very quickly, so the impact of such an optimization on performance would likely be minimal.

2.2.4 Query 4: Sentiment Estimation

Here we begin by taking the tagsRDD and converting them using map to key-value pairs where key is (userId, movieId) and the value is the tag and save it in tagByUserMovie. Then we repeat the same for ratingsRDD, using map to convert to key-value paris where key is (userId, movieId) and the value is the rating. After that we join these two vals, we throw away the (userId, movieId) and replace the (tag, rating) with (tag, (rating, 1)). What we have here is every tag that appears with every rating its movies got, therefore we can calculate the average rating with the last two lines. The result is saved in the averageOfEachTag.

```
1  val tagByUserMovie = tagsRDD     //tagsRDD ->(userId, movieId, tag)
2    .map(t => ((t._1, t._2, t._3))) // (userId, movieId, tag)
3    //.distinct()
4    //we can put distinct here if we don't want to compute duplicate tags by sam
5    //user(if same user spams the same tag multiple times)
6  .map(tuple3 => ((tuple3._1, tuple3._2), tuple3._3))   //((userId, movieId), tag)
7
8  val ratingByUserMovie = ratingsRDD    //ratingsRDD -> (userId, movieId, rating)
9    .map(tuple3 => ((tuple3._1, tuple3._2), tuple3._3))   //((userId, movieId),
         rating)
10
11 val averageOfEachTag = tagByUserMovie
12   .join(ratingByUserMovie)   //( (userId, movieId),(tag, rating) )
13   .map {case (_, (tag, rating)) => (tag, (rating, 1))}
14   //(tag, (rating, 1)), we don't need userId,movieId now
15   .reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2))
16   //compute sum of ratings and the count of those ratings associated with each
         tag
17   .mapValues { case (total, count) => total / count}
18   //compute the average rating associated with each tag
```

2.2.5 Query 5: Multi-Iceberg Skyline Over Genre-Tag-User Triads

To calculate the genre-tag pairs we will use the tagGenreAndRatings from Query 1 and simply filter those entries where the amount of movie ratings is greater than 200.

```
1  val popularGenreTagPairsAndRatings=tagGenreAndRatings
2  //tagGenreAndRatings ((Tag,Genre),List(avg_rating_of_each_movie_associate))
```

```
    .filter(_._2.toSet.size > 200)                          //filter out the tag-genre pair if it is associated with less than 200 unique movies
    .mapValues {list =>
      list
        .map(rating => (rating, 1))
        .reduce((x, y) => (x._1 + y._1, x._2 + y._2))      //Compute the sum and count of the rating associated with each pair
    }
    .map{case((tag,genre),tuple) => ((genre,tag),tuple._1 / tuple._2)}
```

We then compute the user count for each genre-tag pair. This is achieved by first reformatting the tagsRDD to a (movieId, (userId, tag)) format (userTagByMovie), which is joined with movieGenrePairs from Query 1, to associate each tag with its genre. The result is transformed into ((genre, tag), userId) pairs and deduplicated using .distinct() to count only unique users per genre-tag. The .groupByKey() and .mapValues(_.size) produce the final count of unique users per (genre, tag) combination as userByGenreTag.

```
val userTagByMovie = tagsRDD
    .map{ case(userId, movieId, tag) =>
      (movieId,(userId,tag))   //(movieId,(userId,tag))
    }

val userByGenreTag = userTagByMovie//userByGenreTag-> (genre,tag),userCount))
    .join(movieGenrePairs)      //movieGenrePairs -> (movieId,genre)
    .map{ case (movieId, ((userId,tag),genre)) =>
      ((genre,tag), userId)       //(genre,tag), userId)
    }
    .distinct()     //remove duplicate user (for same genre-tag pair)
    .groupByKey()   //group by genre-tag pair
    .mapValues(_.size)  //count the userIds therefor the unique users
```

The next step joins the popularGenreTagPairsAndRatings (which contains the average rating of popular genre-tag pairs) with userByGenreTag to produce avgRatingAndUsersByGenreTag, an RDD containing((genre, tag), (`avg_rating`, `user_count`)). This RDD includes both metrics we need to evaluate dominance in the skyline.

```
val avgRatingAndUsersByGenreTag = popularGenreTagPairsAndRatings
    .join(userByGenreTag)
    .repartition(200) //this is for the cluster
```

To compute the skyline, we apply a Cartesian product on avgRatingAndUsersByGenreTag with itself. This results in every possible pairing of genre-tag pairs. Each pair (a, b) is then filtered with a dominance condition which checks whether pair B dominates pair A in both dimensions (avg rating and user count), and is strictly better in at least one. Only the dominated pairs `(a._1)` are kept from this process. To eliminate redundancy (since many B's may dominate the same A), we call .distinct() on the result. Finally, we subtract the set of dominated keys from the original set of pairs in avgRatingAndUsersByGenreTag, using .subtractByKey(). This leaves us with only the non-dominated pairs.

```
val skylineRDD = avgRatingAndUsersByGenreTag
   .cartesian(avgRatingAndUsersByGenreTag)
   .filter { case (a, b) => a._1 != b._1 }  //we remove the duplicates.
   .filter { case ((_, (ratingA, userA)), ((_, (ratingB, userB)))) =>
     (ratingB >= ratingA && userB >= userA) && (ratingB > ratingA || userB > userA)
   }
```

```
    .map { case (a, _) => (a._1, a._2) }
    .distinct()

val finalSkylineRDD = avgRatingAndUsersByGenreTag.subtractByKey(skylineRDD)
```

Due to the calculation of the Cartesian product being done over many entries, this is by far the most time consuming query in Part A

2.3 PartB - DataFrames

2.3.1 Query 6: Skyline Query

First we have to compute the 3 categories in which we will perform the comparisons for the skyline.

We start by calculating the average rating and the rating count per movie using the ratingsFileDF and save them in movieRatings (movieId, avg rating, ratings count). We do this by taking the ratingsFileDF and select the columns MovieId and Rating. Then we groupBy the MovieId so we have all the ratings for each movie. Now we perform 2 aggregate functions ,first is avg to calculate the avg ratings and the other is count to find the ratings count.

```
val movieRatings = ratingsFileDF
//movieRatings -> |MovieId|avg_rating|rating_count|
    .select(col("MovieId"), col("Rating"))
    .groupBy("MovieId")
    .agg(
      avg("Rating").as("avg_rating"),
      count("Rating").as("rating_count")
    )
```

Then, we calculate the avg tag relevance by using the genomeScoresDF and save them in movieRevelance. We select again the MovieId and Revelance columns and groupBy the MovieId, then using aggregate function avg we take the avg tag relevance.

```
val movieRelevance = genomeScoresDF
    //movieRelevance -> |MovieId|avg_relevance|
    .select(col("MovieId"), col("Relevance"))
    .groupBy("MovieId")
    .agg(avg("relevance").as("avg_relevance"))
```

Finally, we join movieRevelance and movieRatings as movieStats to get our final matrice. In order to perform skyline, we have to compare each entry in our matrice with everyone else, so we will perform a ”self-join”. We do that by creating two aliases of the matrice calling them ”a” and ”b”, saving them in vals statsA and statsB respectively, and perform comparisons which return a column saved in dominationCondition where each entry tells if it is a dominated movie (comparisons returned True) or not dominated (comparisons returned False).

```
val movieStats = movieRatings.join(movieRelevance, "MovieId")

val statsA = movieStats.alias("a")
val statsB = movieStats.alias("b")

val dominationCondition =
    (col("b.avg_rating") >= col("a.avg_rating")) &&
    (col("b.rating_count") >= col("a.rating_count")) &&
    (col("b.avg_relevance") >= col("a.avg_relevance")) &&
```

```
      (
        (col("b.avg_rating") > col("a.avg_rating")) ||
        (col("b.rating_count") > col("a.rating_count")) ||
        (col("b.avg_relevance") > col("a.avg_relevance"))
      )
```

Then we perform a "left anti" join of "b" on "a" using the dominationCondition as filter, which means that the output will be all the entries of "a" where "b" did not match based on the condition, therefore excluding all dominated movies and leaving us with the final skyline.

```
val skylineDF = statsA.join(statsB, dominationCondition, "left_anti")
```

2.3.2 Query 7: Correlation Between Tag Relevance and Average Ratings

For the purpose of Query 7, we begin by calculating two key metrics for each movie: the average user rating and the average genome tag relevance. This is done by grouping the respective DataFrames - `ratingsFileDF` and `genomeScoresDF` - by `movieId`, and applying the `.agg(avg(...))` aggregation function to compute the averages.

```
val avgRatingPerMovieDF = ratingsFileDF                    //ratingsFileDF -> |UserId|
    MovieId|Rating|
  .groupBy(col("MovieId"))
  .agg(avg("Rating").alias("avg_rating_per_movie"))    //avgRatingPerMovieDF -> |
      MovieId|avg_rating_per_movie|

val avgTagRelevancePerMovie = genomeScoresDF                          //
    genomeScoresDF -> |movieId|tagId|relevance|
  .groupBy(col("MovieId"))
  .agg(avg("Relevance").alias("avg_tag_relevance_per_movie"))      //
      avgTagRelevancePerMovie -> |MovieId|avg_tag_relevance_per_movie|
```

All we need to do next is perform an inner join on the two tables using `movieId` as the key, storing the result in a new DataFrame called `joinedResult`. From there, we can compute the Pearson correlation coefficient between the two columns `| avg_rating_per_movie` and `avg_tag_relevance_per_movie |` using the `.stat.corr()` method.

```
val joinedResult = avgRatingPerMovieDF
  .join(avgTagRelevancePerMovie, Seq("MovieId"), "inner")          //joinedResult
      -> |MovieId|avg_rating_per_movie|avg_tag_relevance_per_movie|

val correlation = joinedResult.stat.corr("avg_rating_per_movie", "
    avg_tag_relevance_per_movie")
```

2.3.3 Query 8: Reverse Nearest Neighbor

For the implementation of this query, we first selected a target movie — in this case, `Inception` (with `movieId = "79132"`) — and retrieved its tag profile. This was done by selecting the `TagId` and `Relevance` columns from the `genomeScoresDF` dataframe, filtered(`.where()` transformation) to include only rows corresponding to the chosen `movieId`. After we obtain the DataFrame `likedMoviesDF`, which contains all `(UserId, MovieId)` pairs where the user has rated the movie above 4.0, we join it with `genomeScoresDF`. This join allows us to associate each liked movie with its tag relevance scores. The resulting DataFrame, `joinedByMovies`, has the structure:

`(UserId, MovieId, TagId, Relevance)`

Next, we calculate the average tag relevance score for each user across all the movies they liked. We repartition the data by `UserId` to ensure because the next transformations (`.groupBy()` and aggregation), need to be run efficiently because there is a lot of gigabytes of data shuffling. After the repartition, we group by both `UserId` and `TagId`, and compute the average of the relevance values:

`(UserId, TagId, avg_tag_relevance_per_user)`

This is stored in the `avgTagRelevancePerUser` DataFrame. Before joining with the movie profile, we repartition again by `TagId` to prepare for the next join, since this groupBy is presumably computationally taxing too.
We then join this DataFrame with the tag profile of our chosen movie (Inception, ID `"79132"`), which contains the relevance scores of tags for that movie. This gives us:

`(UserId, TagId, user_score, target_score)`

where `user_score` represents the user's preference for a tag (average relevance), and `target_score` is how relevant that tag is to the chosen movie. We again repartition by `UserId` for better parallelism. Next, we compute the components required for cosine similarity:

- The dot product of the user and target vectors,
- The squared norm of the user vector,
- The squared norm of the target vector.

We perform these computations using new columns created via `.withColumn()`, followed by a `.groupBy("UserId")` and `.agg()` to sum them. The cosine similarity is then calculated using:

$$\texttt{cosine\_similarity} = \frac{\texttt{dot\_product}}{\sqrt{\texttt{user\_norm\_sqr}} \cdot \sqrt{\texttt{target\_norm\_sqr}}}$$

Finally, we select only the columns `UserId` and `cosine_similarity` and use the `.persist()` action on the resulting DataFrame, so it can be efficiently reused later in Query 10.

```
val chosenMovieID = "79132" //Inception

val tagRelevanceOfChosenMovie = genomeScoresDF   //tagRelevanceOfChosenMovie -> |TagId|Relevance| where MovieId == chosenMovieID
  .select(col("TagId"), col("Relevance"))
  .where(col("MovieId") === chosenMovieID)       // This is the tag profile of the chosen movie

val likedMoviesDF = ratingsFileDF          //likedMoviesDF -> |UserId|MovieId| of movies rated by user > 4.0 (user's liked movies)
  .select(col("UserId"), col("MovieId"))
  .where(col("Rating") > 4.0)              //filter only the liked movies of each user (Rating > 4.0)

val joinedByMovies = likedMoviesDF         //joinedByMovies -> |MovieId|UserId|TagId|Relevance|
  .join(genomeScoresDF, Seq("MovieId"))

val avgTagRelevancePerUser = joinedByMovies     //avgTagRelevancePerUser -> |UserId|TagId|avg_tag_relevance_per_user|
  .repartition(400, col("UserId"))
```

```
  .groupBy(col("UserId"), col("TagId"))
  .agg(avg("Relevance").alias("avg_tag_relevance_per_user"))  // Compute the
     average of each tag relevance of all user's liked movies
  .repartition(400, col("TagId"))

val tagProfilesJoined = avgTagRelevancePerUser    //tagProfilesJoined -> |UserId|
   TagId|user_score|target_score|
  .join(tagRelevanceOfChosenMovie, Seq("TagId"))
  .select(
    col("UserId"),
    col("TagId"),
    col("avg_tag_relevance_per_user").alias("user_score"),    //Some renaming
    col("Relevance").alias("target_score")                    //Some renaming
  )
  .repartition(400, col("UserId"))

val cosineComponentsDF = tagProfilesJoined
  .withColumn("dot", col("user_score") * col("target_score"))
     //Create a new column where product between the user score and target score
     is calculated
  .withColumn("user_norm_sqr", col("user_score") * col("user_score"))
     //Create a new column where the square of user score is calculated
  .withColumn("target_norm_sqr", col("target_score") * col("target_score"))
     //Create a new column where the square of target score is calculated
  .groupBy(col("UserId"))
     //Group By UserId
  .agg(
     //Sum each column with itself
    sum("dot").alias("dot_product"),
    sum("user_norm_sqr").alias("user_norm_sqr"),
    sum("target_norm_sqr").alias("target_norm_sqr")
  )
  .withColumn("cosine_similarity",
     //Compute the Cosine Similarity
    col("dot_product") / (sqrt(col("user_norm_sqr")) * sqrt(col("target_norm_sqr"
       )))
  )
     //Until now ->|UserId|dot_product|user_norm_sqr|target_norm_sqr|
     cosine_similarity|
  .select(col("UserId"),col("cosine_similarity"))
     //cosineComponentsDF -> |UserId|cosine_similarity|
  .persist()
```

2.3.4 Query 9: Tag-Relevance Anomaly

This is a simple one. First we will take the tag ids of the famous tags(`action, classic, thriller`) and save them in famous tags. Then we will join the genomeScoresDF based on the TagId and groupBy MovieId. Now we have for each movie those 3 tags and their relevance, therefore we can get their average, from which filter those that are greater or equal 0.8 . Finally, we simply join the avgRatingPerMovieDF from Query 7 on MovieId and filter the movies that have an avg rating less than 2.5.

```
val famous_tags = genomeTagsDF
//we filter only the tagIds for the desired tags
   .filter(col("Tag").isin("action", "classic", "thriller"))

```

```
val overhypedMovies = famous_tags.join(genomeScoresDF, "TagId")
//we join the genome scores table to the filtered tags
  .groupBy("MovieId")
  //then calculate the average relevance of each movie
  .agg(avg("Relevance").as("avg_relevance_to_tags"))
  .filter(col("avg_relevance_to_tags")>= 0.8 )
  .join(avgRatingPerMovieDF, "MovieId")
  //join the average ratings table
  .filter(col("avg_rating_per_movie") < 2.5)
  //filter those with average rating lower than 2.5
```

2.3.5 Query 10: Reverse Top-K Neighborhood

For the last query, we observed that most of what we need to do, is already being done by query 8. Because query 8 is very computationally taxing, we used the `.persist` action on `cosineComponentsDF` so that it does not need to be recomputed again for query 10. We also thought of changing the order of the queries so that query 9 is not being run while a big dataframe(`cosineComponentsDF`) is being cached. Lastly, in order to complete the implementation of the query, all we need to do is use the `.orderBy()` transformation and `.limit(topk)` so that only the top-k results are shown. To comply precisely with the query specifications, we added an extra column indicating the rank of each result.

```
val topK = 10
val topKUsers = cosineComponentsDF
  .orderBy(col("cosine_similarity").desc)
  .limit(topK)
val topKRanked = topKUsers.withColumn("Rank", row_number().over(Window.orderBy(
  col("cosine_similarity").desc))) //adding rank column
```

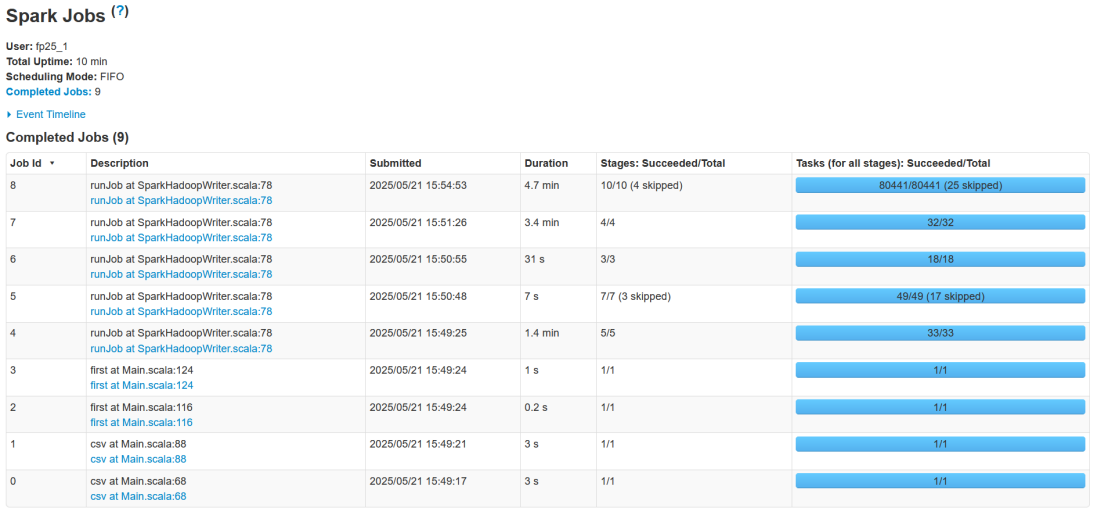
3 Cluster Results

All the results presented below were obtained from executing the JAR files on the TUC cluster. You can review them by visiting:

- PartA-RDDs: `http://clu01.softnet.tuc.gr:18081/history/application_1747035769145_0186/1/jobs/`
- PartB-Dataframes: `http://clu01.softnet.tuc.gr:18081/history/application_1747035769145_0168/1/jobs/`

3.1 Part A - RDDs

First we have all the jobs for PartA.



Spark Jobs (?)

User: fp25_1
Total Uptime: 10 min
Scheduling Mode: FIFO
Completed Jobs: 9

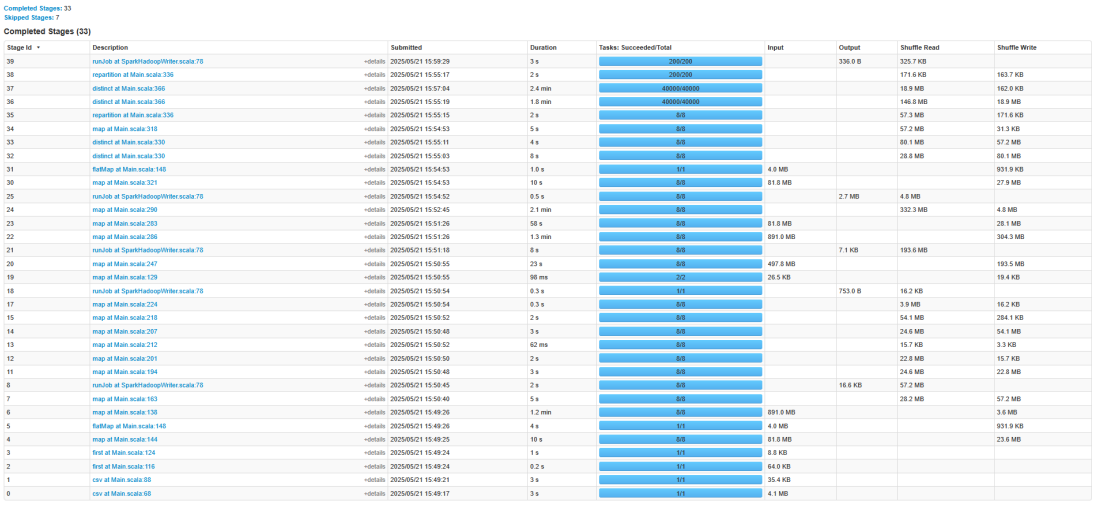
▸ Event Timeline

Completed Jobs (9)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/21 15:54:53	4.7 min	10/10 (4 skipped)	80441/80441 (25 skipped)
7	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/21 15:51:26	3.4 min	4/4	32/32
6	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/21 15:50:55	31 s	3/3	18/18
5	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/21 15:50:48	7 s	7/7 (3 skipped)	49/49 (17 skipped)
4	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/21 15:49:25	1.4 min	5/5	33/33
3	first at Main.scala:124 first at Main.scala:124	2025/05/21 15:49:24	1 s	1/1	1/1
2	first at Main.scala:116 first at Main.scala:116	2025/05/21 15:49:24	0.2 s	1/1	1/1
1	csv at Main.scala:88 csv at Main.scala:88	2025/05/21 15:49:21	3 s	1/1	1/1
0	csv at Main.scala:68 csv at Main.scala:68	2025/05/21 15:49:17	3 s	1/1	1/1

Figure 1: Spark Jobs view for Part A execution (RDD-based queries).

Then we have all the stages for PartA.



Completed Stages: 33
Skipped Stages: 7

Completed Stages (33)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
39	runJob at SparkHadoopWriter.scala:78	+details	2025/05/21 15:59:29	3 s	200/200		336.0 B	325.7 KB	
38	repartition at Main.scala:336	+details	2025/05/21 15:55:17	2 s	200/200			171.6 KB	163.7 KB
37	distinct at Main.scala:366	+details	2025/05/21 15:57:04	2.4 min	40000/40000			18.9 MB	162.0 KB
36	distinct at Main.scala:366	+details	2025/05/21 15:55:19	1.8 min	40000/40000			146.8 MB	18.9 MB
35	repartition at Main.scala:336	+details	2025/05/21 15:55:15	2 s	8/8			57.3 MB	171.6 KB
34	map at Main.scala:318	+details	2025/05/21 15:54:53	5 s	8/8			57.2 MB	31.3 KB
33	distinct at Main.scala:330	+details	2025/05/21 15:55:11	4 s	8/8			80.1 MB	57.2 MB
32	distinct at Main.scala:330	+details	2025/05/21 15:55:03	8 s	8/8			28.8 MB	80.1 MB
31	flatMap at Main.scala:148	+details	2025/05/21 15:54:53	1.0 s	1/1	4.0 MB			931.9 KB
30	map at Main.scala:321	+details	2025/05/21 15:54:53	10 s	8/8	81.8 MB			27.9 MB
25	runJob at SparkHadoopWriter.scala:78	+details	2025/05/21 15:54:52	0.5 s	8/8		2.7 MB	4.6 MB	
24	map at Main.scala:290	+details	2025/05/21 15:52:45	2.1 min	8/8			332.3 MB	4.8 MB
23	map at Main.scala:283	+details	2025/05/21 15:51:26	58 s	8/8	81.8 MB			28.1 MB
22	map at Main.scala:286	+details	2025/05/21 15:51:26	1.3 min	8/8	891.0 MB			304.3 MB
21	runJob at SparkHadoopWriter.scala:78	+details	2025/05/21 15:51:18	8 s	8/8		7.1 KB	193.6 MB	
20	map at Main.scala:247	+details	2025/05/21 15:50:55	23 s	8/8	497.8 MB			193.5 MB
19	map at Main.scala:129	+details	2025/05/21 15:50:55	96 ms	2/2	26.5 KB			19.4 KB
18	runJob at SparkHadoopWriter.scala:78	+details	2025/05/21 15:50:54	0.3 s	1/1		753.0 B	16.2 KB	
17	map at Main.scala:224	+details	2025/05/21 15:50:54	0.3 s	8/8			3.9 MB	16.2 KB
15	map at Main.scala:218	+details	2025/05/21 15:50:52	2 s	8/8			54.1 MB	284.1 KB
14	map at Main.scala:207	+details	2025/05/21 15:50:48	3 s	8/8			24.6 MB	54.1 MB
13	map at Main.scala:212	+details	2025/05/21 15:50:52	62 ms	8/8			15.7 KB	3.3 KB
12	map at Main.scala:201	+details	2025/05/21 15:50:50	2 s	8/8			22.8 MB	15.7 KB
11	map at Main.scala:194	+details	2025/05/21 15:50:48	3 s	8/8			24.6 MB	22.8 MB
8	runJob at SparkHadoopWriter.scala:78	+details	2025/05/21 15:50:45	2 s	8/8		16.6 KB	57.2 MB	
7	map at Main.scala:163	+details	2025/05/21 15:50:40	5 s	8/8			28.2 MB	57.2 MB
6	map at Main.scala:136	+details	2025/05/21 15:49:26	1.2 min	8/8	891.0 MB			3.6 MB
5	flatMap at Main.scala:148	+details	2025/05/21 15:49:26	4 s	1/1	4.0 MB			931.9 KB
4	map at Main.scala:144	+details	2025/05/21 15:49:25	10 s	8/8	81.8 MB			23.6 MB
3	first at Main.scala:124	+details	2025/05/21 15:49:24	1 s	1/1	8.8 KB			
2	first at Main.scala:116	+details	2025/05/21 15:49:24	0.2 s	1/1	64.0 KB			
1	csv at Main.scala:88	+details	2025/05/21 15:49:21	3 s	1/1	35.4 KB			
0	csv at Main.scala:68	+details	2025/05/21 15:49:17	3 s	1/1	4.1 MB			

Figure 2: Spark Jobs view for Part A execution (RDD-based queries).

Jobs **0 to 3** are very short, each consisting of a single stage and only one task. These are related to the dataframes and rdd initialisations in the preprocessing part. They are created with the usage of `.csv` when creating dataframes and the use `.first()`(for determining the headers) after creating an RDD by reading a file with `sc.textFile()`. Thus, Jobs 0 to 3 are not logical jobs (they dont implement any query) but they are helpful physical jobs that spark creates for the initialization of data. Every query was designed as an independent job (physical jobs **4 to 8**), with careful attention to when and how Spark actions were invoked. The use of `.saveAsTextFile()` and `.write().csv()` actions ensured that the lazy transformations were only computed once results were needed.

Before analyzing Job 4 and the subsequent query executions, it's important to briefly explain how Spark defines stages based on the type of transformations applied. Spark distinguishes between **narrow** and **wide** transformations. Narrow transformations (such as `map`, `filter`, `flatMap`, and `mapValues`) operate on partitions independently, allowing for pipelined execution within a single stage. In contrast, wide transformations (like `join`, `groupByKey`, `reduceByKey`, and `distinct`) require data from multiple partitions to be shuffled across the cluster. These shuffles represent stage boundaries, leading Spark to break the computation into multiple stages. Understanding this distinction helps explain the structure of the DAGs and the execution flow in each job that follows below.

3.1.1 PartA-Job4

To better understand the inner workings of a complete RDD-based job, let's examine Job 4 which is the first "real"(logical) job. This job corresponds to a multi-stage query(1st query) involving complex joins, filtering, and reductions.

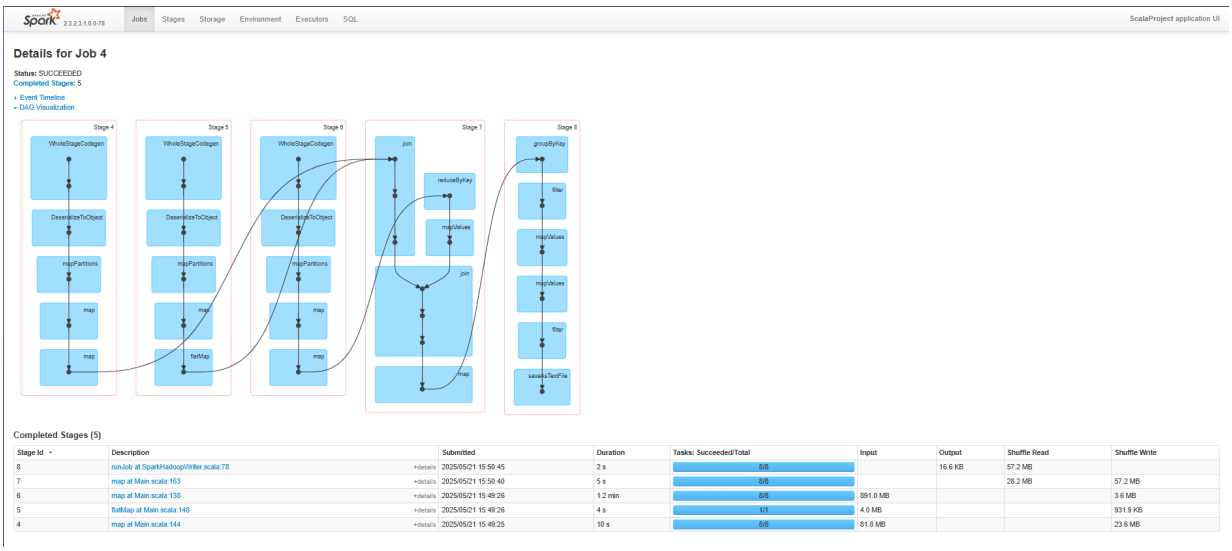


Figure 3: DAG visualization of Job 4 — a multi-stage query

Job 4 comprises 5 stages (Stage IDs 4–8), showing a typical RDD pipeline including multiple transformations and wide operations (e.g., joins and groupByKey), which result in shuffle operations.

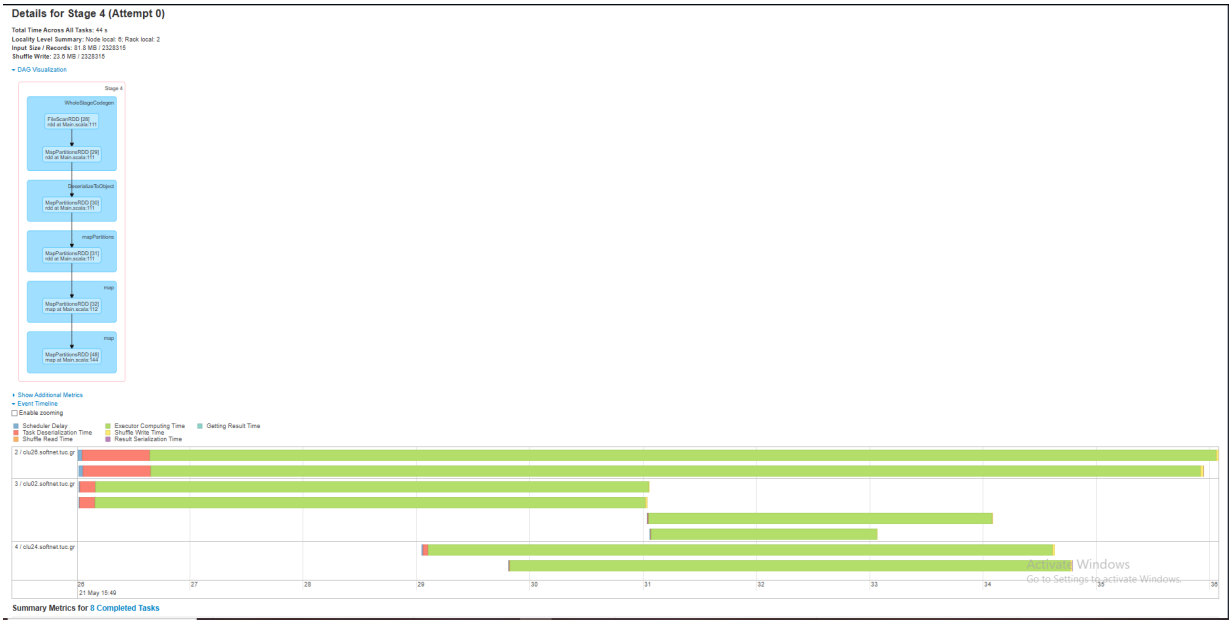


Figure 4: Stage 4 of Job 4 — Mapping over input data

In this stage, the input file is read and deserialized into an rdd form, followed by a few `mapPartitions` and `map` transformations. The executor time is relatively well distributed, indicating uniform task assignment.

