ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Functional Programming, Analytics and Applications Programming Assignment

June 21, 2024

1 Reuters Analytics Application

1.1 Brief Description

Problem Setup:
In this scenario we were provided with a dataset from the Reuters News Agency. It consisted of 3 categories of files:

1. **lyrl** files, which contained the terms (IDs) inside each document (ID) and a weight which we ignored for our analysis
2. **qrels** file, which contains the category each document has been assigned to and an enumeration that has also been ignored
3. **stem** file which contains the stem/token each term ID describes and an idf parameter which we ignore.

The goal of our analysis, was to calculate the Jaccard index for each term-category pair as described in the project requirements, and produce an output file that would contain triplets of the following format :

$$< categoryname >; < stemid >; < JaccardIndex >$$

For this task we are going to use RDDs, since our data don't have a schema.

1.2 Code Analysis

```
val docsPerCategoryCardinality = spark.sparkContext.textFile(pathCategoriesPerDocument)
  .map(line => {
    val fields = line.split("\\s+") // Split whenever one or more whitespaces are found
    (fields(1), fields(0)) // (docID, categoryID)
  })

val docsCountsPerCategory = docsPerCategoryCardinality
  .map(x => (x._2, 1)) // Create a tuple with the categoryID and 1
  .reduceByKey(_ + _) // Count the number of documents per category
```

At the beginning, we processed the qrels file to extract the documentIDs assigned to each categoryID. We created an RDD from the specified path, where each element represented a line from the file. We used the map() HOF to apply the transformation described by the inner code block and convert each line into a tuple. Our lambda function first split each line into fields using whitespace as the delimiter. It then created a tuple where the first element was the categoryID (assumed to be at position 1) and the second element was the docID (assumed to be at position 0). To count the number of documents for each category, we used the map() higher-order function again to pair each categoryID with the count of 1. Finally, we aggregated these counts by summing the values for each key using the reduceByKey() HOF.

```
val termsInDoc =  spark.sparkContext.textFile(pathTermsInDocuments)
    .flatMap(line => {
      val line1 = line.split("\\s+", 2) //Split the line into document ID and term IDs
      val docID = line1(0) //This is our doc ID
      val termIDs = line1(1).split("\\s+").map(_.split(":")(0)) //Extract term IDs, and ignore
          the weights that are not needed for our analysis
      termIDs.map(termID => (docID,termID)) //We return a tuple with the docID and the termID
    })

  val docsCountsInTerms = termsInDoc
    .map(x=>(x._2,1)) //Create a tuple with the termID and 1
    .reduceByKey(_ + _) //So that we can count the number of documents per term as we did in
        Class
```

Next, we processed the **lyrl** files so that we could extract the termIDs contained in each documentID. We first used the flatMap() HOF to apply the transformation described by the inner code block we created and flatten the result. What our lambda function does, is first split each line to two parts with our delimiter being a space (or more, hence the 's+'). Then it assigns the first part docID (which is obviously the document of interest) and it splits the second part which is our termIDs whenever a whitespace is found, with the use of map(_.split(":") it "throws away" the weight parameter after the colon that doesn't interest us for each term and keeps part 0 which is our termID. So essentially "termIDs" is an array of termIDs related to our document. Finally, using map() we correlate each termID with its respective docID with the use of our lambda function 'termIDs.map($termID => (docID, termID)$)' to produce tuples. The flattened result of the code block, and therefore "termsInDoc", is an RDD of tuples (docID,termID).
We then used this RDD to create another RDD "docsCountsInTerms" which will contain the total number of documents related to each term ($|DOC(T)|$). We achieved that by using map() on our previous RDD, where with the use of our lambda function '$x => (x._2, 1)$' we transformed each tuple to a new tuple with the form (termID,1). Afterwards we applied the HOF reduceByKey to our previous result, to get the desired cardinalities for each termID.
The above methodology we used is very similar to the examples we saw during the course.

```
1 val intersectionCardinality = termsInDoc
2     .join(docsPerCategoryCardinality) //Join the terms with the categories, with our key being
          the docID
3     .map { case (docID,(termID,categoryID)) => ((termID, categoryID),1) } //Create tuples with
          the termID and categoryID and 1
4     .reduceByKey(_ + _) //So that we can count the number of documents per unique combination
          of term-category as we did in Class
```

At this point, we calculated the intersection cardinality between terms and categories by counting the number of documents where each term appears within each category.At line 2 we used the join() HOF to associate each term with its corresponding category by matching their common docID. After the join, each element of the RDD consists of a docID paired with a tuple containing a termID and a categoryID.At line 3, we prepared the data for counting the number of documents per unique combination of term and category using the map() HOF. Each element of the joined RDD was transformed into a new tuple ((termID,categoryID),1), where the key represented a combination of term ID and category ID, and the value was set to 1. Finally, at line 4 we aggregated the counts for each unique combination of term ID and category ID by summing the values for each key using the reduceByKey() HOF.

```
1     val termToStem = spark.sparkContext.textFile(pathTermsToStems) //Reading the stem file
2     .map(line => {
3       val fields = line.split("\\s+") //Split depending on whitespaces
4       (fields(1), fields(0)) //termID,stem
5     })
6
7   //termToStem.foreach(println)
```

We then processed the stem file which was needed in order to find the relation between the terms contained in our documents and the respective stem they derive from. We used the map() HOF to apply the transformation described by the inner lambda function. What this lambda does is take each line (record) of the stem file, split it whenever a whitespace character (or more) is found, and then produce tuples where the first element is the termID and the second is the stem it is related to. The result is the RDD "termToStem" which contains all the (termID, stem) tuples.

```
1     val jaccardIndex = intersectionCardinality.map { case ((term, categoryID),
          intersectionCount) =>
2     (term, (categoryID, intersectionCount)) //Transform our intersection so that we can join it
          with the other RDDs, using term as the key
3   }.join(docsCountsInTerms).map { case (term, ((category, intersectionCount), termDocCount)) =>
4     (category, (term, intersectionCount, termDocCount)) //Transform our RDD so that we can join
          it with the other RDDs, using category as the key
5   }.join(docsCountsPerCategory).map { case (category, ((term, intersectionCount, termDocCount),
        categoryDocCount)) =>
```

```
6       val unionCardinality = termDocCount + categoryDocCount - intersectionCount //The union
            cardinality formula
7       val jaccardIdx = Option(unionCardinality).filter(_ != 0).map(uc => intersectionCount.
            toDouble / uc).getOrElse(0.0)
8       (category, term, jaccardIdx) //Return the category, term and Jaccard Index
9     }
```

At this point we have everything we need for the calculation of the Jaccard Index.
We first created the RDD ”jaccardIndex”, where using the map() HOF on the ”intersectionCardinality” RDD we transformed the initial tuples ((term, categoryID), intersectionCount) to tuples with the form (term, (categoryID, intersectionCount)).
This is essential in order to join with the number documents that contain a specific term, using the termID as a key. We did this by joining our previous RDD with ”docsCountsInTerms”, and then using the map() HOF we transformed the resulting tuples of our join with the form (term, ((category, intersectionCount), termDocCount)) to tuples with the form (category, (term, intersectionCount, termDocCount)).
We then joined the result with the RDD ”docsCountsPerCategory”, and performed a ”transformation” for each tuple (category, ((term, intersectionCount, termDocCount), categoryDocCount)), which actually is a series of calculations described in the codeblock. We first calculated ”unionCardinality” with the formula indicated by the instructions, and then we calculated the actual Jaccard index for each tuple containing a category-term combination. We created an option for ”unionCardinality” for cases where the value would be absent, then filtered to get only non-zero values and applied the Jaccard Index calculation as instructed using the map() HOF for each ”unionCardinality”. We also used getOrElse() in the end in order that combinations of category-term that have absolutely no relation, get assigned a ’0’ value as their Jaccard Index. Finally, we returned tuples of the form (category,term, jaccardIdx) which essentially are the content of our RDD jaccardIndex.

```
1   val jaccardIndexFinal = jaccardIndex /*Here we join the jaccard indices for each term-
        category pair,
2                                        with the respective stem so that we can later output
                                             the results with the format <category name>; <
                                             stemid>; <jaccard index>*/
3     .map { case (categoryID,termID, jaccardIndex) => (termID, (categoryID, jaccardIndex)) }
4     .join(termToStem)
5     .map { case (termID, ((categoryID, jaccardIndex), stem)) => (categoryID, stem, jaccardIndex
          ) }
```

For our last step we created the RDD ”jaccardIndexFinal”, in which our goal was to join the previous tuples with the stems provided, and thus create the required triplets which are later going to be output to the file. Using a similar methodology to above, we used map on the tuples of ”jaccardIndex” to transform them from (categoryID , termID , jaccardIndex) to (termID , (categoryID , jaccardIndex)), and then joined with the contents of ”termToStem” using termID as a key. Finally we transform the resulting tuples (termID , ((categoryID , jaccardIndex) , stem)) to (categoryID , stem , jaccardIndex) using the map HOF, which are also the content of RDD ”jaccardIndexFinal”.

```
1   jaccardIndexFinal.map { case (category, stem, jaccardIdx) =>
2     s"$category;$stem;$jaccardIdx"
3   }.saveAsTextFile(outputPath) //Save the results in the output path
```

This bit of code, outputs the tuples one-by-one to the file indicated by our output path.

1.3 Execution Analysis

Everything displayed below is the result of a local execution. The CPU had 12 threads, which as we are going to see is also the maximum parallelization achieved.
After executing our produced JAR file, we observed the following on the Apache Spark History Server:
JOBS

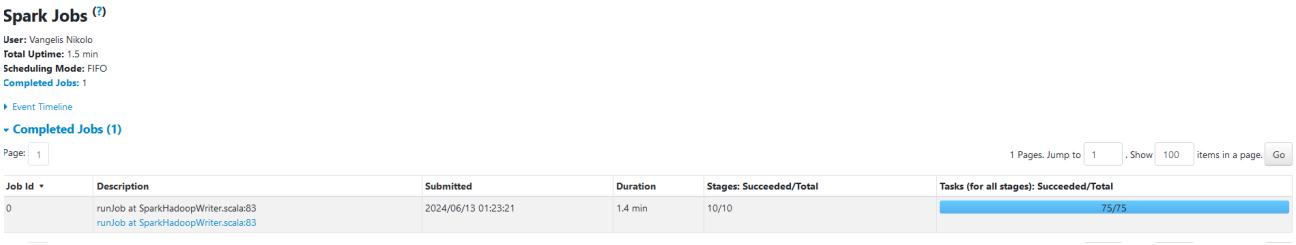


Figure 1: All Physical Jobs as they appear on the History Server

We observe a single Spark Job, which is justified by the fact that in our code we have only included a single action, of outputting the results to a text file with the operation ”saveAsTextFile”.

STAGES

The rest of our code consists of transformations, which Spark is going to execute as a chain inside of this single job. The transformations that compose this job, are essentially our stages, as displayed below:

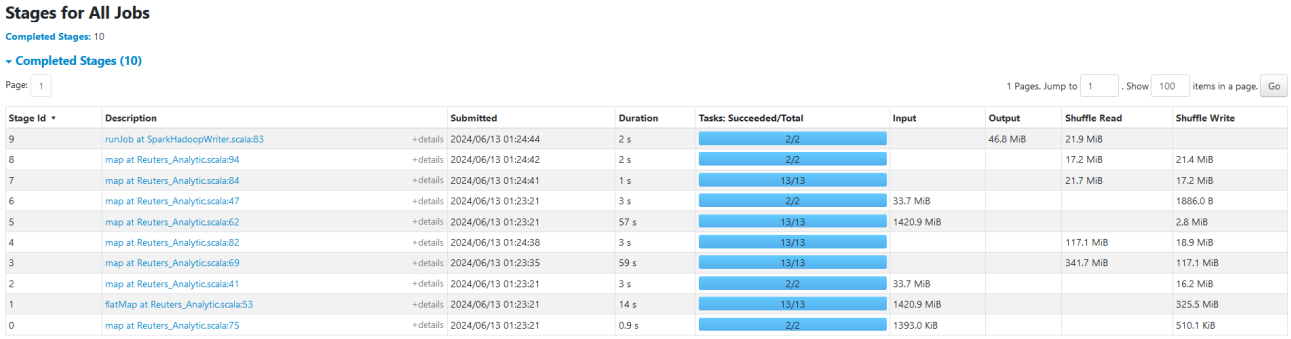


Figure 2: All Stages as they appear on the History Server

Let’s begin by analyzing each stage (transformation) executed as instructed by our Scala code.

STAGE 0

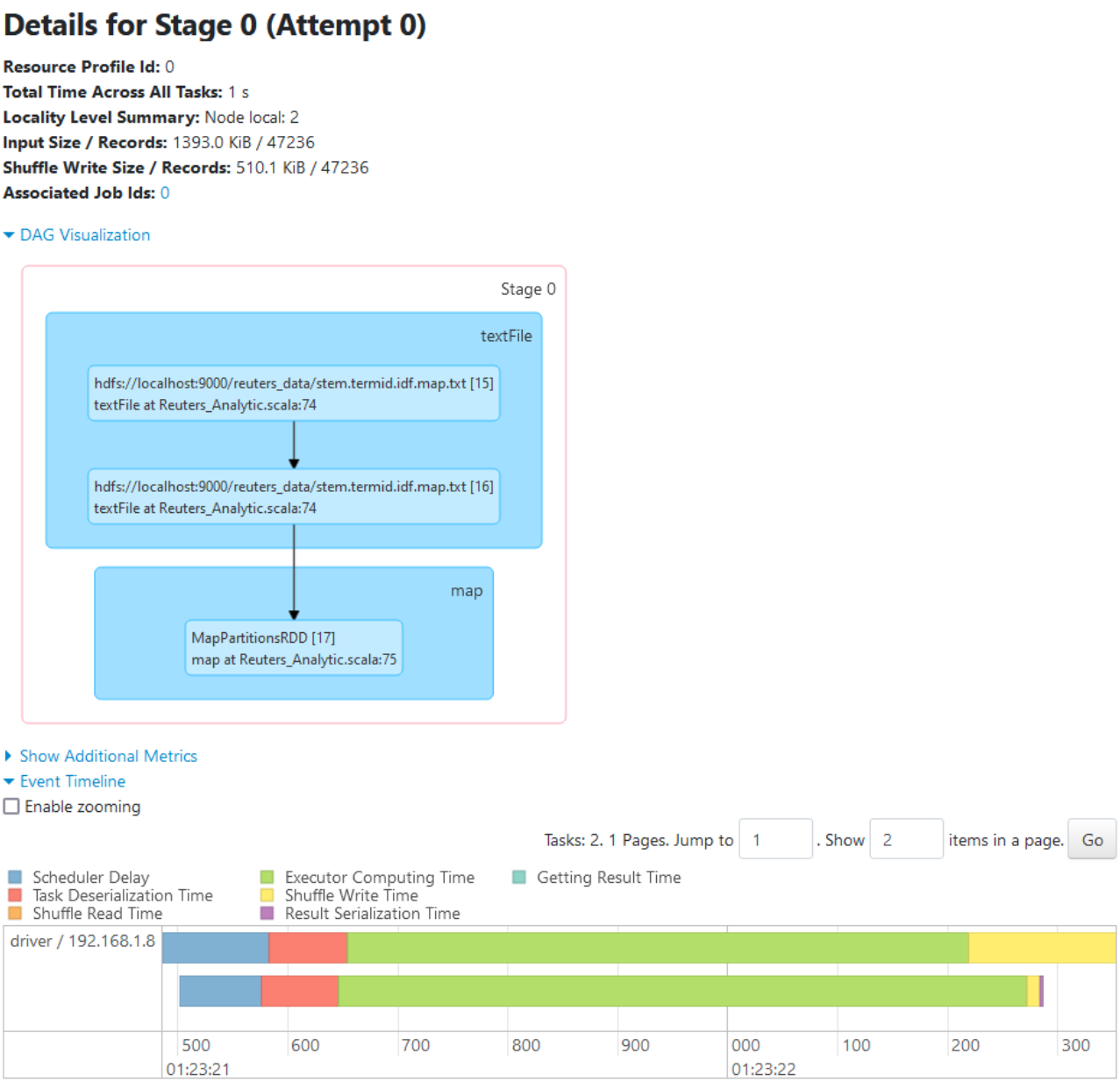


Figure 3: DAG Visualization and Event Timeline of Stage 0

The first stage is the processing of our stem file using the map transformation (line 75 in our code). What we observe here is the parallelization order of 2. This order is achieved because our initial file consists of 2 partitions. Also, map is a narrow transformation that does not require shuffling, therefore maintaining the same number of partitions as the input RDD.

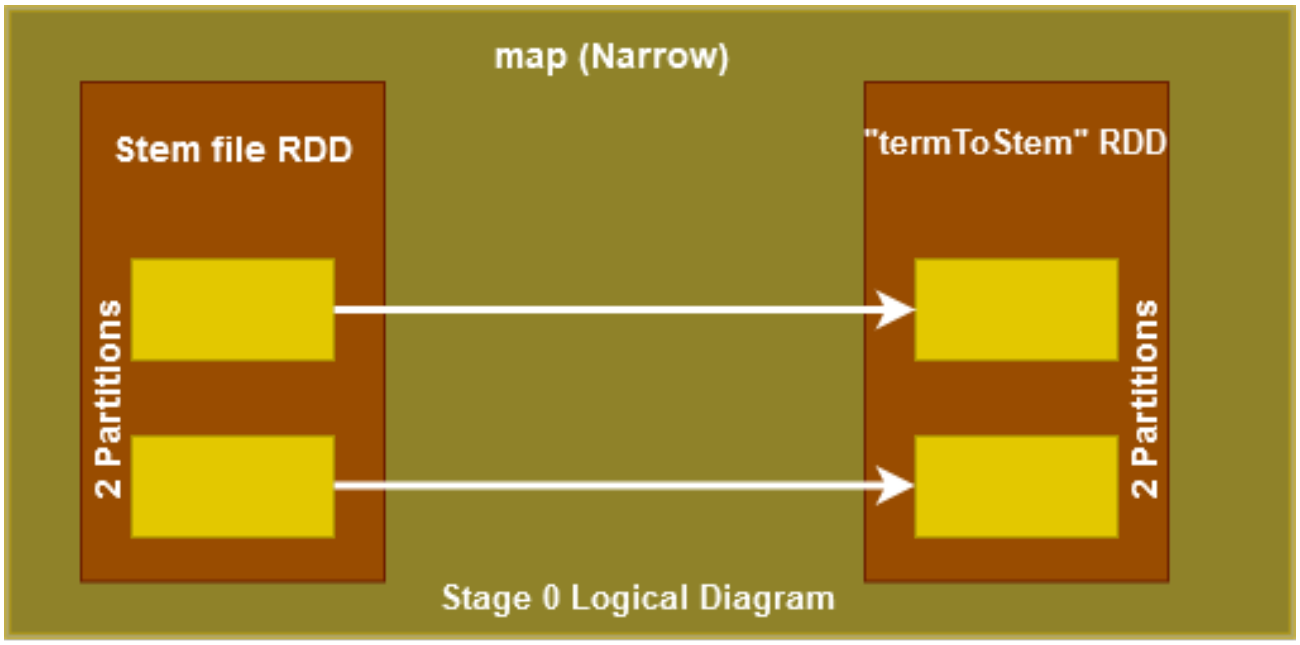


Figure 4: Logical Diagram for Stage 0

STAGE 1

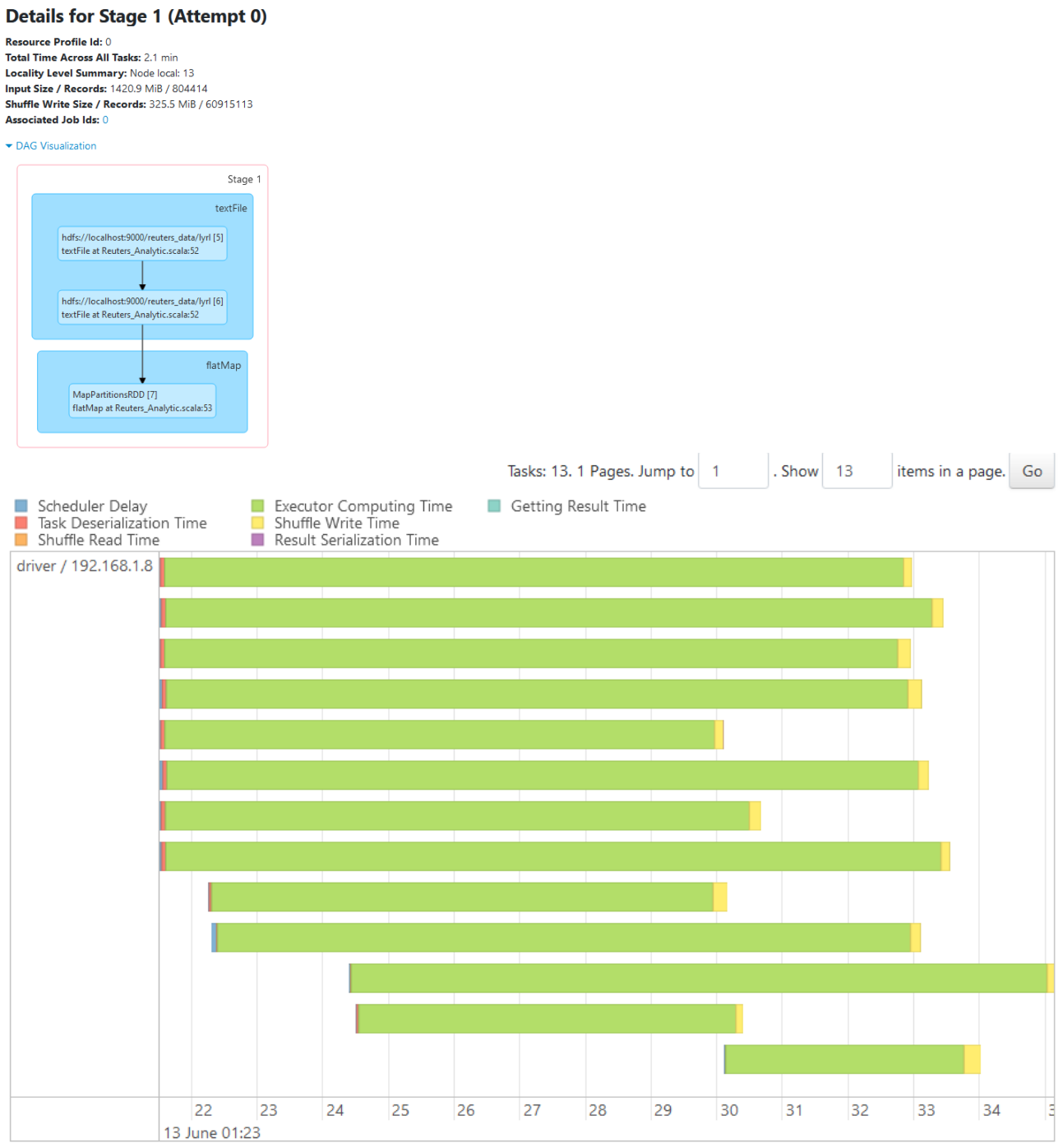
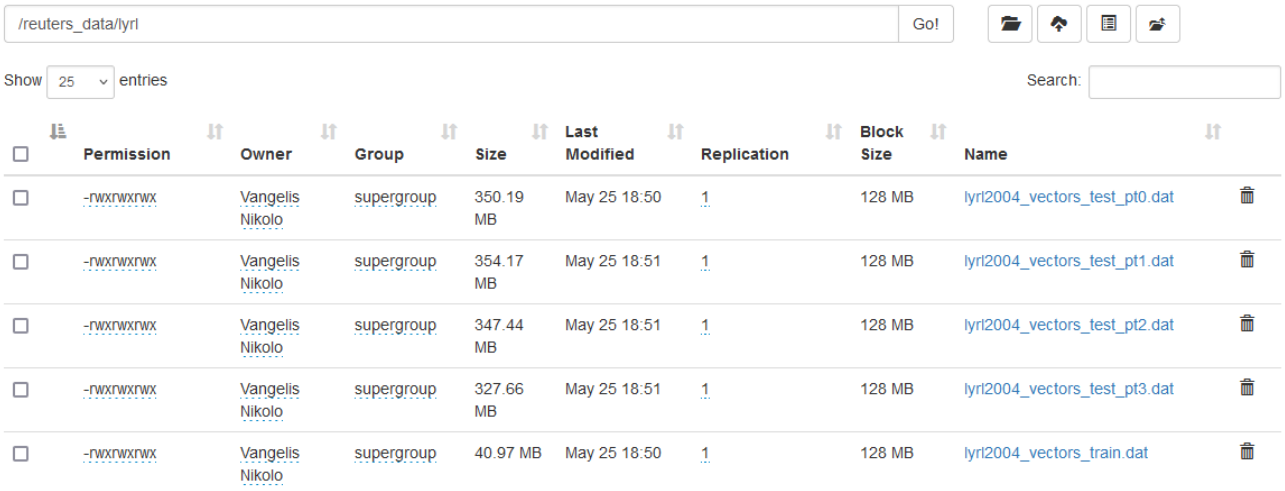


Figure 5: DAG Visualization and Event Timeline of Stage 1

Stage 1 is about processing our lyrl files using the flatmap transformation (line 53 in our code). In our HDFS, the entirety of our lyrl files was distributed amongst a total of 13 partitions, as can be concluded by the following screenshot of our filesystem.



/reuters_data/lyrl Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rwxrwxrwx	Vangelis Nikolo	supergroup	350.19 MB	May 25 18:50	1	128 MB	lyrl2004_vectors_test_pt0.dat
-rwxrwxrwx	Vangelis Nikolo	supergroup	354.17 MB	May 25 18:51	1	128 MB	lyrl2004_vectors_test_pt1.dat
-rwxrwxrwx	Vangelis Nikolo	supergroup	347.44 MB	May 25 18:51	1	128 MB	lyrl2004_vectors_test_pt2.dat
-rwxrwxrwx	Vangelis Nikolo	supergroup	327.66 MB	May 25 18:51	1	128 MB	lyrl2004_vectors_test_pt3.dat
-rwxrwxrwx	Vangelis Nikolo	supergroup	40.97 MB	May 25 18:50	1	128 MB	lyrl2004_vectors_train.dat

Figure 6: lyrl files in HDFS

From the event timeline, we observe that we achieve a parallelization order of 12 (maximum considering our setup), while the 13th task (actually task with id 12) was executed after an executor finished its previous task and was once again available. This parallelization is achieved because of the number of partitions previously described, and since flatmap is a narrow tranformation we are maintaining the same number of partitions as the input RDD.

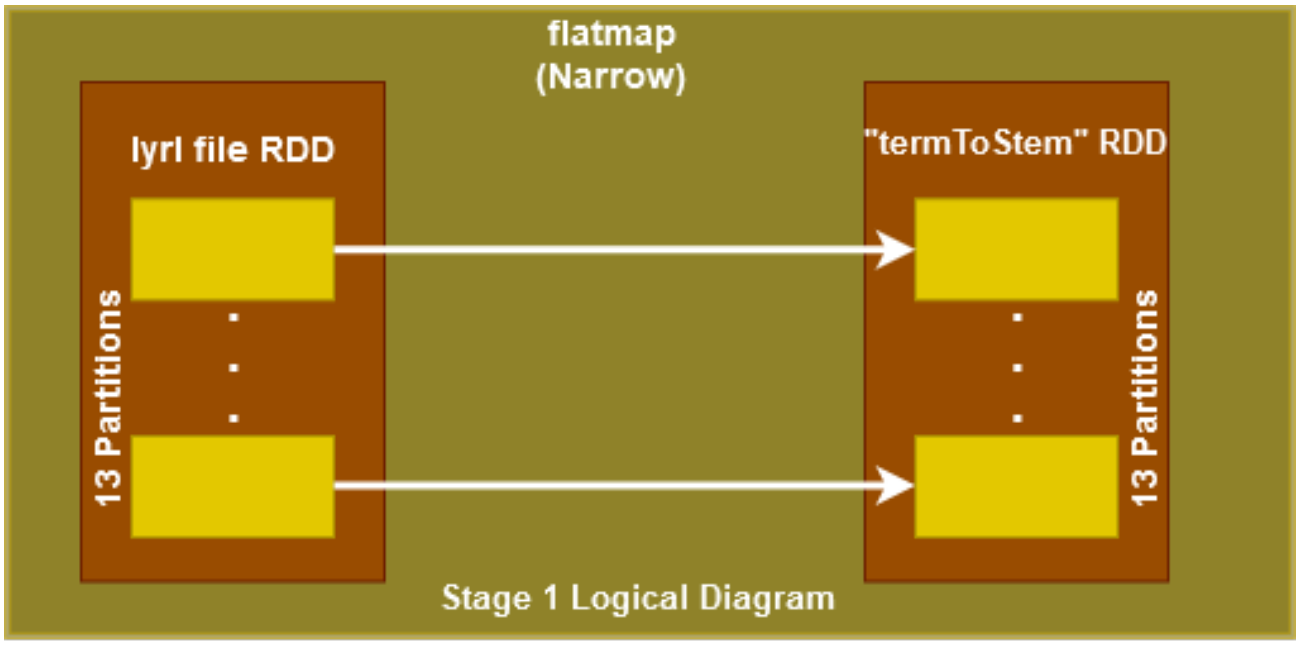


Figure 7: Logical Diagram for Stage 1

STAGE 2

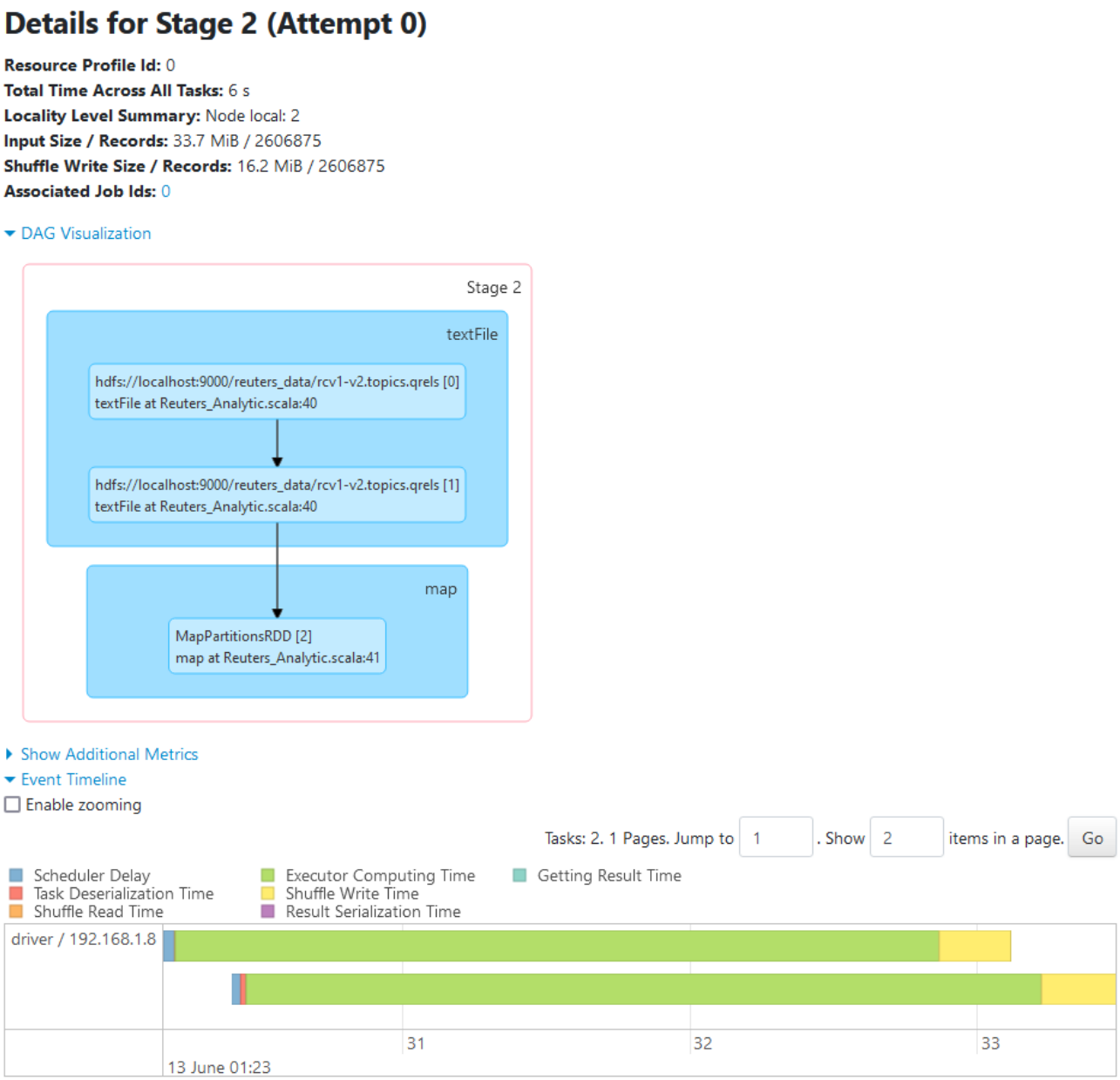


Figure 8: DAG Visualization and Event Timeline of Stage 2

Stage 2 is about processing our rcv file using the map transformation (line 41 in our code). What we observe here is the parallelization order of 2. This order is achieved because our initial file consists of 2 partitions. Also, map is a narrow transformation that does not require shuffling, therefore maintaining the same number of partitions as the input RDD.

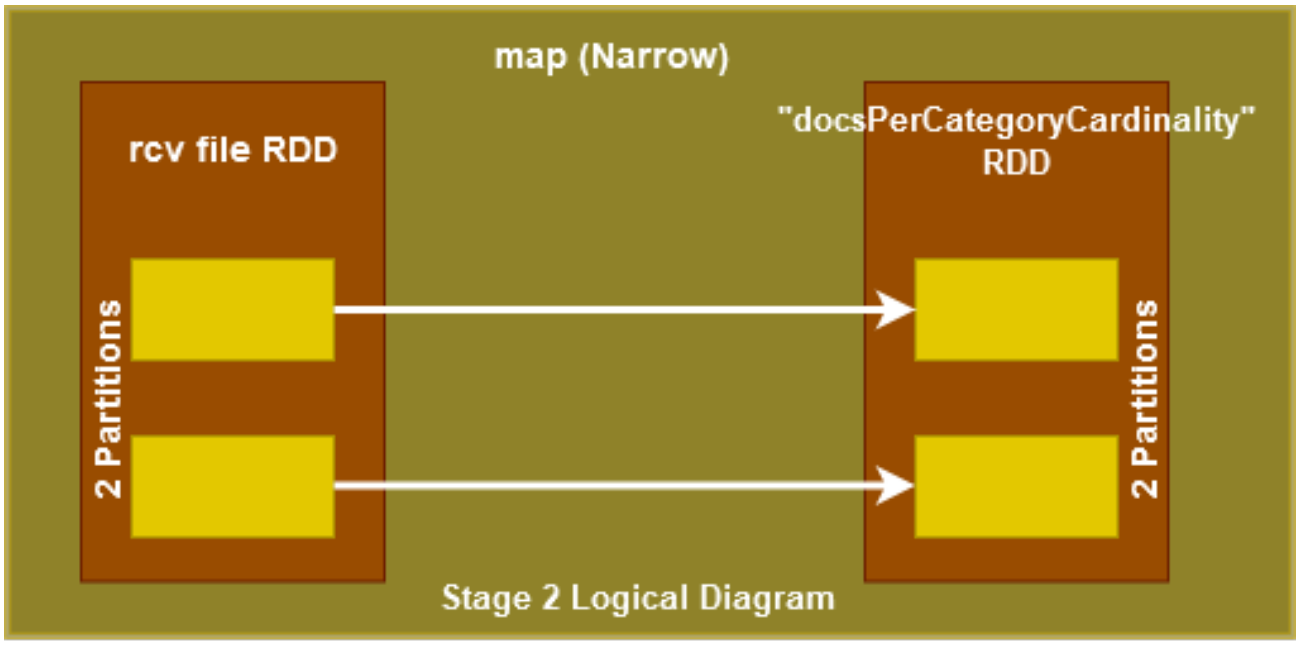


Figure 9: Logical Diagram for Stage 2

STAGE 3

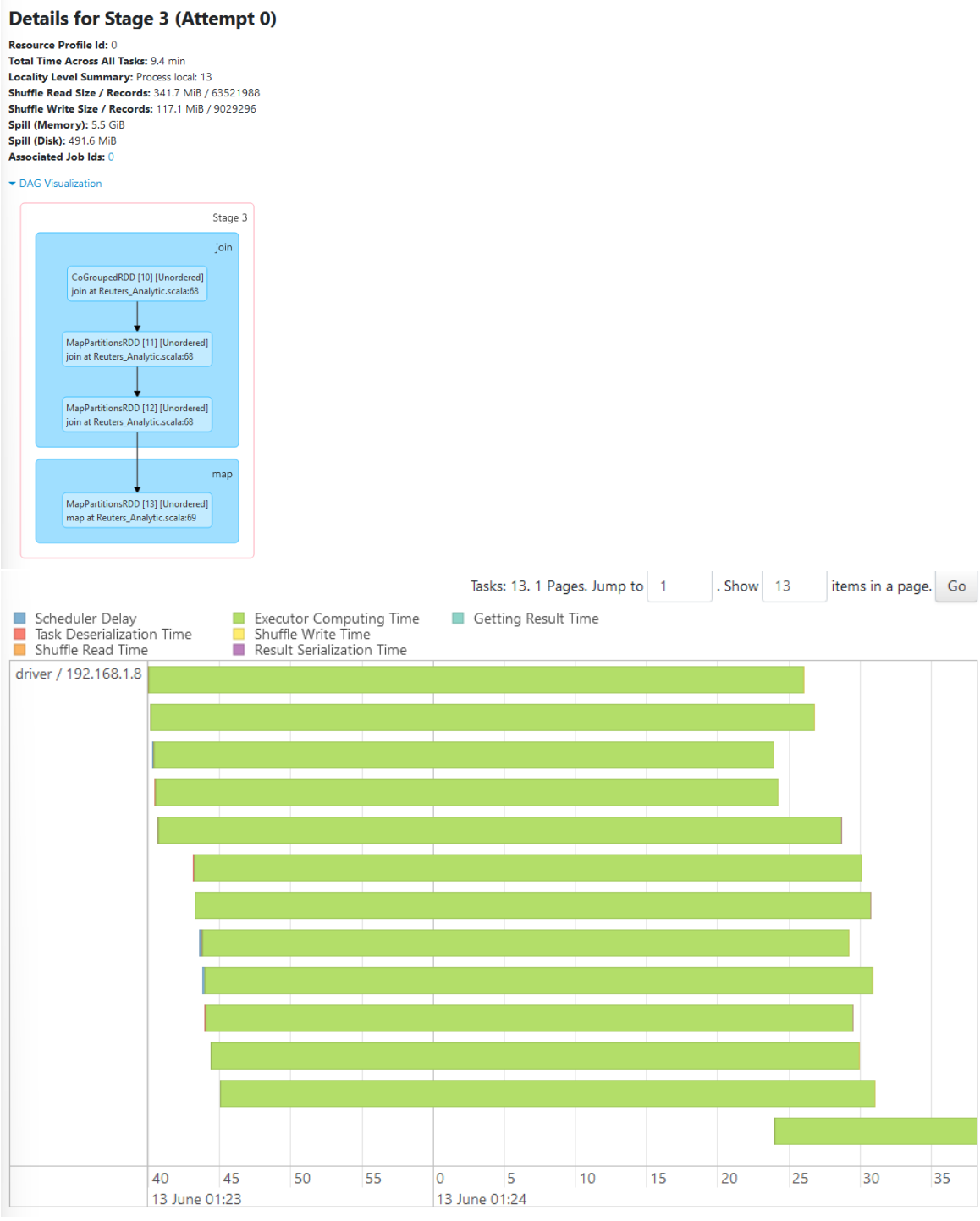


Figure 10: DAG Visualization and Event Timeline of Stage 3

Stage 3 is about creating an RDD that will contain tuples with combinations of terms and categories, by joining the previous RDDs ”termsInDoc” and ”docsPerCategoryCardinality” with the docID as the key.
Something of great interest here, is that although in our code we perform the reduceByKey transformation on line 70 in order to find the cardinality of the intersection, it is not executed in this stage! This is due to the optimizations of Spark, where some transformations are lazily evaluated.
The transformations executed in this stage is a join transformation (wide) and then a map transformation (narrow). Here we achieve a parallelization order of 12 (maximum), since we have 13 tasks, but only 12 available executors.

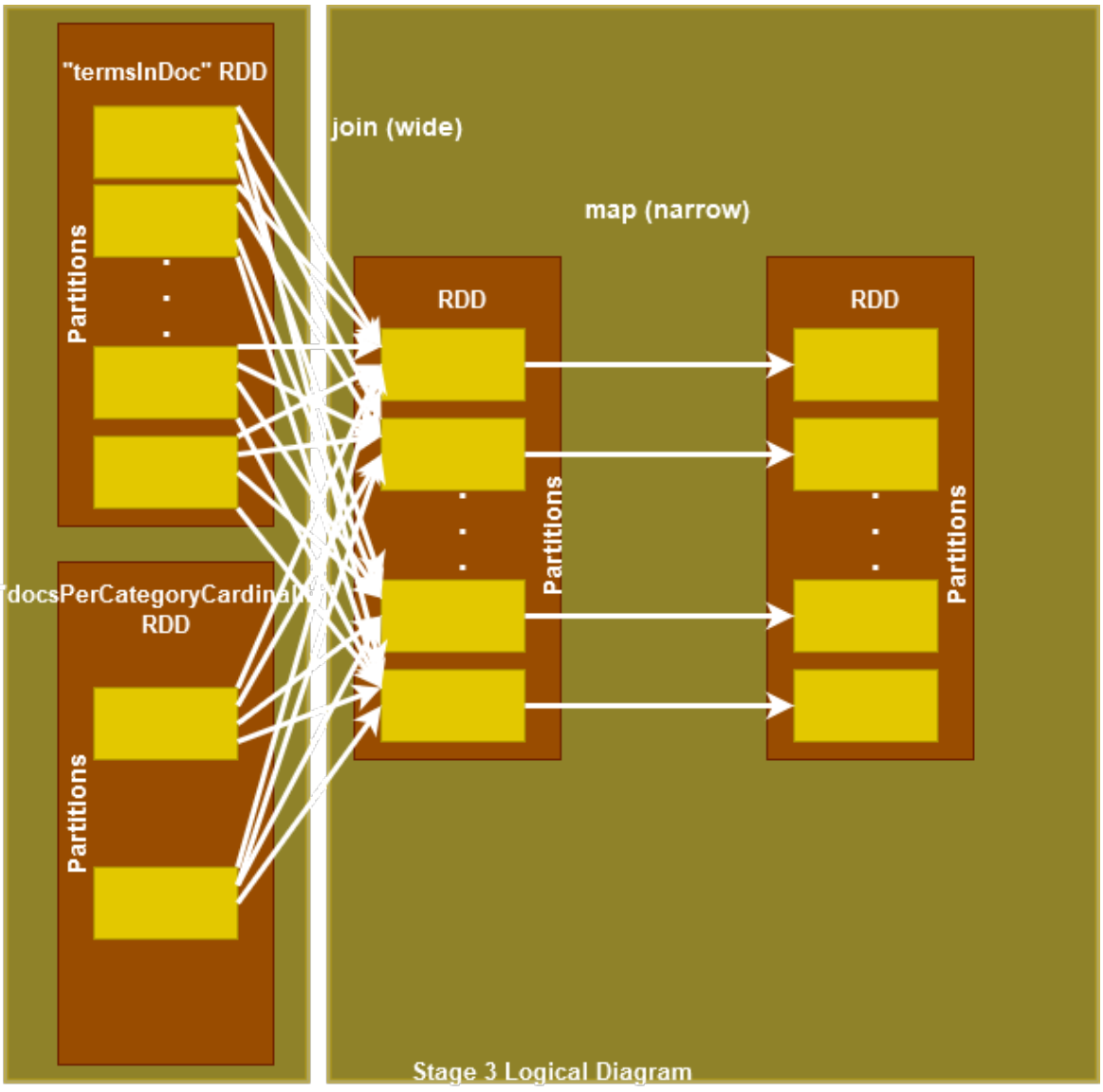


Figure 11: Logical Diagram for Stage 3

STAGE 4

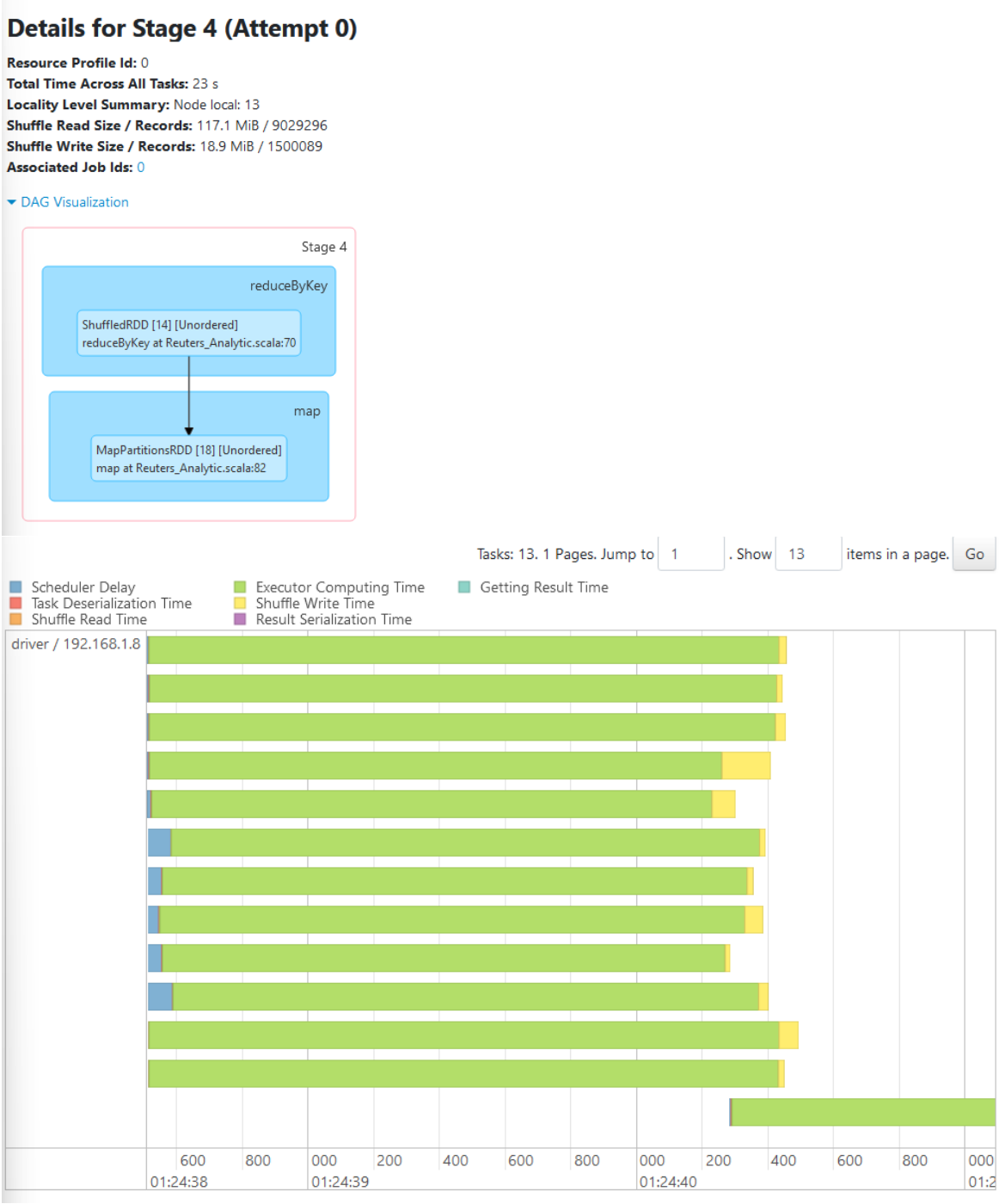


Figure 12: DAG Visualization and Event Timeline of Stage 4

In stage 4, Spark has executed the previous reduceByKey transformation that is on line 70, in order to find the cardinalities of the intersection needed for the next transformation executed which is map on line 82. The parallelization order achieved here is 12, while the tasks are 13. The transformation reduceByKey is wide, while on the other hand, map is narrow.
Below is the logical diagram of stage 3 combined with stage 4, for a better understanding of the execution flow.