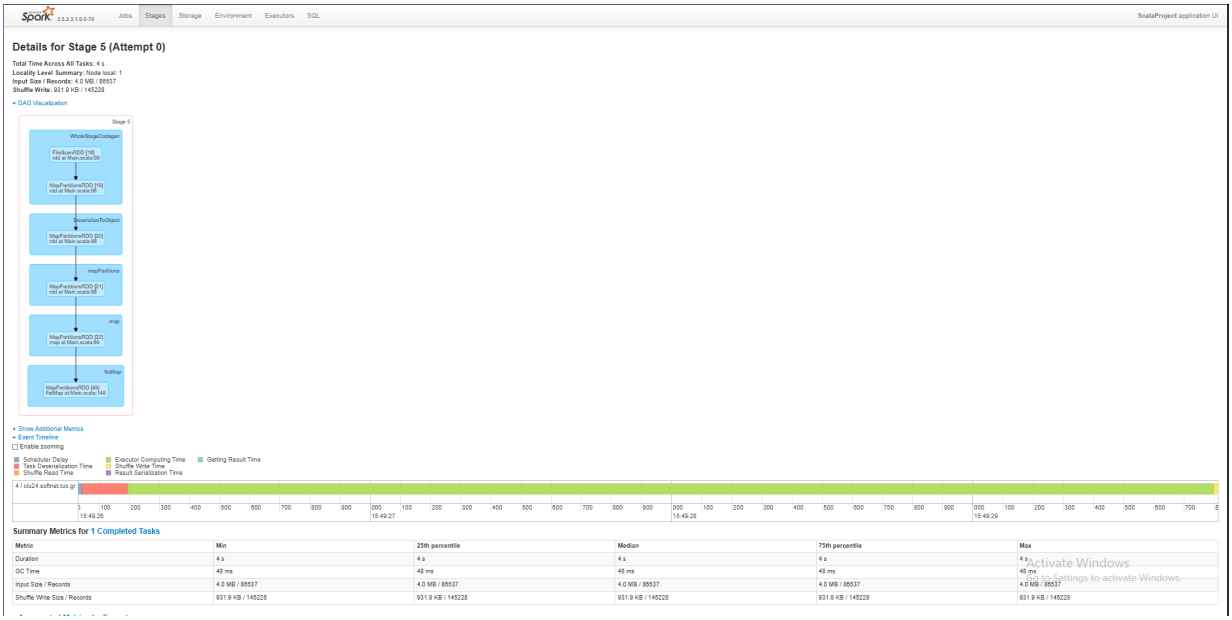


Figure 5: Stage 5 of Job 4 — FlatMapping tags or genres

This stage shows a flatMap transformation after file loading, which corresponds to the creation and manipulation of the movies RDD created by the movies Dataframe. Shuffle write is moderate, which hints at the preparation for future joins.

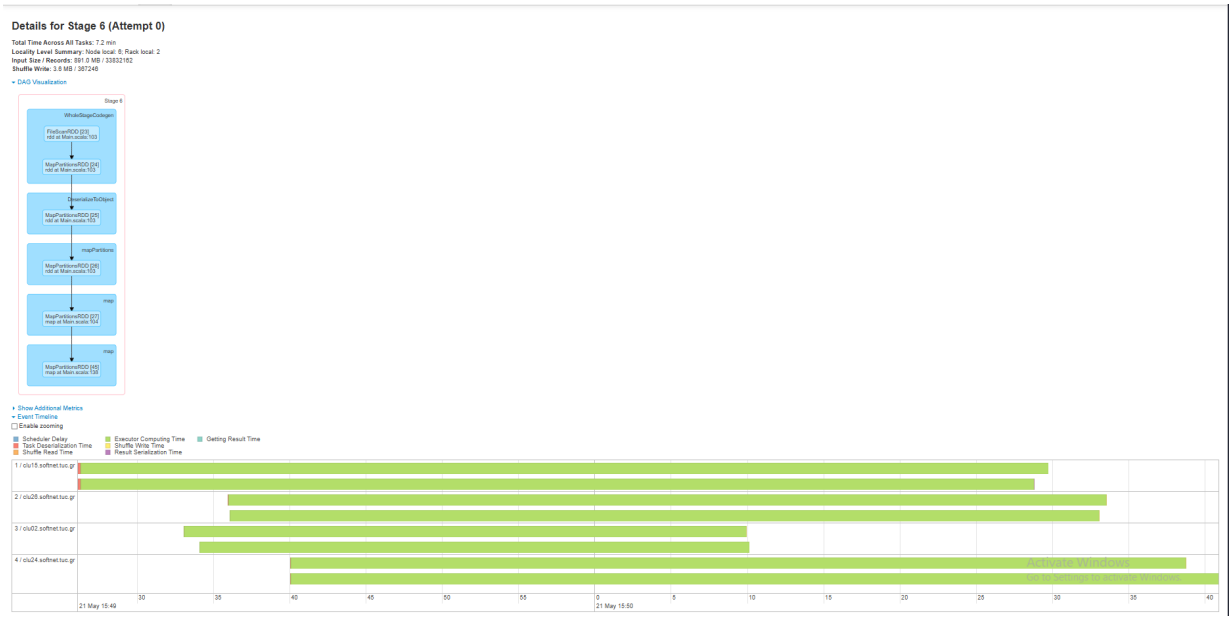


Figure 6: Stage 6 of Job 4 — Additional mapping before join

Stage 6 continues processing from other RDDs that will participate in the join of Stage 7. It features a fairly long executor compute time, with relatively high input size.

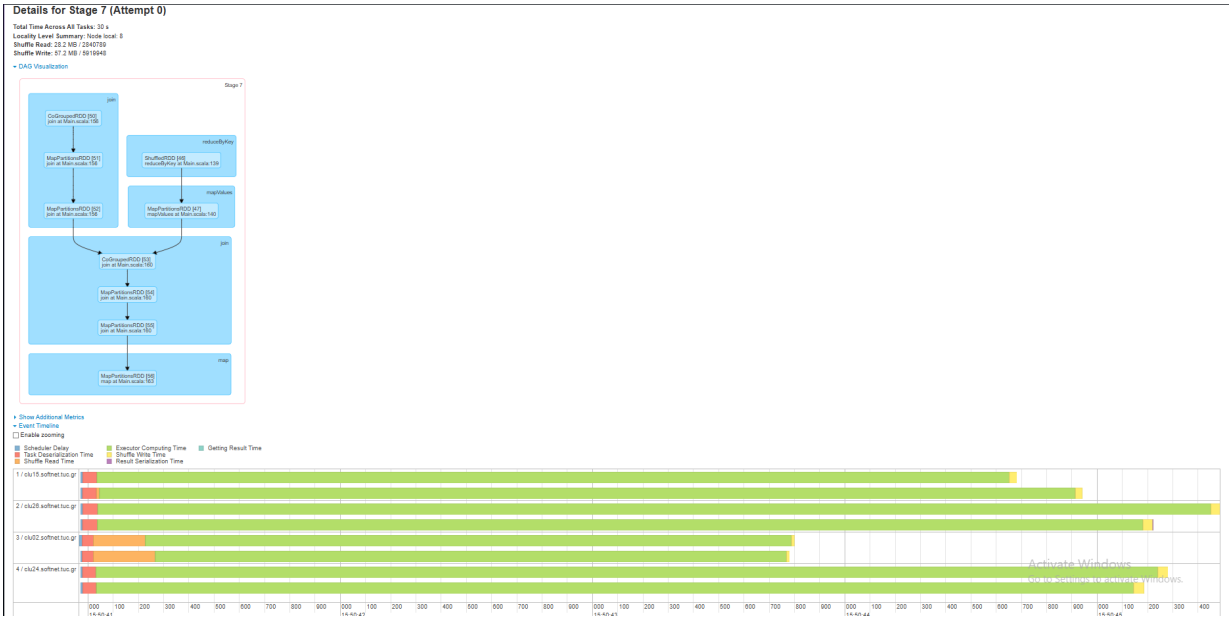


Figure 7: Stage 7 of Job 4 — Join, reduceByKey, and heavy shuffling

This stage is the heaviest in terms of shuffle — both shuffle read and shuffle write are substantial. It performs joins between multiple RDDs (3 to be exact), and includes a `reduceByKey` transformation, which aggregates data based on the ratings that each movie got so we can calculate its average.

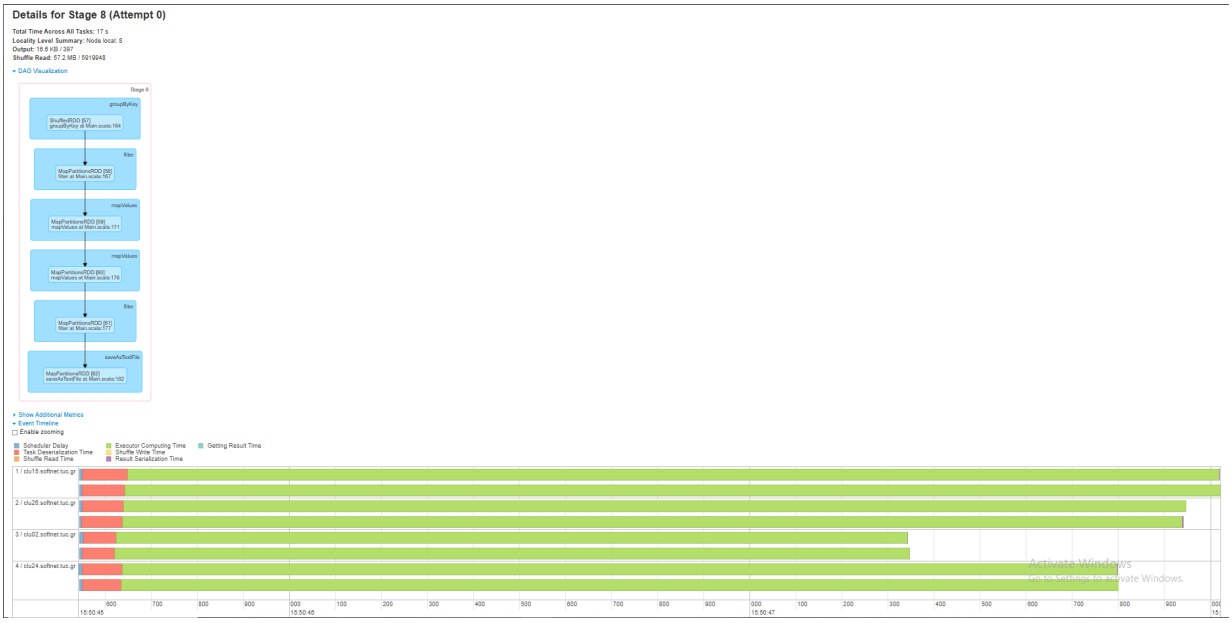


Figure 8: Stage 8 of Job 4 — Final grouping and output

Finally, in Stage 8, the results are grouped and filtered once more, and then written to HDFS using `saveAsTextFile()`. The event timeline of most of the tasks above shows a healthy degree of parallelism (8 concurent tasks), with all executors contributing simultaneously and evenly across tasks. The task durations are pretty similar which indicates effective task scheduling by Spark.

3.1.2 PartA-Job5

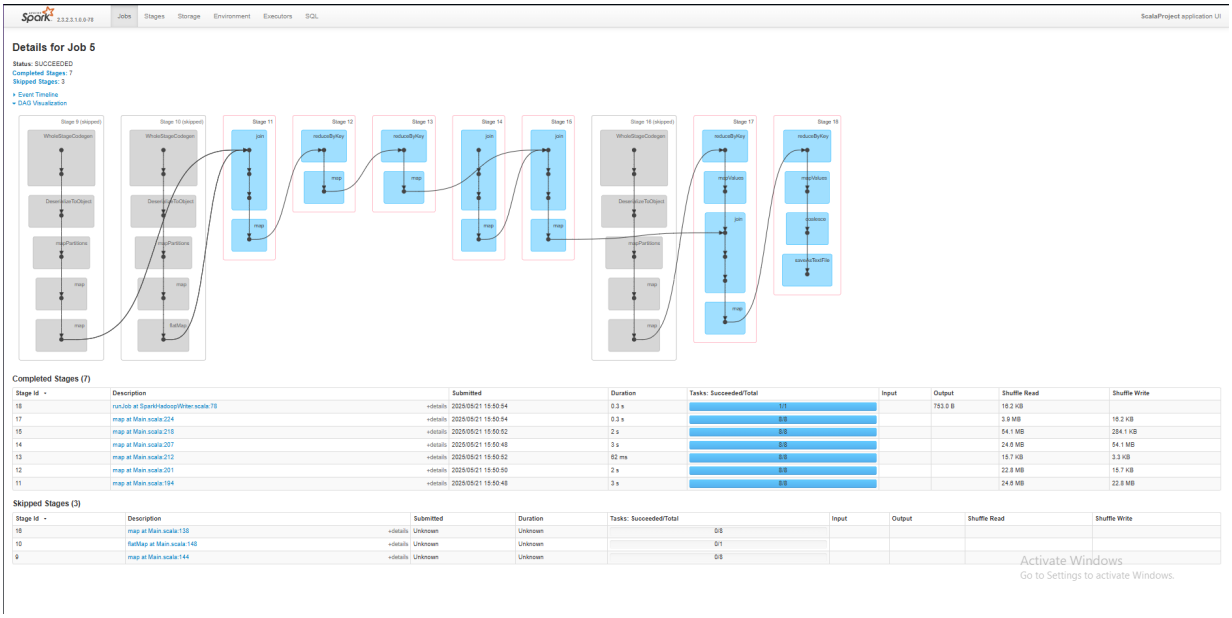


Figure 9: DAG and stage structure for Job 5

Job 5 is composed of 7 completed stages and 3 skipped stages, following the typical lazy evaluation behavior of Spark. The job is triggered by an action, in this case, the saveAsTextFile() we put for

query 2, which forces the chain of transformations to be computed. As the DAG shows, there are multiple joins and reductions that naturally split the job into several stages.

Each stage corresponds to a boundary where a shuffle occurs, i.e., Spark needs to exchange data across partitions to satisfy wide operations such as joins.

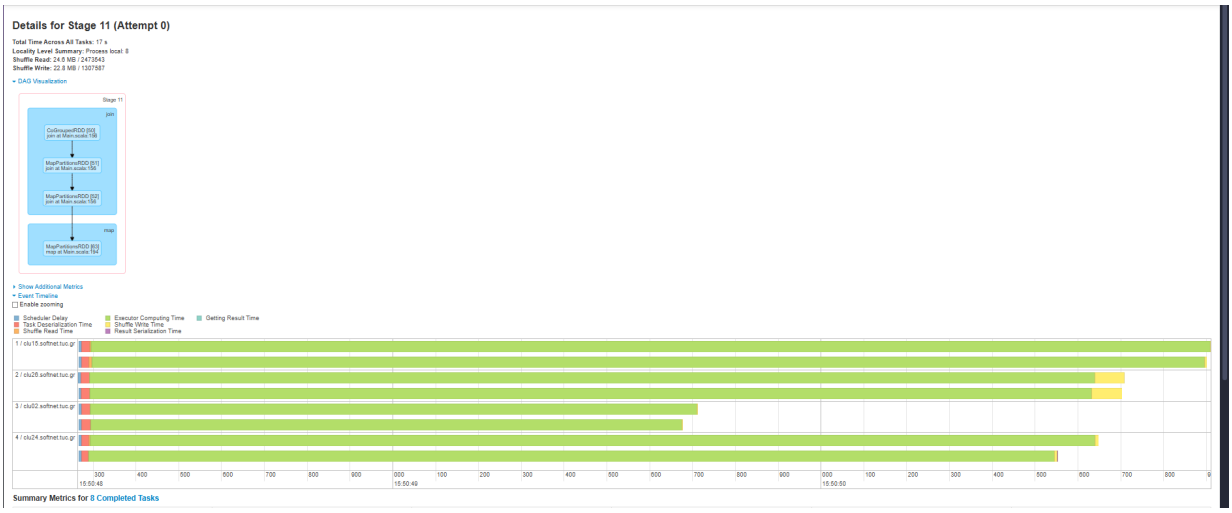


Figure 10: Stage 11: CoGroup + Map after shuffle

Stage 11 performs a shuffle join via CoGroupedRDD. The event timeline at the bottom shows 8 parallel tasks, which is most likely matches the default number of partitions or executors. Most tasks begin and end in a tightly clustered timeframe, indicating high parallelism and efficient cluster resource usage.

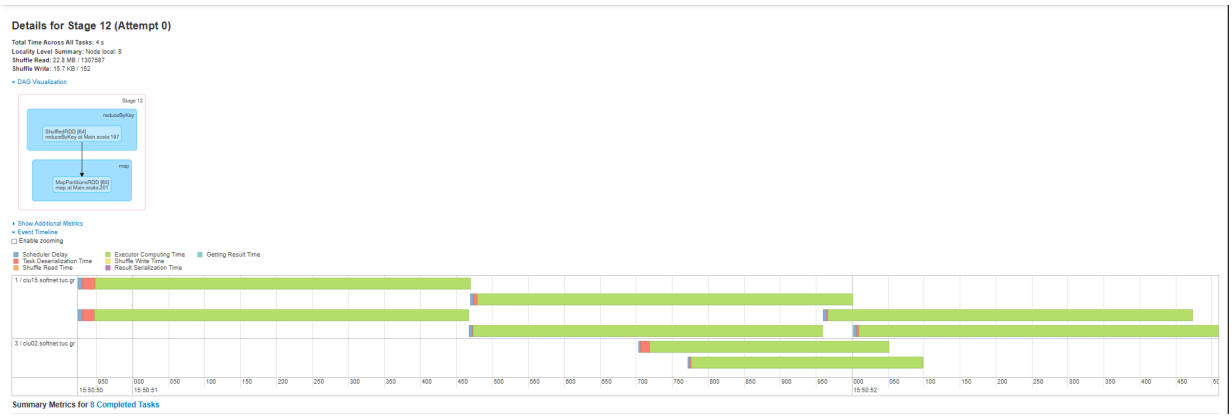


Figure 11: Stage 12: Reduce and Map following join

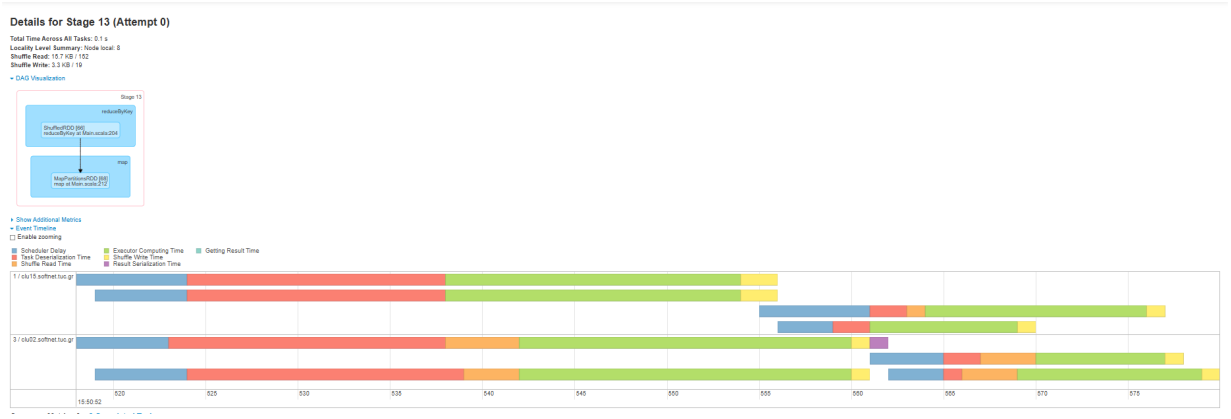


Figure 12: Stage 13: Further aggregation (low data volume)

These stages represent reduceByKey and follow-up map operations. In Stage 12, we again observe 8 parallel tasks, with a moderate data shuffle. Stage 13, however, operates on a significantly smaller data subset (only a few kilobytes), and thus the cluster executes just 2 tasks, reflecting reduced parallelism due to limited data rather than configuration.

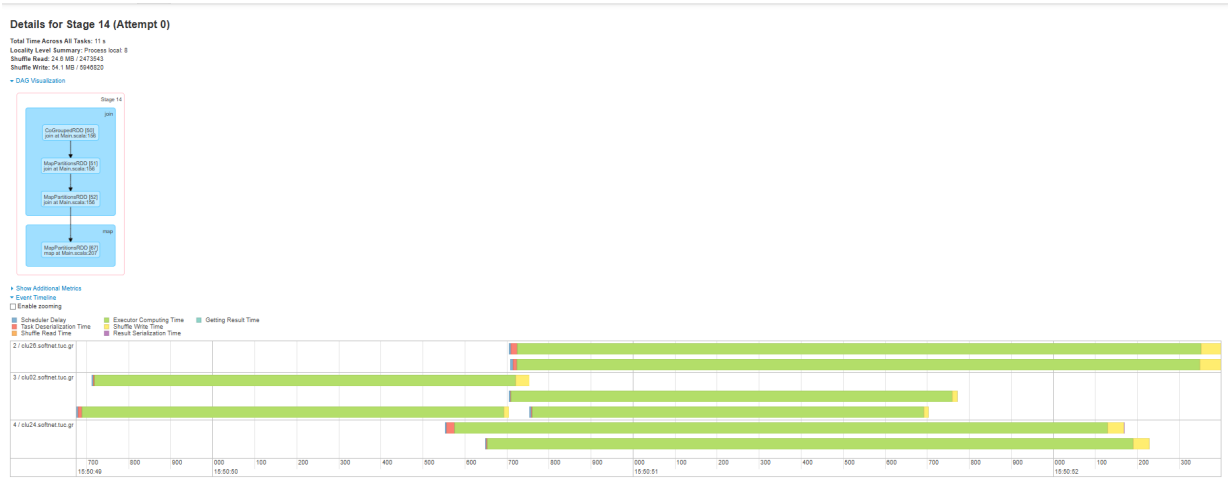


Figure 13: Stage 14: Small join and transformation

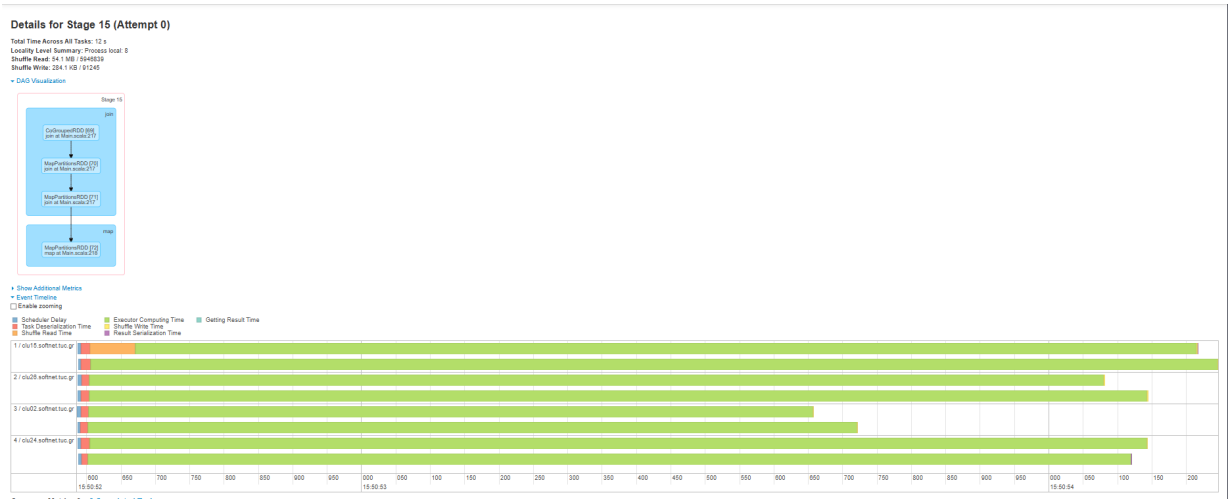


Figure 14: Stage 15: Join on wider dataset

Stage 14 works over a lighter dataset and spawns only 3 tasks. Meanwhile, Stage 15 once again scales up to 8 parallel tasks. This demonstrates that Spark automatically adjusts task parallelism based on the number of partitions, high data shuffle and computationally-heavy work.

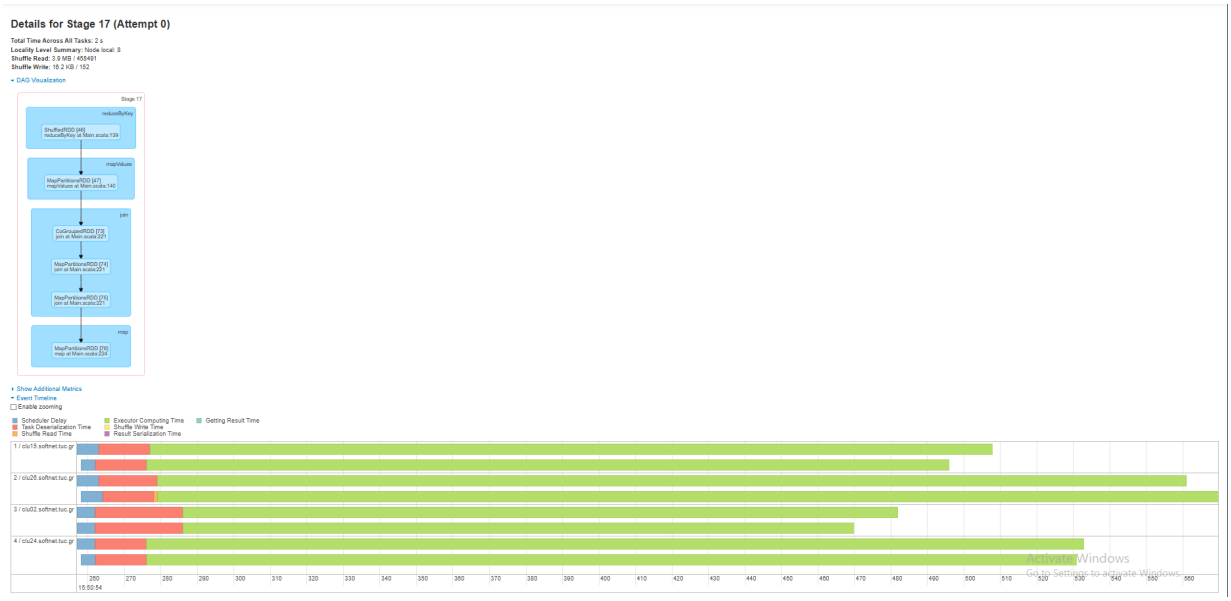


Figure 15: Stage 17: Final map-reduce before write

This stage concludes the computation-heavy part of the job. It includes a reduceByKey, followed by a map, and another shuffle join. The timeline shows a clean parallel execution of 8 tasks.

Figure 16: Stage 18: Output writing stage with coalesce

Finally, this stage uses a coalesce(1) transformation, which reduces the number of partitions to one for single-file output. Consequently, only 1 task is executed, which is expected and optimal for write efficiency when we know the output will be a relatively small amount of data.

3.1.3 PartA-Job6

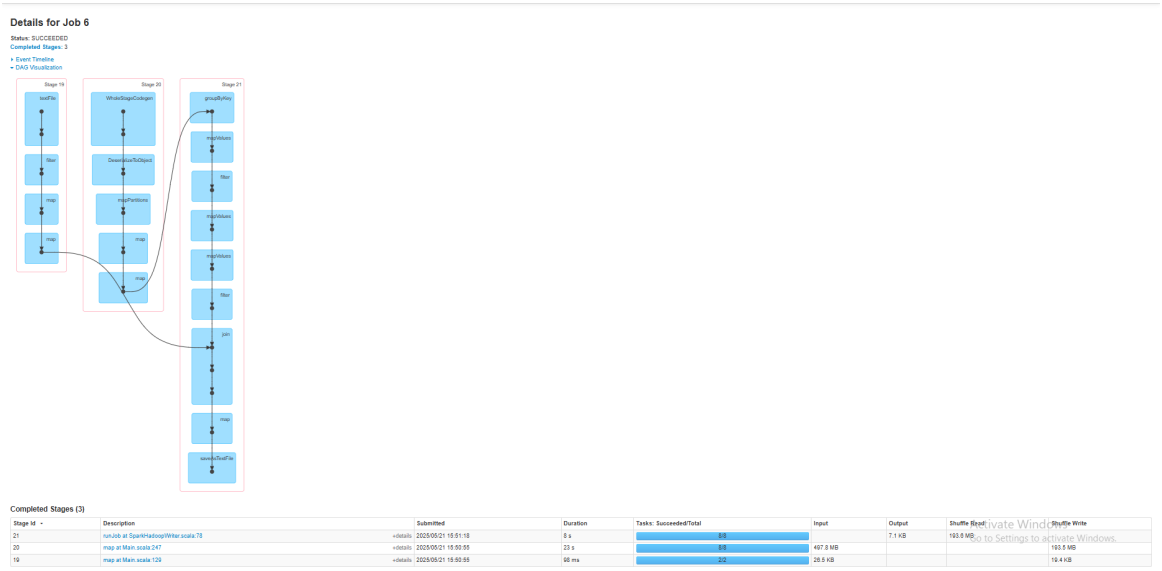


Figure 17: DAG visualization for Job 6

Job 6 consists of 3 stages, is addressing query 3 and is triggered by the `saveAsTextFile` action. This job performs filtering and transformations on tag data and joins it with genome tags RDD. The stage breakdown as usual, is determined by wide operations like groupByKey and join, which require data shuffling and thus separate stages.

Figure 18: Stage 19: Input preprocessing and light filtering

Stage 19 is a narrow stage that reads a small file, filters, and maps it. It runs quickly with only two tasks, showing low parallelism. This is expected due to the small file size.

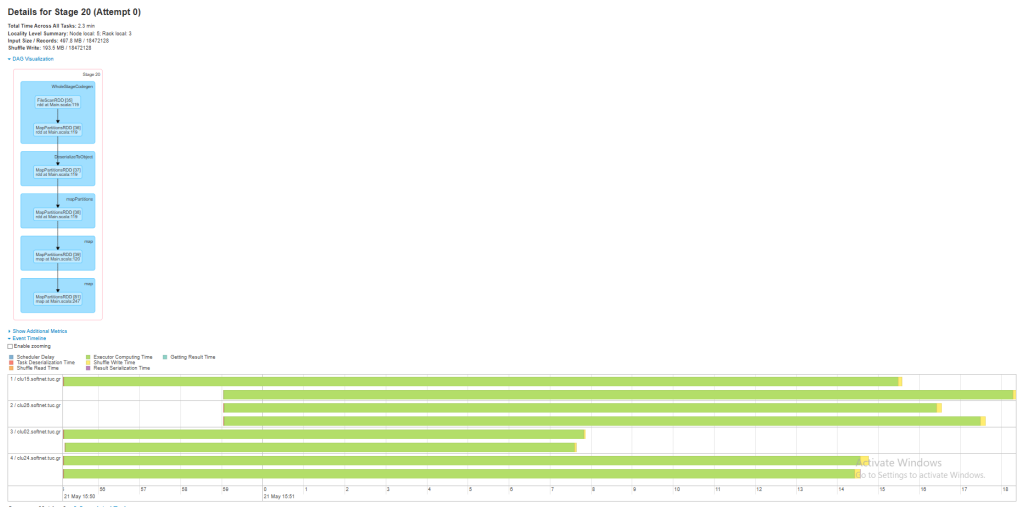


Figure 19: Stage 20: Ratings read and transformation

Stage 20 performs file read and mapping operations on a much larger dataset (497 MB). It uses 8 tasks, reflecting the default Spark parallelism.

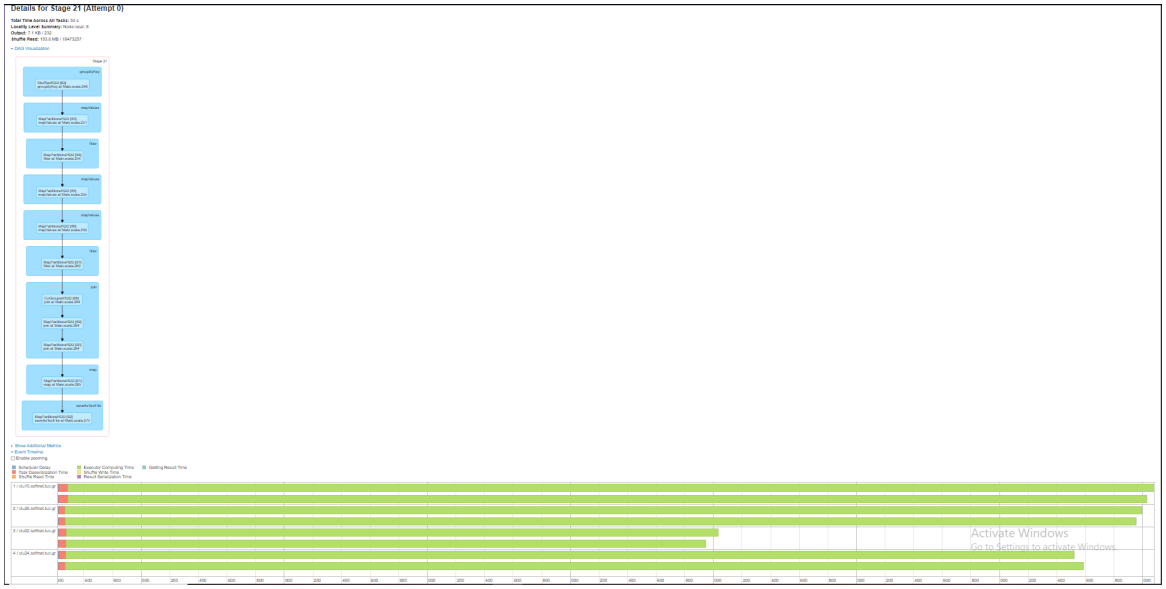


Figure 20: Stage 21: Join and filtering with shuffles

Stage 21 performs a groupByKey and join, which triggers a shuffle. It again runs with 8 tasks,the maximum default parallelism, and demonstrates good concurrency. Despite the higher shuffle read (193.8 MB), task durations are fairly consistent.