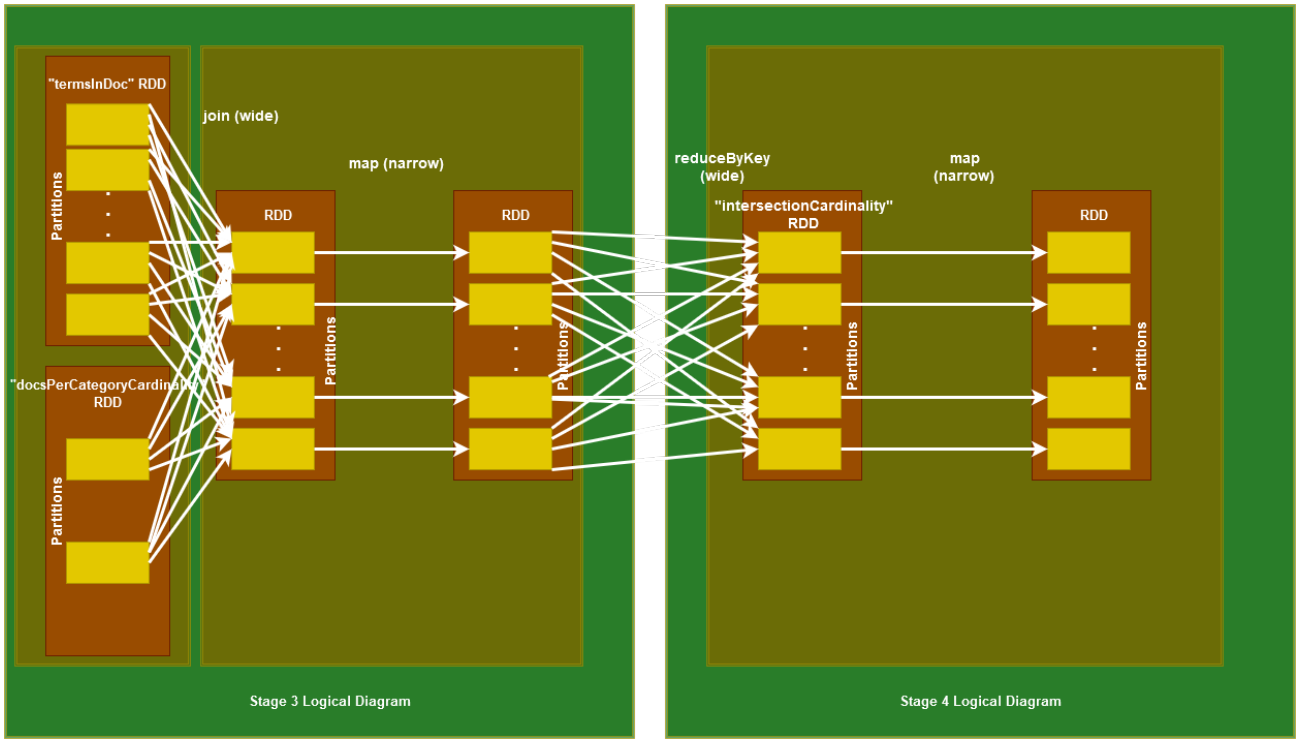


Figure 13: Logical Diagram for Stage 4

STAGE 5

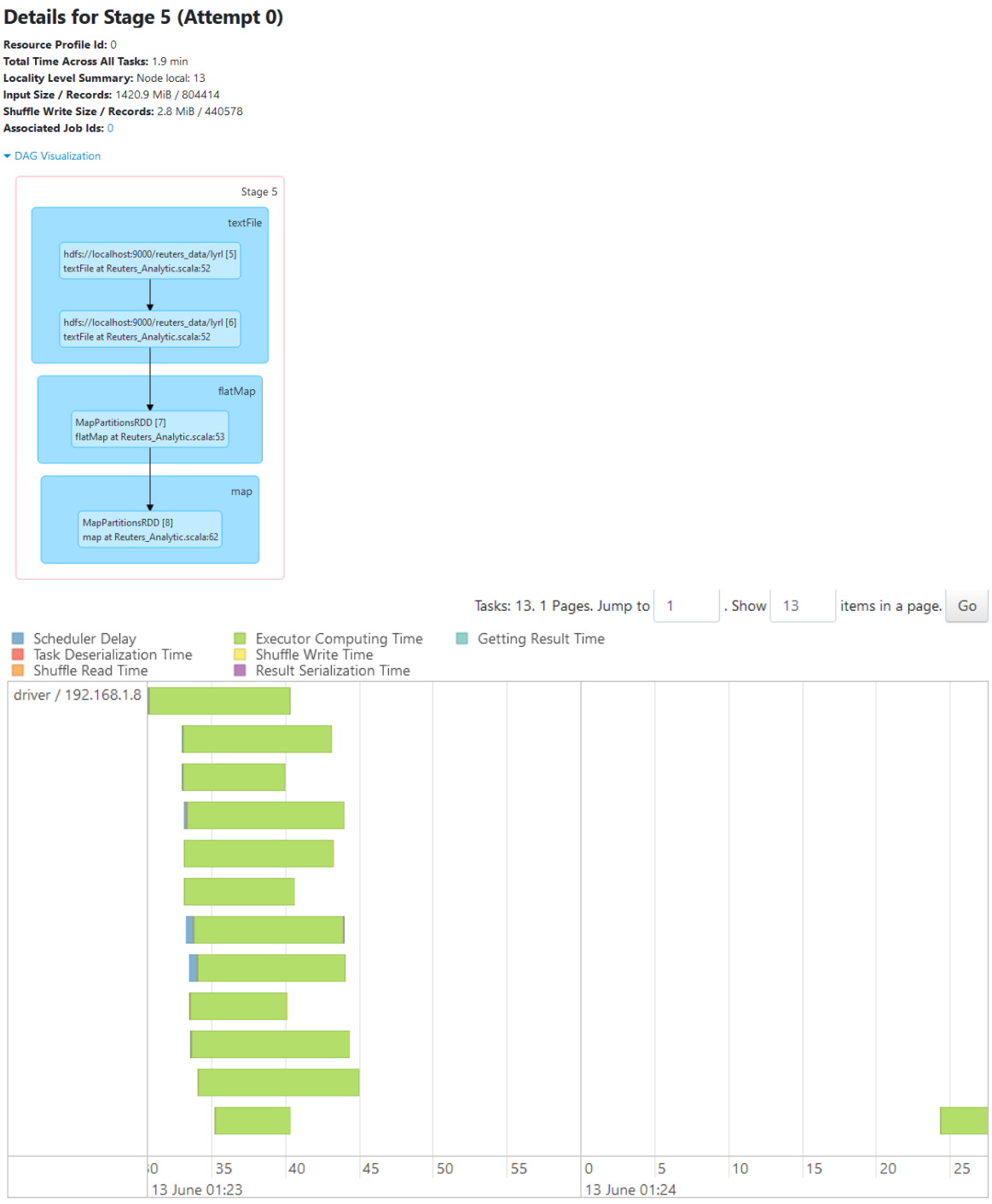


Figure 14: DAG Visualization and Event Timeline of Stage 5

In stage 5, we observe that there are 2 transformations happening. The first is the flatMap transformation on the lyrl files as described by line 53 in our code and the second one is a map transformation on the RDD result of flatMap. **The flatMap transformation though, was executed previously on Stage 1! The reason why it is contained in this stage's DAG Visualization aswell, is that if in a stage Spark has available executors that aren't currently being used, it might "reevaluate" previously evaluated RDDs that are needed for this stage, instead of retrieving them.**
We came to this conclusion by initially changing the line the second transformation map() happened, to execute it directly after the first transformation flatMap.

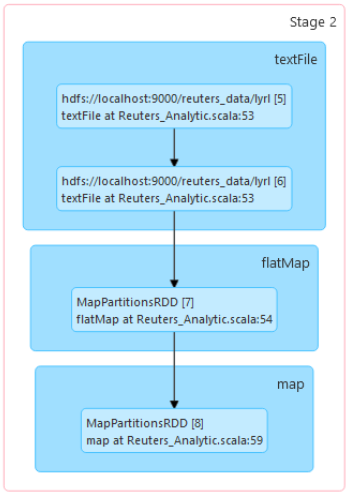
```
1       val termsInDoc =  spark.sparkContext.textFile(pathTermsInDocuments)
2     .flatMap(line => {
3       val line1 = line.split("\\s+", 2) //Split the line into document ID and term IDs
4       val docID = line1(0) //This is our doc ID
5       val termIDs = line1(1).split("\\s+").map(_.split(":")(0)) //Extract term IDs, and ignore
            the weights that are not needed for our analysis
6       termIDs.map(termID => (docID,termID)) //We return a tuple with the docID and the termID
7     }).map(x=>(x._2,1)) //Create a tuple with the termID and 1
8
9
10  val docsCountsInTerms = termsInDoc
11    .reduceByKey(_ + _) //So that we can count the number of documents per term as we did in
          Class
```

Below is what we observed on the History Server, which actually shows that in Stage 2 the first transformation flatMap is applied and afterwards the second transformation map() happens subsequntly. Then on Stage 3 the reduceByKey transformation is happening, without repeating the evaluation of the 2 former transformations.

Details for Stage 2 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 1.8 min
Locality Level Summary: Node local: 13
Input Size / Records: 1420.9 MiB / 804414
Shuffle Write Size / Records: 2.8 MiB / 440578
Associated Job Ids: 1

▾ DAG Visualization



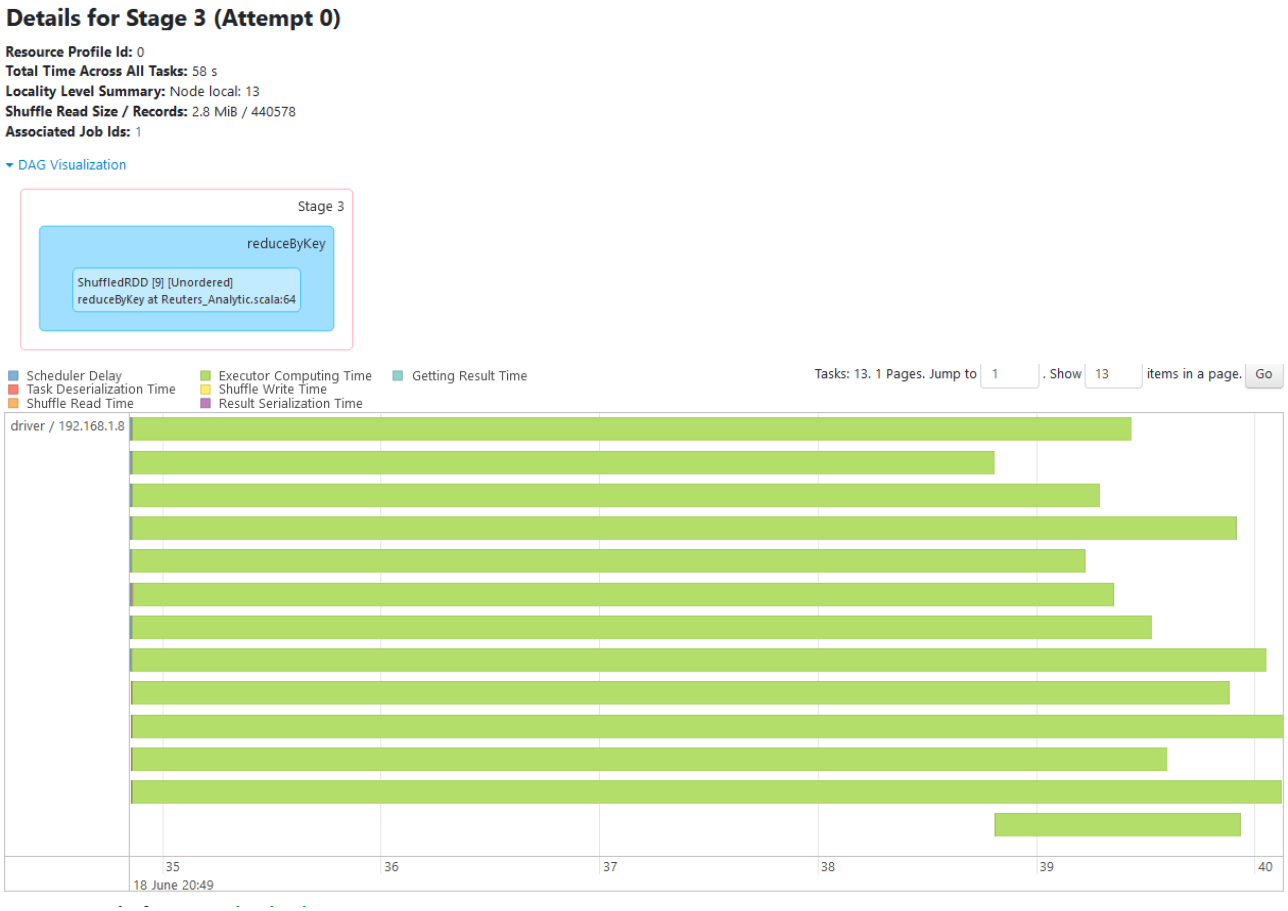


Figure 15: DAG Visualization and event timeline of our modified code

Then we executed the original code, this time only allocating 2 of our 12 available executors to observe the behaviour. What we faced on the History Server was the absolute same execution flow.

Therefore in this stage, what is actually performed is first the narrow transformation flatMap, and then the narrow transformation map (line 62 in our code) on the RDD that was produced by flatMap on stage 1, namely ”termsInDoc” in our code. Here we observe a parallelization order of 12, with the tasks being 13 in total (12 executors allocated by our system).
Below is the logical diagram of stage 1 combined with stage 5, for a better understanding of the execution flow.

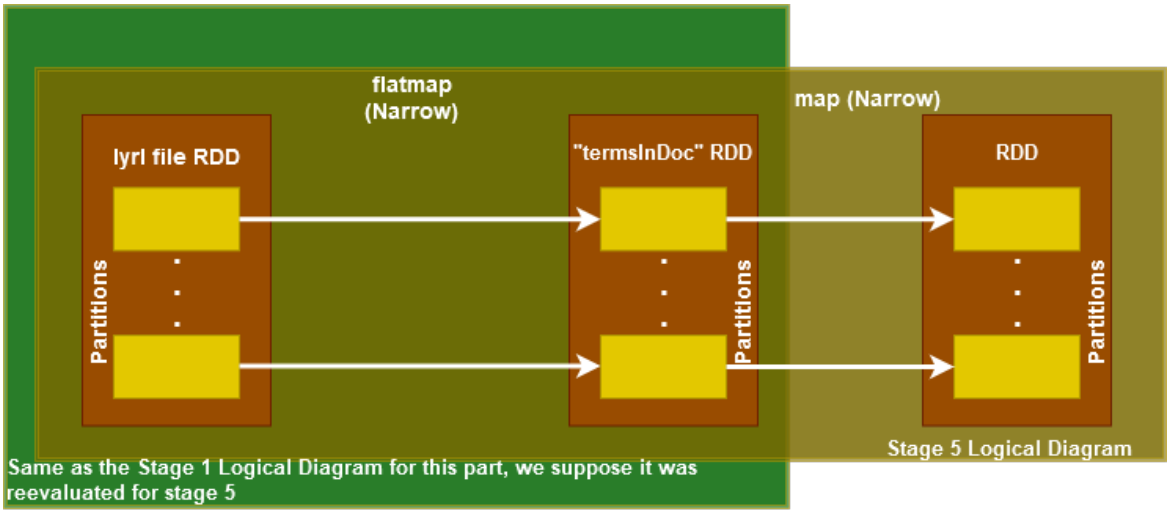


Figure 16: Logical Diagram for Stage 5

STAGE 6

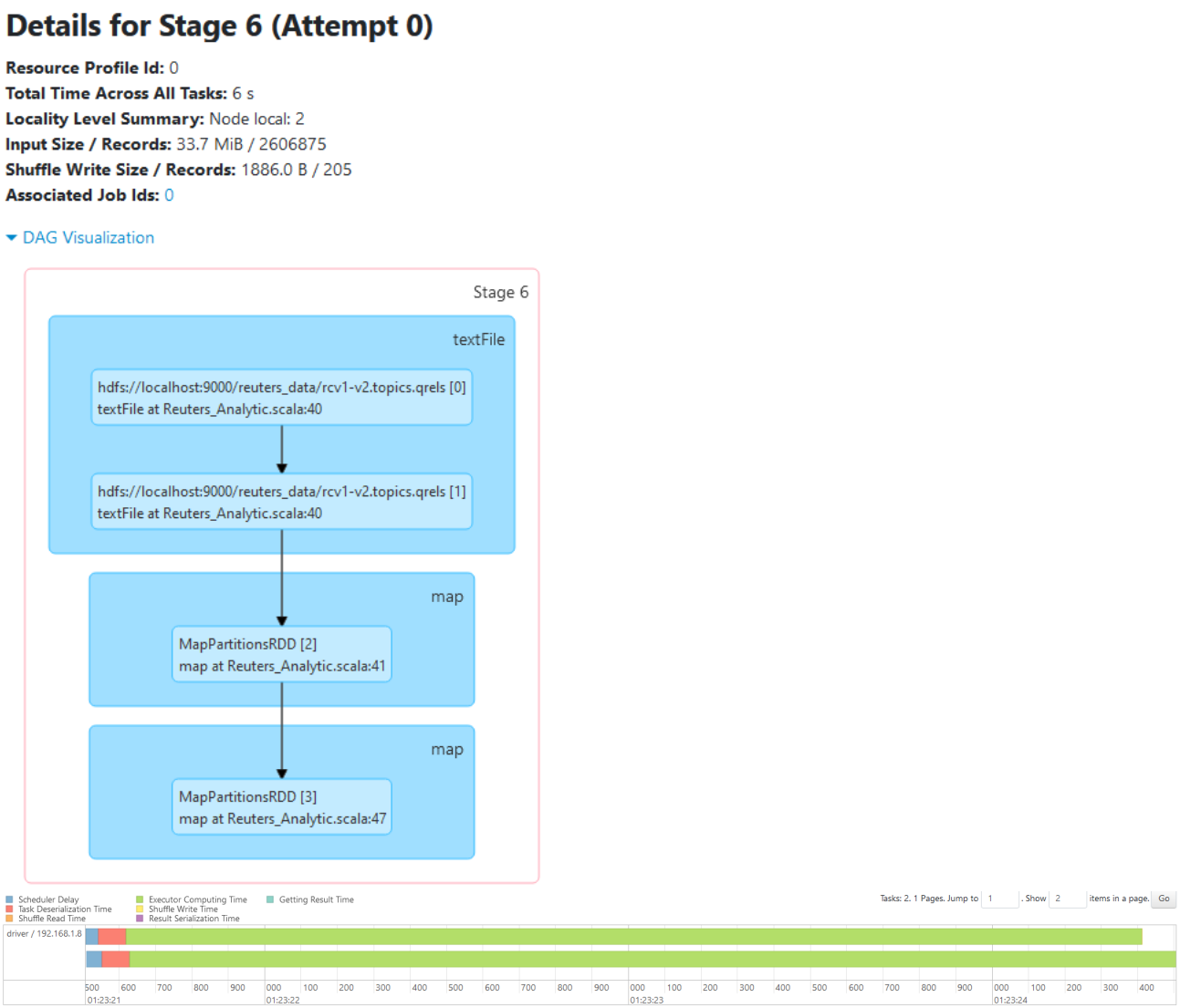


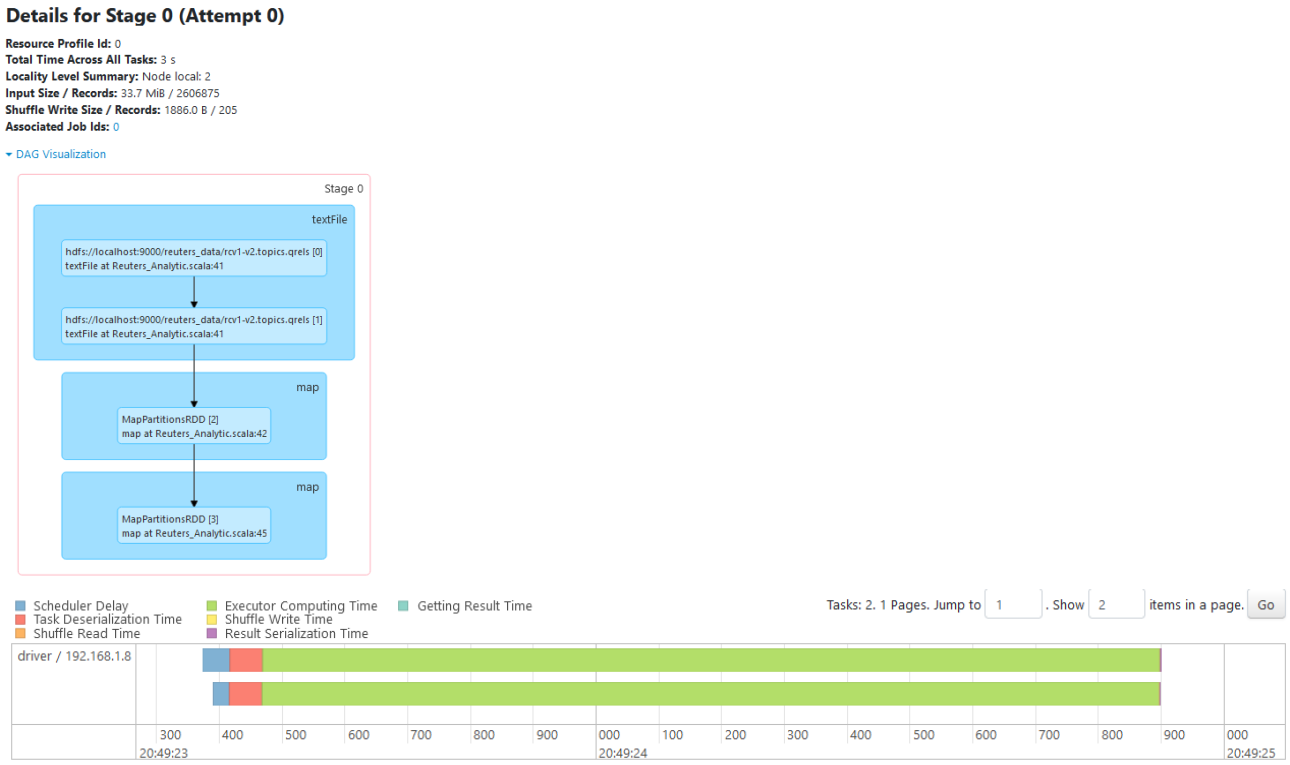
Figure 17: DAG Visualization and Event Timeline of Stage 6

In stage 6, we observe that there are 2 transformations happening. The first is the map transformation on the rcv files as described by line 41 in our code and the second one is also a map transformation on the RDD result of the previous map. **The first map transformation though (line 41), was executed previously on Stage 2! With the same logic described above, we once again modified our code to place the second map() transformation right after the first one.**

```
1       val docsPerCategoryCardinality = spark.sparkContext.textFile(pathCategoriesPerDocument)
2     .map(line => {
3       val fields = line.split("\\s+") //Split whenever one or more whitespaces are found
4       (fields(1),fields(0))//(docID,categoryID)
5     }).map(x=>(x._2,1)) //Create a tuple with the categoryID and 1
6
```

```
7   val docsCountsPerCategory = docsPerCategoryCardinality
8
9     .reduceByKey(_ + _) //So that we can count the number of documents per category as we did
          in Class
```

Below is what we observed on the History Server, which actually shows that in Stage 0 the first transformation map() is applied and afterwards the second transformation map() happens subsequntly. Then on Stage 1 the reduceByKey transformation is happening, without repeating the evaluation of the 2 former transformations.