3.1.4 PartA-Job7

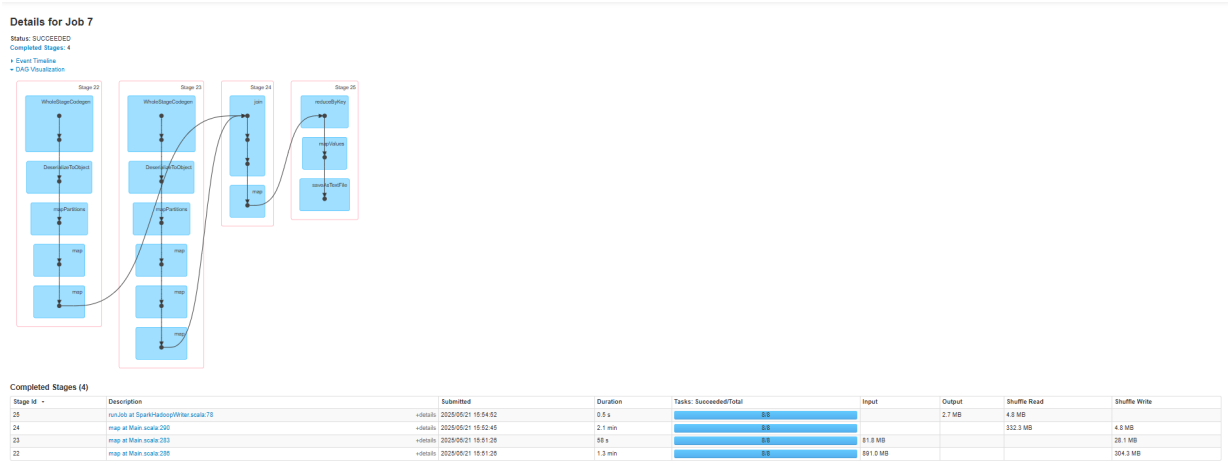


Figure 21: DAG visualization for Job 7

Job 7 consists of 4 stages, is addressing query 4 and is triggered by the `saveAsTextFile` action. It reads two separate datasets, applies transformations, joins them, and performs a final aggregation before writing the result to disk. The boundaries between stages are dictated by wide operations such as joins and reduceByKey, which require shuffling across the cluster.

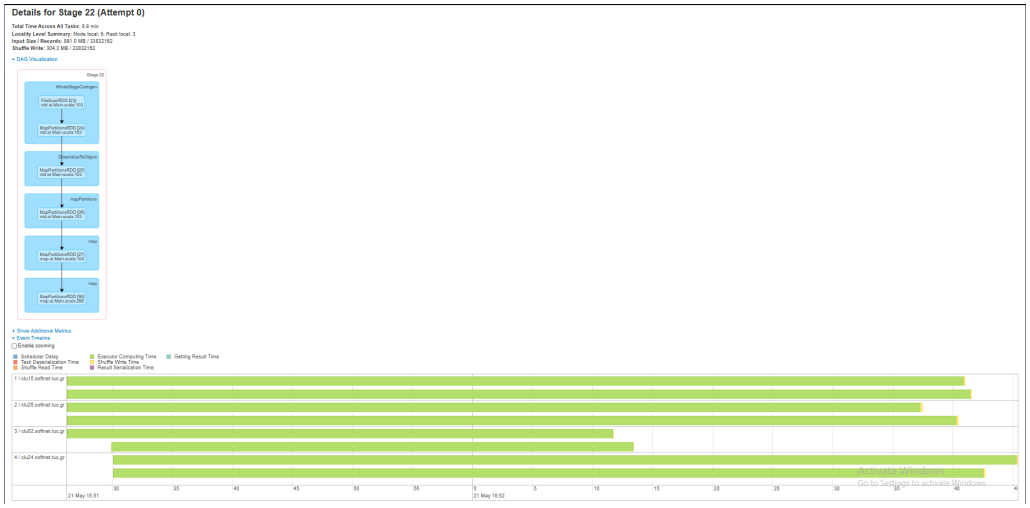


Figure 22: Stage 22: Ratings file read and preprocessing

Stage 22 performs a read and transformation over a large ratings dataset (891 MB). It consists of 8 tasks, reflecting the cluster's default parallelism. Tasks show consistent and balanced executor time.

Figure 23: Stage 23: Processing of genome tags dataset

Stage 23 reads and transforms the genome tags input (81.8 MB). It also uses 8 tasks and completes efficiently, preparing data for the upcoming join.

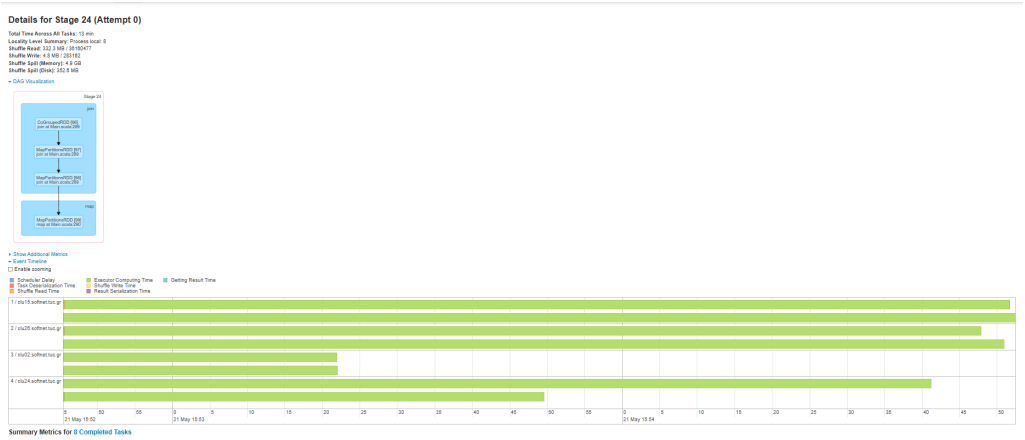


Figure 24: Stage 24: Join operation and mapping

Stage 24 performs a join between the outputs of the previous stages, producing a larger intermediate result. This triggers a shuffle read of over 330 MB and involves 8 tasks, which are executed concurrently with relatively balanced durations.

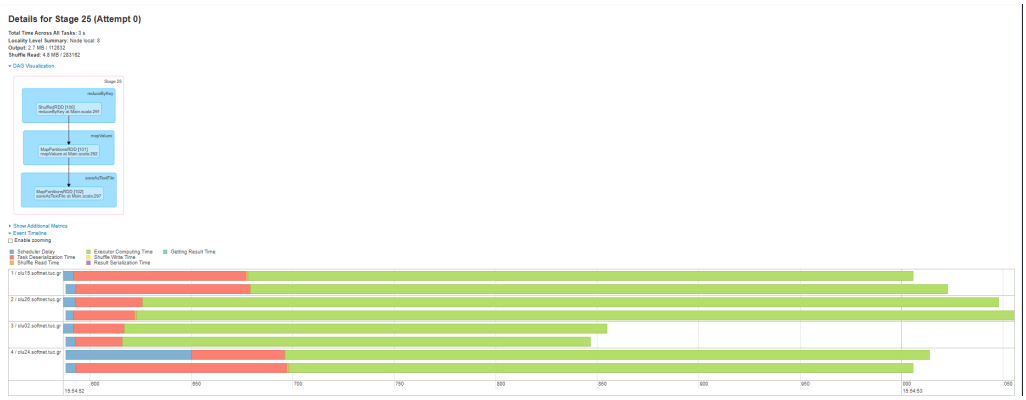


Figure 25: Stage 25: Final aggregation and save

Stage 25 applies reduceByKey and mapValues operations before saving the result. It's a short stage with 8 tasks completing in parallel and quickly finalizing the job output.

3.1.5 PartA-Job8

Figure 26: DAG visualization for Job 8

Job 8 consists of 10 completed stages, corresponds to query 5, and is triggered by the `saveAsTextFile` action. The job performs multiple transformations including distinct operations, joins, and groupings over several RDDs. Multiple wide operations, such as joins and groupByKey operations, cause data shuffling and produce so many different stages.

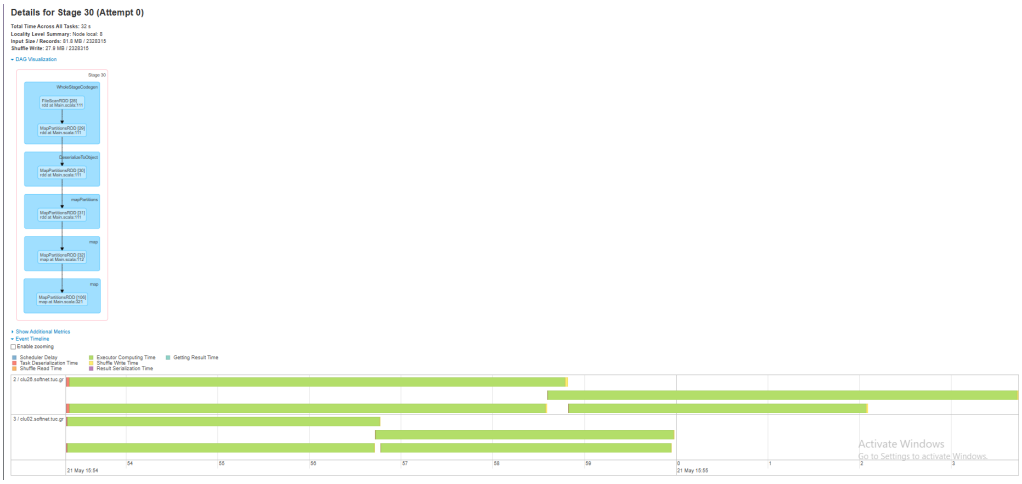


Figure 27: Stage 30: File read and transformation

Stage 30 involves reading a moderately sized dataset (81.8 MB) and applying map transformations. It executes with 8 tasks and performs efficiently.

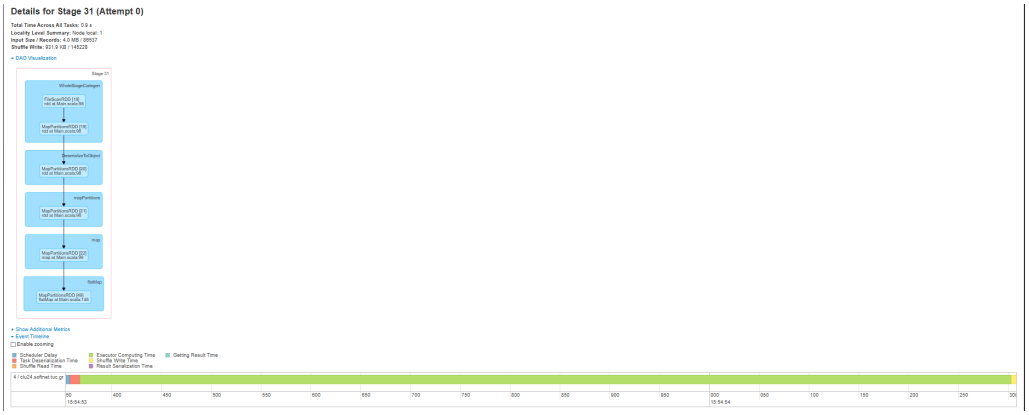


Figure 28: Stage 31: FlatMap transformation on metadata

Stage 31 processes a small file (4.0 MB) and applies a `flatMap`. It executes quickly with a single task, due to the limited input size.

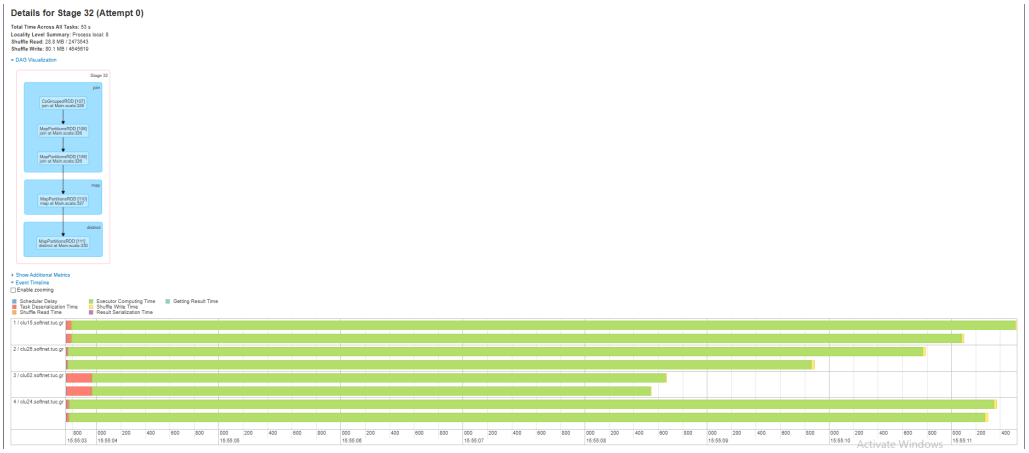


Figure 29: Stage 32: Join followed by distinct operation

Stage 32 performs a join and subsequent mapping on shuffled data (28.8 MB read). The stage outputs to a `distinct` operation, preparing for throwing out duplicate data.

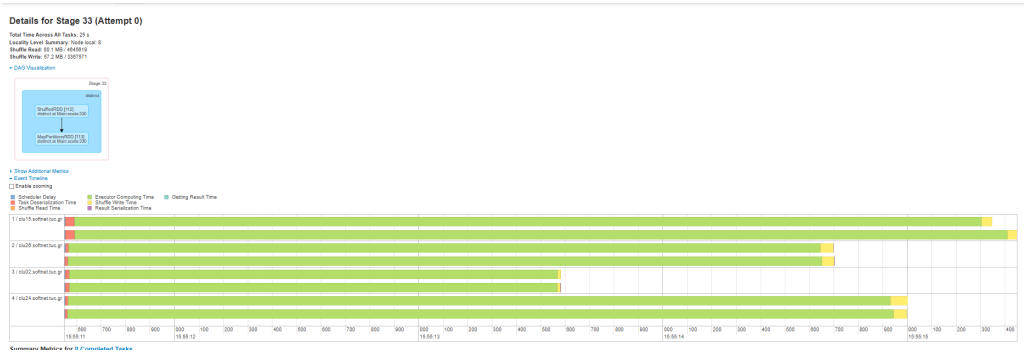


Figure 30: Stage 33: Distinct on previous output

Stage 33 reads 80.1 MB of shuffled data and removes duplicates using the `distinct` transformation. It runs with 8 tasks and completes in 25 seconds.

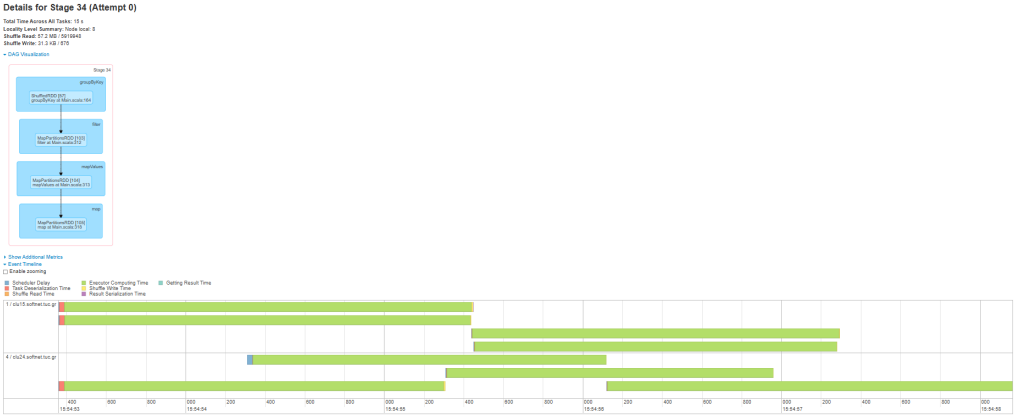


Figure 31: Stage 34: Grouping, filtering and mapping

Stage 34 groups the distinct output by key, filters it, and maps the results. It reads 57.2 MB of shuffled data and executes with 8 tasks.

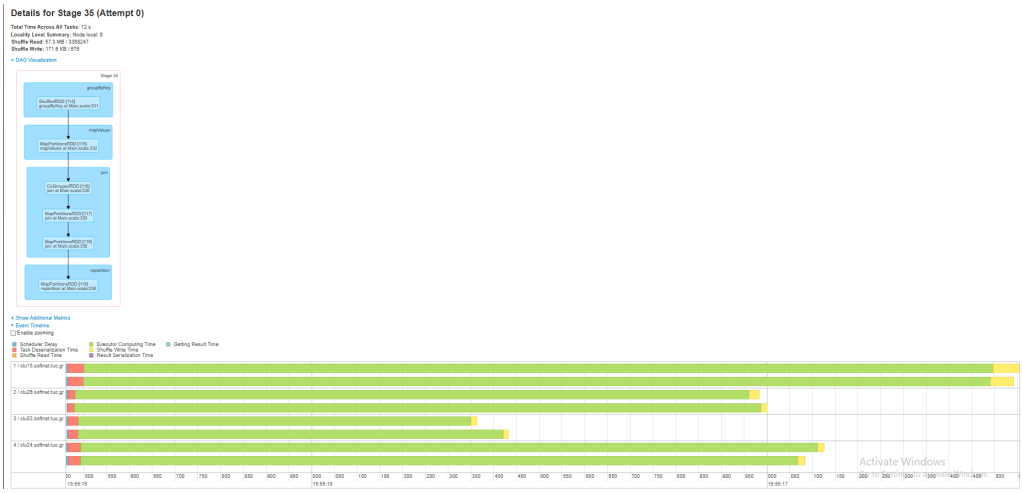


Figure 32: Stage 35: Join and repartitioning

Stage 35 performs a groupByKey, a join, and a repartition. This stage handles 57.3 MB of shuffled input and executes efficiently across 8 tasks.

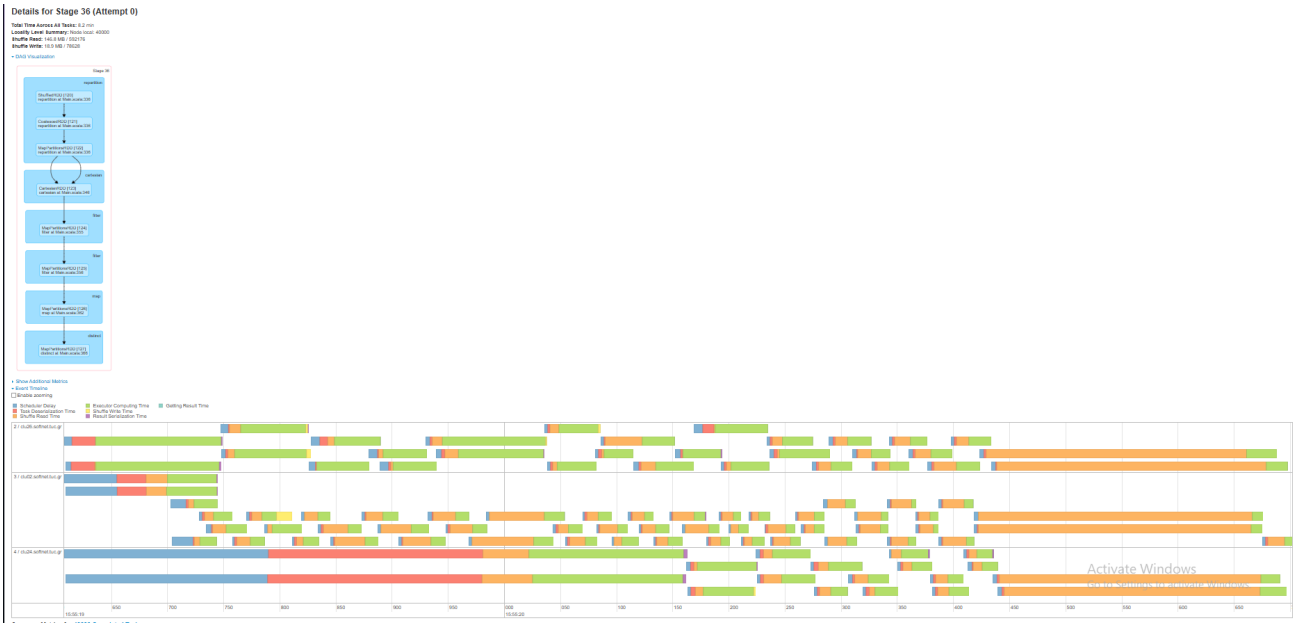


Figure 33: Stage 36: Cartesian operation and distinct

Stage 36 is a heavy stage that performs a Cartesian product followed by mapping and a distinct operation. It reads 146.8 MB and writes 188.7 MB of shuffle data over 40,000 tasks, indicating significant parallel computation. This huge amount of tasks was determined by Spark because of our use of `repartition(200)`. Before the use of repartition, when we tried to run it on the cluster, the amount of tasks were a lot less.

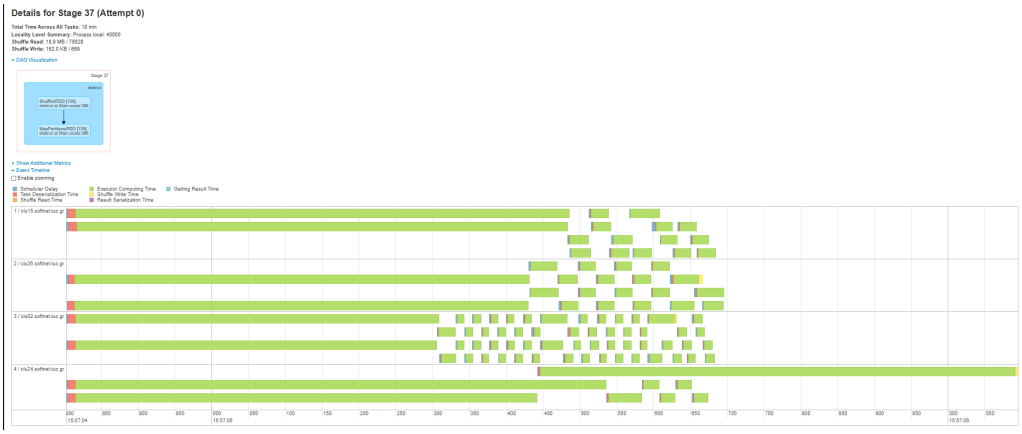


Figure 34: Stage 37: Deduplication after Cartesian

Stage 37 throws out the duplicates that the Cartesian output produced from Stage 36 by applying a `distinct`. The stage processes 18.9 MB of shuffle data across 40,000 tasks.

Figure 35: Stage 38: Repartitioning of results

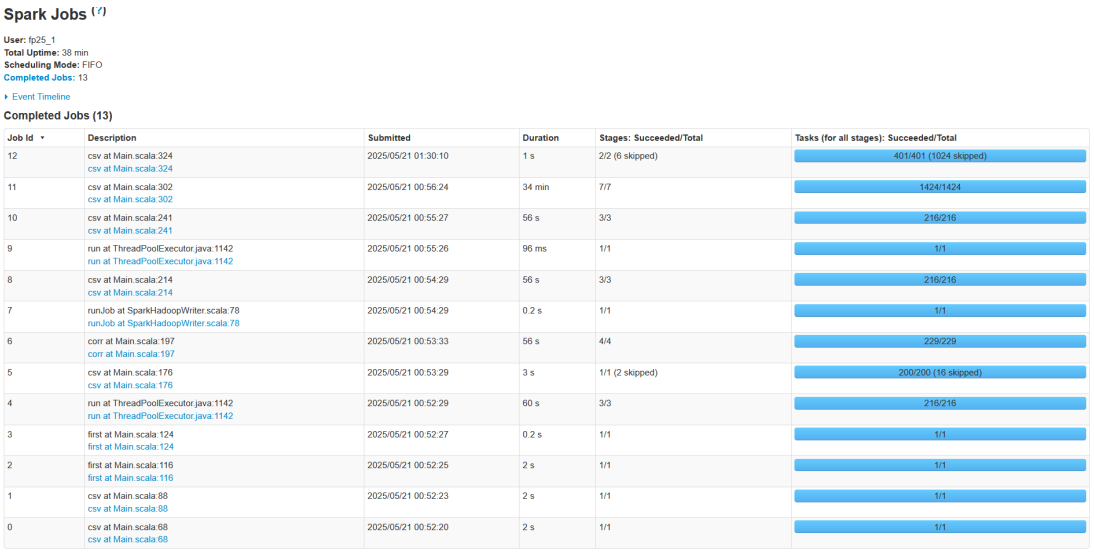
Stage 38 repartitions the data, distributing the workload for the final step. It reads 171.6 KB and prepares the output for writing.

Figure 36: Stage 39: Final subtraction and save

Stage 39 subtracts keys from two datasets and writes the final output to disk using `saveAsTextFile`. This final stage processes 325.7 KB of shuffle read data using 200 tasks.
We can see in the last couple stages that the level of parallelism has ramped up to 16. This means that either the cluster has suddenly less work to do for other users or our code is way more computationally taxing than usual.

3.2 Part B - DataFrames

First we have all the jobs for PartB.



Spark Jobs (?)

User: fp25_1
Total Uptime: 38 min
Scheduling Mode: FIFO
Completed Jobs: 13

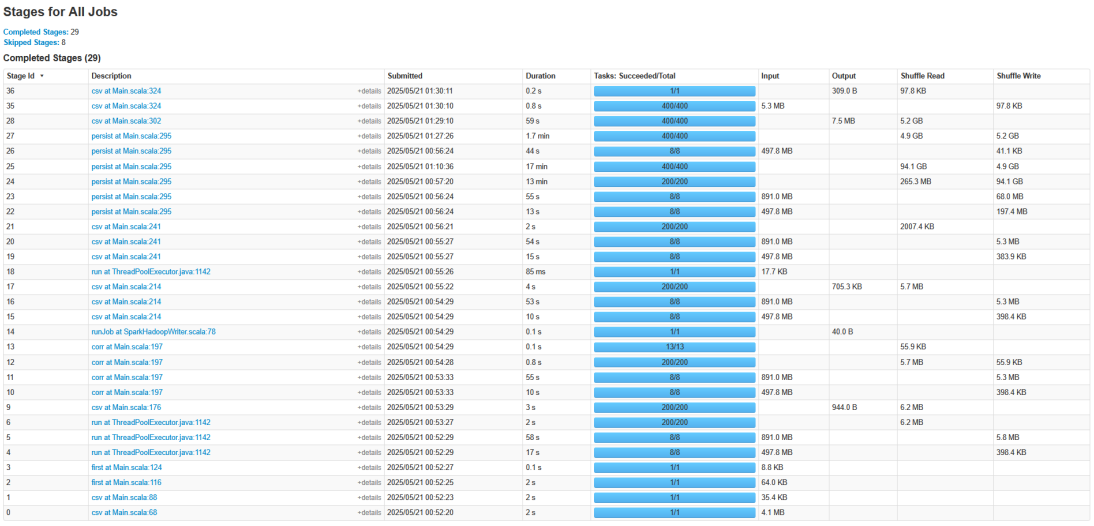
▸ Event Timeline

Completed Jobs (13)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
12	csv at Main.scala:324 csv at Main.scala:324	2025/05/21 01:30:10	1 s	2/2 (6 skipped)	401/401 (1024 skipped)
11	csv at Main.scala:302 csv at Main.scala:302	2025/05/21 00:56:24	34 min	7/7	1424/1424
10	csv at Main.scala:241 csv at Main.scala:241	2025/05/21 00:55:27	56 s	3/3	216/216
9	run at ThreadPoolExecutor.java:1142 run at ThreadPoolExecutor.java:1142	2025/05/21 00:55:26	96 ms	1/1	1/1
8	csv at Main.scala:214 csv at Main.scala:214	2025/05/21 00:54:29	56 s	3/3	216/216
7	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2025/05/21 00:54:29	0.2 s	1/1	1/1
6	corr at Main.scala:197 corr at Main.scala:197	2025/05/21 00:53:33	56 s	4/4	229/229
5	csv at Main.scala:176 csv at Main.scala:176	2025/05/21 00:53:29	3 s	1/1 (2 skipped)	200/200 (16 skipped)
4	run at ThreadPoolExecutor.java:1142 run at ThreadPoolExecutor.java:1142	2025/05/21 00:52:29	60 s	3/3	216/216
3	first at Main.scala:124 first at Main.scala:124	2025/05/21 00:52:27	0.2 s	1/1	1/1
2	first at Main.scala:116 first at Main.scala:116	2025/05/21 00:52:25	2 s	1/1	1/1
1	csv at Main.scala:88 csv at Main.scala:88	2025/05/21 00:52:23	2 s	1/1	1/1
0	csv at Main.scala:68 csv at Main.scala:68	2025/05/21 00:52:20	2 s	1/1	1/1

Figure 37: Spark Jobs view for Part A execution (RDD-based queries).

Then we have all the stages for PartB.



Stages for All Jobs

Completed Stages: 29
Skipped Stages: 8

Completed Stages (29)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
36	csv at Main.scala:324 +details	2025/05/21 01:30:11	0.2 s	1/1		309.0 B	97.8 KB	
35	csv at Main.scala:324 +details	2025/05/21 01:30:10	0.8 s	400/400	5.3 MB			97.8 KB
28	csv at Main.scala:302 +details	2025/05/21 01:29:10	59 s	400/400		7.5 MB	5.2 GB	
27	persist at Main.scala:295 +details	2025/05/21 01:27:26	1.7 min	400/400			4.9 GB	5.2 GB
26	persist at Main.scala:295 +details	2025/05/21 00:56:24	44 s	8/8	497.8 MB			41.1 KB
25	persist at Main.scala:295 +details	2025/05/21 01:10:36	17 min	400/400			94.1 GB	4.9 GB
24	persist at Main.scala:295 +details	2025/05/21 00:57:20	13 min	200/200			265.3 MB	94.1 GB
23	persist at Main.scala:295 +details	2025/05/21 00:56:24	55 s	8/8	891.0 MB			68.0 MB
22	persist at Main.scala:295 +details	2025/05/21 00:56:24	13 s	8/8	497.8 MB			197.4 MB
21	csv at Main.scala:241 +details	2025/05/21 00:56:21	2 s	200/200			2007.4 KB	
20	csv at Main.scala:241 +details	2025/05/21 00:55:27	54 s	8/8	891.0 MB			5.3 MB
19	csv at Main.scala:241 +details	2025/05/21 00:55:27	15 s	8/8	497.8 MB			383.9 KB
18	run at ThreadPoolExecutor.java:1142 +details	2025/05/21 00:55:26	85 ms	1/1	17.7 KB			
17	csv at Main.scala:214 +details	2025/05/21 00:55:22	4 s	200/200		705.3 KB	5.7 MB	
16	csv at Main.scala:214 +details	2025/05/21 00:54:29	53 s	8/8	891.0 MB			5.3 MB
15	csv at Main.scala:214 +details	2025/05/21 00:54:29	10 s	8/8	497.8 MB			398.4 KB
14	runJob at SparkHadoopWriter.scala:78 +details	2025/05/21 00:54:29	0.1 s	1/1		40.0 B		
13	corr at Main.scala:197 +details	2025/05/21 00:54:29	0.1 s	13/13			55.9 KB	
12	corr at Main.scala:197 +details	2025/05/21 00:54:28	0.8 s	200/200			5.7 MB	55.9 KB
11	corr at Main.scala:197 +details	2025/05/21 00:53:33	55 s	8/8	891.0 MB			5.3 MB
10	corr at Main.scala:197 +details	2025/05/21 00:53:33	10 s	8/8	497.8 MB			398.4 KB
9	csv at Main.scala:176 +details	2025/05/21 00:53:29	3 s	200/200		944.0 B	6.2 MB	
6	run at ThreadPoolExecutor.java:1142 +details	2025/05/21 00:53:27	2 s	200/200			6.2 MB	
5	run at ThreadPoolExecutor.java:1142 +details	2025/05/21 00:52:29	58 s	8/8	891.0 MB			5.8 MB
4	run at ThreadPoolExecutor.java:1142 +details	2025/05/21 00:52:29	17 s	8/8	497.8 MB			398.4 KB
3	first at Main.scala:124 +details	2025/05/21 00:52:27	0.1 s	1/1	8.8 KB			
2	first at Main.scala:116 +details	2025/05/21 00:52:25	2 s	1/1	64.0 KB			
1	csv at Main.scala:88 +details	2025/05/21 00:52:23	2 s	1/1	35.4 KB			
0	csv at Main.scala:68 +details	2025/05/21 00:52:20	2 s	1/1	4.1 MB			

Figure 38: Spark Jobs view for Part A execution (RDD-based queries).

Jobs **0 to 3** are same as in PartA: very short, each consisting of a single stage and only one task. These are related to the dataframes and rdd initialisations in the preprocessing part. They are created with the usage of `.csv` when creating dataframes and the use `.first()` after creating an RDD by reading a file with `sc.textFile()`.

3.2.1 PartB-Job4

Job 4 displays `run at ThreadPoolExecutor.java:1142` on the history Server and it is related to the join operation at Query 6. It consists of stages 4, 5 and 6.

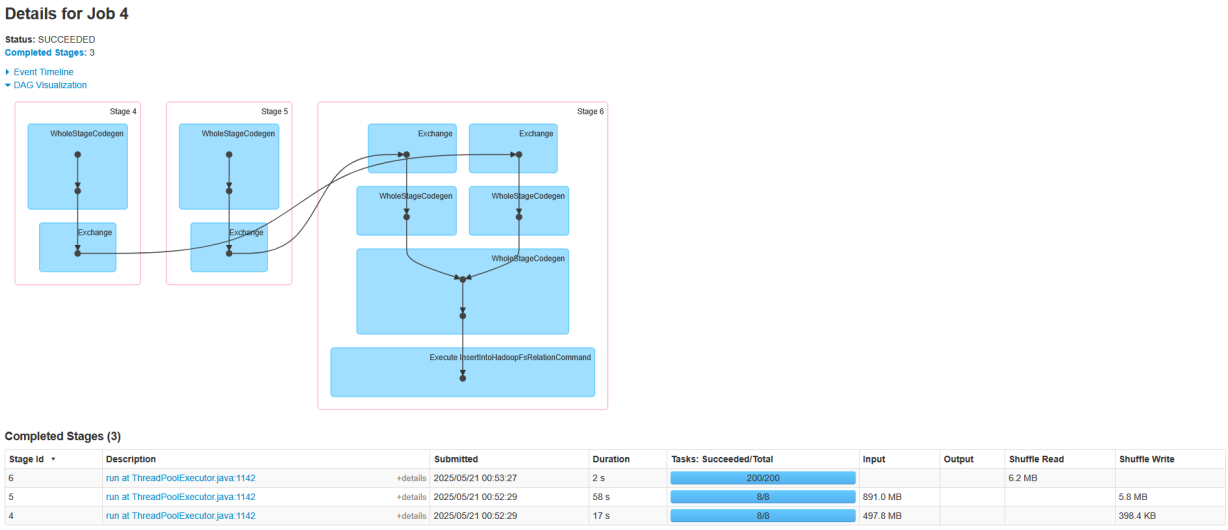


Figure 39: DAG visualization of Job 4

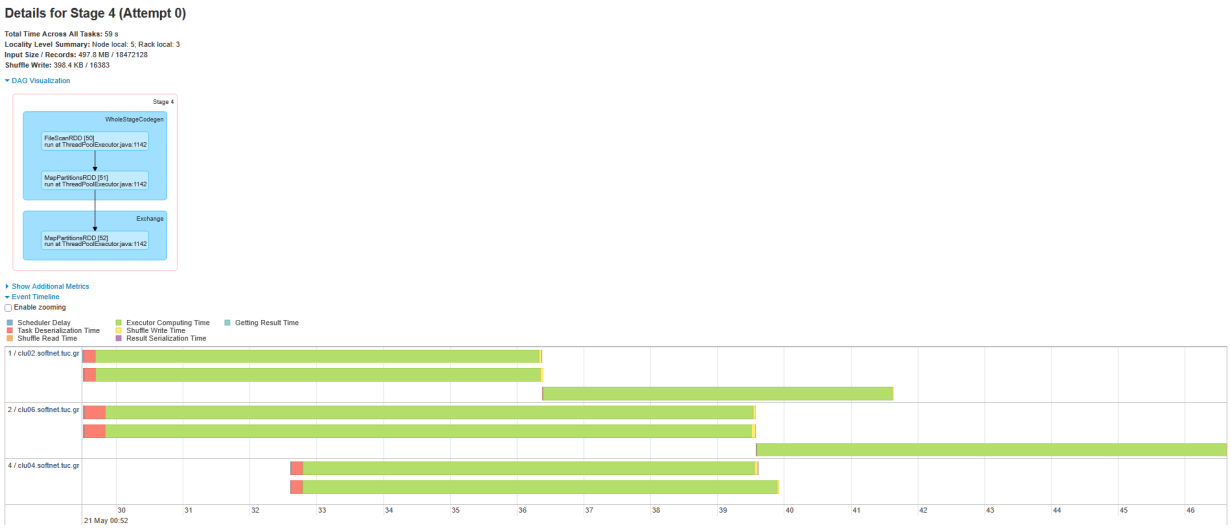


Figure 40: DAG/Event Timeline visualization of Stage 4

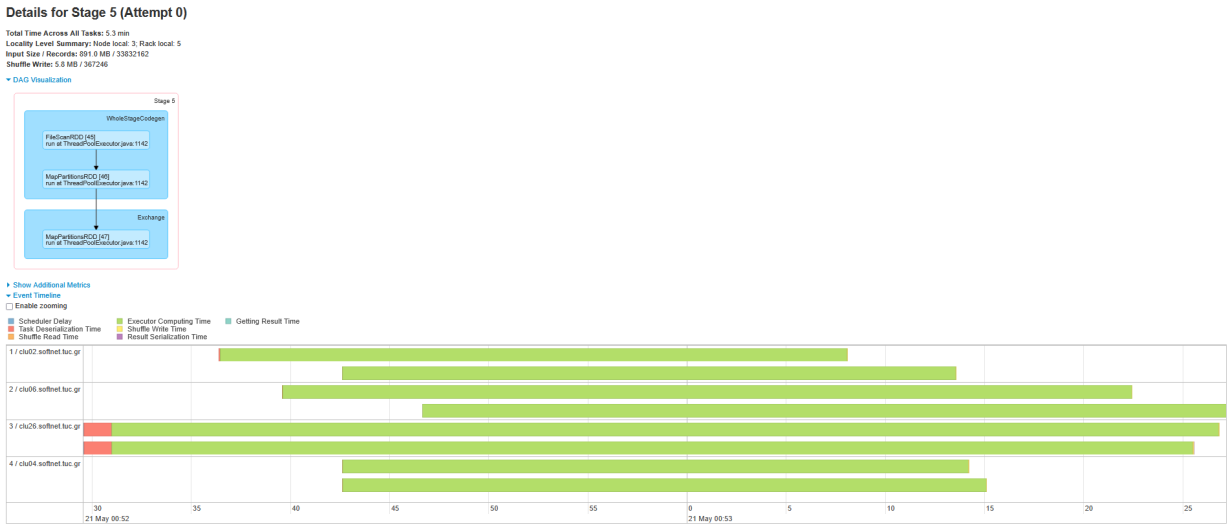


Figure 41: DAG/Event Timeline visualization of Stage 5

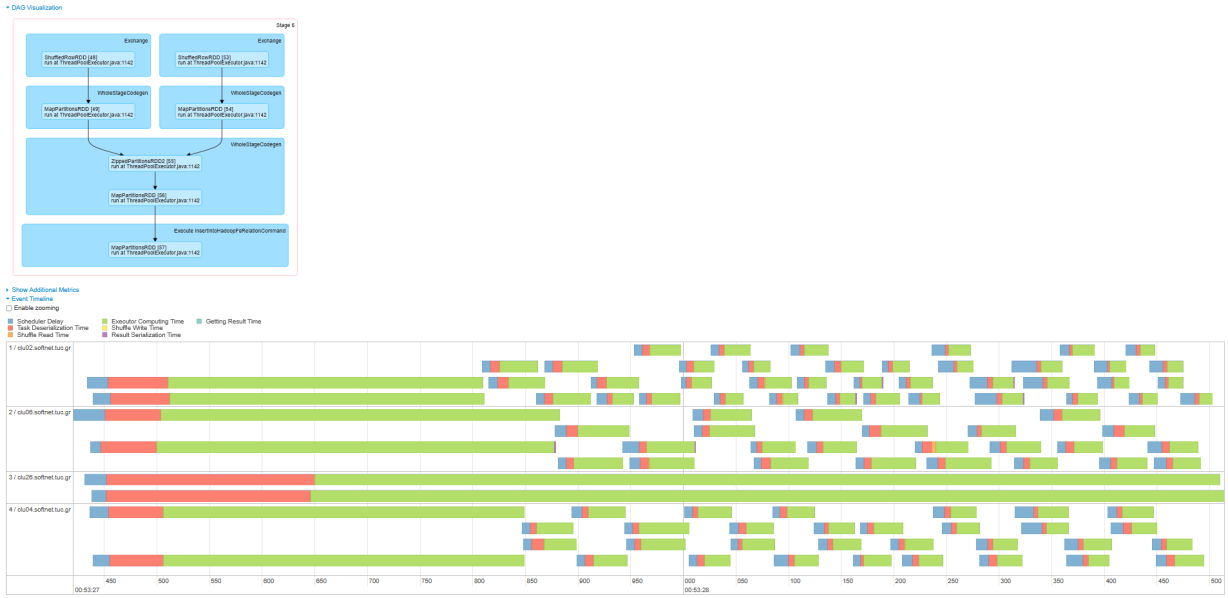


Figure 42: DAG/Event Timeline visualization of Stage 6

We can see that all the above stages are being executed pretty efficiently, with Stages 4 and 5 having a parallelism level of 8, while stage 6 has a parallelism level of 14.

3.2.2 PartB-Job5

Here in Job 5 we have only 1 stage which is directly related to Query 6.

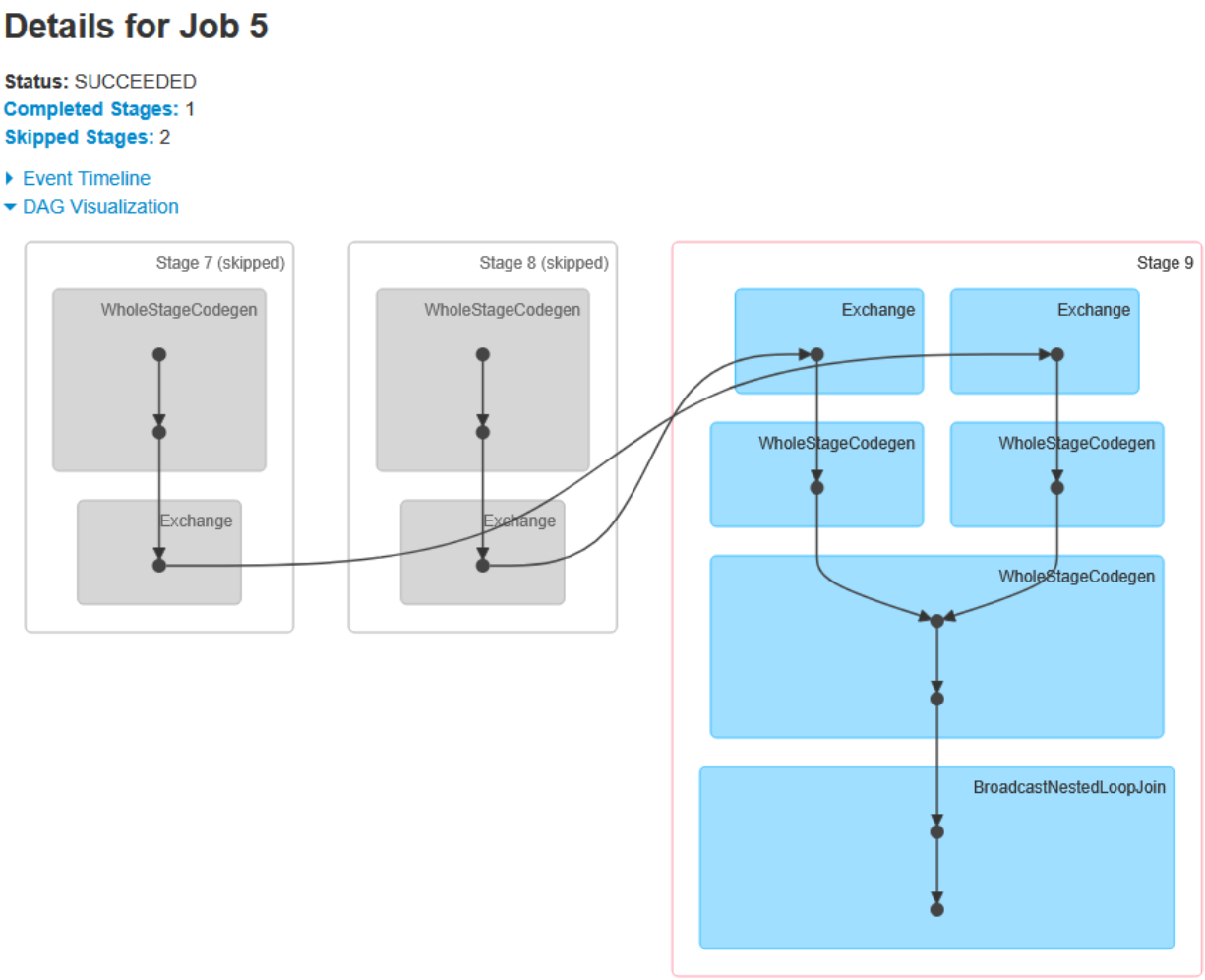


Figure 43: DAG visualization of Job 5

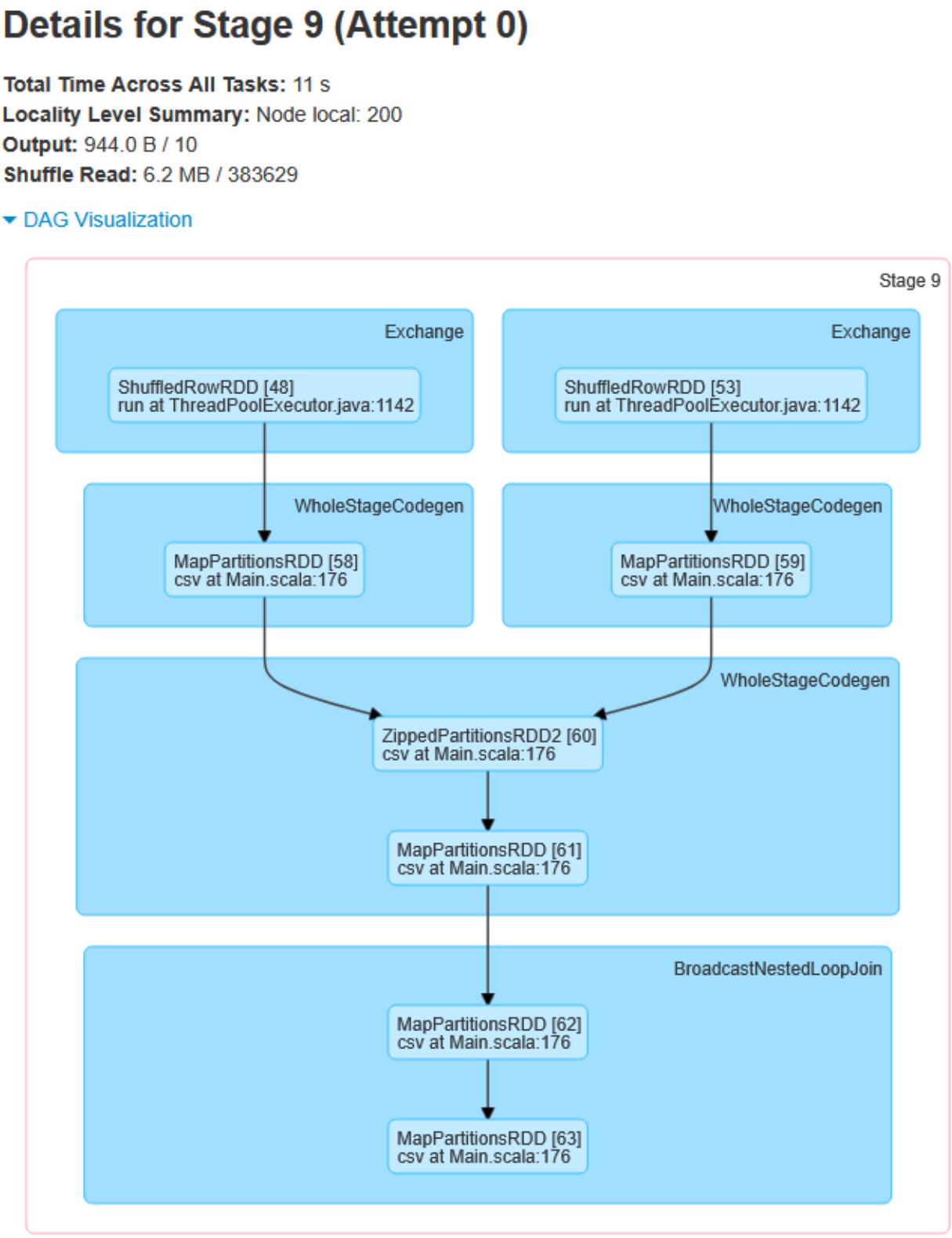


Figure 44: DAG visualization of Stage 9

In Stage 9, the execution begins with two shuffled inputs, ShuffledRowRDD[48] and ShuffledRowRDD[53], representing the two copies of the movieStats DataFrame (statsA and statsB) used in the self-join. These are processed by MapPartitionsRDD[58] and [59], which perform the initial row projections and filtering. The join logic is executed using ZippedPartitionsRDD2[60]. This is followed by MapParti-

tionsRDD[61], where the domination condition of the skyline query is evaluated. Finally, MapPartitionsRDD[62] and [63] represent the final transformation and formatting of the result, culminating in the .write.csv() output action.

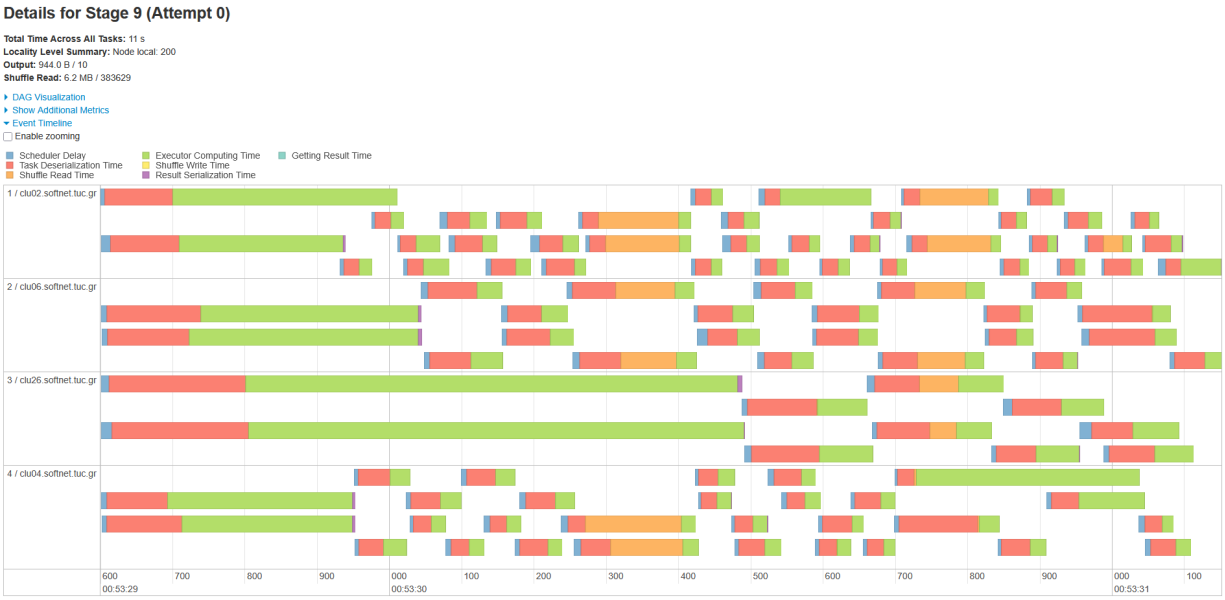


Figure 45: Event Timeline of Stage 9

3.2.3 PartB-Job6

Here in Job 6 we have 4 stages which are directly related to Query 7.
The job begins by aggregating the ratings.csv and genome-scores.csv files to compute the two averages. Stages 10 and 11 read and aggregate each file independently. These results are joined in Stage 12, which performs the heaviest work, involving two exchanges (shuffles) and a treeAggregate to compute the correlation. Finally, Stage 13 collects the scalar correlation result into a single partition using coalesce(1) and writes it to HDFS. The DAG shows clear data dependencies and efficient task execution, with 200 parallel tasks in Stage 12

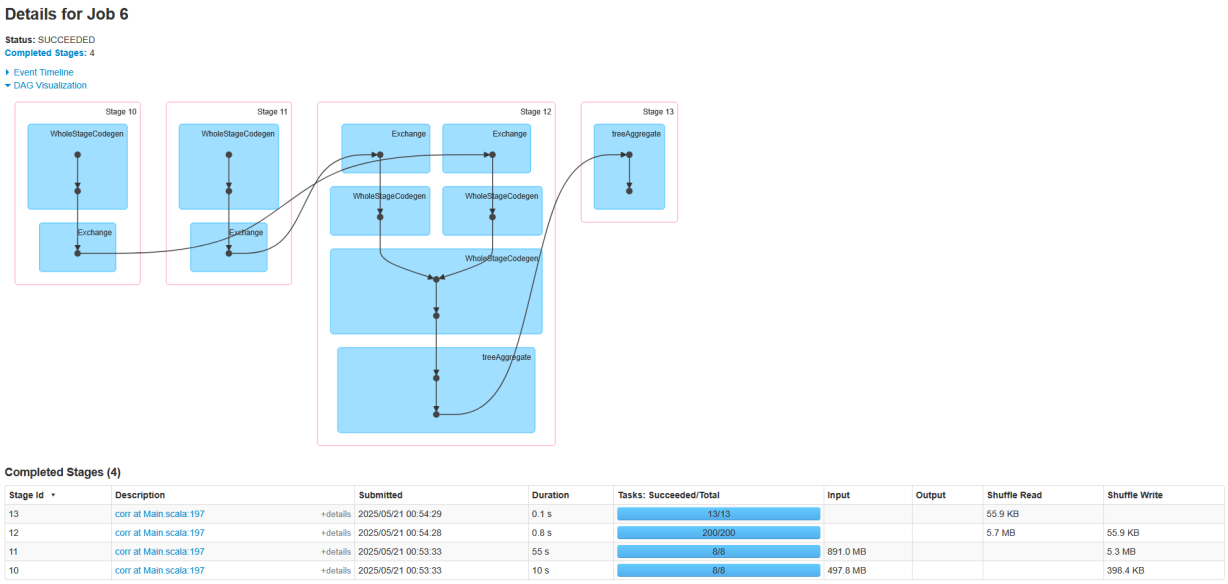


Figure 46: DAG visualization of Job 6

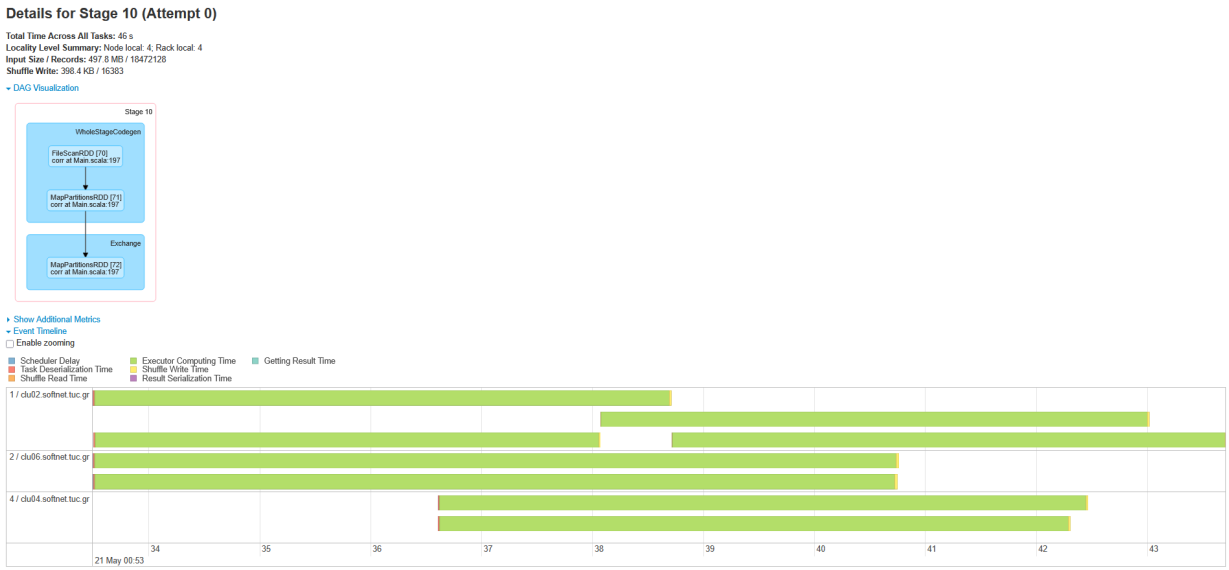


Figure 47: DAG/Event Timeline visualization of Stage 10

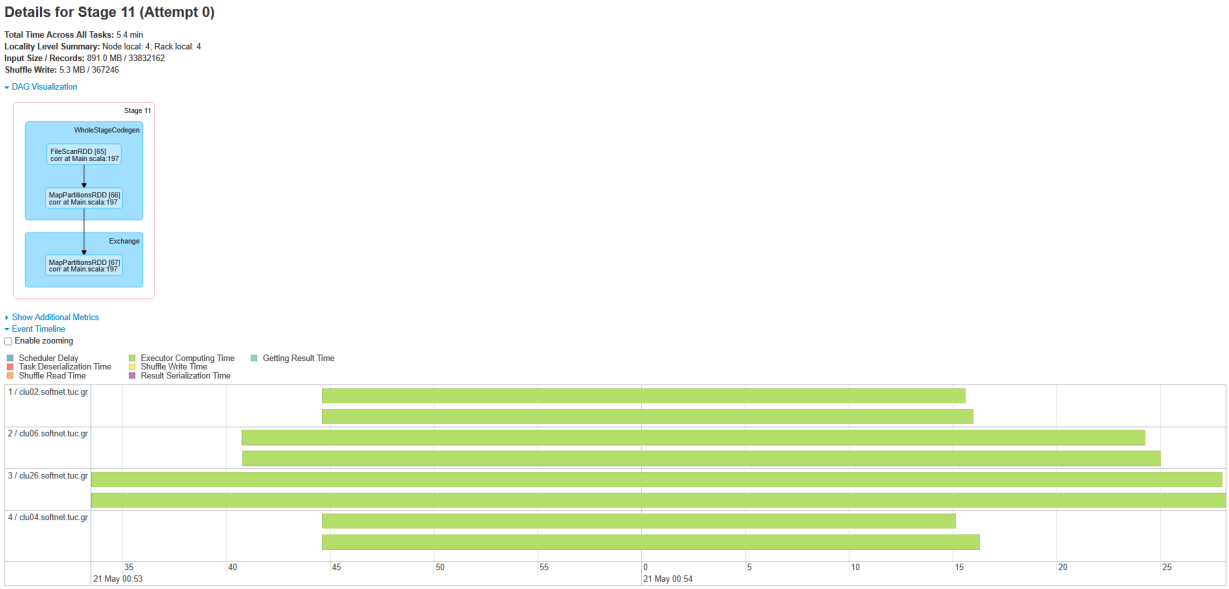


Figure 48: DAG/Event Timeline visualization of Stage 11

Details for Stage 12 (Attempt 0)

Total Time Across All Tasks: 4 s
Locality Level Summary: Node local: 200
Shuffle Read: 5.7 MB / 383629
Shuffle Write: 55.9 KB / 200

▾ DAG Visualization

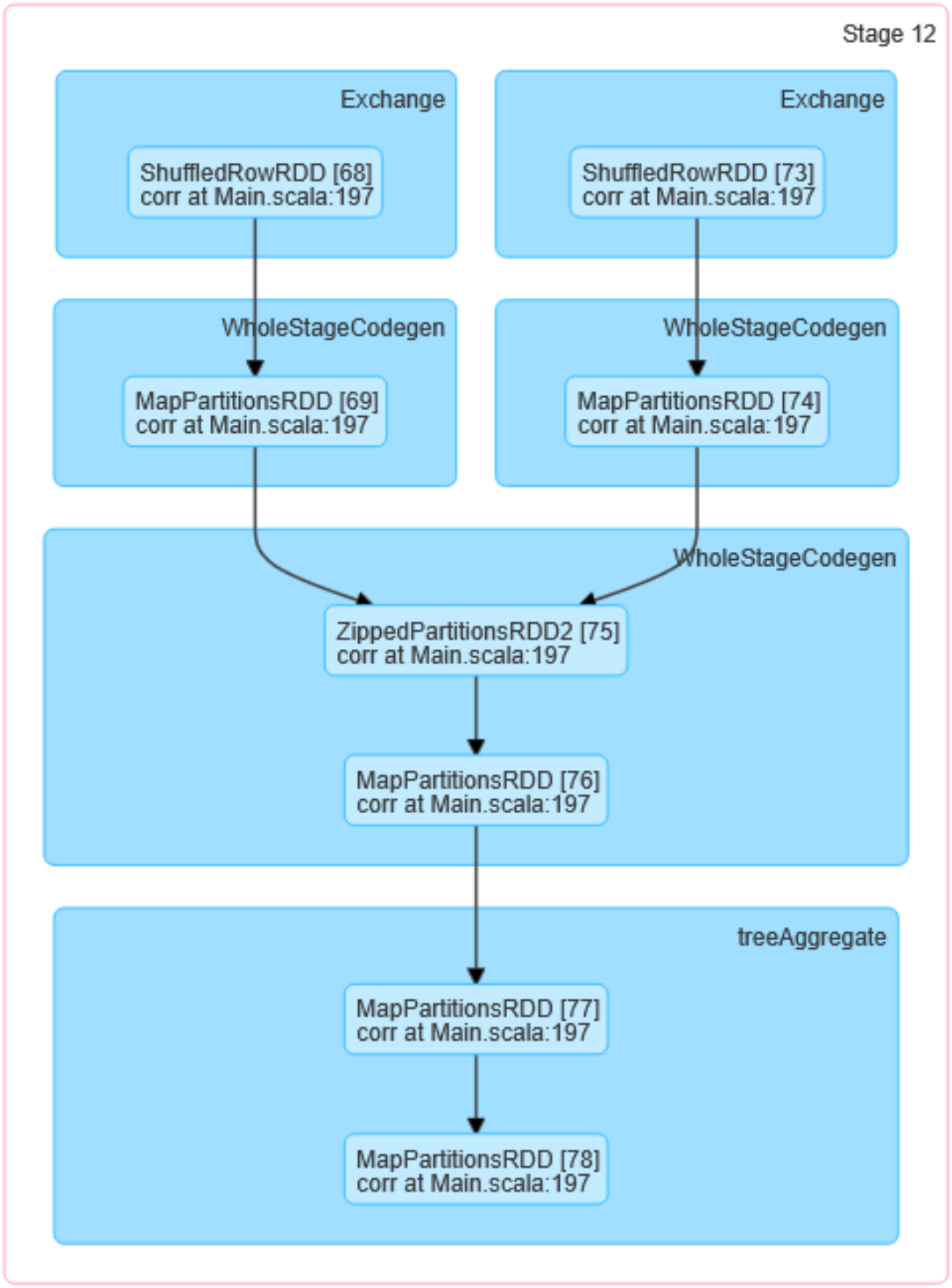


Figure 49: DAG visualization of Stage 12

Figure 50: Event Timeline visualization of Stage 12

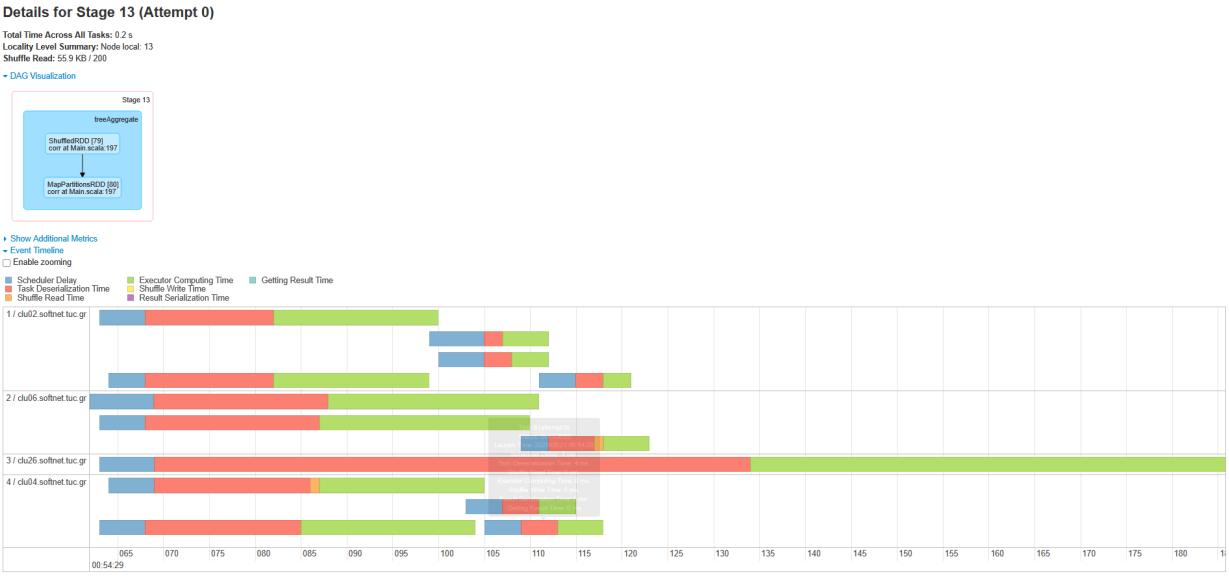


Figure 51: DAG/Event Timeline visualization of Stage 13

3.2.4 PartB-Job7

Job 7 is related to Query 7 and it is responsible for writing the Pearson correlation result to HDFS as a text file. It consists of only one stage, stage 14.