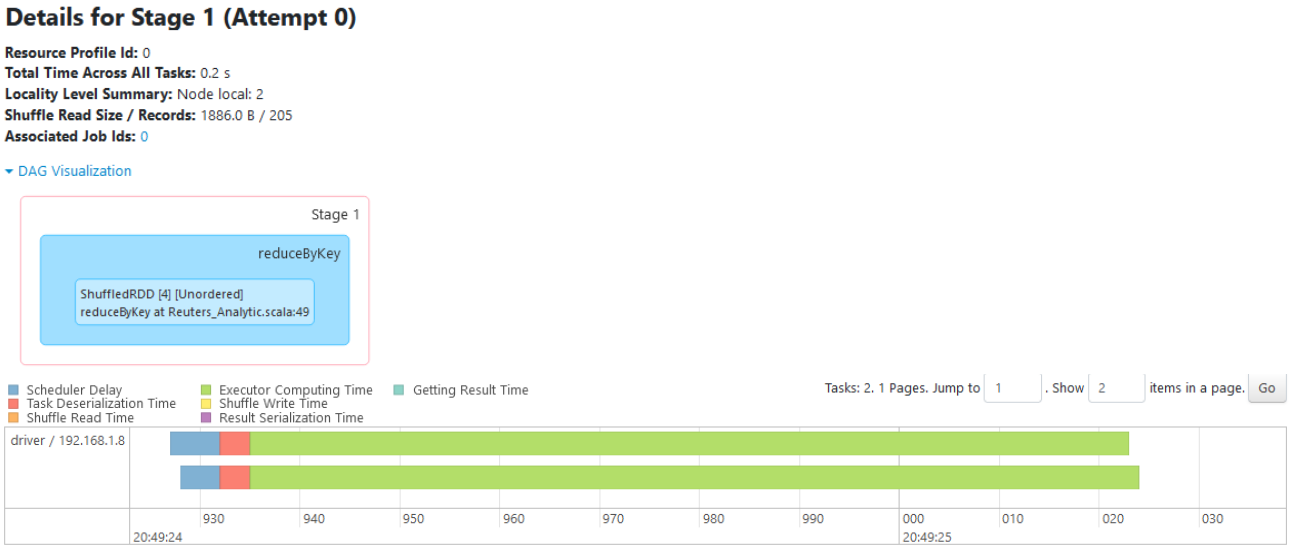


Figure 18: DAG Visualization and event timeline of our modified code

Then we executed the original code, this time only allocating 2 of our 12 available executors to observe the behaviour. What we faced on the History Server was the absolute same execution flow.

Therefore in this stage, what is actually performed is first the reevaluation of the first narrow transformation map, and then the second narrow transformation map (line 47 in our code) on the RDD that was produced by the map on stage 2, namely ”docsPerCategoryCardinality” in our code. Here we observe a parallelization order of 2, equal to the number of total tasks and also equal to the number of partitions of RDD ”docsPerCategoryCardinality”.
Below is the logical diagram of stage 2 combined with stage 6, for a better understanding of the execution flow.

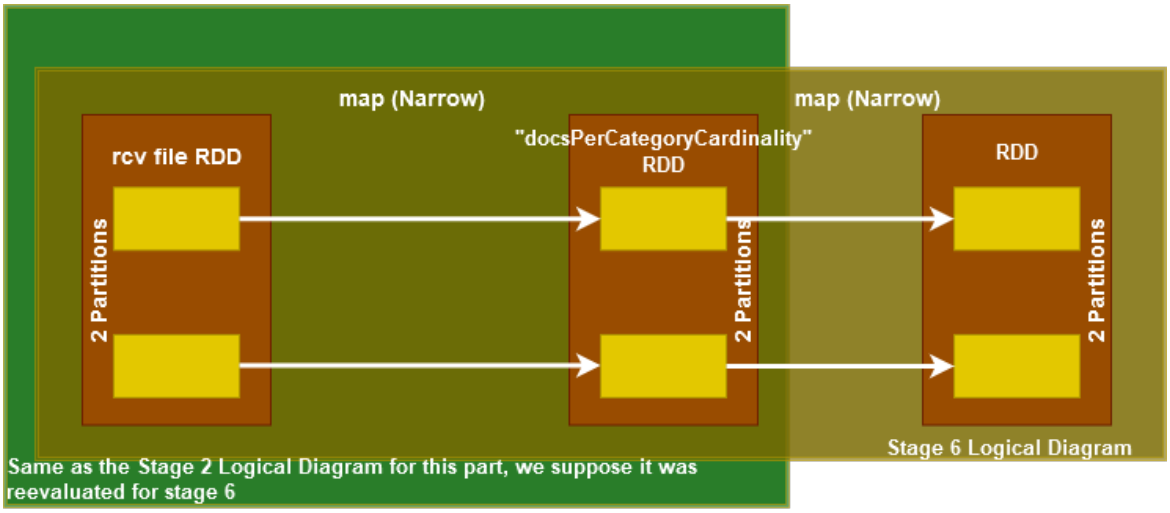


Figure 19: Logical Diagram for Stage 6

STAGE 7

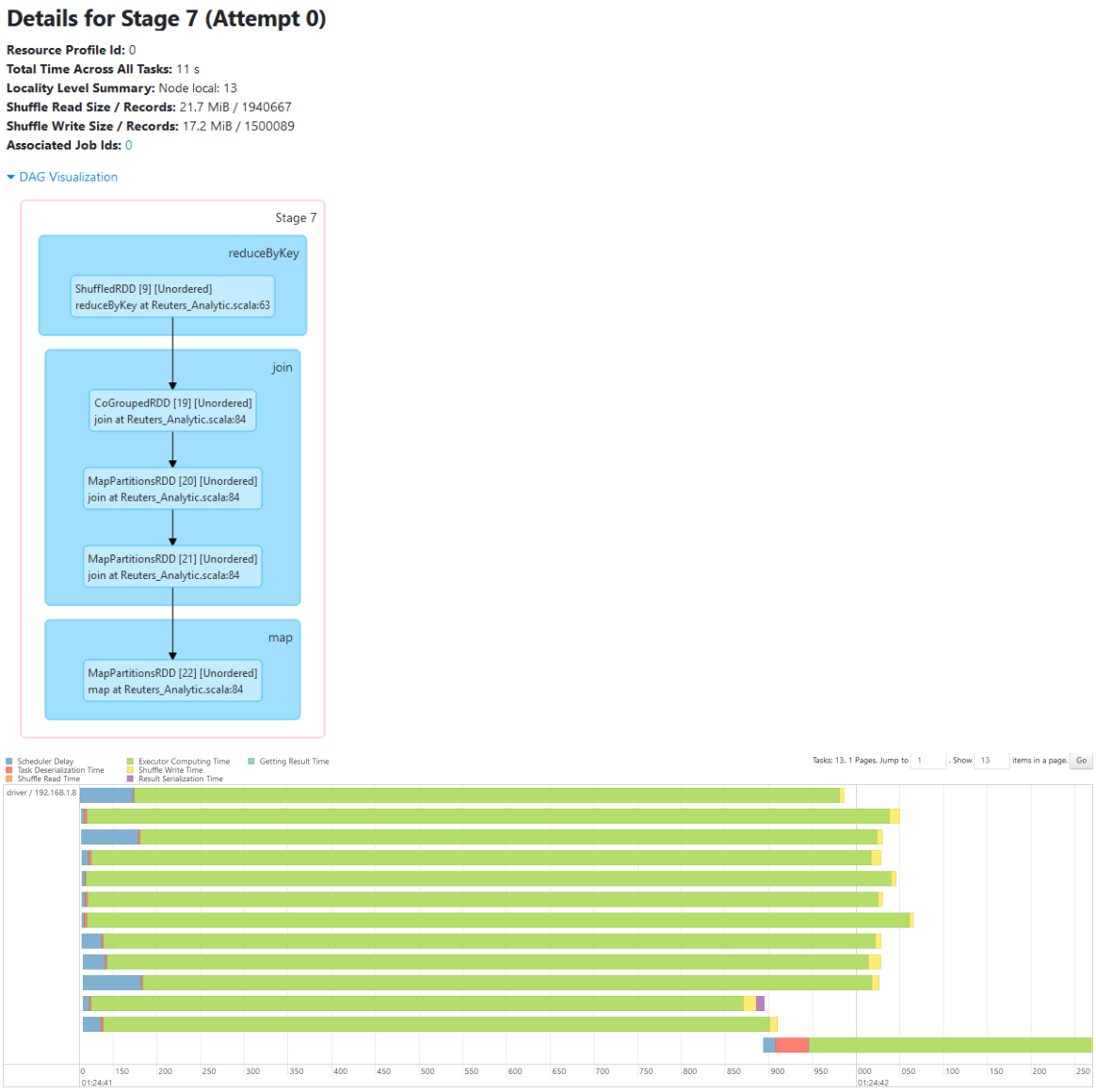


Figure 20: DAG Visualization and Event Timeline of Stage 7

In stage 7, there are 3 transformations executed. The first one, is the wide transformation reduceBykey (line 63 in our code) which is applied to the result RDD of stage 5. Therefore the RDD ”docsCountsInTerms” is created. The next transformation is the wide join transformation (line 84 in our code) which is performed between the partitions of the result RDD of Stage 4 and the partitions of RDD ”docsCountsInTerms”. On the resulting RDD of the join transformation, is applied the third and final transformation of this stage which is map (line 84 in our code). In this stage, we have a total of 13 tasks, and we achieve a parallelization order of 12 (12 executors). **Below is the logical diagram of stage 7, combined with parts from different stages for visualization purposes and better understanding.**

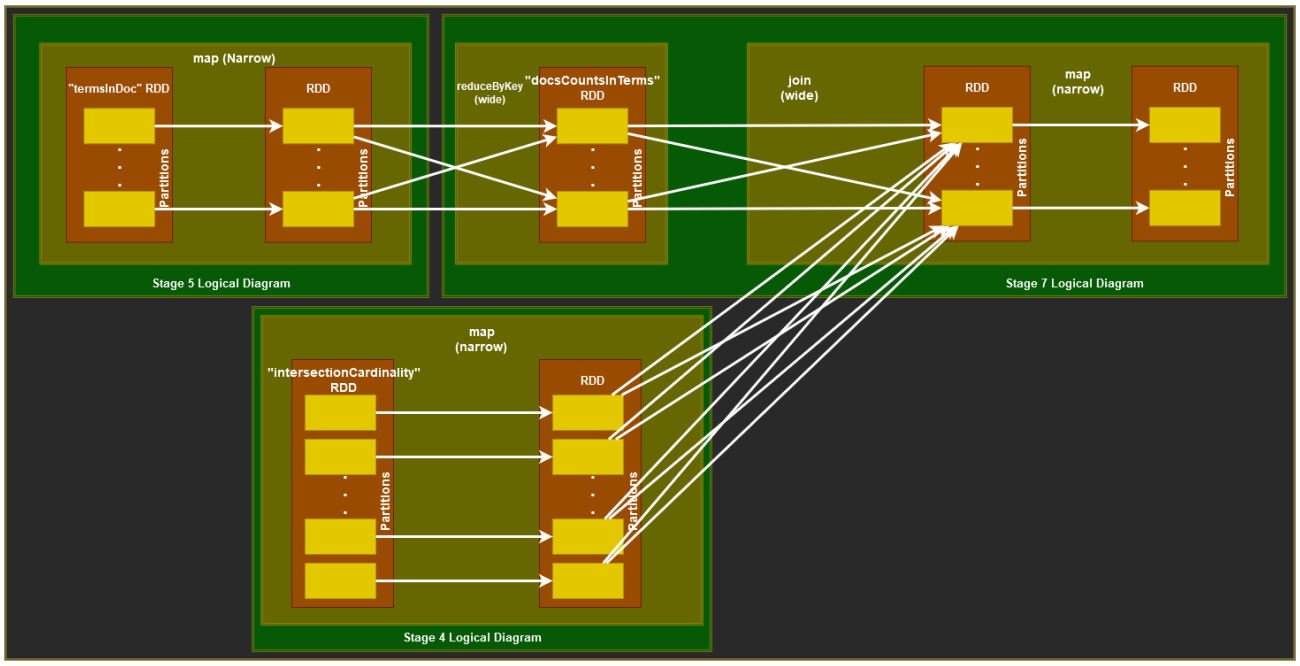


Figure 21: Logical Diagram for Stage 7

STAGE 8

Details for Stage 8 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 3 s
Locality Level Summary: Node local: 2
Shuffle Read Size / Records: 17.2 MiB / 1500294
Shuffle Write Size / Records: 21.4 MiB / 1500089
Associated Job Ids: 0

▾ DAG Visualization

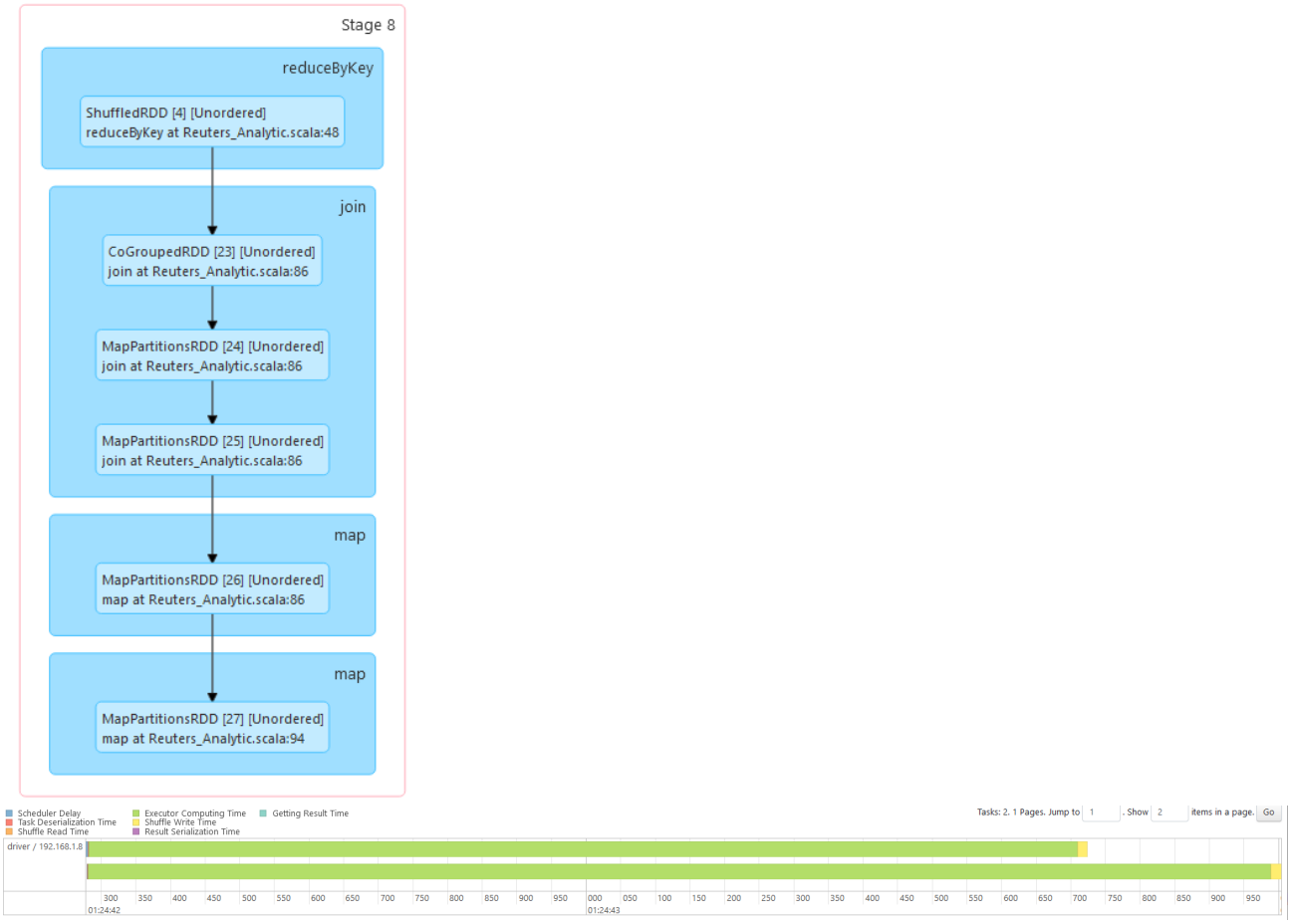


Figure 22: DAG Visualization and Event Timeline of Stage 8

In stage 8, there are 4 transformations executed. The first one, is the wide transformation reduceBykey (line 48 in our code) which is applied to the result RDD of stage 6. Therefore the RDD ”docsCountsPerCategory” is created. The next transformation is the wide join transformation (line 86 in our code) which is performed between the partitions of the result RDD of Stage 7 and the partitions of RDD ”docsCountsPerCategory”. On the resulting RDD of the join transformation, is applied the third transformation of this stage which is the narrow map (line 86 in our code). By now the RDD ”jaccardIndex” has been created, to which we apply the 4th and final narrow transformation, map (line 94 in our code). In this stage, we have a total of 2 tasks, and we achieve a parallelization order of 2. **Below is the logical diagram of stage 8, combined with parts from different stages for visualization purposes and better understanding.**

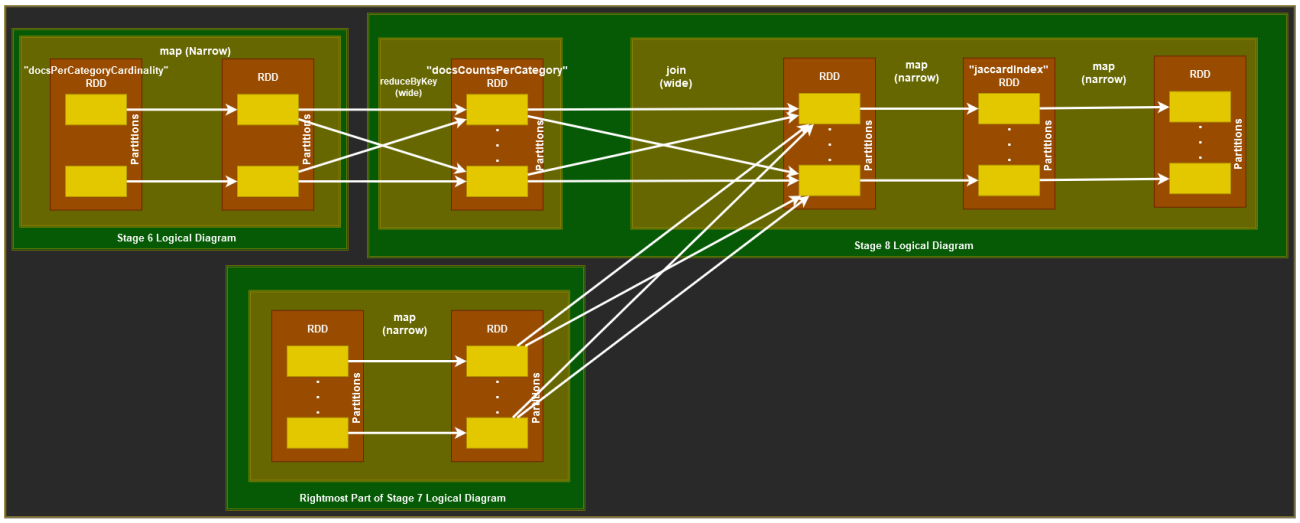


Figure 23: Logical Diagram for Stage 8

STAGE 9

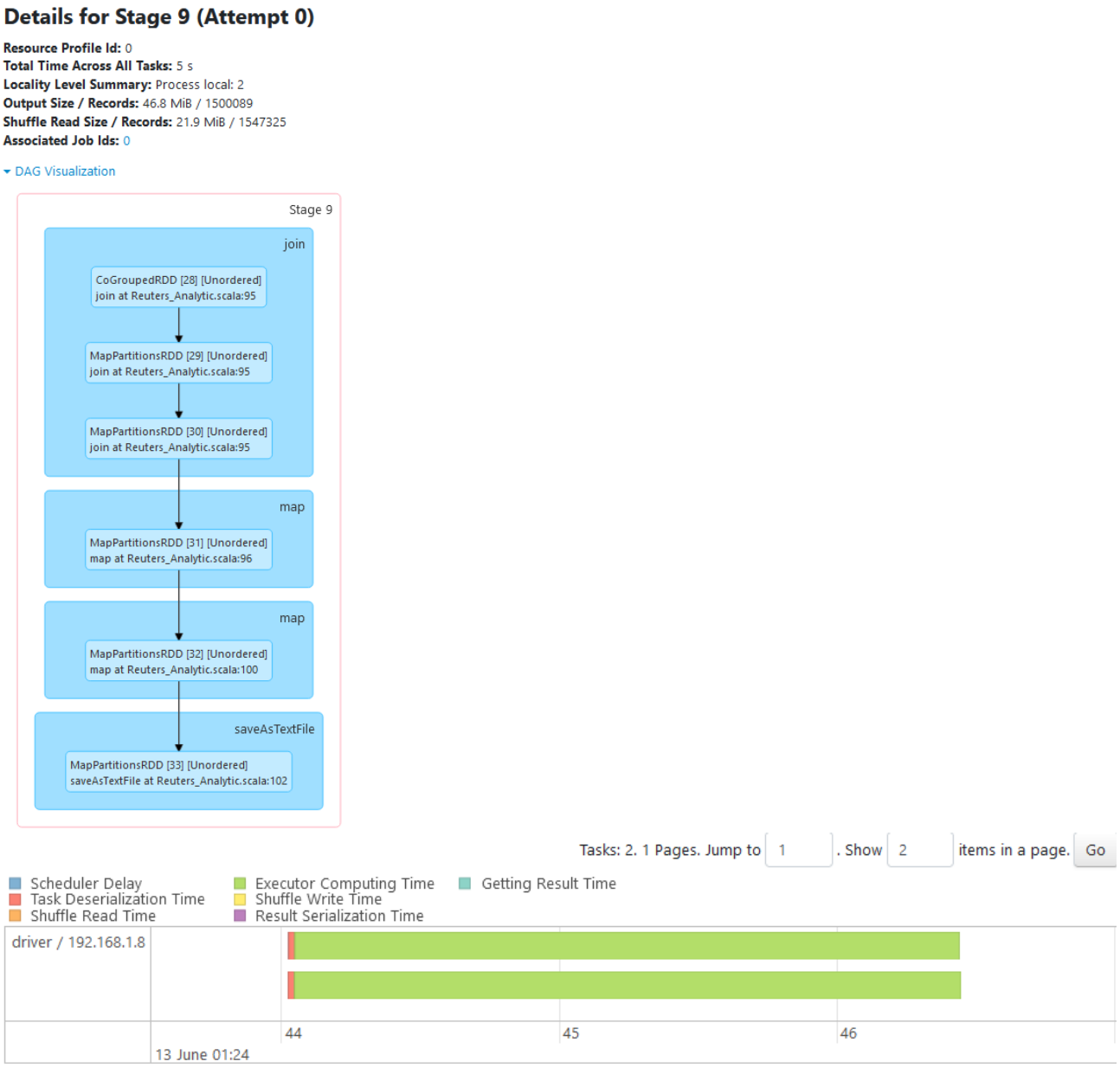


Figure 24: DAG Visualization and Event Timeline of Stage 9

In stage 9, there are 4 transformations executed. The first one, is the wide transformation join (line 95 in our code) which is applied to the result RDD of stage 8 and the ”termToStem” RDD. On the resulting RDD is applied the next transformation which is the narrow map transformation (line 96 in our code). Therefore, the RDD ”jaccardIndexFinal” is created. The third transformation is the narrow map (line 100 in our code) which is applied to the contents of the RDD ”jaccardIndexFinal” in order to transform each tuple to a string of desired format. At last, the action saveAsTextFile is performed. In this stage, we have a total of 2 tasks, and we achieve a parallelization order of 2.