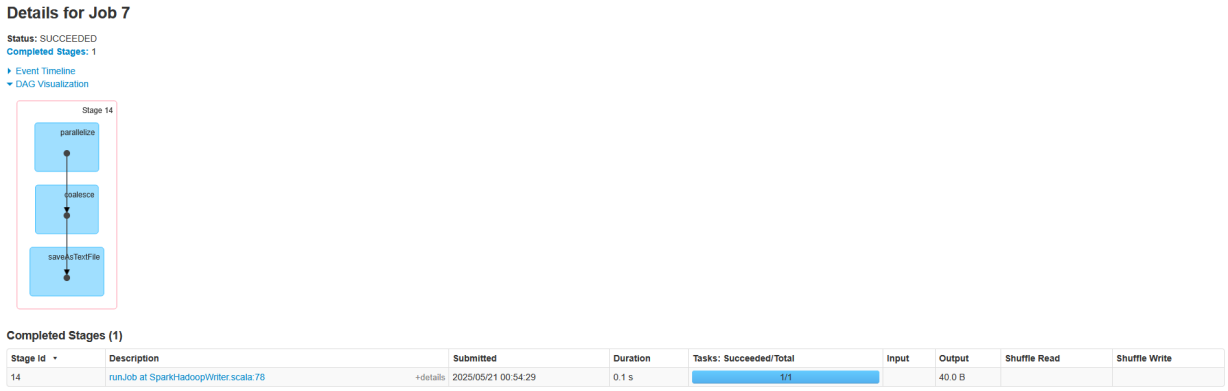


Figure 52: Event Timeline visualization of Stage 12

The DAG of stage 14 shows a linear sequence of operations: `parallelize` to create an RDD, `coalesce(1)` to ensure a single output file, and `saveAsTextFile` to write the result. Since no wide transformations or shuffles occur, Spark executes everything within a single stage. The job runs a single task and completes in 0.1 seconds.

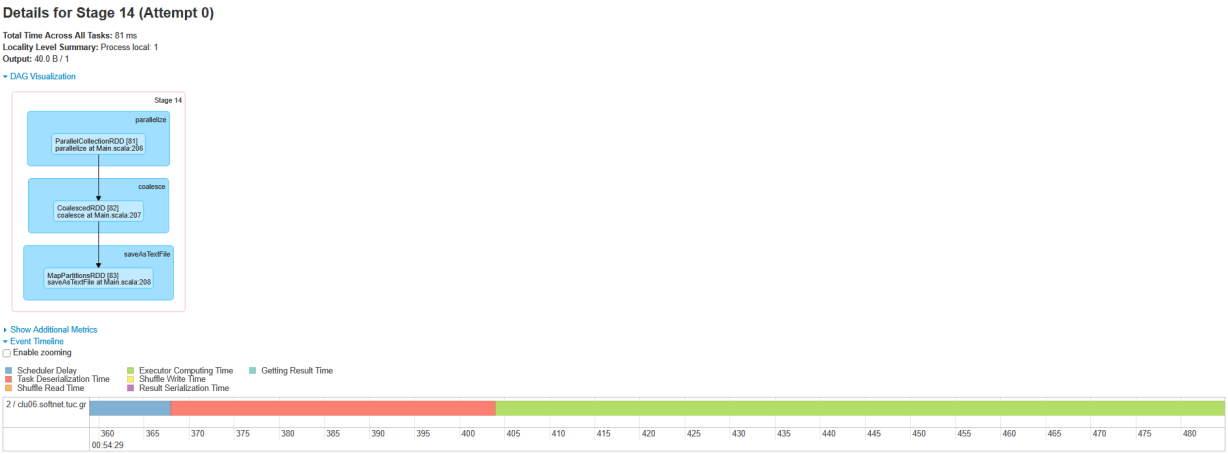


Figure 53: DAG/Event Timeline visualization of Stage 14

3.2.5 PartB-Job8

Job 8 is the 2nd action of Query 7 and it consists of 3 stages.

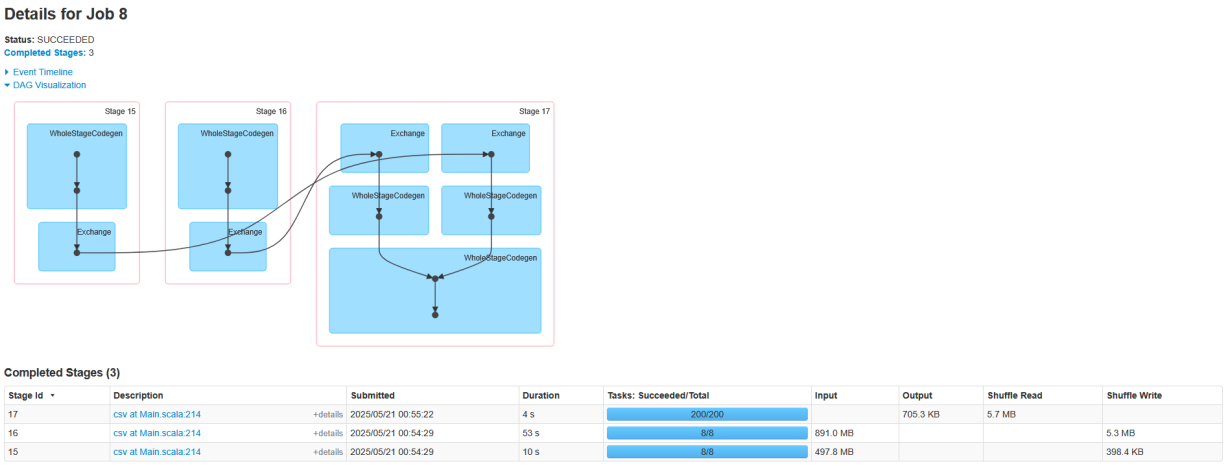


Figure 54: DAG visualization of Job 8

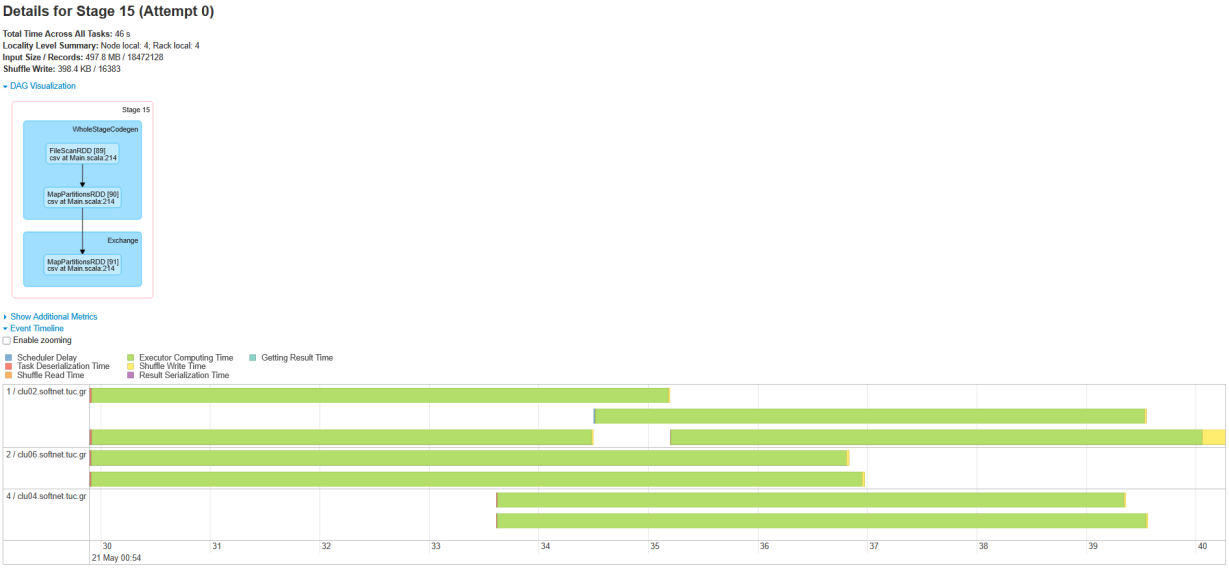


Figure 55: DAG/Event Timeline visualization of Stage 15

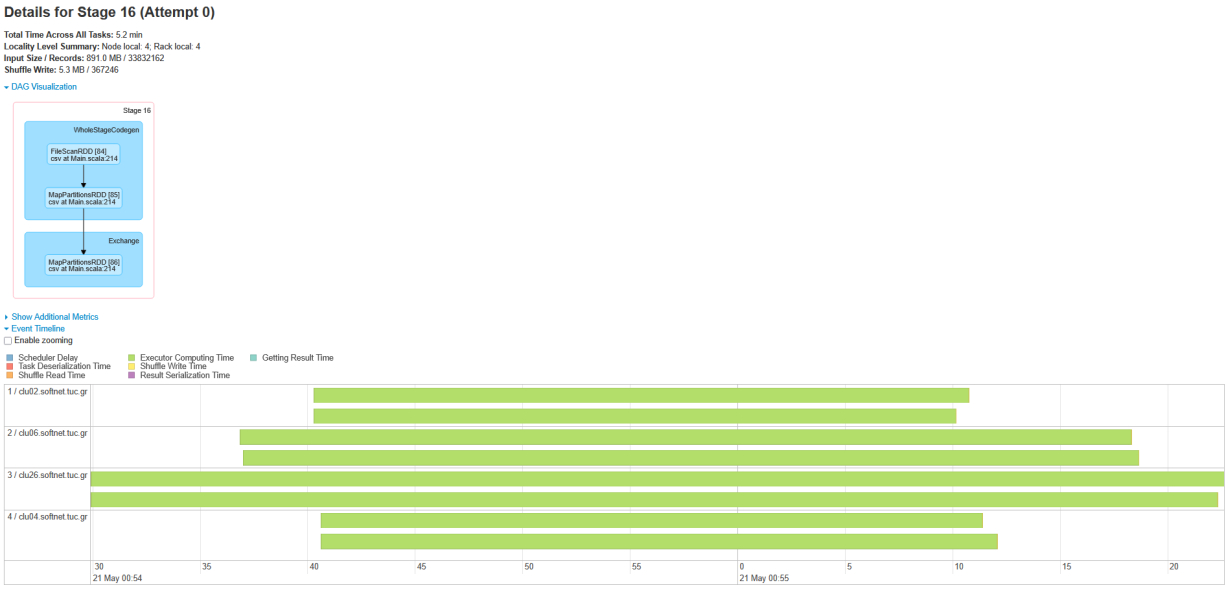


Figure 56: DAG/Event Timeline of Stage 16

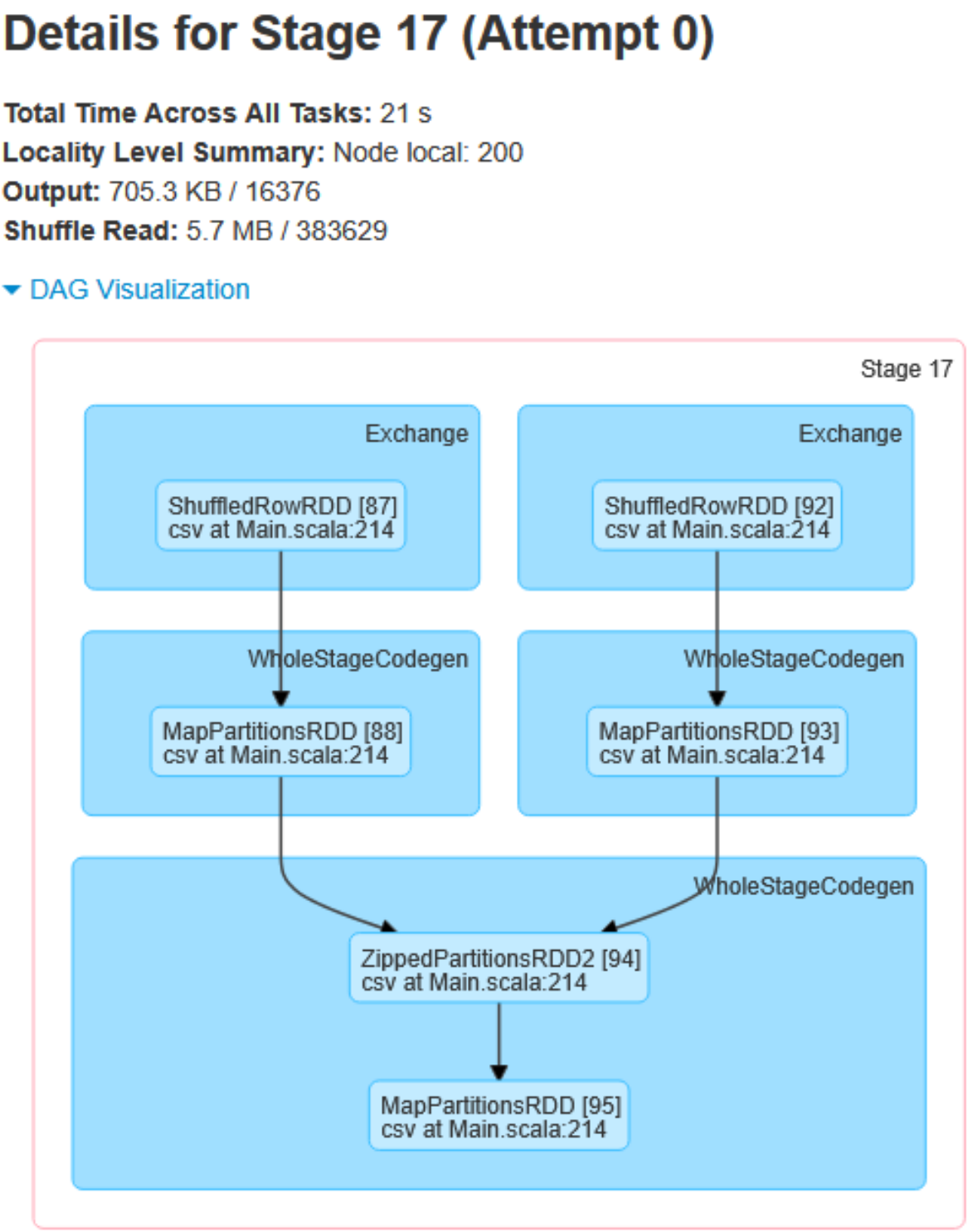


Figure 57: DAG visualization of Stage 17

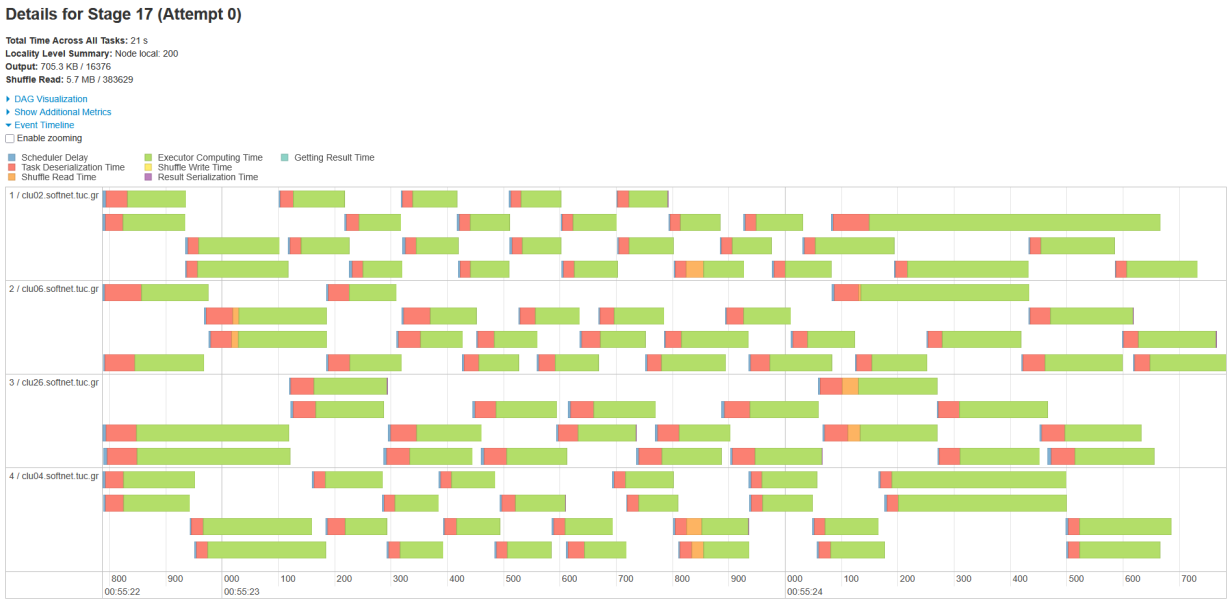


Figure 58: Event Timeline visualization of Stage 17

3.2.6 PartB-Job9

Job 9 also displays `run at ThreadPoolExecutor.java:1142` and it is related to the join operation at Query 9

`Note: Query 9 is executed before Query 8 in our code.`

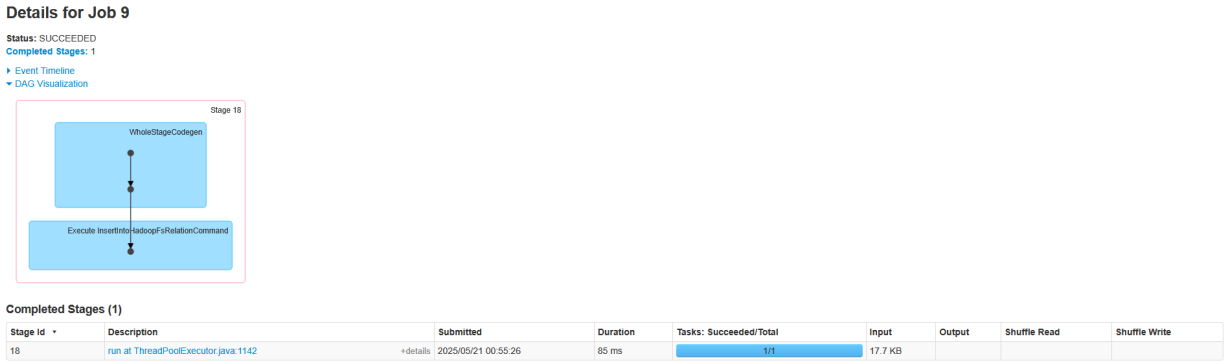


Figure 59: DAG visualization of Job 9

Job 9 consists only of Stage 18. Its DAG shows a file scan, followed by a simple map, and finally a write to HDFS.

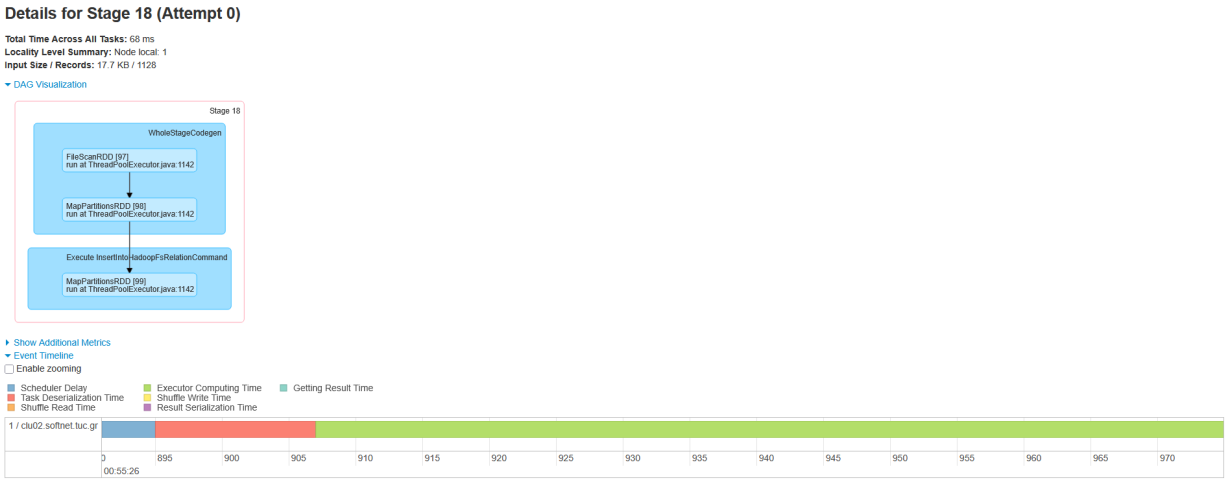


Figure 60: DAG/Event Timeline visualization of Stage 18

3.2.7 PartB-Job10

Job 10 is the 2nd action of Query 9 and consists of 3 stages as well.

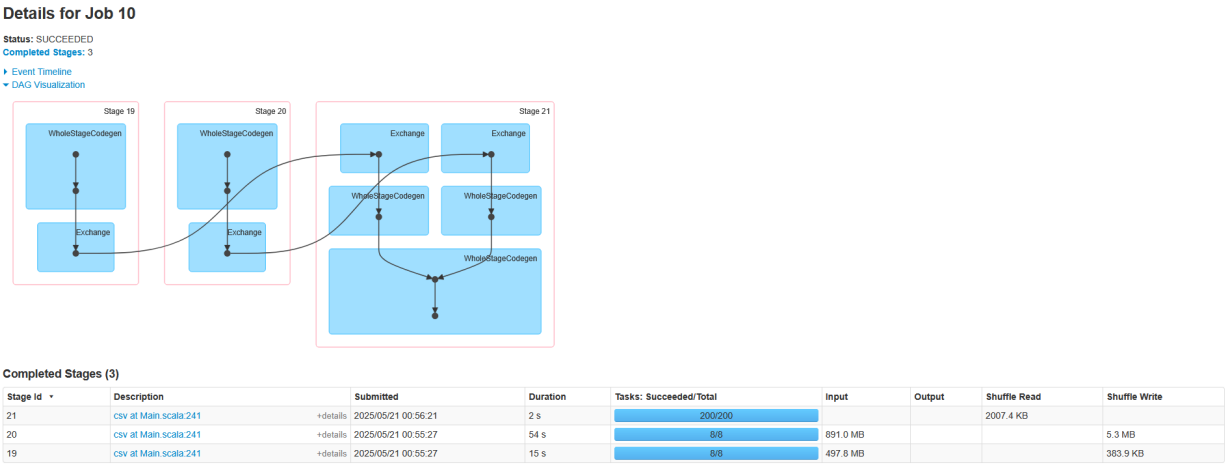


Figure 61: DAG visualization of Job 10

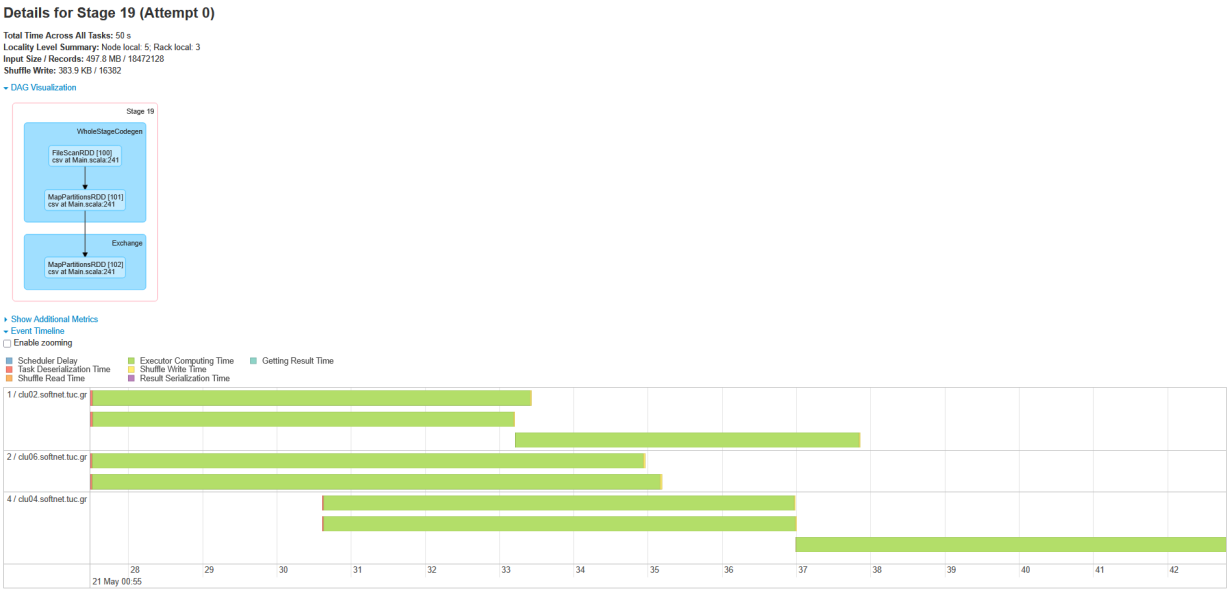


Figure 62: DAG/Event Timeline visualization of Stage 19

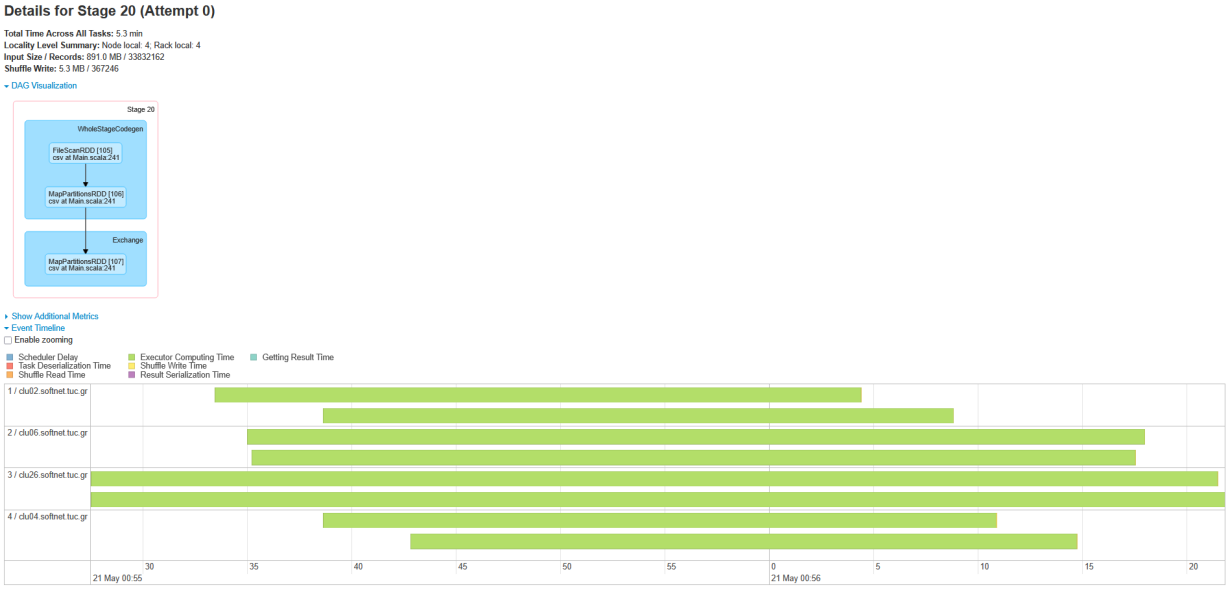


Figure 63: DAG/Event Timeline visualization of Stage 20

Details for Stage 21 (Attempt 0)

Total Time Across All Tasks: 3 s
Locality Level Summary: Node local: 200
Shuffle Read: 2007.4 KB / 38646

▾ DAG Visualization

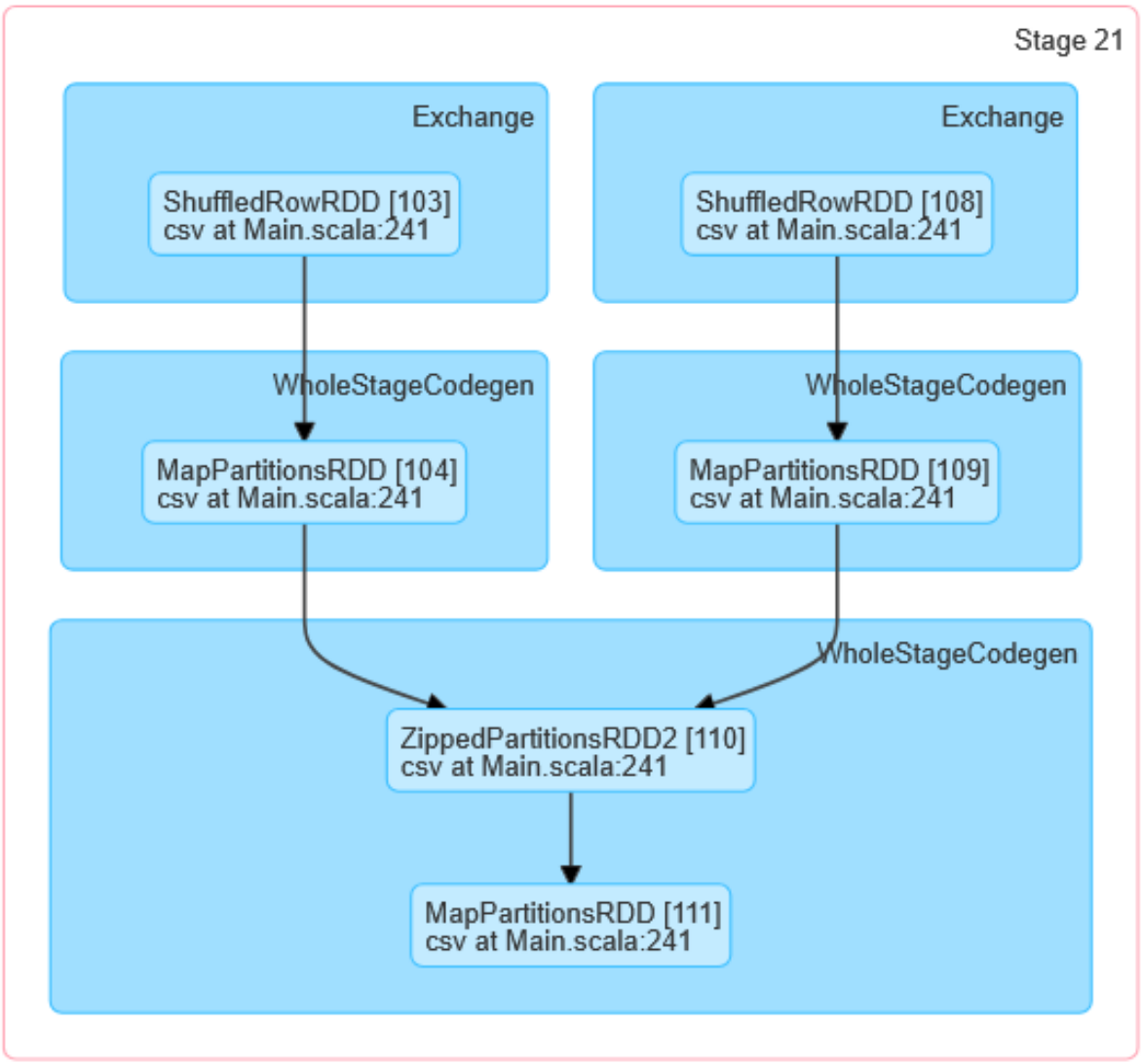


Figure 64: DAG visualization of Stage 21

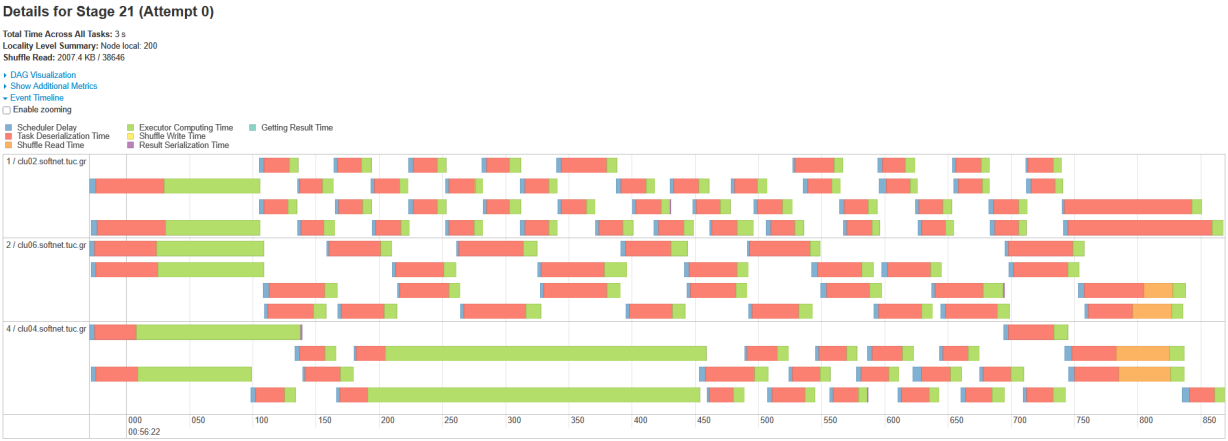


Figure 65: Event Timeline visualization of Stage 21

3.2.8 PartB-Job11

Here, we enter Query 8 with Job 11, which, as evident from the DAG and event timeline screenshots, is the most computationally intensive query in the entire project.

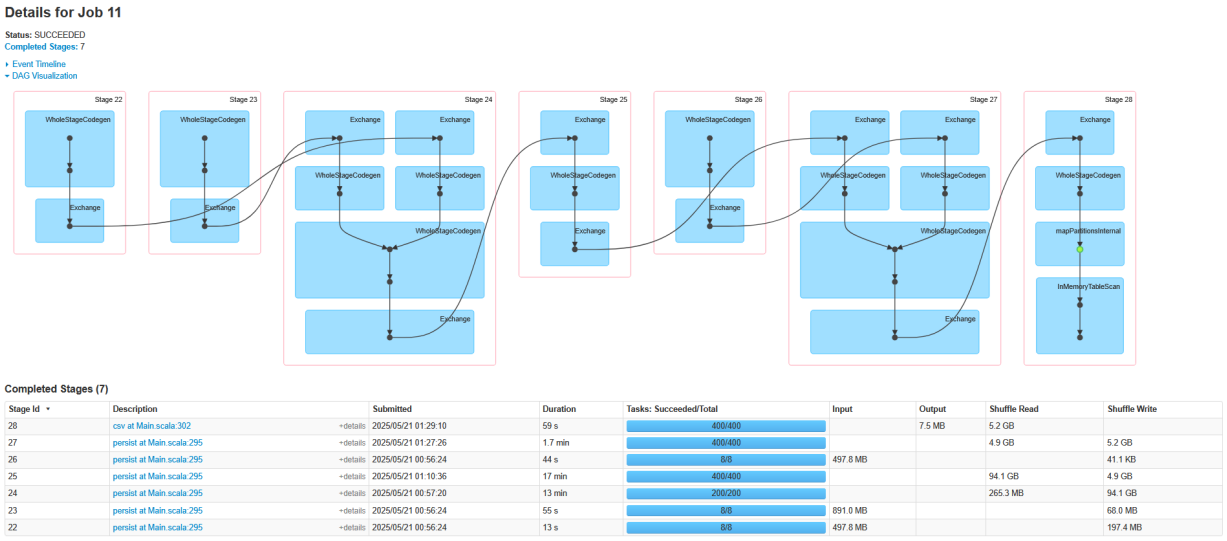


Figure 66: DAG visualization of Job 11

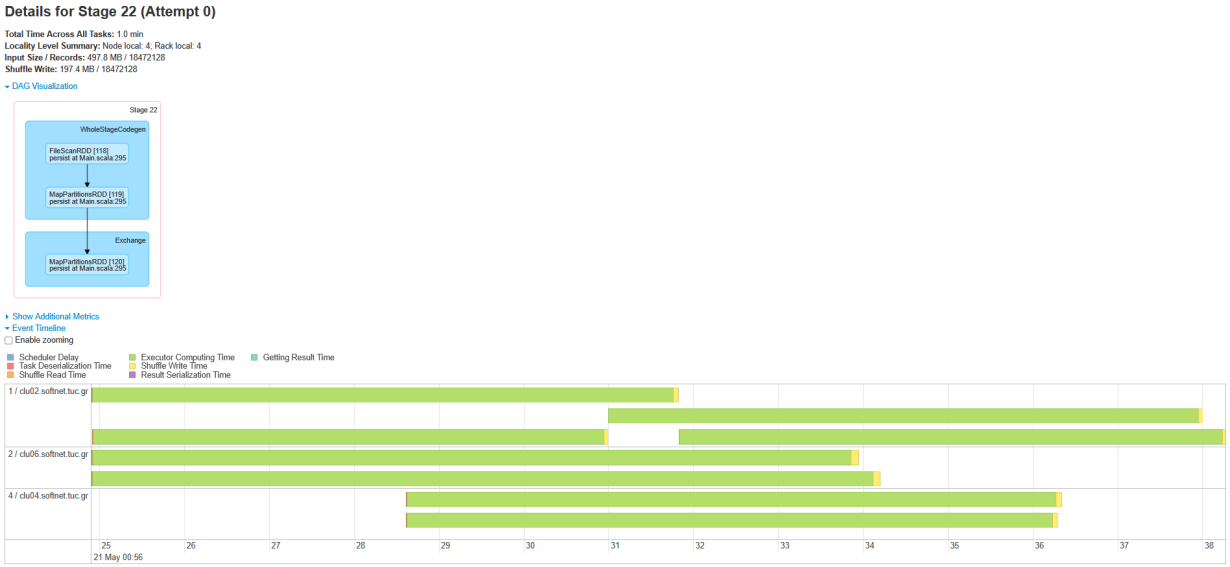


Figure 67: DAG/Event Timeline visualization of Stage 22

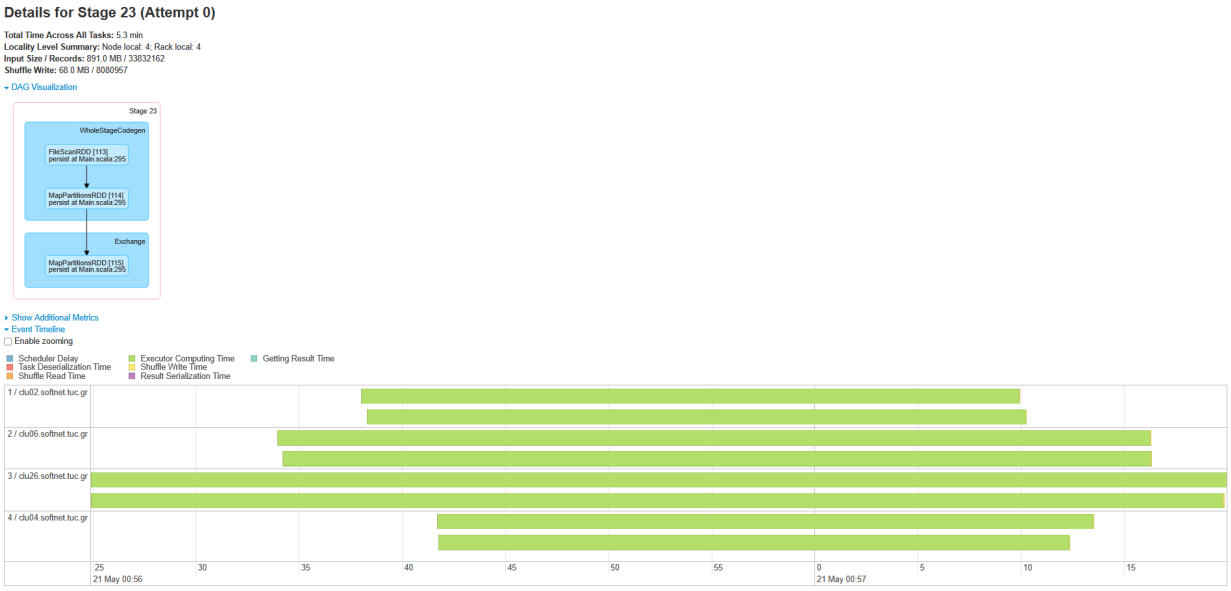


Figure 68: DAG/Event Timeline visualization of Stage 23

Details for Stage 24 (Attempt 0)

Total Time Across All Tasks: 1.7 h
Locality Level Summary: Node local: 200
Shuffle Read: 265.3 MB / 26553085
Shuffle Write: 94.1 GB / 9046644600
Shuffle Spill (Memory): 396.4 GB
Shuffle Spill (Disk): 79.0 GB

▾ DAG Visualization

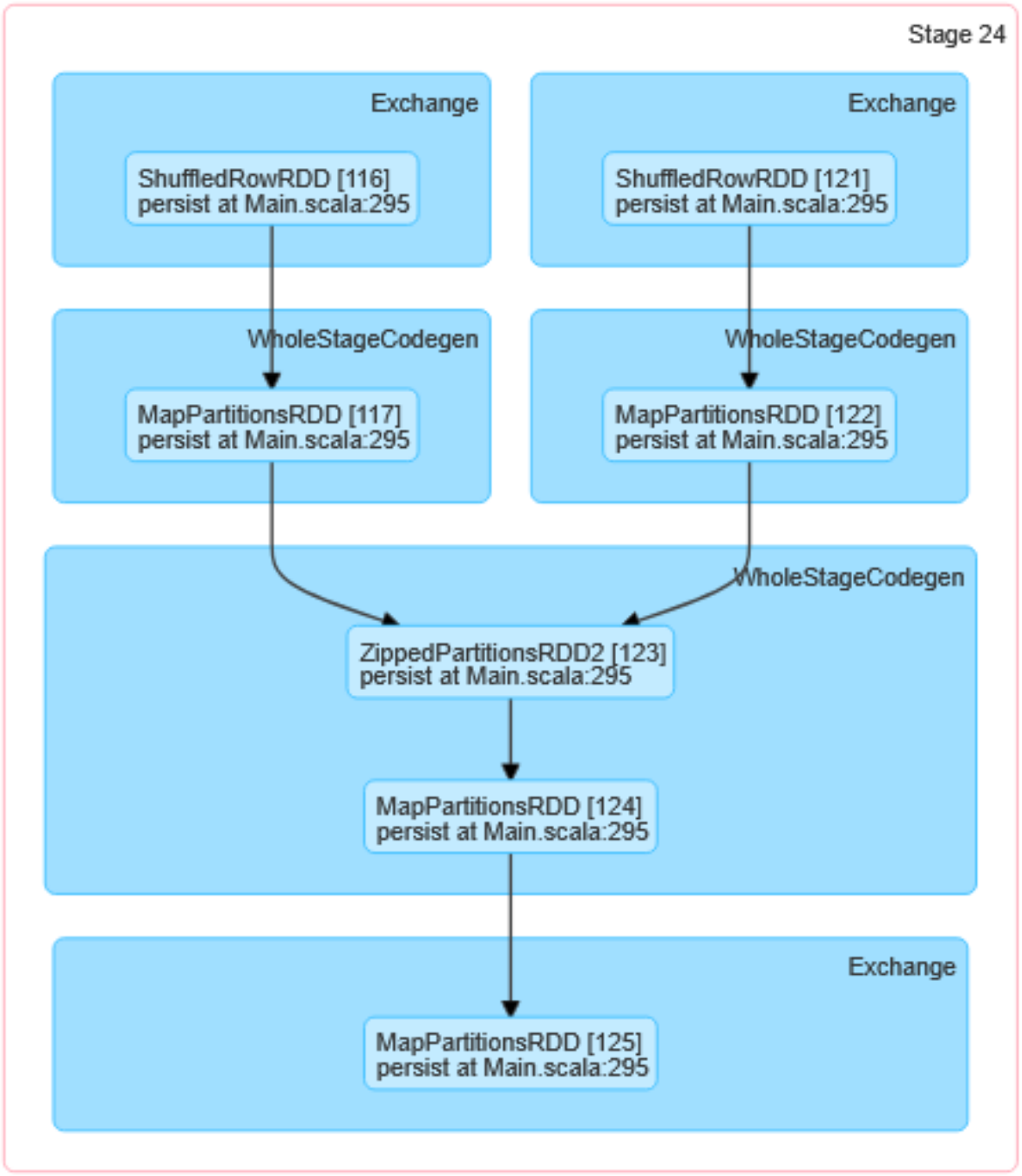


Figure 69: DAG visualization of Stage 24

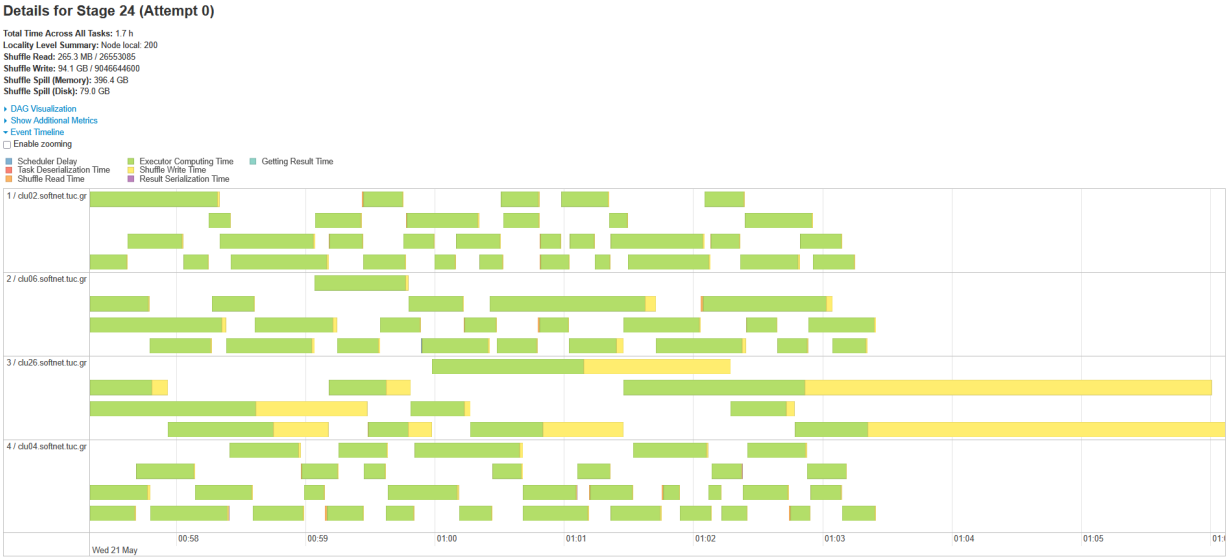


Figure 70: Event Timeline visualization of Stage 24

Details for Stage 25 (Attempt 0)

Total Time Across All Tasks: 2.2 h
Locality Level Summary: Node local: 303; Rack local: 97
Shuffle Read: 94.1 GB / 9046644600
Shuffle Write: 4.9 GB / 344731464

▾ DAG Visualization

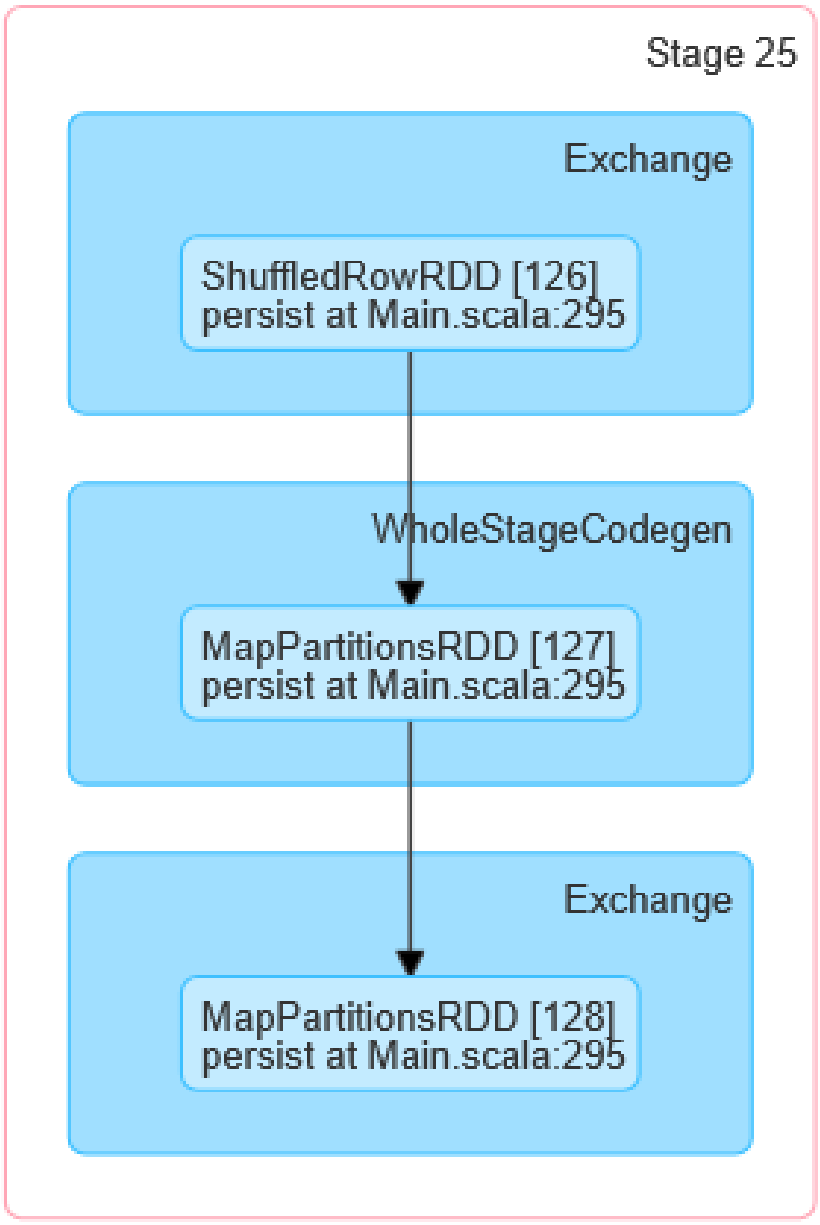


Figure 71: DAG visualization of Stage 25

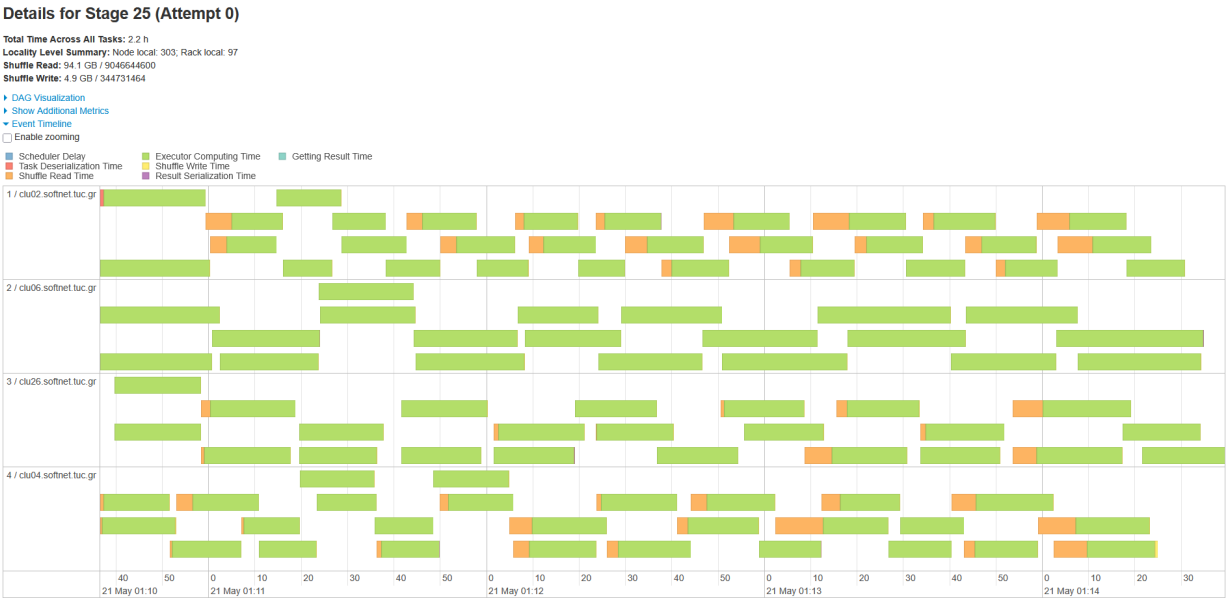


Figure 72: Event Timeline visualization of Stage 25

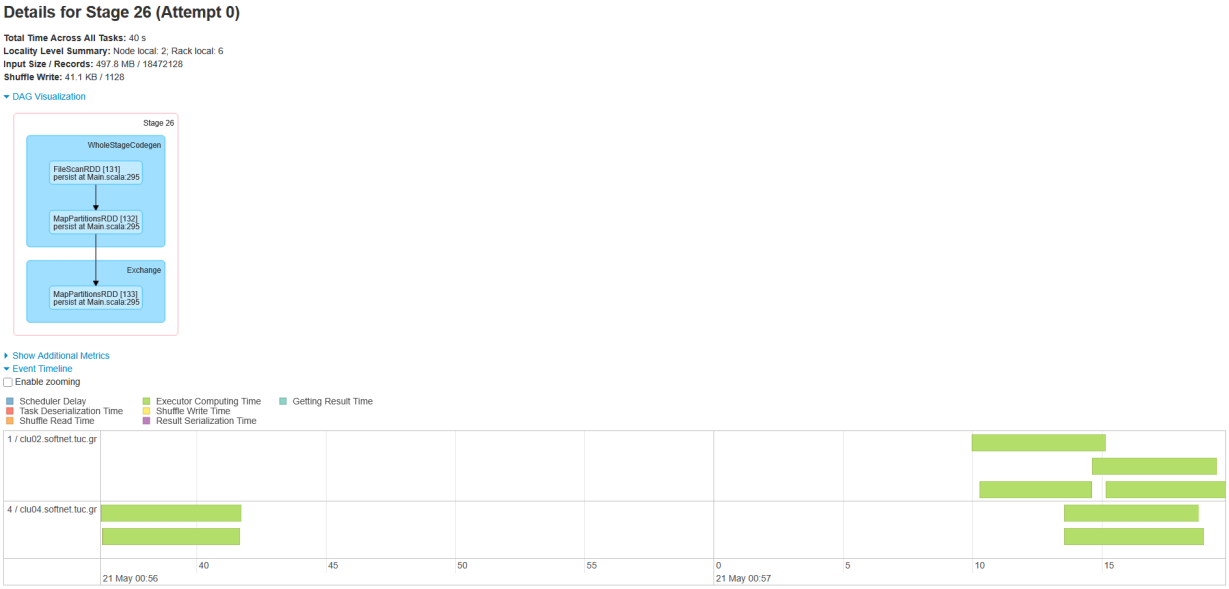


Figure 73: DAG/Event Timeline visualization of Stage 26

Details for Stage 27 (Attempt 0)

Total Time Across All Tasks: 14 min
Locality Level Summary: Node local: 375; Process local: 25
Shuffle Read: 4.9 GB / 344732592
Shuffle Write: 5.2 GB / 344731464

▾ DAG Visualization

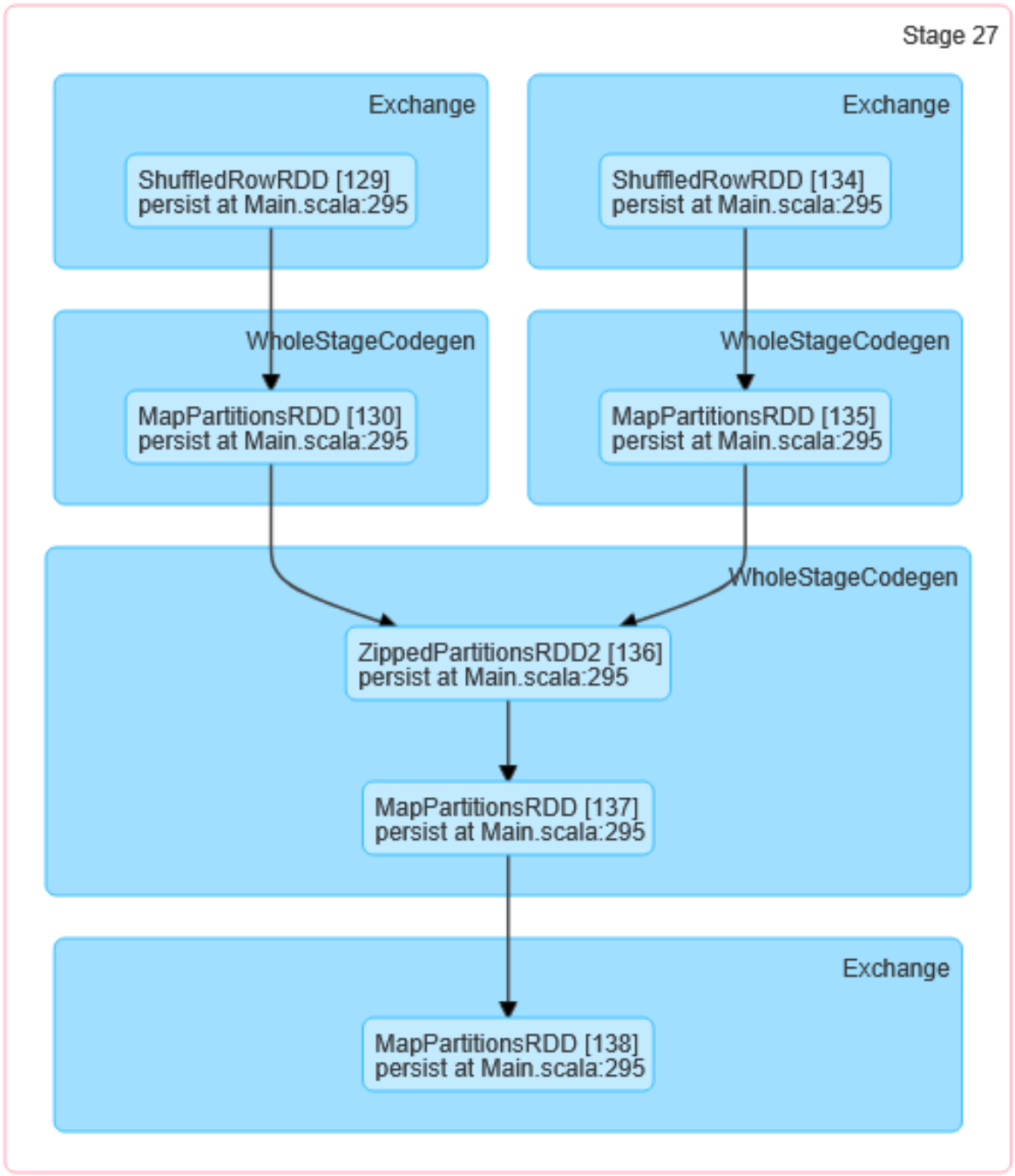


Figure 74: DAG visualization of Stage 20

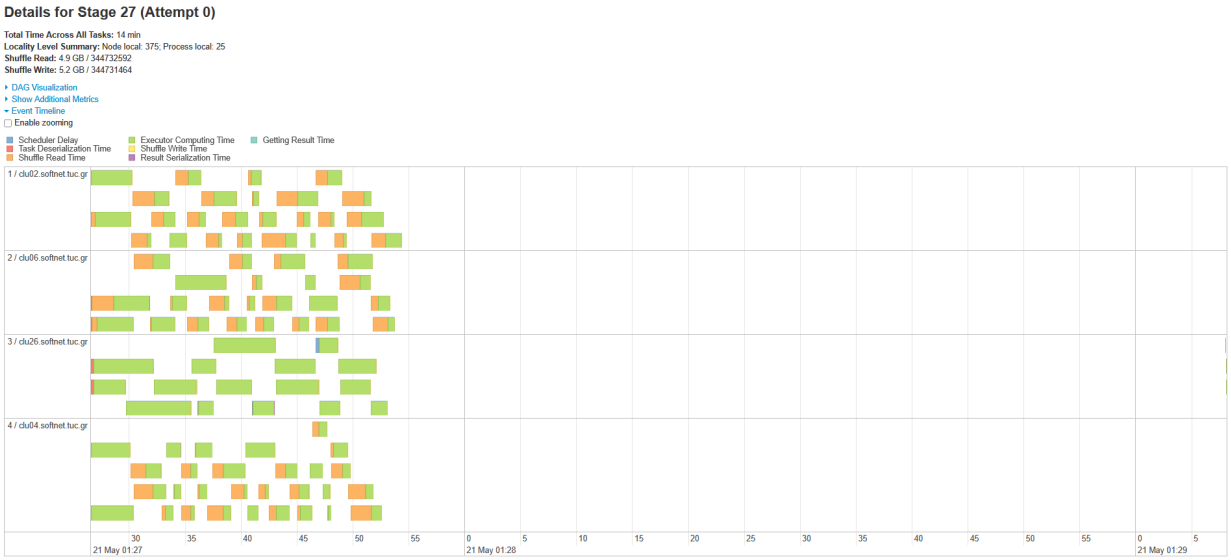


Figure 75: Event Timeline visualization of Stage 20

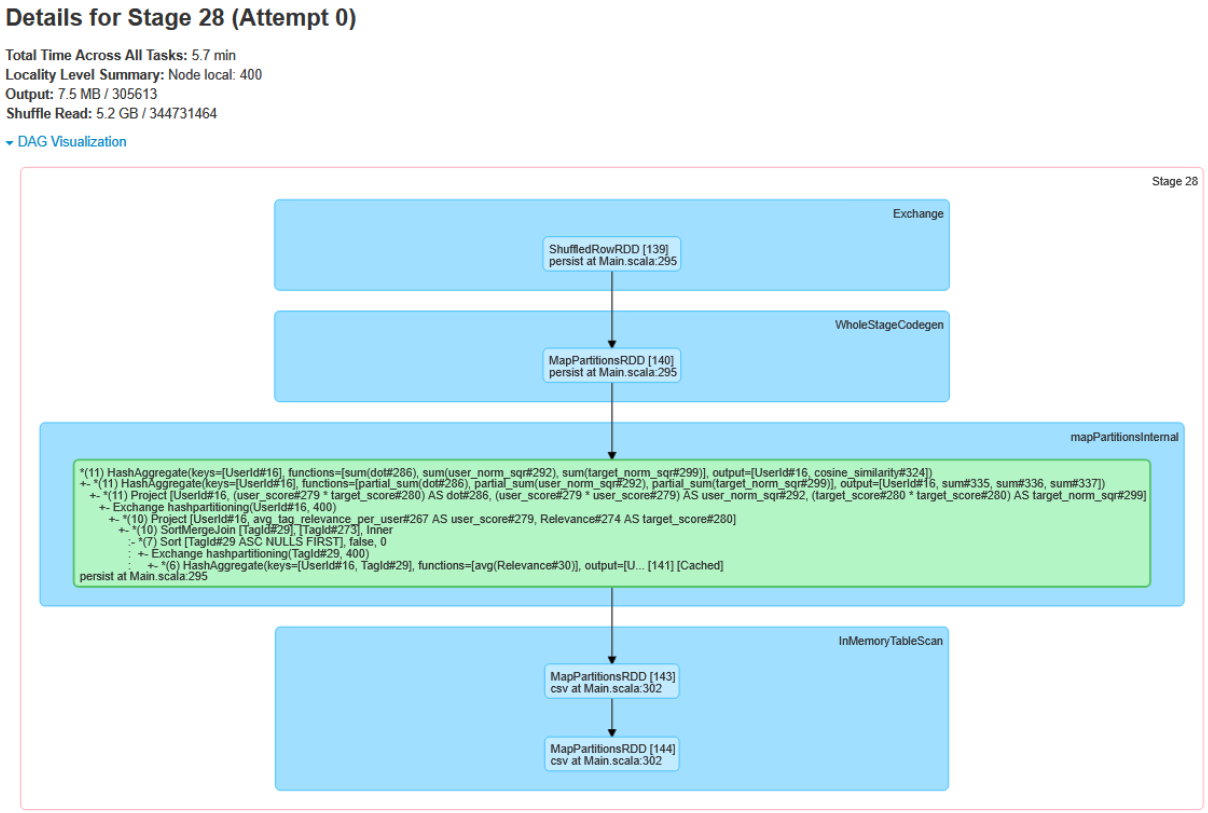


Figure 76: DAG visualization of Stage 20

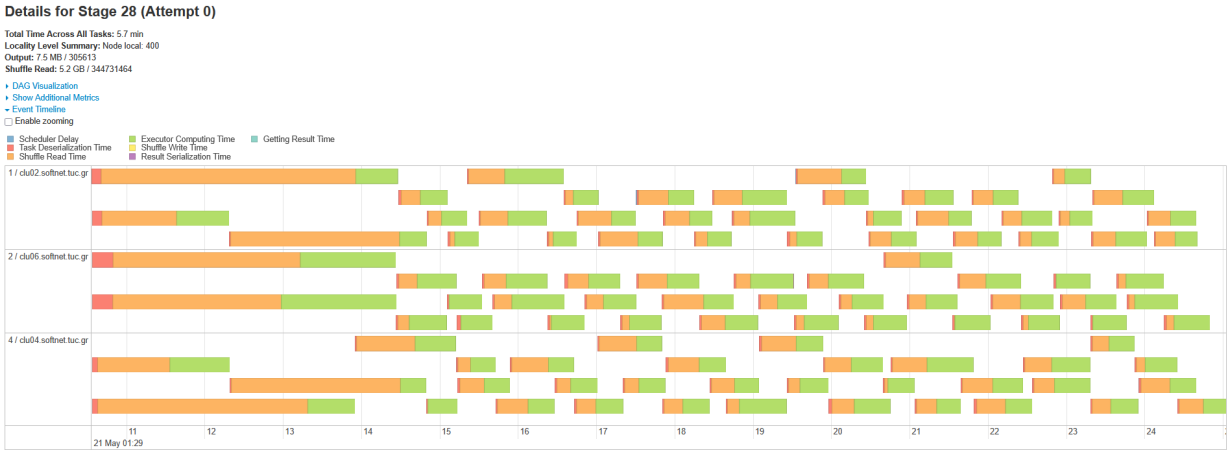


Figure 77: Event Timeline visualization of Stage 20

The above screenshots, while they might show a high level of efficient parallelism, the query still takes approximately 40 minutes. This is natural because both the amount of times there needs to be a shuffling(evident by the number of stages) and the size of those shuffles are substantial.