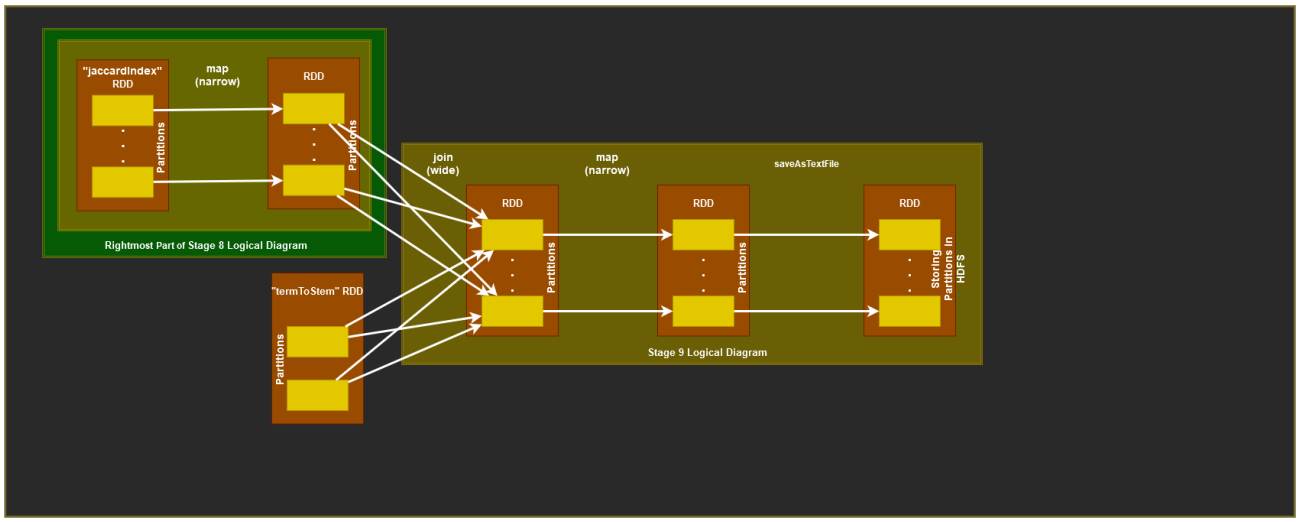


Figure 25: Logical Diagram for Stage 9

2 Aegean Analytics Application

2.1 Brief Description

Problem Setup:
In this scenario we were provided with a dataset collected from a set of stations located in Piraeus and Syros Island. It consisted of the following file:

nmea_aegeanlogs file, which contained the information below:

- **timestamp**: the timestamp of the acquired vessel position
- **station**: the id of the station which acquired the corresponding vessel position
- **mmsi**: maritime mobile service identity, consider this as the vessel Id
- **longitude**:geographic longitude of each timestamped vessel position
- **latitude**: geographic latitude of each timestamped vessel position
- **Speed Over Ground (SOG)**: is the speed of the vessel relative to the surface of the earth
- **Course Over Ground (COG)**: is the actual direction of progress of a vessel, between two points, with respect to the surface of the earth
- **Heading**: is the compass direction in which the vessel bow is pointed
- **Status**:
  - 0 = underway using engine
  - 1 = at anchor
  - 2 = not under command
  - 3 = restricted maneuverability
  - 4 = constrained by her draught
  - 5 = moored
  - 6 = aground
  - 7 = engaged in fishing
  - 8 = underway sailing
  - 9 & 10 = reserved for future amendment
  - 11 = power-driven vessel towing astern
  - 12 = power-driven vessel pushing ahead or towing alongside
  - 13 = reserved for future use
  - 14 =AIS-SART Active (Search and Rescue Transmitter), AIS-MOB (Man Overboard), AISEPIRB (Emergency Position Indicating Radio Beacon)
  - 15 = undefined = default (also used by AIS-SART, MOB-AIS, and EPIRB-AIS under test)

For this task, we are going to use DataFrames.

2.2 Code Analysis

The goal of our analysis was to provide answers to the following queries. Before addressing these questions, we processed the data to ensure the integrity and accuracy of the dataset. Specifically, we typecasted the columns speedoverground, heading, and courseoverground to DoubleType because, as string types, we could not apply mathematical operations such as averaging. Additionally, we filtered heading and courseoverground to be between 0 and 360 because we considered that these measurements are in degrees.

```
1  //Here we sanitize our data, typecasting on columns that actually contain numeric values and
       filtering outliers (heading,sog > 360)
2  val SanitizedDf = df.withColumn("speedoverground", col("speedoverground").cast(DoubleType))
3    .withColumn("heading", col("heading").cast(DoubleType))
4    .withColumn("courseoverground", col("courseoverground").cast(DoubleType))
5    .filter(col("heading") <= 360)
6    .filter(col("courseoverground") <= 360)
```

In this code snippet, we utilized the withColumn function to transform specific columns in the DataFrame df. For each column, we provided the column name as the first argument and an expression to cast its values to DoubleType as the second argument. Additionally, we applied the filter HOF to ensure that their values fall within the range of 0 to 360 degrees.
After completing the data sanitization process, we proceeded to address the queries listed below.

- **Q1:** What is the number of tracked vessel positions per station per day?

```
1    //Question_1
2    val numberOfTrackedVesselPositions =SanitizedDf
3      .groupBy((dayofmonth(col("timestamp"))),col("station")) //Group by day and station
4      .count().alias("Number of Tracked Vessels") //Count the number of tracked vessels for
           each day and station
5
6    numberOfTrackedVesselPositions.show()//Print the result
7
8    numberOfTrackedVesselPositions.rdd.saveAsTextFile(outputPath1)//Convert the dataFrame to
         an rdd and  save the results in the output path
```

For Question 1, we first grouped the sanitized data frame by the day of the month and the station using the groupBy HOF. The dayofmonth function in Spark SQL was employed to extract the day of the month from the ”timestamp” column for each row. Typically, the timestamp column is in the format ’YYYY-MM-DDTHH:MM’.Then, we counted the number of tracked vessels for each day and station by applying the count function and after that we renamed the resulting column using the alias function, assigning it the name ”Number of Tracked Vessels”. After the data processing, we printed the result with show() and we saved it as a text file after first transforming it to rdd.

- **Q2:**What is the vessel id with the highest number of tracked positions?

```
1  //Question_2
2    val maximumTrackedPositions = SanitizedDf
3     .groupBy(col("mmsi")) // Group by vesselID
4      .count().withColumnRenamed("count", "Number of Tracked Vessels in Total") // Count
           the number of tracked positions for each vessel
5     .orderBy(col("Number of Tracked Vessels in Total").desc) // Order by the number of
           tracked positions in descending order
6     .limit(1) // Get the vessel with the most tracked positions
7
8    maximumTrackedPositions.show()//Print the result
9
10   maximumTrackedPositions.rdd.saveAsTextFile(outputPath2)//Convert the dataFrame to an
         rdd and  save the results in the output path
```

For Question 2, we began by grouping the sanitized DataFrame by the vessel ID. Next, we counted the number of rows for each vessel in line 4, representing the number of tracked positions for each vessel using the count() function. We then renamed the resulting ”count” column to ”Number of Tracked Vessels in Total” using the withColumnRenamed() function. After that, we ordered the results in descending order based on the number of tracked positions for each vessel, using orderBy with the desc argument. The limit(1) at line 6 allowed us to select the first result, which corresponds to the vessel with the maximum number of tracked positions. Finally, as with all queries, we printed the result with show() and we saved it as a text file after first transforming it to rdd.

- **Q3:**What is the average SOG of vessels that appear in both station 8006 and station 10003 in the same day?

```
1  //Question_3
2  val station10003 = SanitizedDf
3    .filter(col("station") === 10003).withColumnRenamed("speedoverground","
         speedoverground1") //Renaming needed so that we can later calculate the average
         SOG
4    .select(dayofmonth(col("timestamp")).alias("day"),col("mmsi"),col("speedoverground1"))
          //Retrieving the day, vesselID and SOG for vessels tracked in station 10003
5
6  val station8006 = SanitizedDf
7    .filter(col("station") === 8006).withColumnRenamed("speedoverground","speedoverground2
         ") //Renaming needed so that we can later calculate the average SOG
8    .select(dayofmonth(col("timestamp")).alias("day"),col("mmsi"),col("speedoverground2"))
          //Retrieving the day, vesselID and SOG for vessels tracked in station 8006
9
10 val joinedDF = station10003
11   .join(station8006, Seq("day","mmsi"),"inner") //Inner join ensures that we only get
         the vessels (with the same ID) that were tracked in both stations on the same day
12   .select("day","mmsi","speedoverground1","speedoverground2") //Selecting the SOG of the
          vessels tracked in both stations
13   .agg(((avg("speedoverground1") + avg("speedoverground2")) / 2).alias("Average_SOG"))
        //Calculating the average SOG of the vessels tracked in both stations
14
15 joinedDF.show()//Print the result
16
17 joinedDF.rdd.saveAsTextFile(outputPath3)//Convert the dataFrame to an rdd and  save the
       results in the output path
```

For Question 3, we began by extracting specific information for each of the two stations, ”8006” and ”10003,” from the sanitized DataFrame. Initially, we applied the filter higher-order function (HOF) to isolate data rows where the ”station” column matched each respective station ID. Following this, we used withColumnRenamed to rename the column ”speedoverground” to ”speedoverground1” for station ”10003” and ”speedoverground2” for station ”8006.”

This step was essential for later calculations to differentiate the speed over ground (SOG) values from each station. Subsequently, using the select function, we narrowed down the extracted data to include only the essential columns required for subsequent processing: ”dayofmonth” (extracted from the ”timestamp” column), ”mmsi” (vessel ID), and the newly renamed ”speedoverground” columns corresponding to each station. The next step involved joining these two DataFrames using an inner join operation (join) based on the common columns ”day” and ”mmsi.” The inner join ensures that we only consider vessels that were tracked in both stations on the same day. After the join, we used select to focus on the relevant columns ”speedoverground1” and ”speedoverground2” and applied the agg function. Inside agg, we calculated the average SOG of each station using the avg() function and then computed the overall average SOG for both stations by summing up the averages of the stations and dividing by two. The resulting column was given the alias ”Average_SOG”. Finally we printed the result and we saved it as a text file after first transforming it to rdd.

Observation: The result for this question was **NULL**. This likely occurs because the two stations do not have common elements in the columns ”day” and ”mmsi,” so the join returns no results. To confirm this, we ran the following code to see the days each station appears.

```
val daysStation10003 = SanitizedDf
   .filter(col("station") === 10003)
   .select(dayofmonth(col("timestamp")).alias("day"),col("station"))
   .distinct()
   .orderBy("day")
   .show()

 val daysStation8006 = SanitizedDf
   .filter(col("station") === 8006)
   .select(dayofmonth(col("timestamp")).alias("day"),col("station"))
   .distinct()
   .orderBy("day")
   .show()
```

What we are doing above is filtering the SanitizedDf to only include rows where the station is either 10003 or 8006 and extracting the day of the month and station for each one using select. After that, we used distinct fucntion to get unique combinations of (station, day) and order the results by day. The results we took are shown below.

day	station
17	10003
18	10003
19	10003
20	10003
21	10003
22	10003
23	10003
24	10003
25	10003
26	10003
27	10003
28	10003
29	10003
30	10003

(a) Results for Station ’10003’

day	station
15	8006
16	8006

(b) Results for Station ’8006’

As we observed in the screenshots above, the two stations do not have any common days, so it is expected that the final result is **NULL**.

- **Q4:**What is the average Abs (Heading – COG) per station?

```
//Question_4
 val averageAbs = SanitizedDf
   .groupBy(col("station")) //Group by station
   .agg(avg(abs(col("heading")-col("courseoverground")))).alias("Average Abs") //
       Calculate the average absolute difference between heading and COG for each station

 averageAbs.show()//Print the result

 averageAbs.rdd.saveAsTextFile(outputPath4)//Convert the dataFrame to an rdd and  save
     the results in the output path
```

For Question 4, we started by grouping the sanitized DataFrame SanitizedDf by the ”station” column using the groupBy function. Next, we used the agg function to aggregate the grouped data. Inside agg, we applied the avg and abs function to calculate the average absolute difference between the ”heading” and ”courseoverground” columns for each station. The resulting column was given the alias ”Average Abs”. Finally,we printed the result with show() and we saved it as a text file after first transforming it to rdd.

- **Q5:**What are the Top-3 most frequent vessel statuses?

```
  //Question_5
 val mostFrequentStatuses = SanitizedDf
   .groupBy(col("status")) //Group by status
   .count().withColumnRenamed("count","Most Frequents Statuses")//Count the number of
       records for each status
   .orderBy(col("Most Frequents Statuses").desc) //Order by the number of records in
       descending order
   .limit(3) //Take the top 3

 mostFrequentStatuses.show()//Print the result

 mostFrequentStatuses.rdd.saveAsTextFile(outputPath5)//Convert the dataFrame to an rdd
     and  save the results in the output path
```

For Question 5, we first grouped the sanitized DataFrame by the ”status” column using the groupBy function. Next, we applied the count function to calculate the number of records for each status group. The resulting column from count was renamed to ”Most Frequents Statuses” using withColumnRenamed.We then ordered the DataFrame by the ”Most Frequents Statuses” column in descending order using the orderBy function. This ensures that the statuses with the highest count appear first.To focus on the top 3 most frequent statuses, we used the limit(3) function. Finally,we printed the result with show() and we saved it as a text file after first transforming it to rdd.

2.3 Execution Analysis

Everything displayed below is the result of a local execution. The CPU had 12 threads, which as we are going to see is also the maximum parallelization achieved. It also depends on the number of partitions each dataFrame takes up which reflects the number of tasks.

After executing our produced JAR file, we observed the following on the Apache Spark History Server:
JOBS

Spark Jobs (?)

User: Ermina
Total Uptime: 34 s
Scheduling Mode: FIFO
Completed Jobs: 27

▸ Event Timeline

▾ **Completed Jobs (27)**

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

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
26	runJob at SparkHadoopWriter.scala:83 runJob at SparkHadoopWriter.scala:83	2024/06/16 19:56:00	89 ms	1/1 (1 skipped)	1/1 (12 skipped)
25	rdd at Aegean_Analytics.scala:85 rdd at Aegean_Analytics.scala:85	2024/06/16 19:55:58	1 s	1/1	12/12
24	show at Aegean_Analytics.scala:83 show at Aegean_Analytics.scala:83	2024/06/16 19:55:58	35 ms	1/1 (1 skipped)	1/1 (12 skipped)
23	show at Aegean_Analytics.scala:83 show at Aegean_Analytics.scala:83	2024/06/16 19:55:56	2 s	1/1	12/12
22	runJob at SparkHadoopWriter.scala:83 runJob at SparkHadoopWriter.scala:83	2024/06/16 19:55:56	0.1 s	1/1 (1 skipped)	1/1 (12 skipped)
21	rdd at Aegean_Analytics.scala:74 rdd at Aegean_Analytics.scala:74	2024/06/16 19:55:54	2 s	1/1	12/12
20	show at Aegean_Analytics.scala:72 show at Aegean_Analytics.scala:72	2024/06/16 19:55:54	63 ms	1/1 (1 skipped)	1/1 (12 skipped)
19	show at Aegean_Analytics.scala:72 show at Aegean_Analytics.scala:72	2024/06/16 19:55:53	2 s	1/1	12/12
18	runJob at SparkHadoopWriter.scala:83 runJob at SparkHadoopWriter.scala:83	2024/06/16 19:55:52	0.2 s	1/1 (2 skipped)	1/1 (13 skipped)
17	rdd at Aegean_Analytics.scala:65 rdd at Aegean_Analytics.scala:65	2024/06/16 19:55:52	56 ms	1/1 (1 skipped)	1/1 (12 skipped)
16	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264 $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2024/06/16 19:55:51	0.5 s	1/1 (1 skipped)	1/1 (12 skipped)
15	rdd at Aegean_Analytics.scala:65 rdd at Aegean_Analytics.scala:65	2024/06/16 19:55:49	4 s	1/1	12/12
14	rdd at Aegean_Analytics.scala:65 rdd at Aegean_Analytics.scala:65	2024/06/16 19:55:48	3 s	1/1	12/12

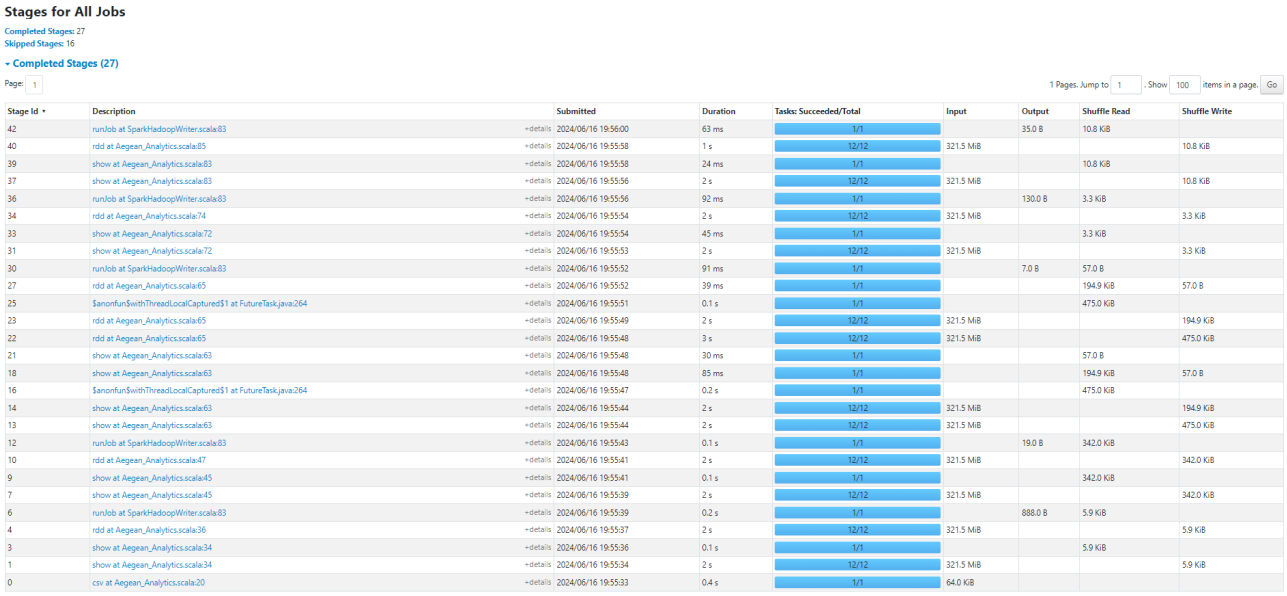
Figure 27: All Physical Jobs as they appear on the History Server(1)

13	show at Aegean_Analytics.scala:63 show at Aegean_Analytics.scala:63	2024/06/16 19:55:48	41 ms	1/1 (2 skipped)	1/1 (13 skipped)
12	show at Aegean_Analytics.scala:63 show at Aegean_Analytics.scala:63	2024/06/16 19:55:48	0.1 s	1/1 (1 skipped)	1/1 (12 skipped)
11	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264 $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2024/06/16 19:55:47	0.8 s	1/1 (1 skipped)	1/1 (12 skipped)
10	show at Aegean_Analytics.scala:63 show at Aegean_Analytics.scala:63	2024/06/16 19:55:44	4 s	1/1	12/12
9	show at Aegean_Analytics.scala:63 show at Aegean_Analytics.scala:63	2024/06/16 19:55:44	2 s	1/1	12/12
8	runJob at SparkHadoopWriter.scala:83 runJob at SparkHadoopWriter.scala:83	2024/06/16 19:55:43	0.2 s	1/1 (1 skipped)	1/1 (12 skipped)
7	rdd at Aegean_Analytics.scala:47 rdd at Aegean_Analytics.scala:47	2024/06/16 19:55:41	2 s	1/1	12/12
6	show at Aegean_Analytics.scala:45 show at Aegean_Analytics.scala:45	2024/06/16 19:55:41	0.1 s	1/1 (1 skipped)	1/1 (12 skipped)
5	show at Aegean_Analytics.scala:45 show at Aegean_Analytics.scala:45	2024/06/16 19:55:39	2 s	1/1	12/12
4	runJob at SparkHadoopWriter.scala:83 runJob at SparkHadoopWriter.scala:83	2024/06/16 19:55:39	0.2 s	1/1 (1 skipped)	1/1 (12 skipped)
3	rdd at Aegean_Analytics.scala:36 rdd at Aegean_Analytics.scala:36	2024/06/16 19:55:37	2 s	1/1	12/12
2	show at Aegean_Analytics.scala:34 show at Aegean_Analytics.scala:34	2024/06/16 19:55:36	0.2 s	1/1 (1 skipped)	1/1 (12 skipped)
1	show at Aegean_Analytics.scala:34 show at Aegean_Analytics.scala:34	2024/06/16 19:55:34	2 s	1/1	12/12
0	csv at Aegean_Analytics.scala:20 csv at Aegean_Analytics.scala:20	2024/06/16 19:55:33	0.6 s	1/1	1/1

Figure 28: All Physical Jobs as they appear on the History Server(2)

We observe that we have 27 physical jobs, whereas we expect only 11 logical jobs. Essentially, we expect one job for each saveAsTextFile operation, one for each show operation, and one for reading the CSV file. Since we performed both the show and saveAsTextFile operations for each of the five questions, plus the initial read operation, we expect a total of 11 logical jobs. However, Spark optimizes the execution by splitting one logical job into multiple physical jobs. For example, in our case, the show operation is split into more than one physical job. Additionally, while RDD operations might not be explicitly recognized as actions, Spark internally processes and optimizes them, leading to the creation of more physical jobs.