3.2.9 PartB-Job12

Job 12 is triggered by our final Query via `.cvs()` operation.

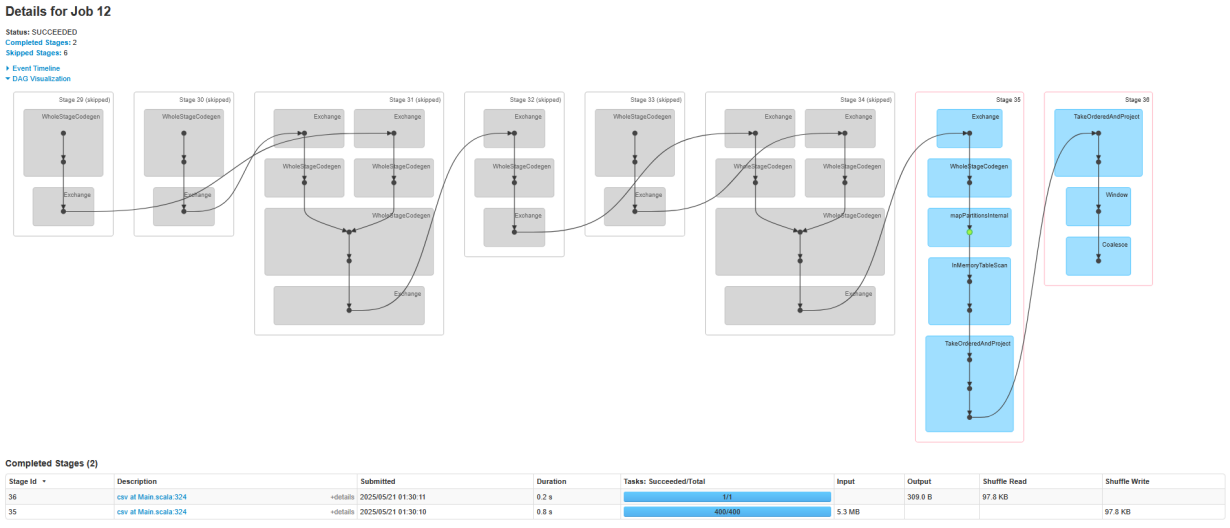


Figure 78: DAG visualization of Job 12

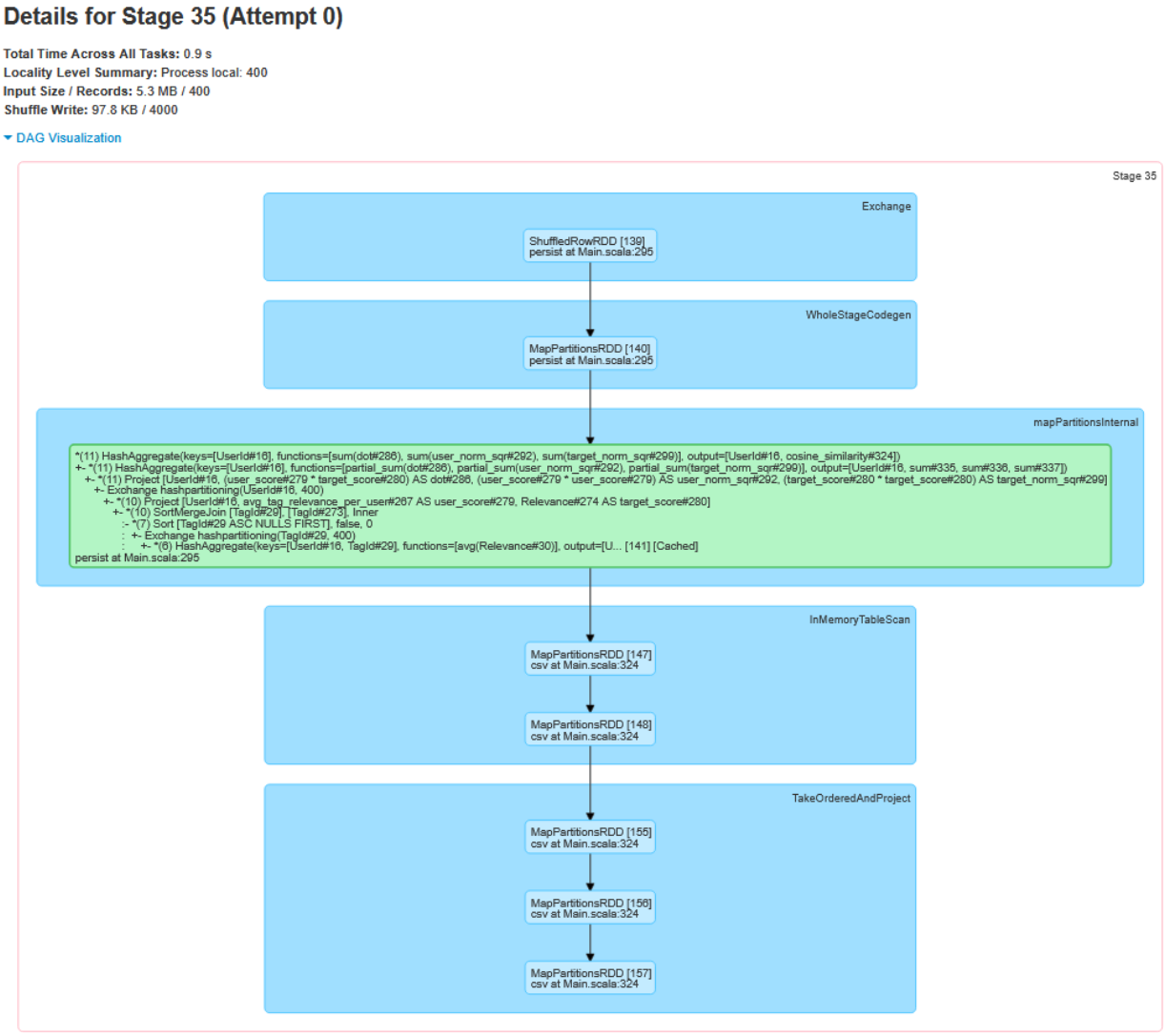


Figure 79: DAG visualization of Stage 35

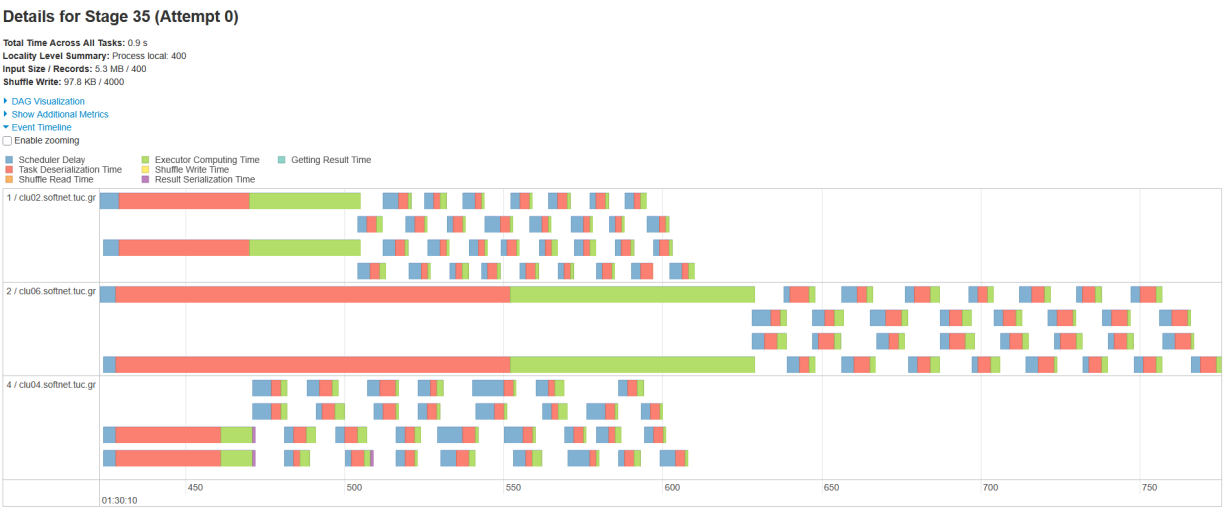


Figure 80: Event Timeline visualization of Stage 35

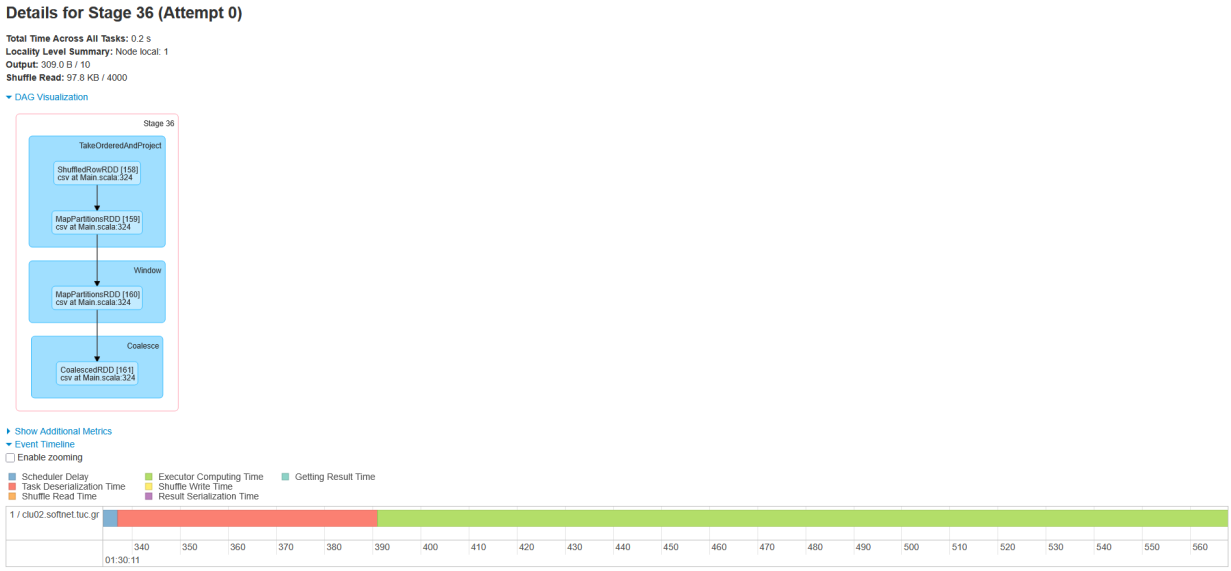


Figure 81: DAG/Event Timeline visualization of Stage 36

From the screenshot that shows the job, we can see that most of the stages are very similar with the stages of query 8 that were executed. These same stages we can see that are being skipped by the Spark's optimizer. This probably happens because spark had already that data loaded(because of out use of persist) and does not need to re-compute all those stages.

4 Sample Output

4.1 Output Files

It is important to note that in order for the above computations to be executed and their results to be materialized as files, we must invoke a Spark action on the respective RDDs or DataFrames.

In our implementation:

- For RDDs, we use the action `.saveAsTextFile()` to write the output to HDFS as text file.
- For DataFrames, we use `.write().csv()` to save the data in CSV format.

However, Spark will throw an exception if the specified output path already exists. To avoid such failures, we handle this differently depending on the method used:

- For DataFrames, we chain the method `.mode("overwrite")` before writing to ensure that any existing output directory is automatically replaced.
- For RDDs, which don't support overwrite mode directly, we manually check for the existence of the path and delete it beforehand using:

```
if (hdfs.exists(new org.apache.hadoop.fs.Path(outputPathName))) {
    hdfs.delete(new org.apache.hadoop.fs.Path(outputPathName), true)
}
```

This ensures that each query can safely write its output without manual intervention, regardless of whether the directory already exists from a previous execution.

4.2 Output File Partitioning

The number of output files generated by Spark depends directly on the number of partitions in the final RDD or DataFrame at the time the `.saveAsTextFile()` or `.write().csv()` action is invoked. Below are three representative examples from our project:

- **Query 1**: This query had no repartitioning before writing the output, and therefore the number of part files reflects the default parallelism level — in our case, 8 files were produced (see Figure 82).
- **Query 5**: Here, we explicitly used `.repartition(200)` to improve parallel processing and load balancing for the intensive aggregation step. As expected, the output folder contains exactly 200 part files (see Figure 83).
- **Query 7 (Pearson)**: Since this query results in a single line of output, we used `.coalesce(1)` before saving to force Spark to consolidate the data into a single output file (see Figure 84).

This shows how important it is to manage the number of partitions in Spark. Using `repartition()` helped us split heavy workloads across many executors and speed up the computation, while `coalesce()` was useful when the output was small and we just wanted a single, clean file instead of many part files. So, depending on the query and its size, adjusting the number of partitions gave us better performance and more organized results.

After transferring all the results from the cluster to our local machine, we can now analyze and discuss them in more detail.

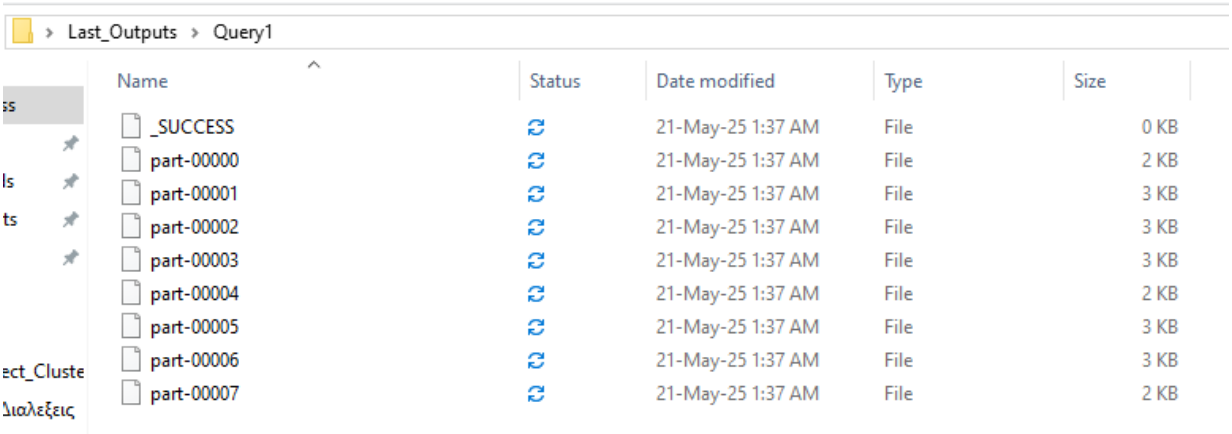


Figure 82: Output files of Query 1 (no manual repartitioning)

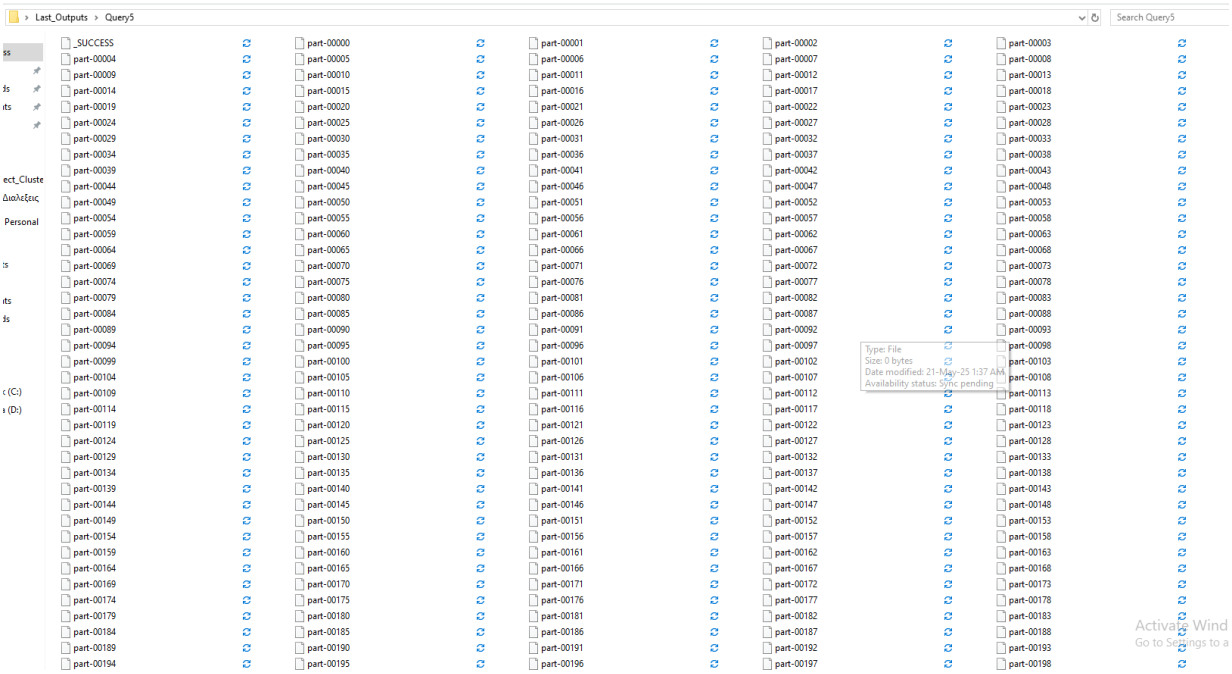


Figure 83: Output files of Query 5 (after `repartition(200)`)

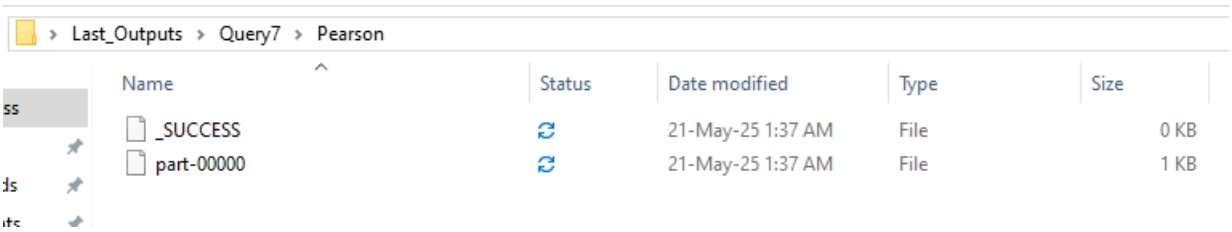


Figure 84: Output files of Query 7 (after `coalesce(1)`)

4.3 Part A

4.3.1 Query 1: Iceberg Query

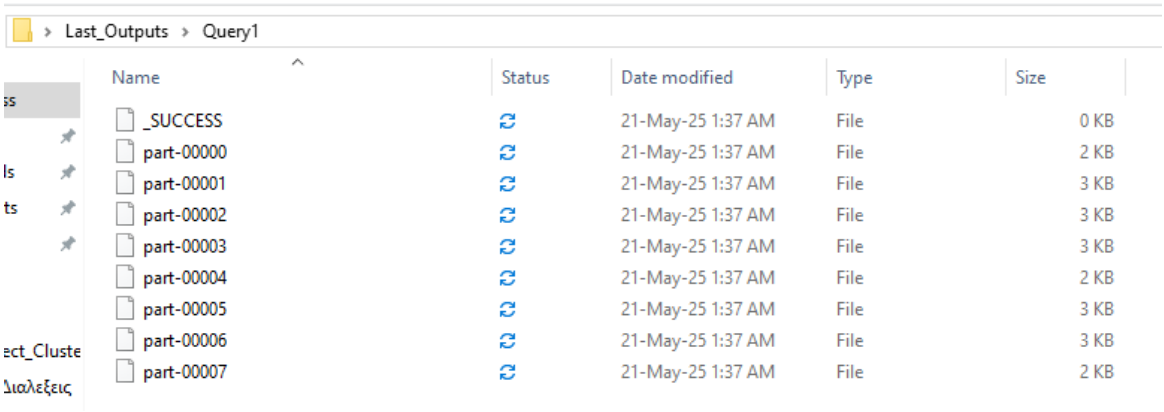


Figure 85: Output files for Query 1

As we can see, Query 1 generated only 8 part files. This relatively low number is due to the lack of manual repartitioning, allowing Spark to use its default level of parallelism. The data size is also small (a few kilobytes each), which reflects the nature of the aggregation task in this query.

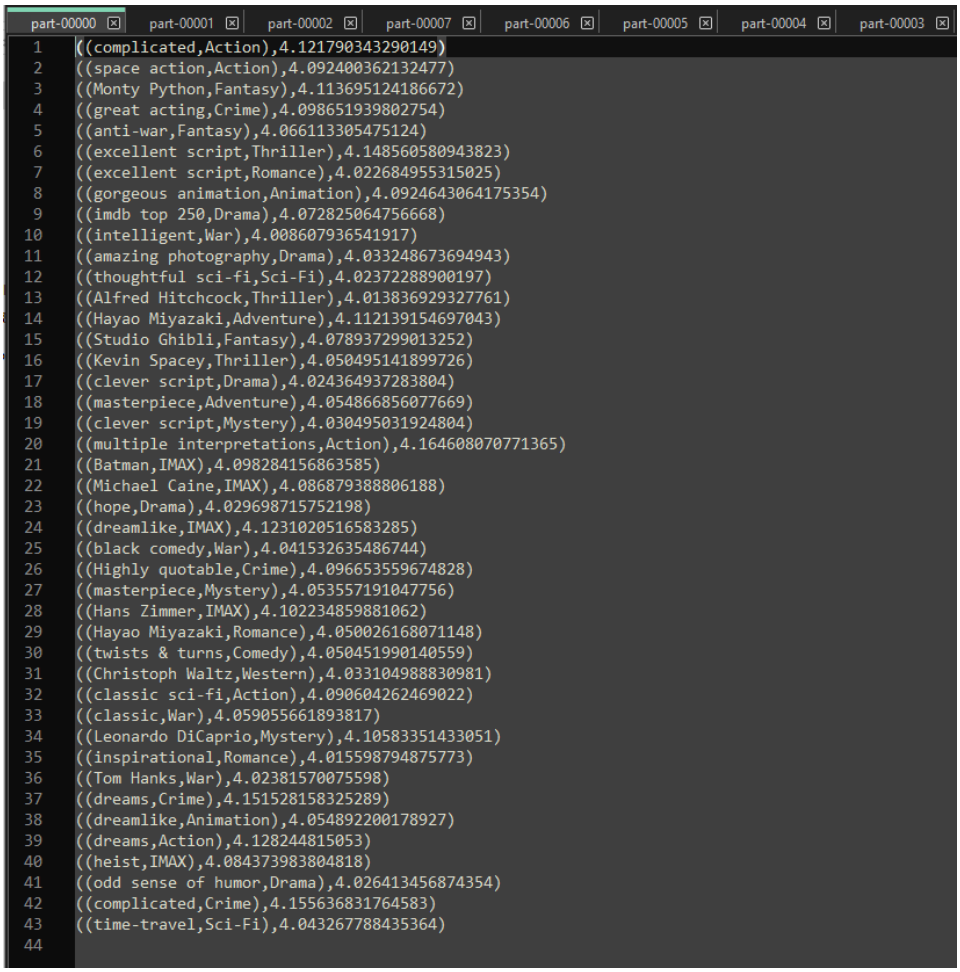


Figure 86: Sample output of Query 1

The output consists of `((Genre, Tag), AverageRating)` tuples. These results represent the average rating of movies where a specific tag dominates a particular genre. These results have strong analytical value for understanding tag sentiment or tag preferences across genres — e.g., "Sci-Fi" movies tagged with "IMAX" tend to receive higher ratings.

4.3.2 Query 2: Tag Dominance per Genre

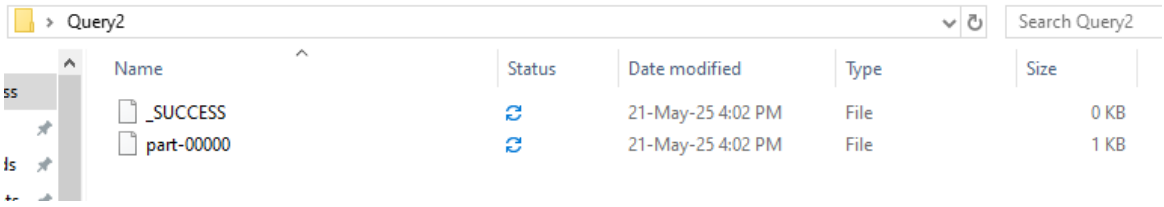


Figure 87: Output files for Query 2

As shown above, Query 2 produced only one part file due to the use of `coalesce(1)`. We used `coalesce(1)` here because we know in advance that the maximum amount of rows will be as much as the maximum number of movie genres there are. The small file size (1KB) suggests the result is compact and easy to analyze.

part-00000

```
((sci-fi,IMAX),3.6750153526890106)
((western,Western),3.719762429443877)
((sci-fi,Adventure),3.6814697476180633)
((twist ending,Mystery),3.9162217116898477)
((animation,Animation),3.6251035254349113)
((sci-fi,Sci-Fi),3.67506321825999)
((dark comedy,Crime),3.9298858663036986)
((animation,Children),3.5986174171078926)
((World War II,War),3.843665869224708)
((action,Action),3.573472621730471)
((comedy,Comedy),3.5043258945894706)
((film noir,Film-Noir),3.773405829050057)
((horror,Horror),3.370511177906561)
((documentary,Documentary),3.675197335653596)
((twist ending,Thriller),3.8919487514004296)
((atmospheric,Drama),3.870067528376439)
((musical,Musical),3.449903254554407)
((romance,Romance),3.624768743848848)
((fantasy,Fantasy),3.669370846789935)
```

Figure 88: Sample output of Query 2

This output shows the most dominant tag per genre, for example, the tag "IMAX" appears most frequently in Sci-Fi movies. This result has great value in applications like genre-based tag recommendation, where we want to suggest the most relevant tags that define user expectations within a genre.

4.3.3 Query 3: Iceberg

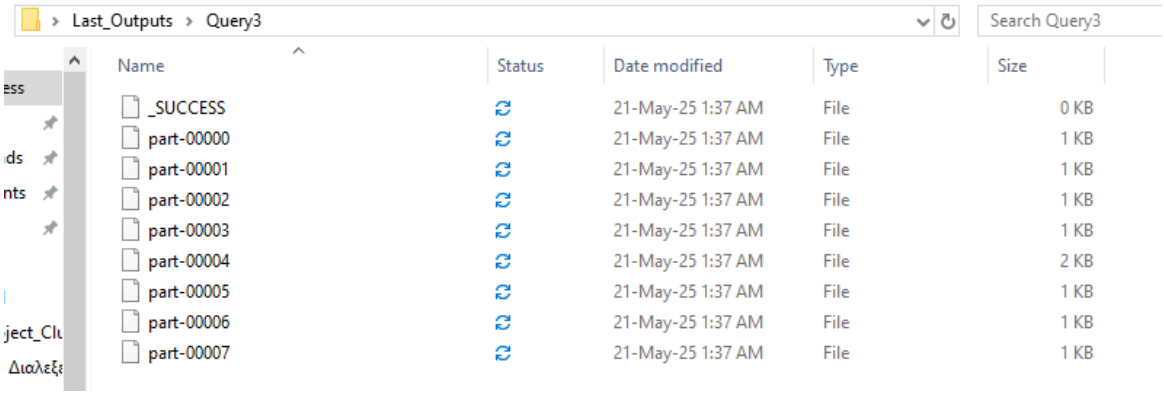


Figure 89: Output files for Query 3

Query 3 generated 8 output part files with each file containing several tag-relevance pairs. No repartitioning or coalescing was performed, allowing Spark to choose the number of output files based on default partitioning.

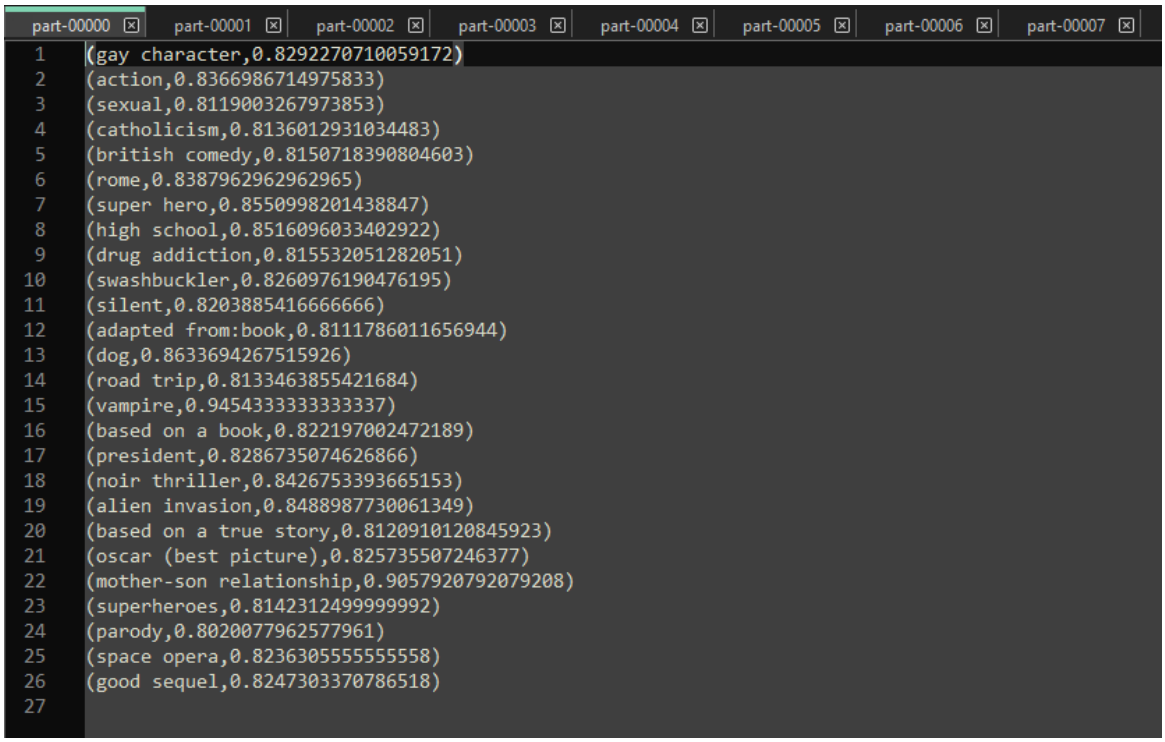


Figure 90: Sample output of Query 3

This result maps each tag to its average relevance score (e.g., "gay character", 0.83). It helps us identify the most semantically significant tags across the dataset, i.e., tags that are consistently

relevant across many movies. This can be useful in filtering or curating tag vocabularies for content-based filtering or search optimization.

4.3.4 Query 4: Sentiment Estimation

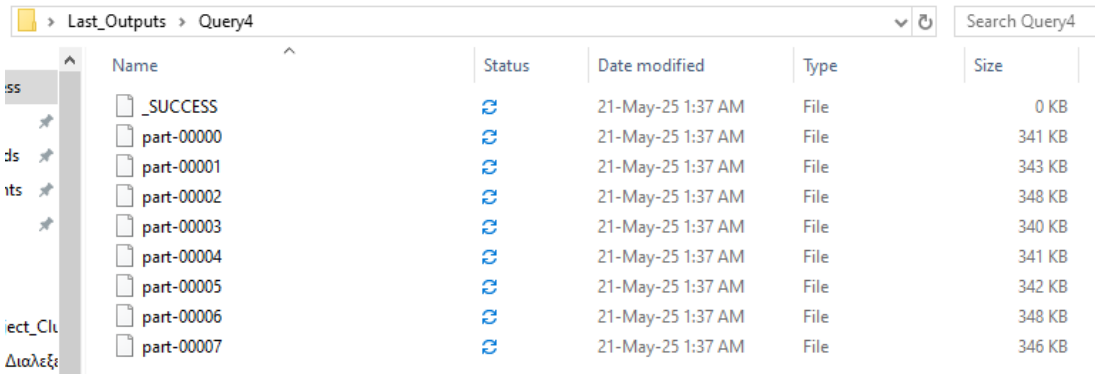


Figure 91: Output files for Query 4

Query 4 produced 8 output part files, each approximately 340KB in size. This indicates a high number of results, reflecting the wide variety of tags in the dataset and the number of ratings associated with them. Since the goal was to estimate sentiment by computing the average rating for each tag, the output density is expected.

part-00000 part-00001 part-00002 part-00003 part-00004 part-00005 part-00006 part-00007

```
(Josh Brolin,3.7263157894736842)
(Ed Decter,2.5)
(StereoVision,3.0)
(too short,4.069444444444445)
(specism,2.5)
(bombast,3.0)
(r:brief violent image,3.0)
(calling someone baby,1.5)
(Dynamics of Power,5.0)
(french suspense,3.9375)
(blue star,1.0)
(Boat,4.5)
(wasp sting,5.0)
(Ahney Her,3.8333333333333335)
(reference to edith piaf,5.0)
(picturesque village,5.0)
(slow plot,3.357142857142857)
(growling stomach,3.5)
(jade,2.5)
(crying,3.3847926267281108)
(gold tooth,3.1666666666666665)
(Dorothy Atkinson,4.0)
(breath,3.25)
(San Francisco,3.552287581699346)
(alive and kicking,2.5)
(dead police officer,3.75)
(reference to emile zola,1.0)
(for fans - best entertainment,5.0)
(U.S.S. Ticonderoga (CV-14),4.0)
(lamp post,4.0)
(energizing,4.381578947368421)
(#boring,2.0)
(worst superhero,3.5)
(former nun,5.0)
(Overshadowed by Darkest Hour,3.5)
(sad man,3.5)
(Adventurous,4.0)
(dead body in a car trunk,2.8181818181818183)
(teacher unions terrorizing children,2.25)
(young heroes,3.142857142857143)
(dance along,4.0)
(atrocious cinematography,0.5)
(san francisco california,2.7916666666666665)
(panties slip,4.0)
(motive for murder,4.0)
(Folk,4.5)
(machine pistol,2.0)
(sumer,4.0)
```

Figure 92: Sample output of Query 4

In this sample, we see tags like `"Josh Brolin"` and `"San Francisco"` paired with average rating scores. These scores represent how positively or negatively movies associated with these tags are generally rated. For example words like ”Adventurous” or ”energizing” is natural to be paired with high ratings whereas ”bad actor play” is obviously getting a bad score. This type of analysis can be used in sentiment-aware tagging systems or for understanding public reception of concepts/topics represented by tags.

4.3.5 Query 5: Multi-Iceberg Skyline

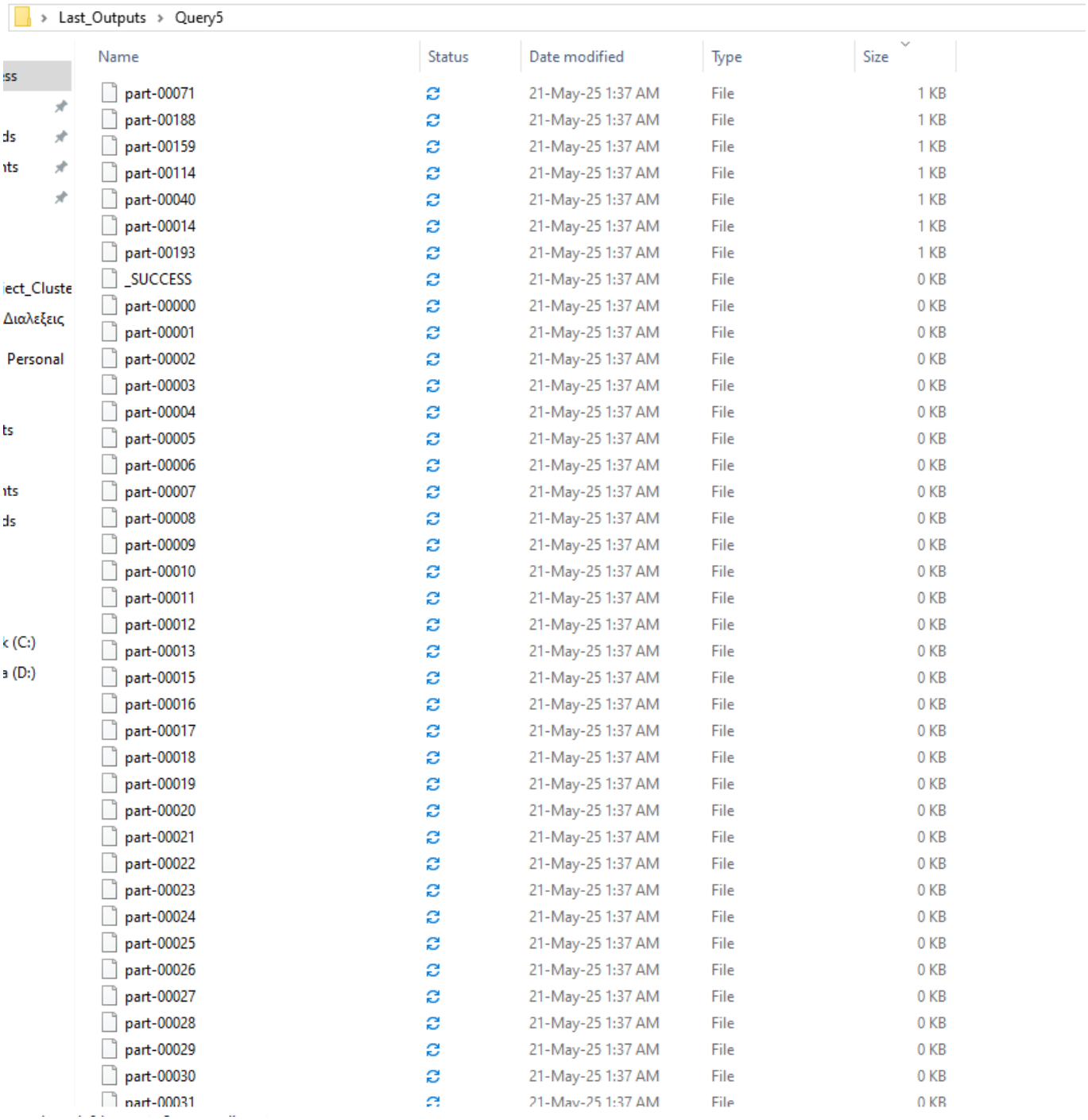


Figure 93: Output files for Query 5

Query 5 produced over 200 output part files due to the use of `repartition(200)` earlier in the computation to improve parallelism during heavy joins and aggregations. However, most of these output files are empty (0KB), and only a small number actually contain results (typically around 1KB). This is because the final output includes only a few qualified `((tag, genre), (avg_rating, count))` pairs that satisfy the Skyline restrictions — in our case fewer than 20 in total. The large

number of output files is a byproduct of the high degree of repartitioning done earlier for performance, without an explicit `coalesce()` before writing the result. This choice was intentional, as the actual number of qualifying rows could vary significantly depending on the data, and we opted to preserve parallelism flexibility.

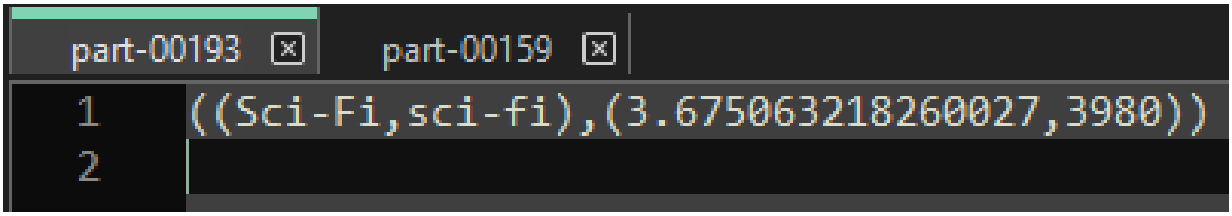
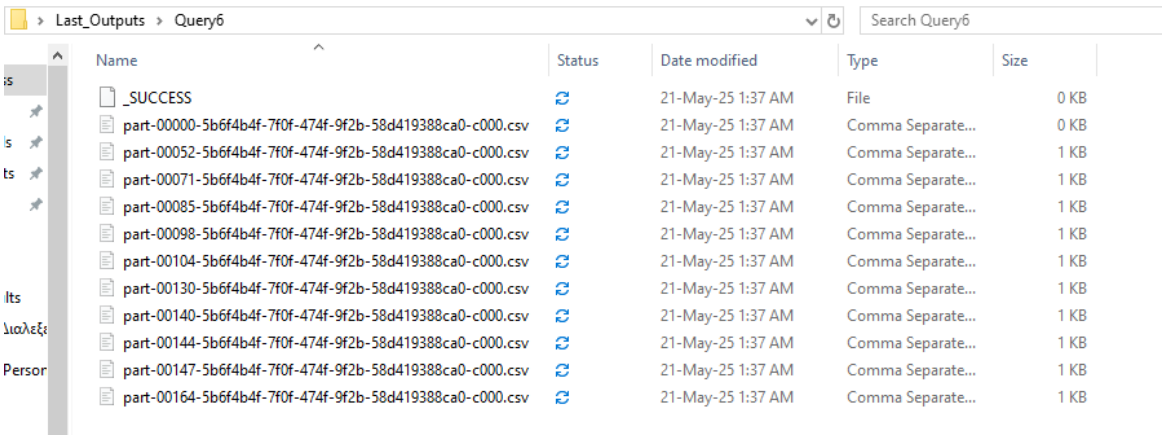


Figure 94: Sample output of Query 5

From the sample output, we see meaningful results like `((Drama, great acting), (3.959, 1599))` and `((Sci-Fi, sci-fi), (3.675, 3980))`. These represent high-traffic genre-tag combinations with strong average ratings, which are particularly valuable for content recommendation and analytics. These tuples suggest what combinations resonate most with users and can be used to guide both content creation and user engagement strategies.

4.4 Part B

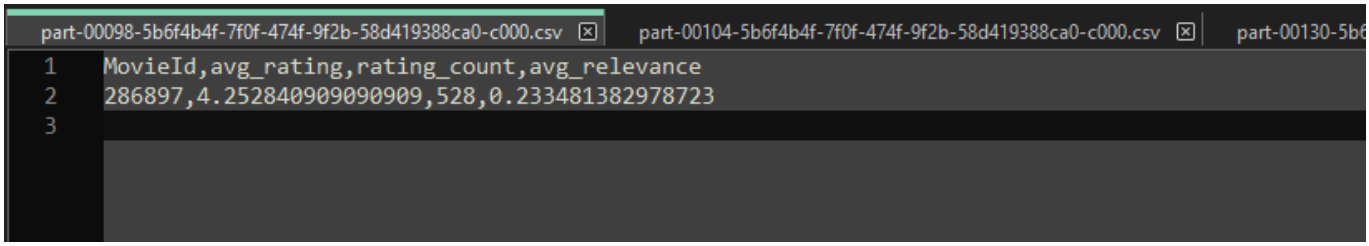
4.4.1 Query 6: Skyline Query



Query 6 produced a relatively small number of part files (all of them being around 0KB-1KB). This is expected, although the total number of movies is large, only a small fraction of them satisfy the skyline condition, i.e., they are not dominated in all three dimensions:

- `avg_rating`
- `rating_count`
- `avg_relevance`

Since the skyline is inherently sparse and filters out the majority of the data, most files are only holding 1 row of data (1 KB), and even 1 of the files contain zero data(0 KB).

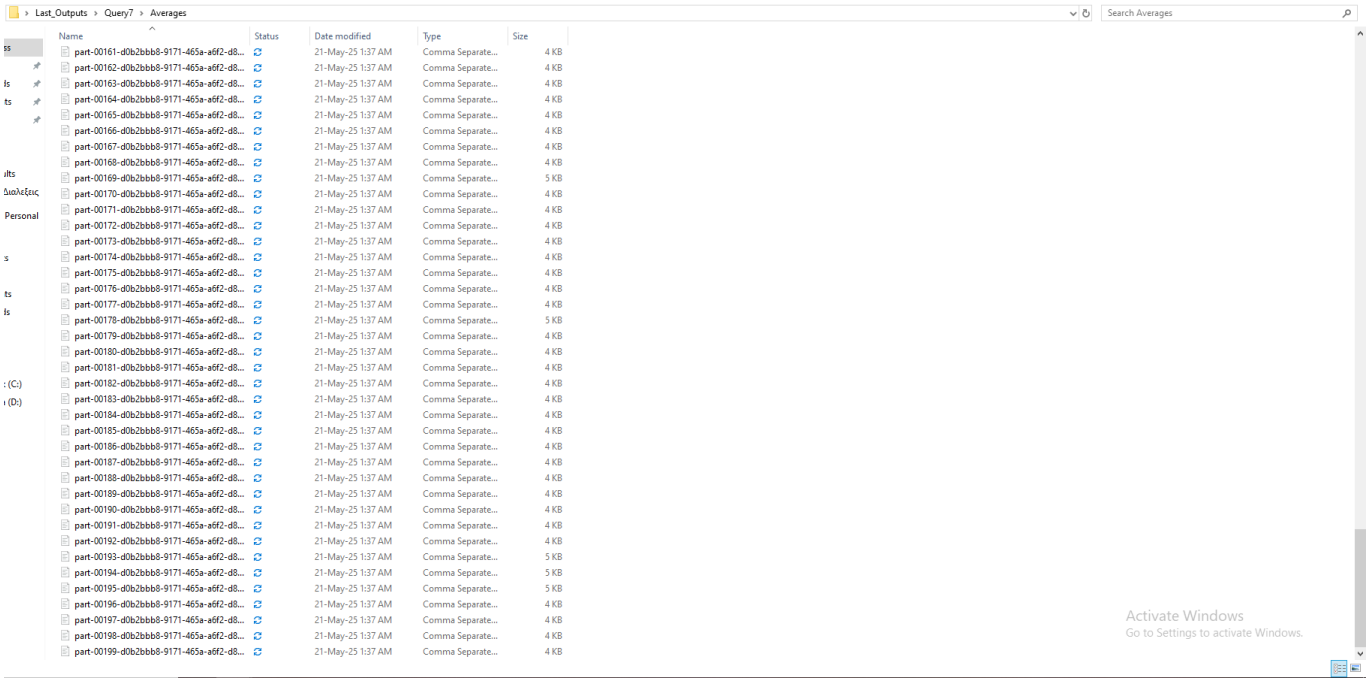


Each row here corresponds to a non-dominated movie, one that is not worse in all three criteria compared to any other movie. For instance, movie `286897`(Spider-Man: Across the Spider-Verse (2023)) has:

- An average rating of over `4.25`
- A high rating count (`528`)
- A decent average tag relevance score of `0.23`

Skyline queries like this are extremely valuable for multi-criteria filtering and personalized recommendations. Instead of looking at only the top-rated or most-rated movies in isolation, the skyline gives us a list of well-rounded choices, films that stand out on more than one metric without being outperformed across the board.

4.4.2 Query 7: Correlation Between Tag Relevance and Average Ratings

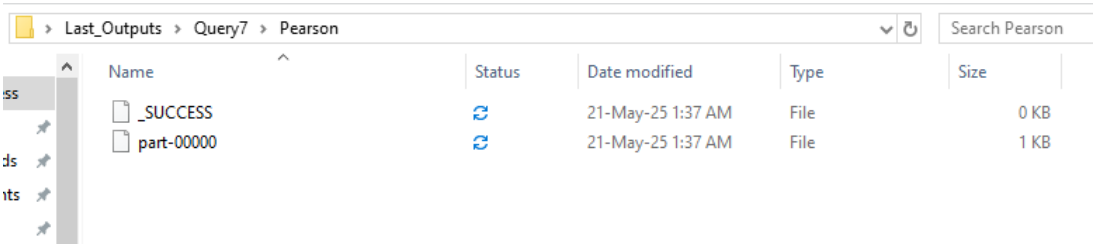


As expected, this query generated many small files containing average relevance and rating per movie. These results stem from grouping by `MovieId`, which is a fairly wide key space.

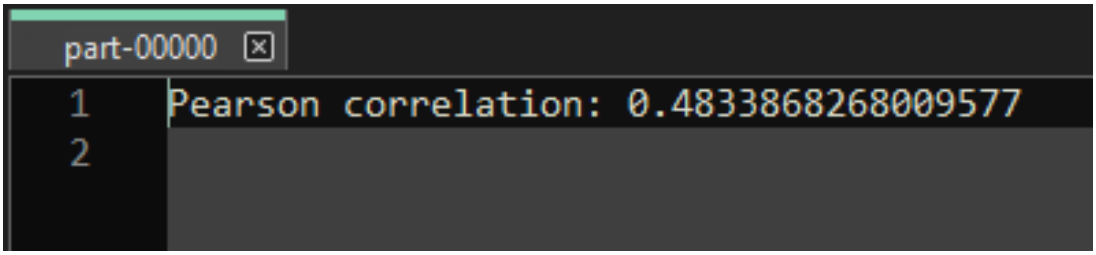
part-00000-d0b2bbb8-9171-465a-a6f2-d8a788bab520-c000.csv part-00001-d0b2bbb8-9171-465a-a6f2-d8a788bab520-c000.csv

```
MovieId,avg_rating_per_movie,avg_tag_relevance_per_movie
102684,3.125779625779626,0.10732579787234028
103050,3.349056603773585,0.10905961879432625
106022,3.352861035422343,0.10882136524822683
108791,3.588235294117647,0.11050997340425543
1090,3.902013422818792,0.20033133865248245
112911,2.789772727272727,0.11027703900709201
115713,3.9907429107101007,0.21088120567375898
115770,3.688976377952756,0.09716179078014182
119655,2.7680776014109347,0.11158421985815613
120478,4.016566265060241,0.12304765070921987
121370,2.492857142857143,0.08009064716312055
134851,2.9936708860759493,0.15466799645390095
141418,3.0833333333333335,0.10406249999999997
141950,3.170886075949367,0.05810150709219858
1436,2.986666666666667,0.08197384751773056
144656,3.3545454545454545,0.06432291666666656
144982,2.584070796460177,0.07082424645390066
147244,3.5483870967741935,0.08095257092198578
1572,3.784491440080564,0.11638031914893614
158813,3.0262968299711814,0.09549556737588658
164905,3.1574074074074074,0.1342832446808509
173535,4.124,0.15327681737588644
173627,3.2413793103448274,0.12696830673758872
2069,3.8074324324324325,0.10718107269503555
2088,2.5695630336514315,0.08993195921985826
2136,2.8188819167142043,0.10628523936170202
2162,2.4949879711307137,0.0817189716312058
2294,3.210627400768246,0.10972761524822697
26005,3.6812080536912752,0.13593262411347504
26082,4.1918876755070205,0.1676981382978726
27317,3.615297321833863,0.14948204787234037
27884,3.6468253968253967,0.09415203900709218
2904,3.571875,0.09146697695035445
296,4.191777924896098,0.21452548758865253
3210,3.636603524644237,0.15813320035460976
3414,3.2611940298507465,0.09124179964538999
3606,3.848498635122839,0.1241655585106382
3959,3.682651016318351,0.15425531914893642
39659,3.5695364238410594,0.08332557624113465
4032,2.977777777777778,0.09654033687943256
467,3.41921664626683,0.0819299645390071
47092,3.2936507936507935,0.0658663563829787
47940,3.2916666666666665,0.10269126773049636
4821,3.1841609050911375,0.10126484929078021
48738,3.8394851539090165,0.17960017730496455
4937,2.8493150684931505,0.07905385638297861
50802,2.9324324324324325,0.08017841312056743
51063,3.340909090909091,0.06854233156028376
5325,3.7377706495589416,0.12885970744680847
55498,2.9434782608695653,0.0730336879432623
```

Each row includes a movie's average user rating and average genome tag relevance. This structure allows us to perform statistical correlation.

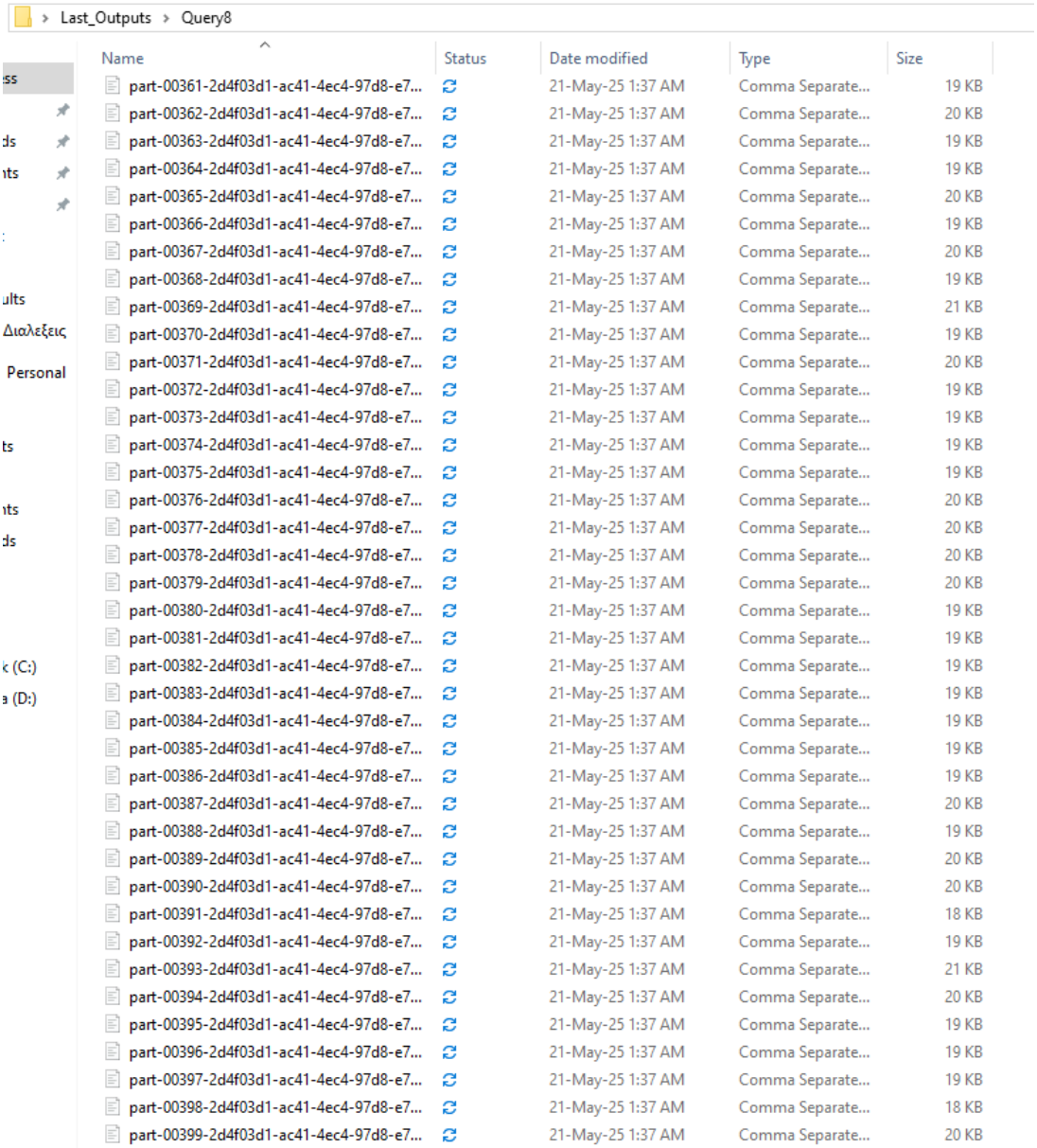


Because the correlation value is always just a single scalar result, we used `coalesce(1)` before saving. As a result, only one small file is generated.



The computed Pearson correlation is approximately `0.483`, suggesting a moderate positive relationship between semantic tag relevance and user ratings. This confirms that while tags are meaningful, other subjective factors also influence movie ratings.

4.4.3 Query 8: Reverse Nearest Neighbor



Due to the complexity of cosine similarity computations and the need to handle millions of user-tag relevance comparisons, we used multiple `repartition()` calls throughout the pipeline. This is

evident in the large number of sizable part files. Our number of part files are exactly 400 which is obvious because we use that exact number as an argument in our `repartition()` calls.

part-00399-2d4f03d1-ac41-4ec4-97d8-e7d76ec38554-c000.csv

```
UserId,cosine_similarity
174379,0.856459881418073
306357,0.8733407666069585
247350,0.8596482945101969
14515,0.9010897449429016
92950,0.8823468906941964
137853,0.8732037714024742
179490,0.8945380440781465
182151,0.8496938048780402
212708,0.8928554984106551
236333,0.9088591139880866
33403,0.8738202357441255
252206,0.854189453287563
282840,0.876975892891499
43694,0.9133137353911379
89996,0.8978091302758816
98120,0.8838376906227493
225486,0.8098219068442926
212566,0.8717706409148334
62063,0.8306035809020057
138312,0.8671200015634505
194465,0.841436868985883
77475,0.8579453883961777
151452,0.8843200619313679
130087,0.8395217010232012
172696,0.8844802892222461
189470,0.8766120759726745
311595,0.8918587782885776
233466,0.8878427523152104
16689,0.9350320016904583
250588,0.9179957673319935
261001,0.9202360590594869
278951,0.8716797314446789
131141,0.8988928541238583
329358,0.87153335256164
110684,0.8553311631494287
2854,0.8753176165465302
51778,0.835745086099745
74301,0.8435654643274162
309326,0.7992323614552518
313948,0.7987498267435407
316372,0.8731673074770526
325656,0.8798816394775684
230889,0.8486202306326611
240679,0.7505761833809744
4379,0.8520609046477197
18777,0.8946057478391175
252852,0.7961963615452852
253760,0.8504086015046588
256817,0.6218011799482303
150552,0.8236608963041533
```

The final output is a list of users along with their cosine similarity to the selected movie (Inception). This allows us to identify the top users whose preferences align most closely with the movie's tag profile, enabling personalized marketing or targeted recommendations.

4.4.4 Query 9: Tag-Relevance Anomaly

This query aimed to find cases where a movie is semantically linked to popular tags like `action`, `classic`, or `thriller`, with a high tag relevance ($>= 0.8$), but ends up being poorly rated by users (average rating < 2.5).

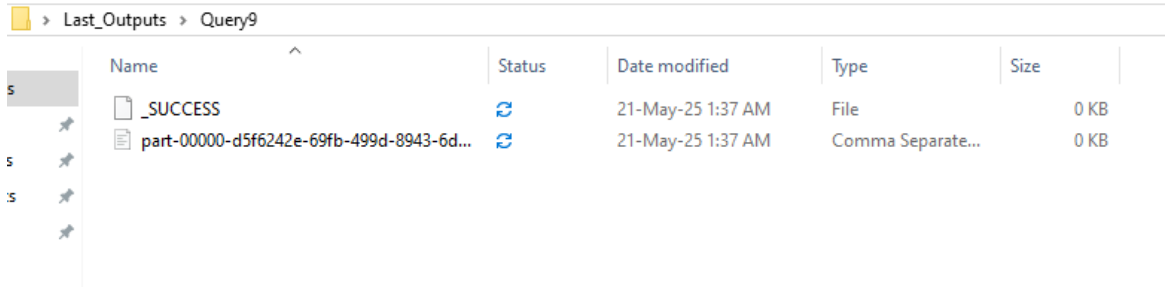


Figure 95: Output files for Query 9

As we can see, Query 9 produced just a single output file. However, this file is completely empty, meaning that no movies satisfied both conditions at once. While this might seem like we did something wrong at first, it's actually a correct and valuable result.

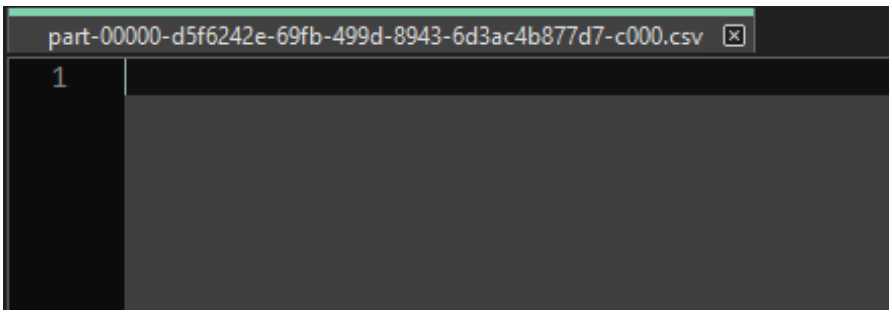


Figure 96: Sample output of Query 9 (empty)

This outcome suggests two things:

- Either no movies in our dataset were both over-hyped and low rated..
- Or most likely, the thresholds to determine what it means for a movie to be over-hyped were too strict.

4.4.5 Query 10: Reverse Top-K Neighborhood Users

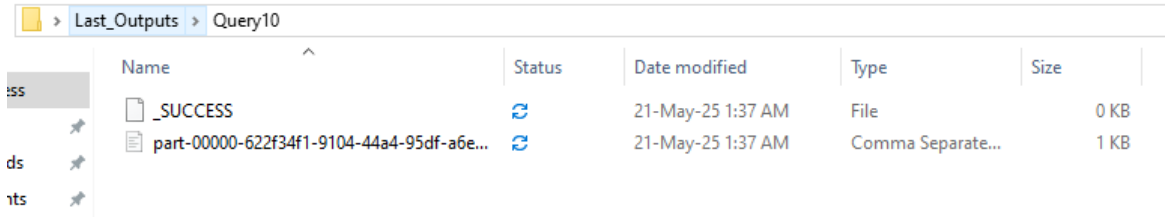


Figure 97: Output files for Query 10

As shown in the figure above, Query 10 produced a single output file. This was expected, as we used `coalesce(1)` before writing to disk, since the result contains only the top-10 users ranked by cosine similarity.

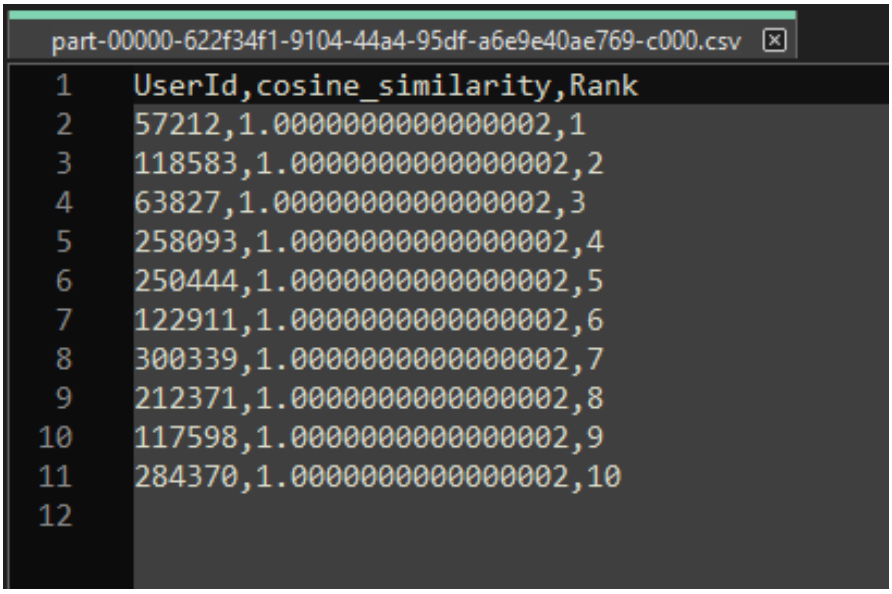


Figure 98: Sample output of Query 10

The output contains tuples in the form `(UserId, cosine_similarity, Rank)`. These are the top 10 users whose tag preferences most closely match the tag profile of the target movie (Inception). Interestingly, all cosine similarities are very close to 1.0, indicating a strong alignment between these users' preferences and the selected movie's semantics. From a recommendation-analytic perspective, this is extremely useful, these users are ideal candidates for recommending similar movies, and for more user personalization.

5 References

- Spark Documentation, RDDs programming guide: `https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds`
- Spark Documentation, DataFrames programming guide: `https://spark.apache.org/docs/latest/sql-programming-guide.html`
- `https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.repartition.html`
- `https://spark.apache.org/docs/latest/api/python/reference/pyspark.pandas/api/pyspark.pandas.DataFrame.spark.persist.html`
- Professor's Lectures