STAGES



Stages for All Jobs

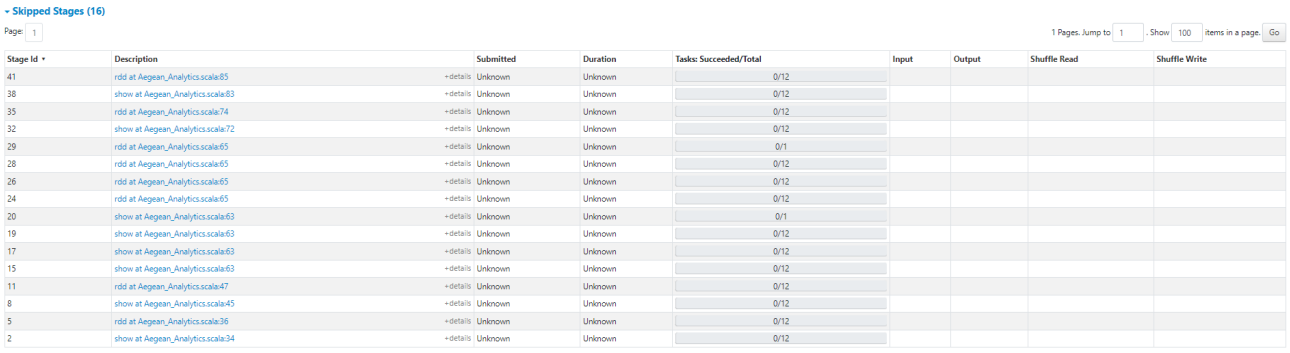
Completed Stages: 27
Skipped Stages: 16

▾ Completed Stages (27)

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

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
42	runJob at SparkHadoopWriter.scala:83 +details	2024/06/16 19:56:00	63 ms	1/1		35.0 B	10.8 KiB	
40	rdd at Aegean_Analytics.scala:85 +details	2024/06/16 19:55:58	1 s	12/12	321.5 MiB			10.8 KiB
39	show at Aegean_Analytics.scala:83 +details	2024/06/16 19:55:58	24 ms	1/1			10.8 KiB	
37	show at Aegean_Analytics.scala:83 +details	2024/06/16 19:55:56	2 s	12/12	321.5 MiB			10.8 KiB
36	runJob at SparkHadoopWriter.scala:83 +details	2024/06/16 19:55:56	92 ms	1/1		130.0 B	3.3 KiB	
34	rdd at Aegean_Analytics.scala:74 +details	2024/06/16 19:55:54	2 s	12/12	321.5 MiB			3.3 KiB
33	show at Aegean_Analytics.scala:72 +details	2024/06/16 19:55:54	45 ms	1/1			3.3 KiB	
31	show at Aegean_Analytics.scala:72 +details	2024/06/16 19:55:53	2 s	12/12	321.5 MiB			3.3 KiB
30	runJob at SparkHadoopWriter.scala:83 +details	2024/06/16 19:55:52	91 ms	1/1		7.0 B	57.0 B	
27	rdd at Aegean_Analytics.scala:65 +details	2024/06/16 19:55:52	39 ms	1/1			194.9 KiB	57.0 B
25	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264 +details	2024/06/16 19:55:51	0.1 s	1/1			475.0 KiB	
23	rdd at Aegean_Analytics.scala:65 +details	2024/06/16 19:55:49	2 s	12/12	321.5 MiB			194.9 KiB
22	rdd at Aegean_Analytics.scala:65 +details	2024/06/16 19:55:48	3 s	12/12	321.5 MiB			475.0 KiB
21	show at Aegean_Analytics.scala:63 +details	2024/06/16 19:55:48	30 ms	1/1			57.0 B	
18	show at Aegean_Analytics.scala:63 +details	2024/06/16 19:55:48	85 ms	1/1			194.9 KiB	57.0 B
16	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264 +details	2024/06/16 19:55:47	0.2 s	1/1			475.0 KiB	
14	show at Aegean_Analytics.scala:63 +details	2024/06/16 19:55:44	2 s	12/12	321.5 MiB			194.9 KiB
13	show at Aegean_Analytics.scala:63 +details	2024/06/16 19:55:44	2 s	12/12	321.5 MiB			475.0 KiB
12	runJob at SparkHadoopWriter.scala:83 +details	2024/06/16 19:55:43	0.1 s	1/1		19.0 B	342.0 KiB	
10	rdd at Aegean_Analytics.scala:47 +details	2024/06/16 19:55:41	2 s	12/12	321.5 MiB			342.0 KiB
9	show at Aegean_Analytics.scala:45 +details	2024/06/16 19:55:41	0.1 s	1/1			342.0 KiB	
7	show at Aegean_Analytics.scala:45 +details	2024/06/16 19:55:39	2 s	12/12	321.5 MiB			342.0 KiB
6	runJob at SparkHadoopWriter.scala:83 +details	2024/06/16 19:55:39	0.2 s	1/1		888.0 B	5.9 KiB	
4	rdd at Aegean_Analytics.scala:36 +details	2024/06/16 19:55:37	2 s	12/12	321.5 MiB			5.9 KiB
3	show at Aegean_Analytics.scala:34 +details	2024/06/16 19:55:36	0.1 s	1/1			5.9 KiB	
1	show at Aegean_Analytics.scala:34 +details	2024/06/16 19:55:34	2 s	12/12	321.5 MiB			5.9 KiB
0	csv at Aegean_Analytics.scala:20 +details	2024/06/16 19:55:33	0.4 s	1/1	64.0 KiB			

Figure 29: All completed Stages as they appear on the History Server



▾ Skipped Stages (16)

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

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
41	rdd at Aegean_Analytics.scala:85 +details	Unknown	Unknown	0/12				
38	show at Aegean_Analytics.scala:83 +details	Unknown	Unknown	0/12				
35	rdd at Aegean_Analytics.scala:74 +details	Unknown	Unknown	0/12				
32	show at Aegean_Analytics.scala:72 +details	Unknown	Unknown	0/12				
29	rdd at Aegean_Analytics.scala:65 +details	Unknown	Unknown	0/1				
28	rdd at Aegean_Analytics.scala:65 +details	Unknown	Unknown	0/12				
26	rdd at Aegean_Analytics.scala:65 +details	Unknown	Unknown	0/12				
24	rdd at Aegean_Analytics.scala:65 +details	Unknown	Unknown	0/12				
20	show at Aegean_Analytics.scala:63 +details	Unknown	Unknown	0/1				
19	show at Aegean_Analytics.scala:63 +details	Unknown	Unknown	0/12				
17	show at Aegean_Analytics.scala:63 +details	Unknown	Unknown	0/12				
15	show at Aegean_Analytics.scala:63 +details	Unknown	Unknown	0/12				
11	rdd at Aegean_Analytics.scala:47 +details	Unknown	Unknown	0/12				
8	show at Aegean_Analytics.scala:45 +details	Unknown	Unknown	0/12				
5	rdd at Aegean_Analytics.scala:36 +details	Unknown	Unknown	0/12				
2	show at Aegean_Analytics.scala:34 +details	Unknown	Unknown	0/12				

Figure 30: All skipped Stages as they appear on the History Server

One job is associated with one or more stages. However, as we noticed in the screenshot above, some stages are skipped due to optimization reasons(due to the existence of Catalyst).

The first logical job in our code is reading the csv file.

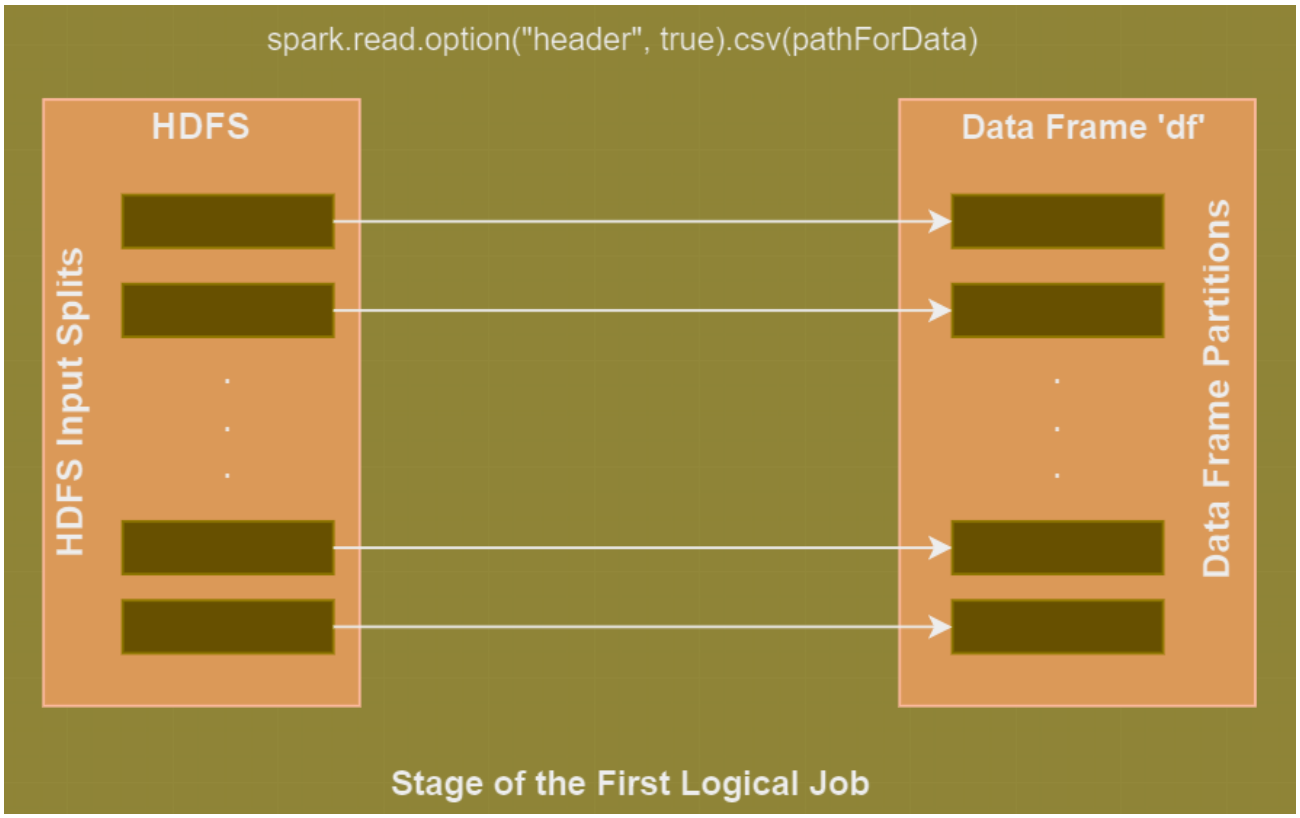


Figure 31: Stage Of Logical Job 1

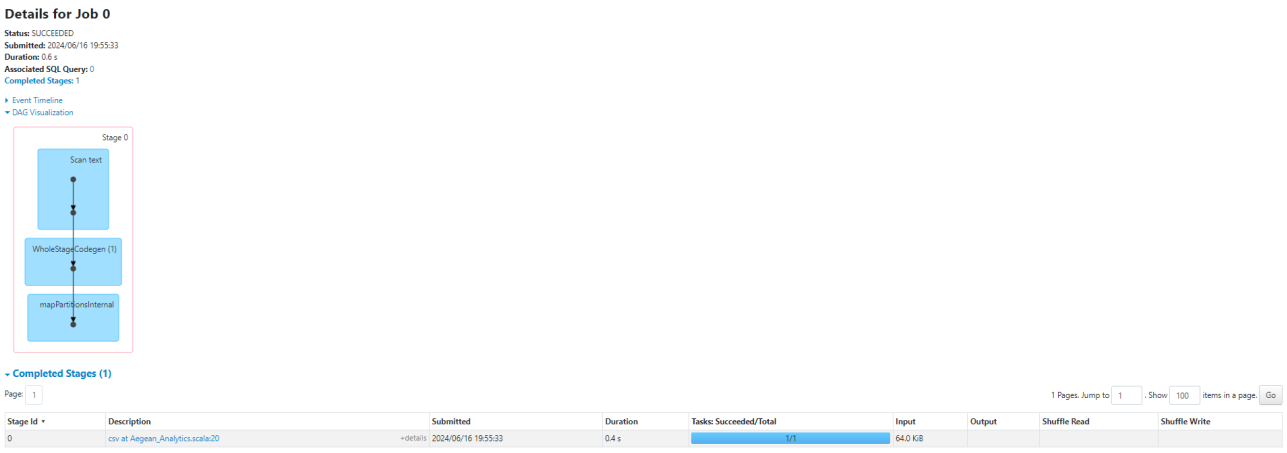


Figure 32: Physical Job 0

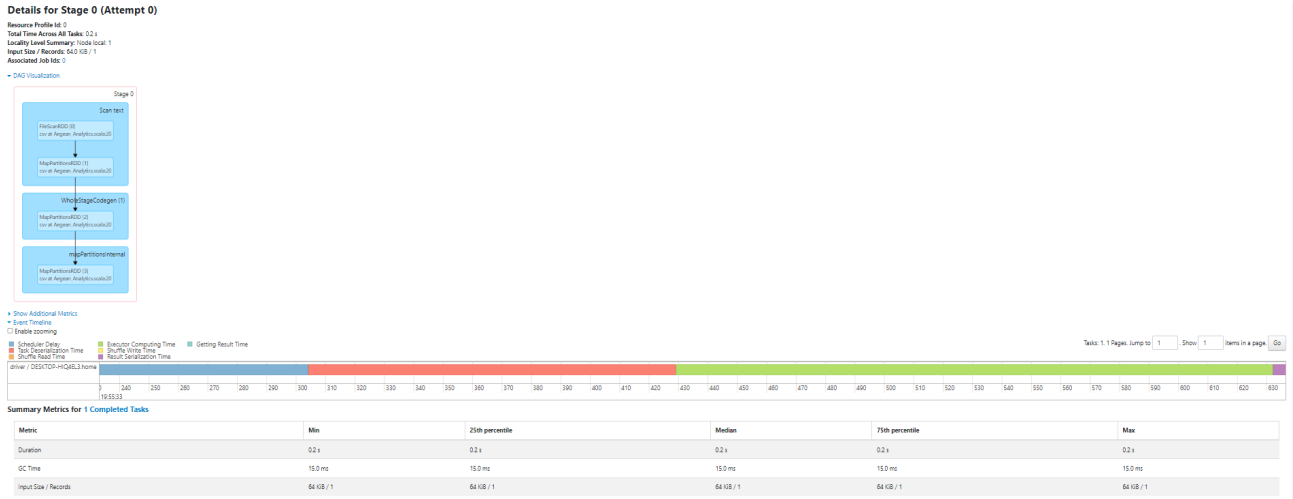


Figure 33: Stage 0 of Physical Job 0

After that for all operations we do after we use the Data Frame 'sanitizedDf', which is calculated once and serves as input for each of the ten jobs.

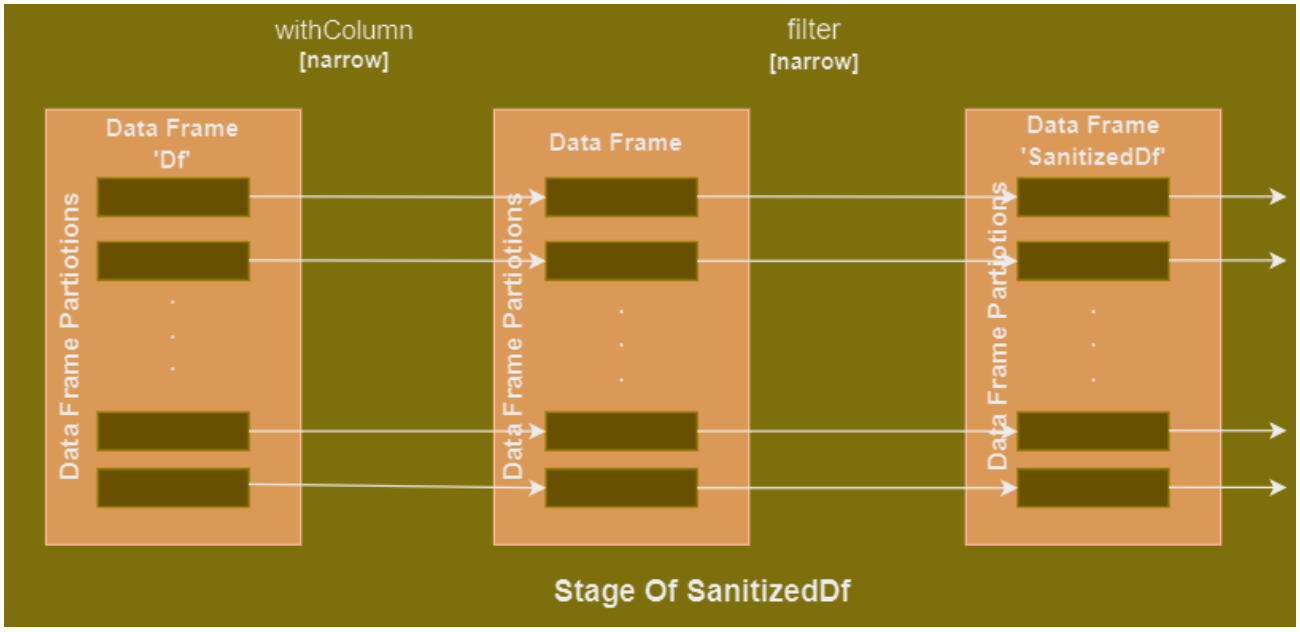


Figure 34: Stage Of Sanitized Df

- **Question 1**

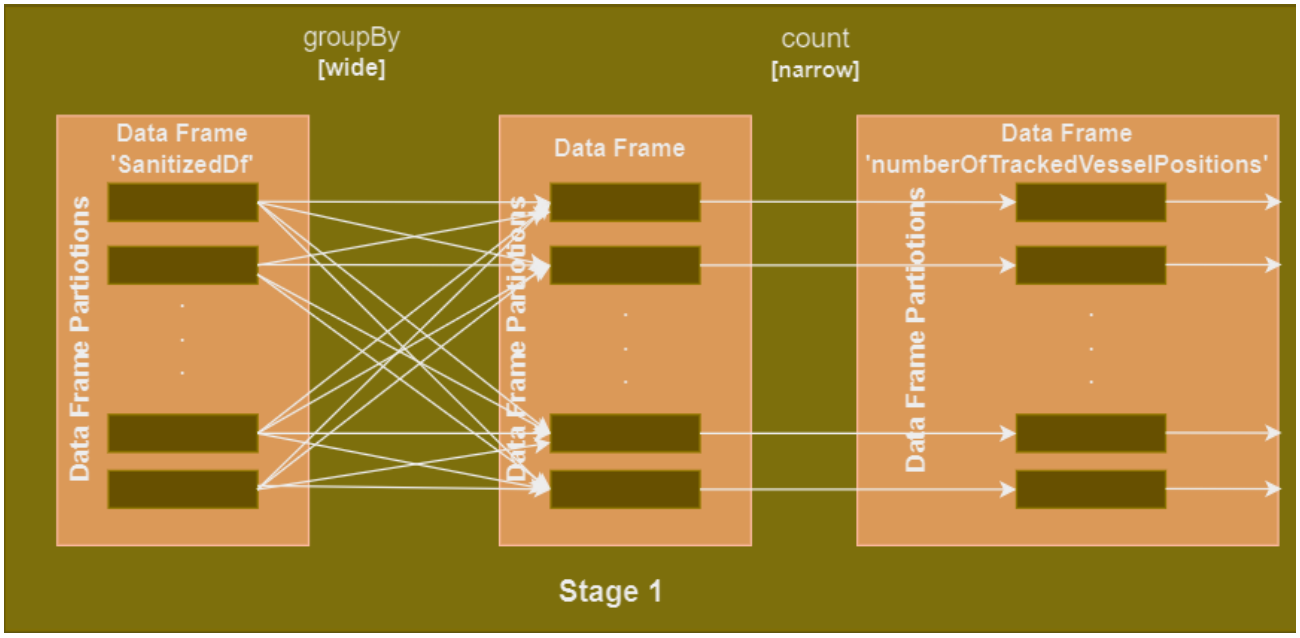


Figure 35: Stage 1

The groupBy operation is a wide transformation because it involves shuffling data across partitions to group records by day and station. On the other hand, the count operation, which follows the groupBy, is a narrow transformation. Since the data is already co-partioned, the count occurs within each partition without needing to shuffle the data again.

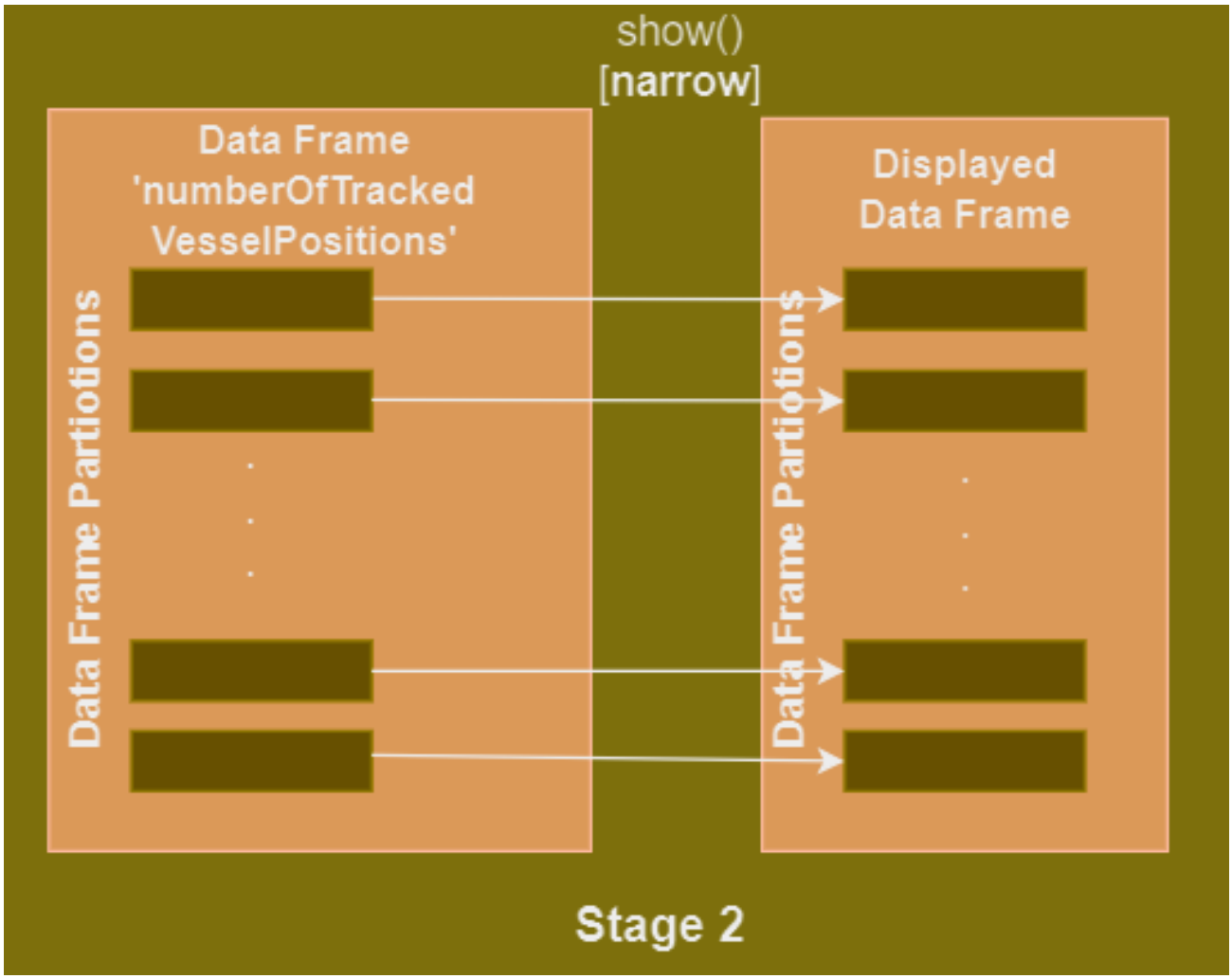


Figure 36: Stage of Logical Job 2

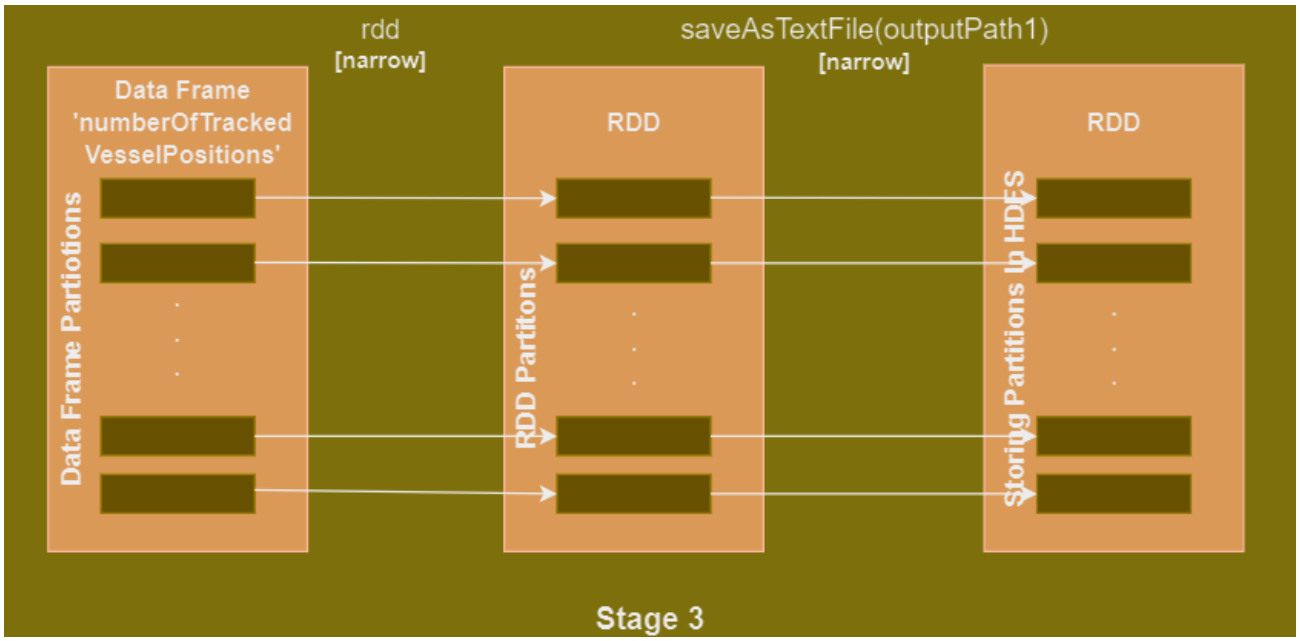


Figure 37: Stage of Logical Job 3

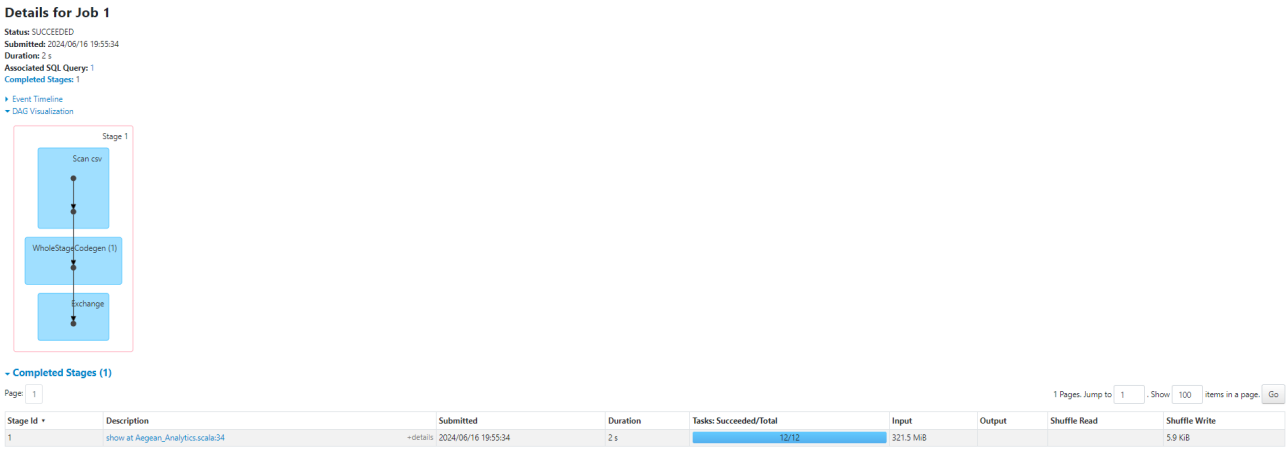


Figure 38: Physical Job 1

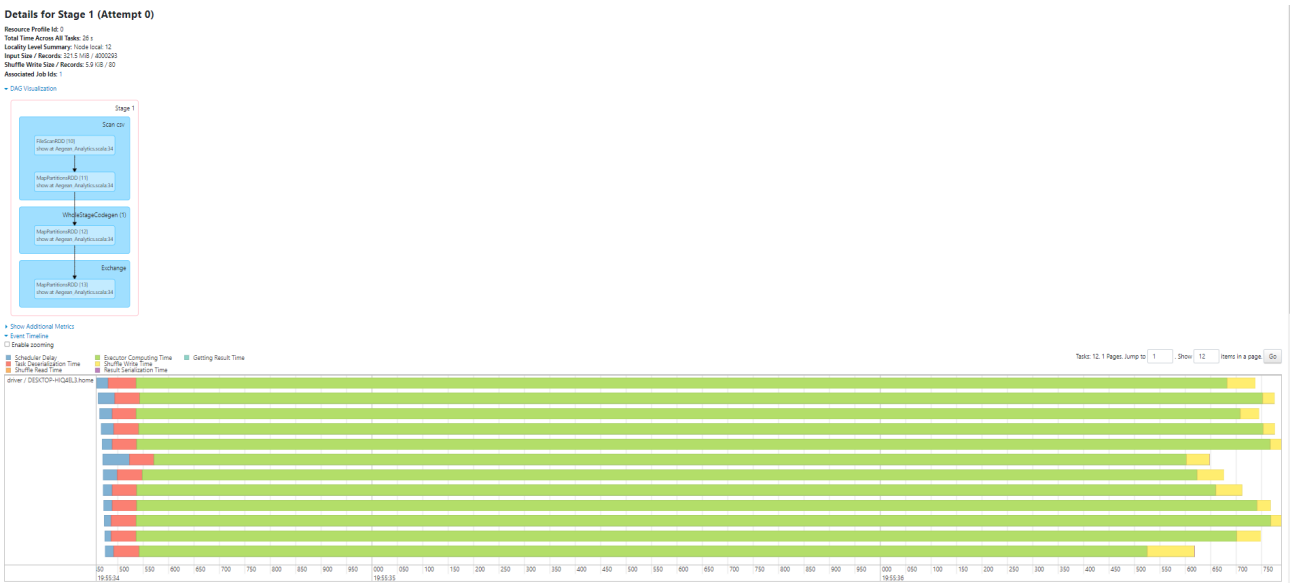


Figure 39: Stage 1 of Physical Job 1

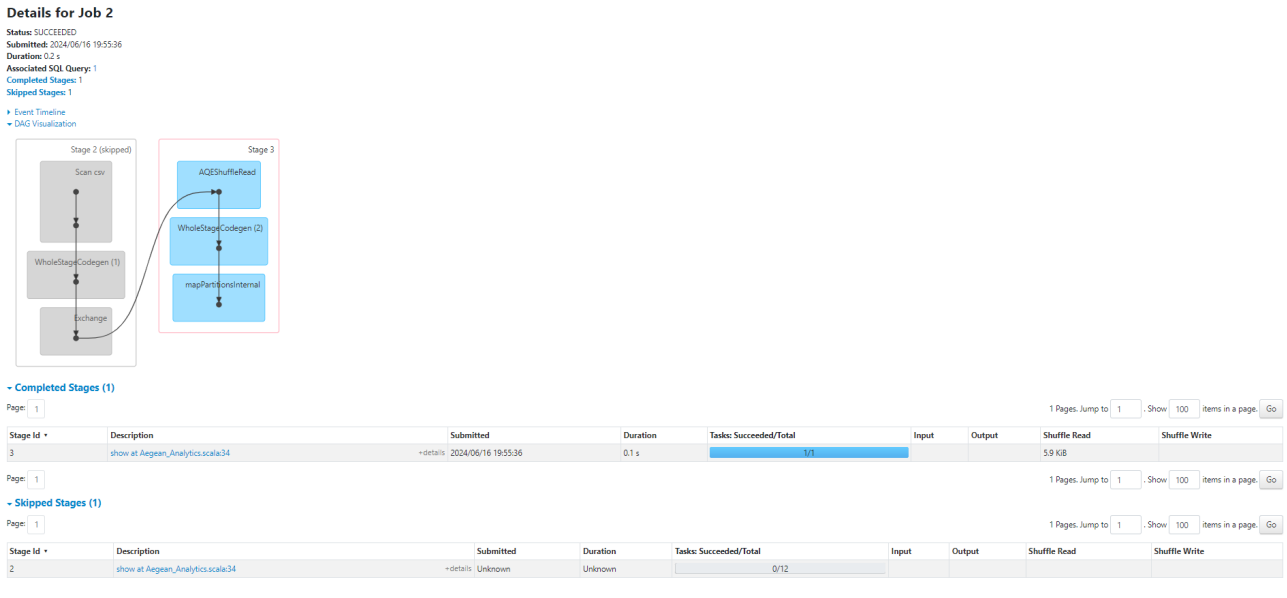


Figure 40: Physical Job 2

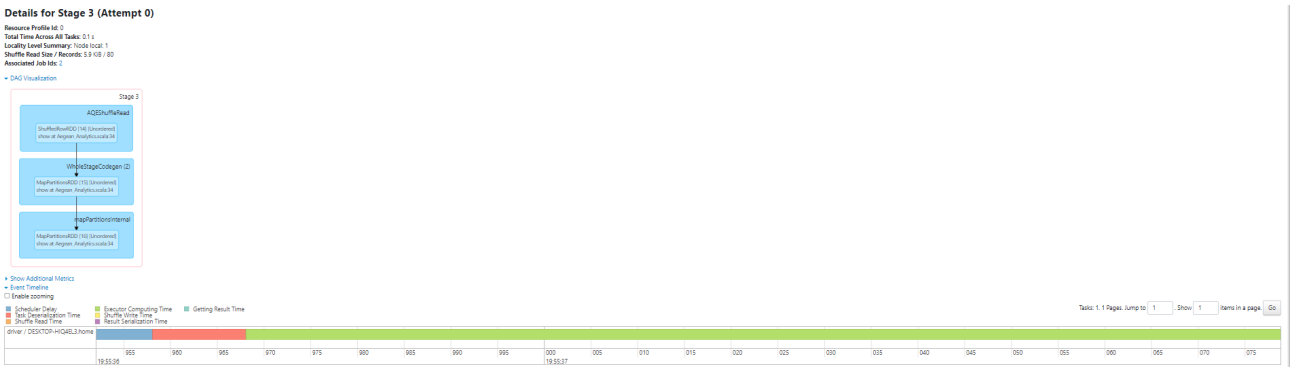


Figure 41: Stage 3 of Physical Job 2

Physical Job 1 in Spark is associated with Stage 1, while Physical Job 2 is associated with stages 2 and 3. However, Stage 2 is skipped. Both of these Jobs are related to the execution of the show method as outlined in question 1.

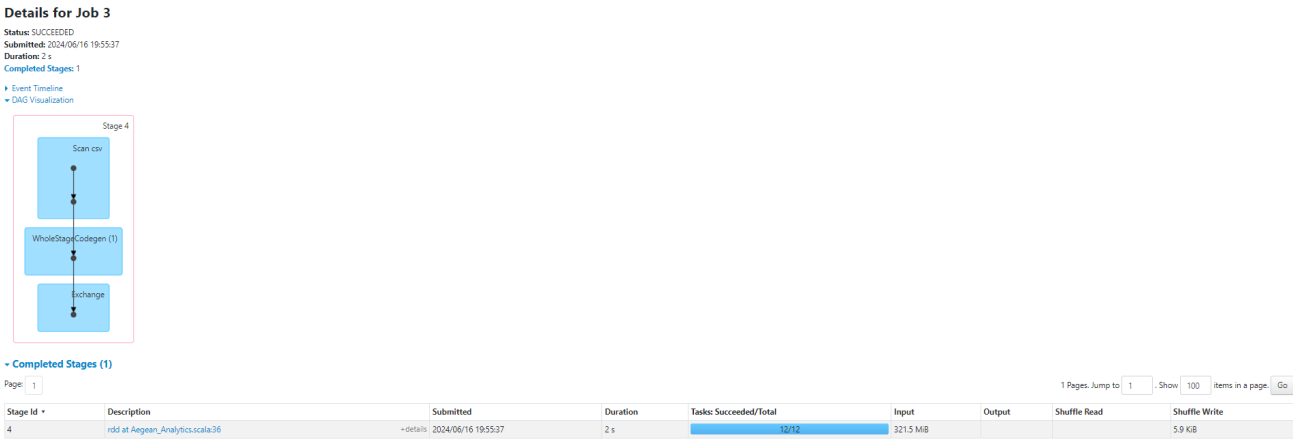


Figure 42: Physical Job 3

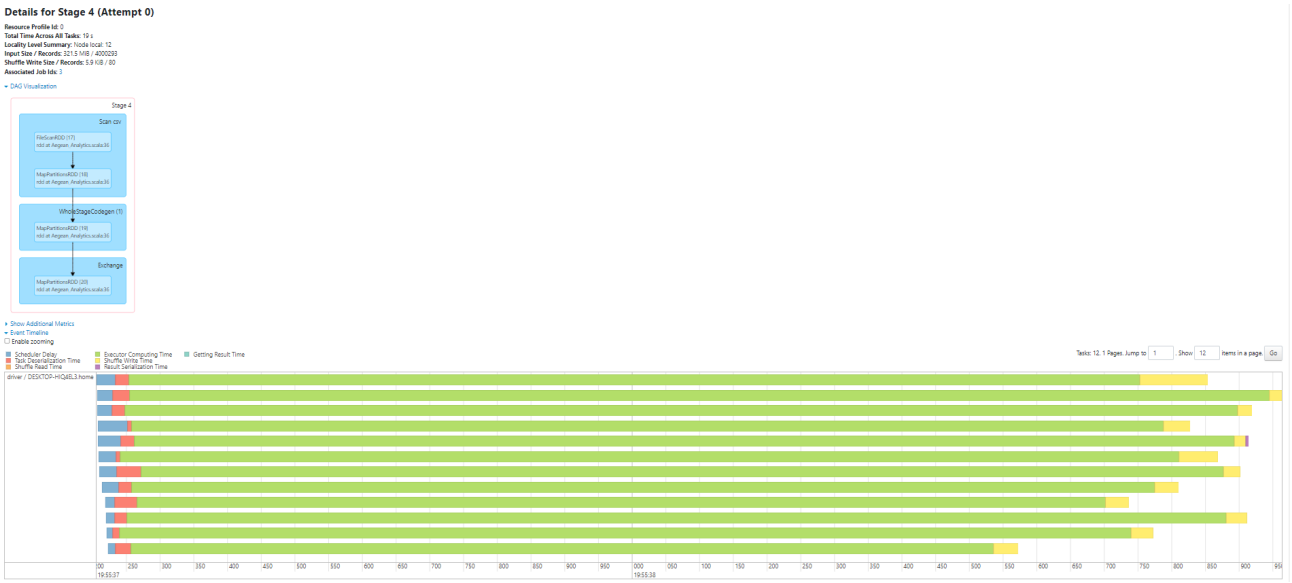


Figure 43: Stage 4 of Physical Job 3

Physical Job 3 is associated with Stage 4. This Job is about transforming the result DataFrame of Question 1 into RDD and saving it.

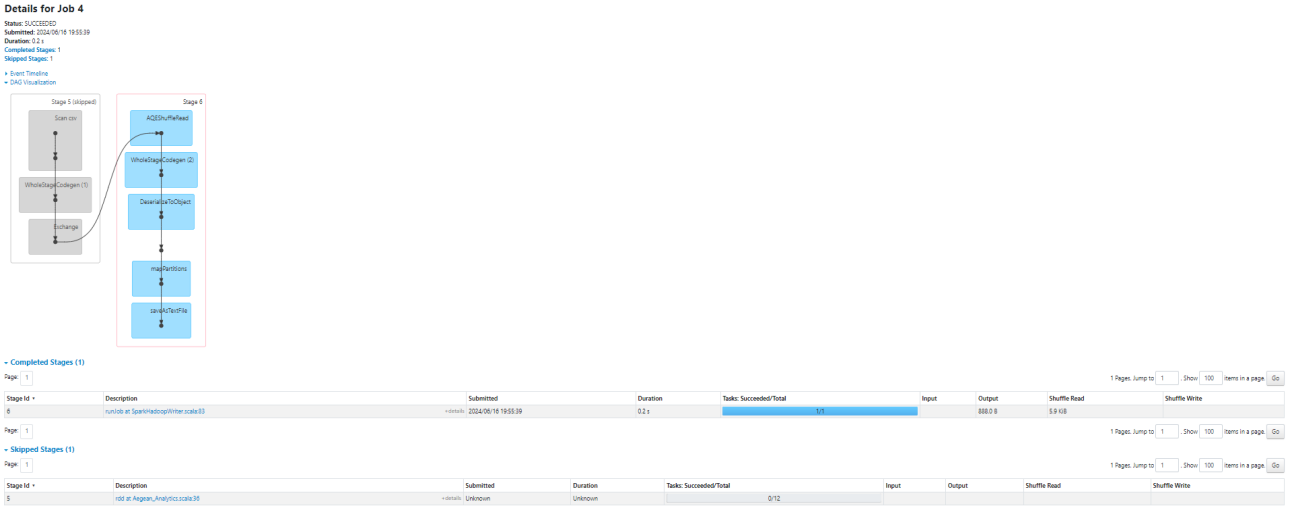


Figure 44: Physical Job 4

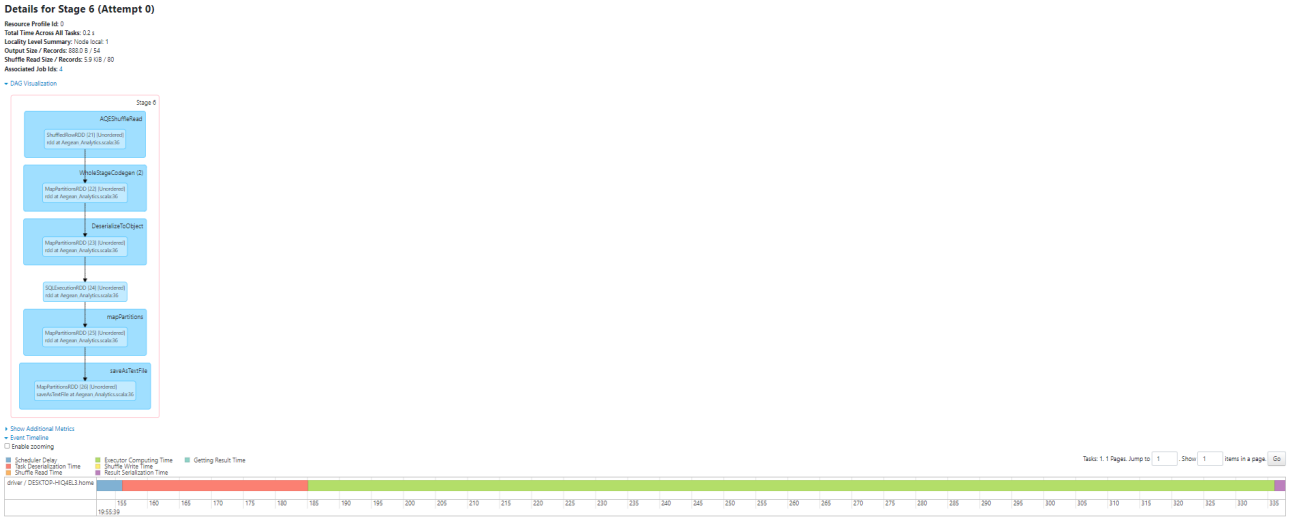


Figure 45: Stage 6 of Physical Job 4

Physical Job 4 is associated with Stage 6 and Stage 5 that it is skipped. Job 4 is about writing our result to an output file.

- **Question 2**

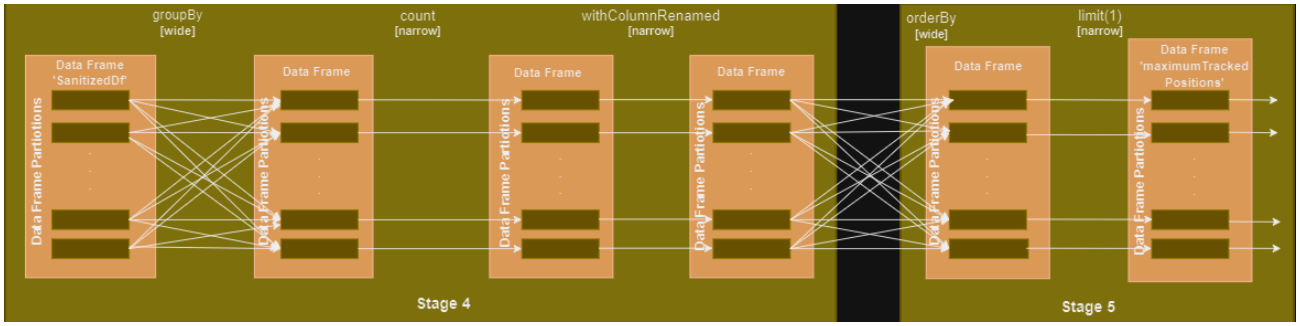


Figure 46: Stages 4 and 5

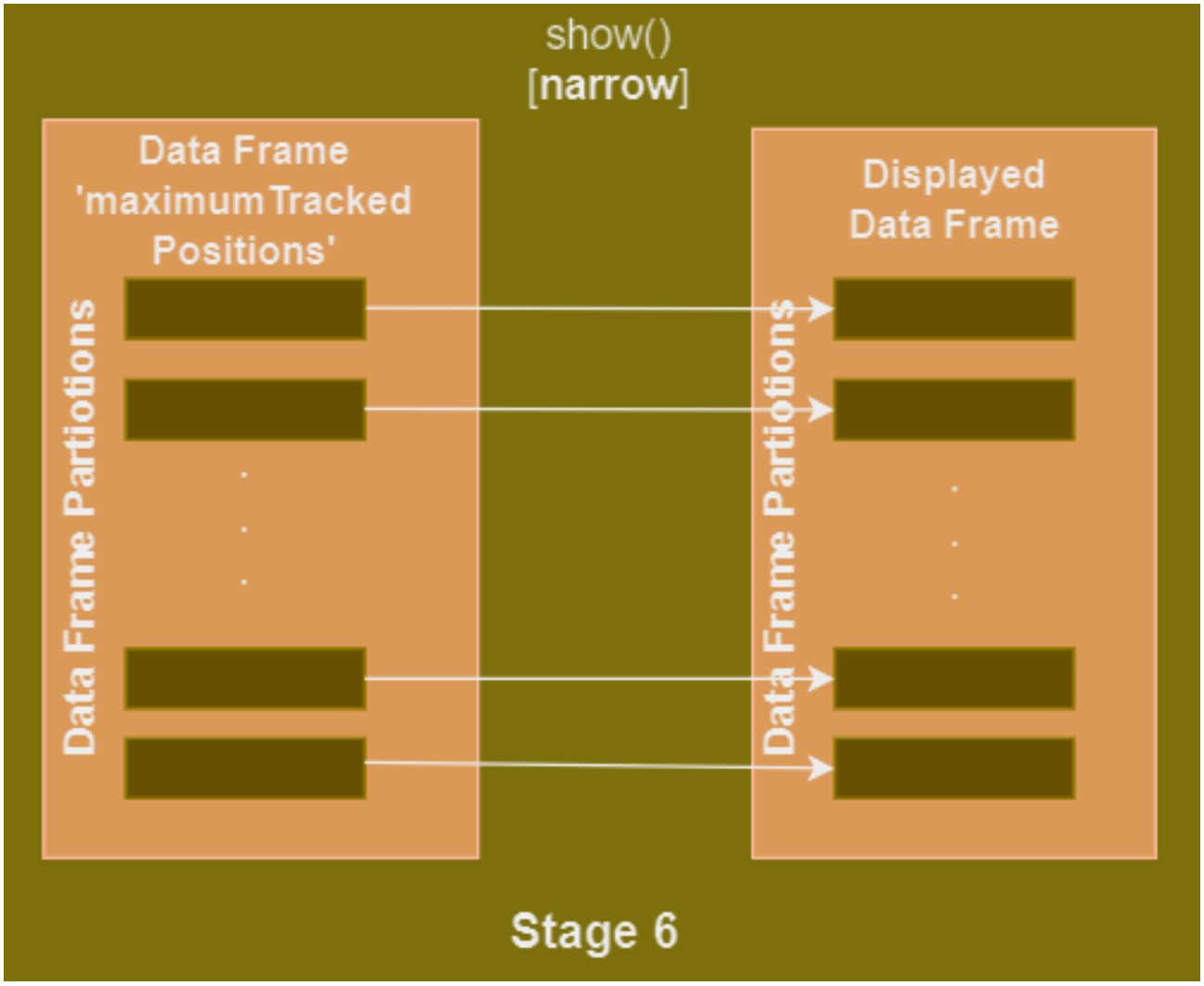


Figure 47: Stage 6 of Logical Job 4

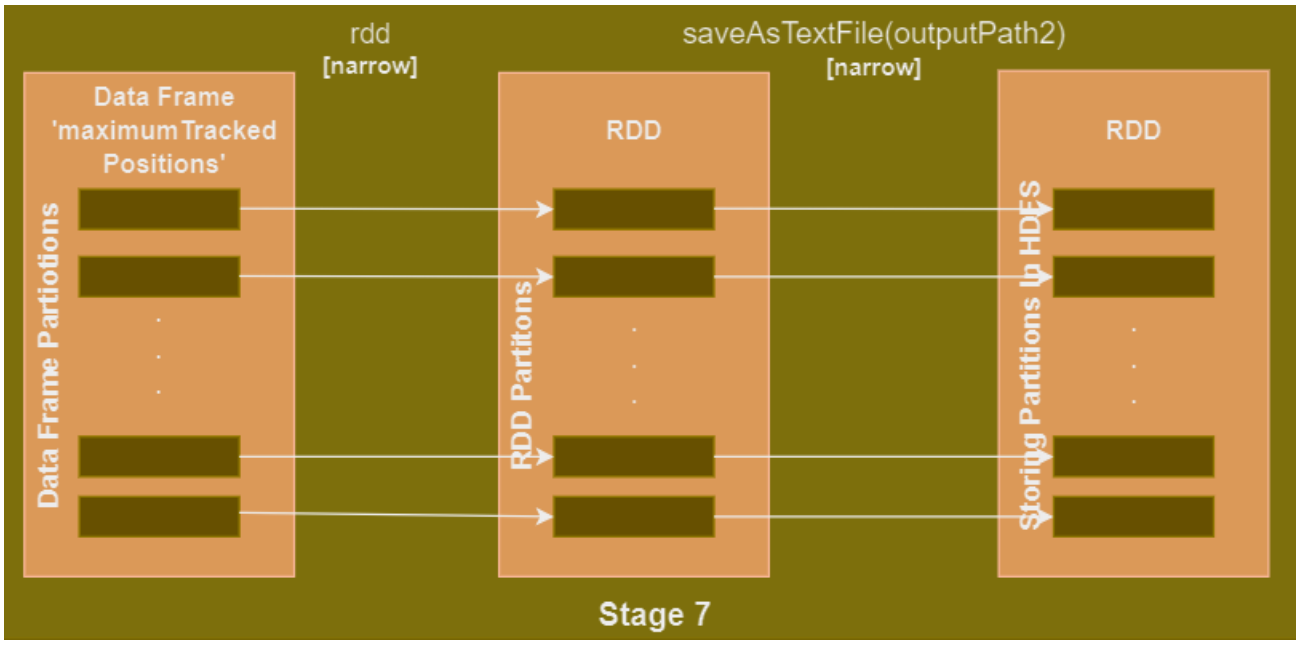


Figure 48: Stage 7 of Logical Job 5

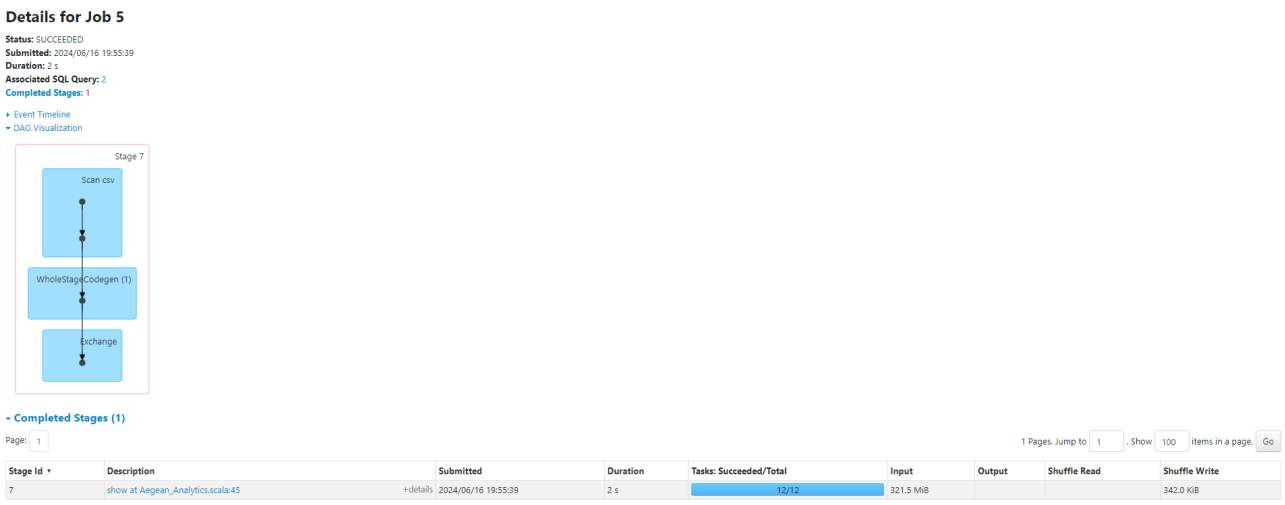


Figure 49: Physical Job 5

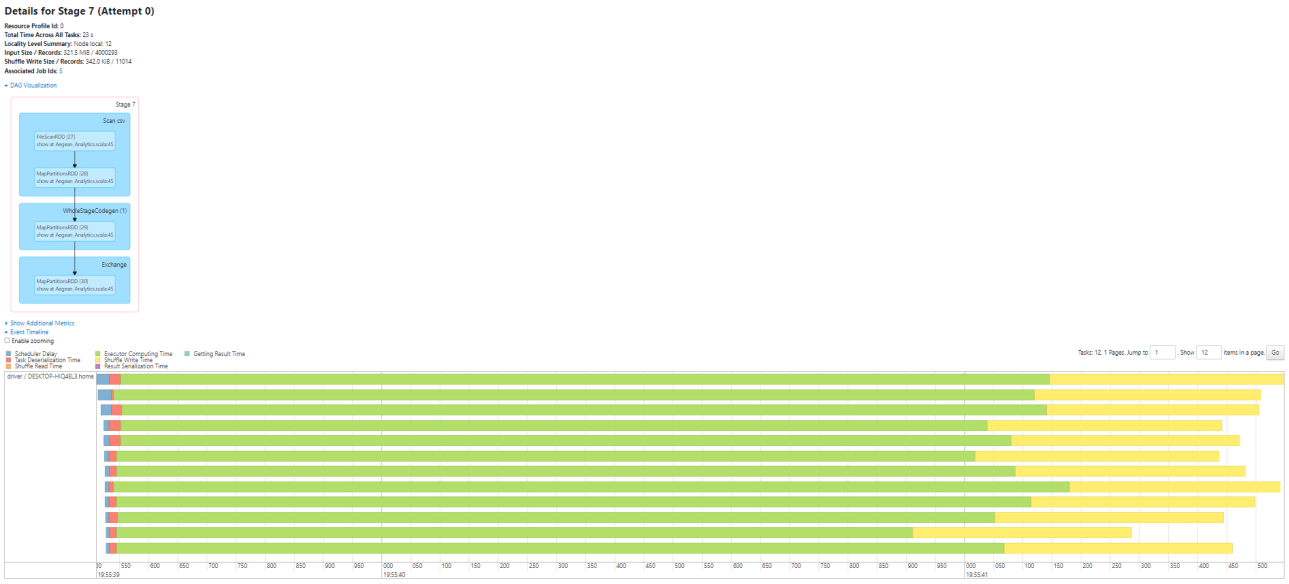


Figure 50: Stage 7 of Physical Job 5

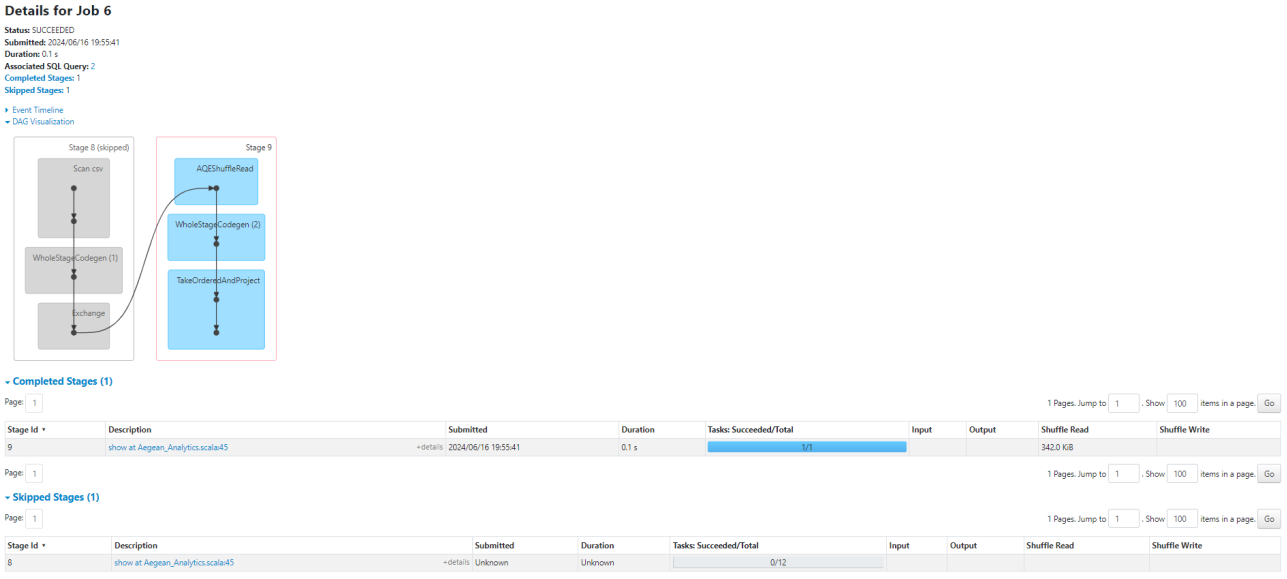


Figure 51: Physical Job 6

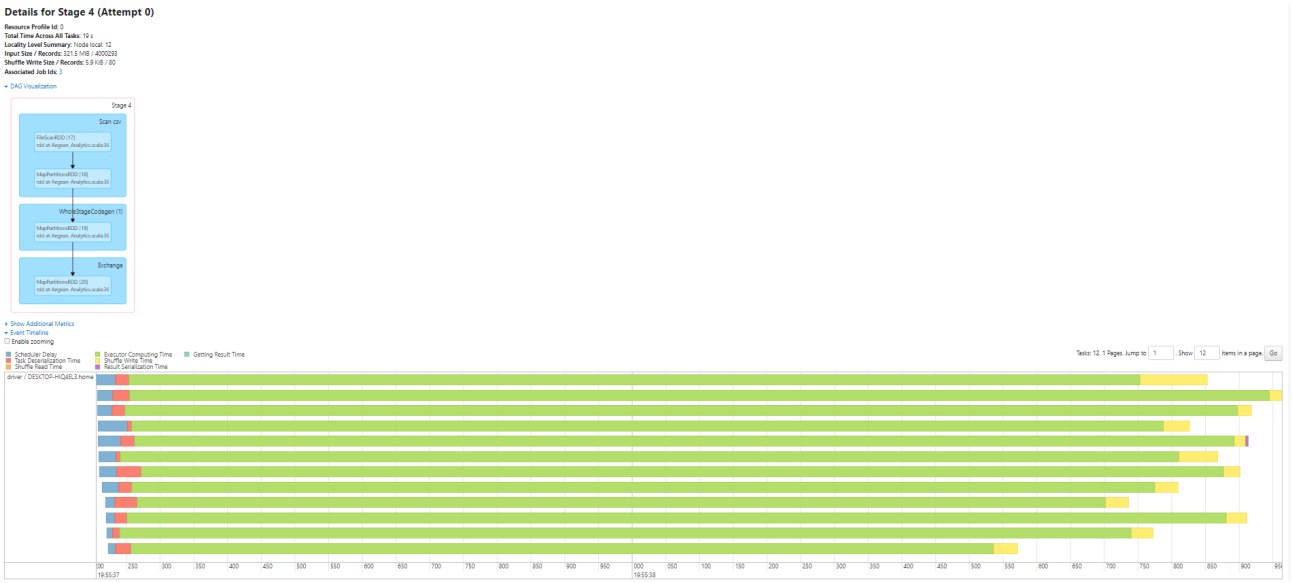


Figure 52: Stage 9 of Physical Job 6

Physical Job 5 is associated with Stage 7 , while physical Job 6 is associated with Stage 8 that it is skipped and Stage 9. Both of these jobs are related to the execution of the show method as outlined in question 2.

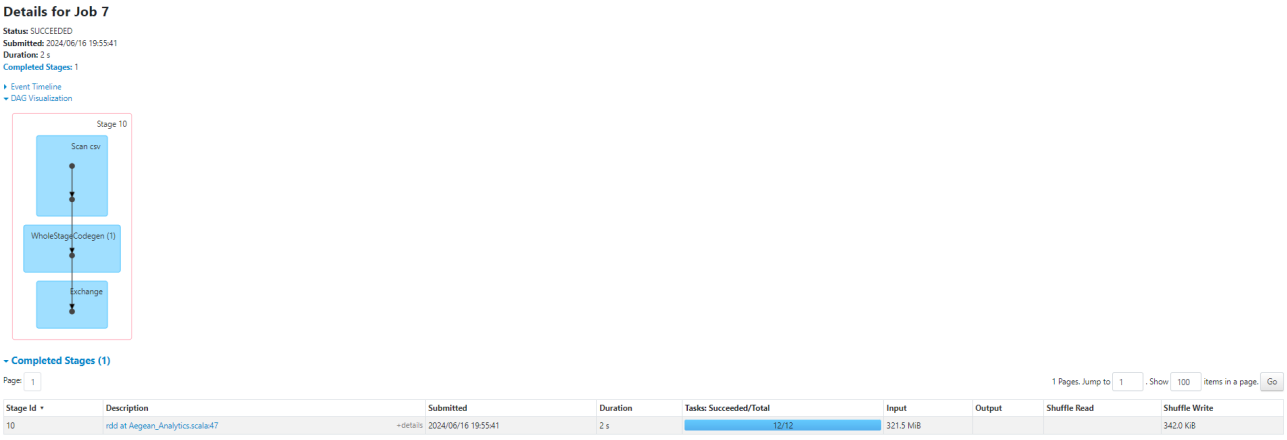


Figure 53: Physical Job 7

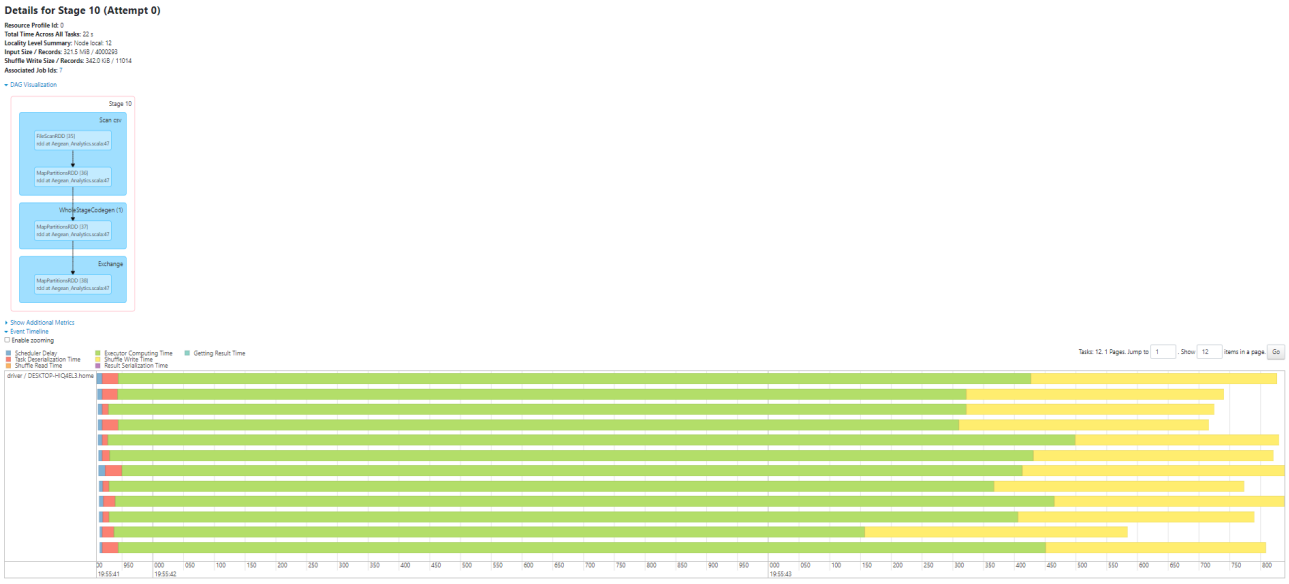


Figure 54: Stage 10 of Physical Job 7

Physical Job 7 is associated with stage 10. This Job is about transforming the result DataFrame of Question 2 to RDD and saving it.

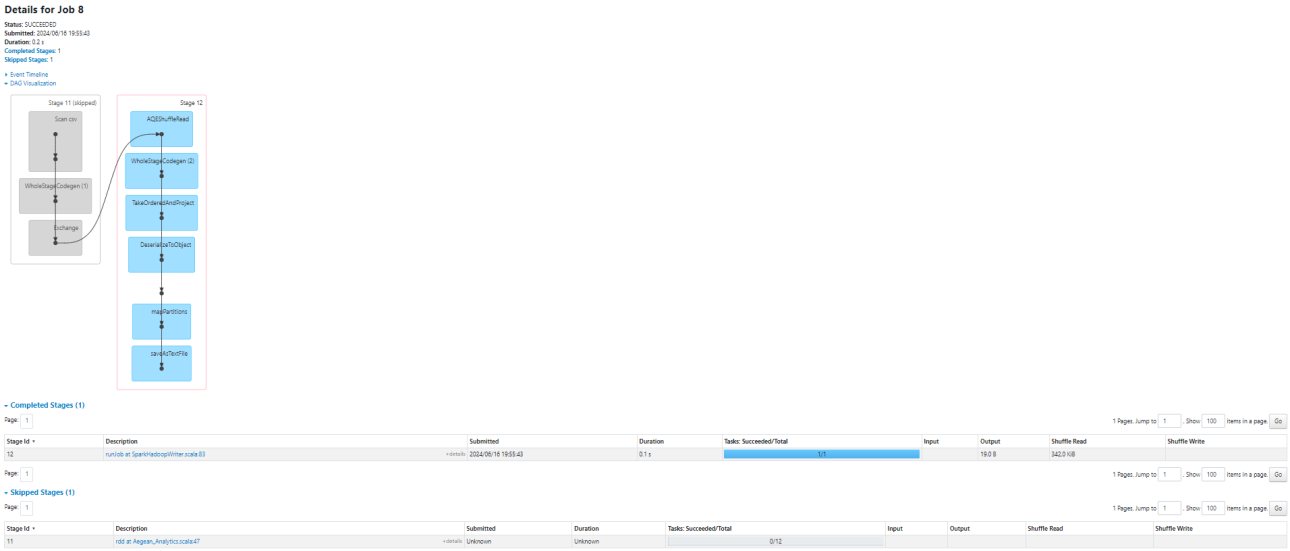


Figure 55: Physical Job 8

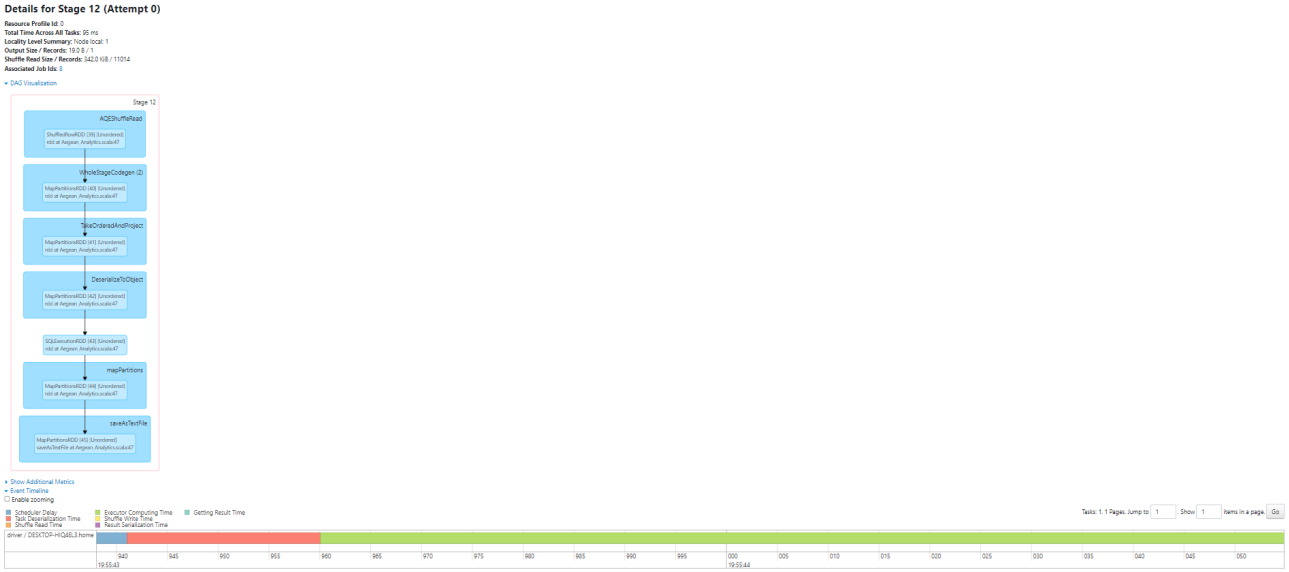


Figure 56: Stage 12 of Physical Job 8

Physical Job 8 is associated with Stage 11 that is skipped and with stage 12. This Job is about writing our result to an output file.

- **Question 3**

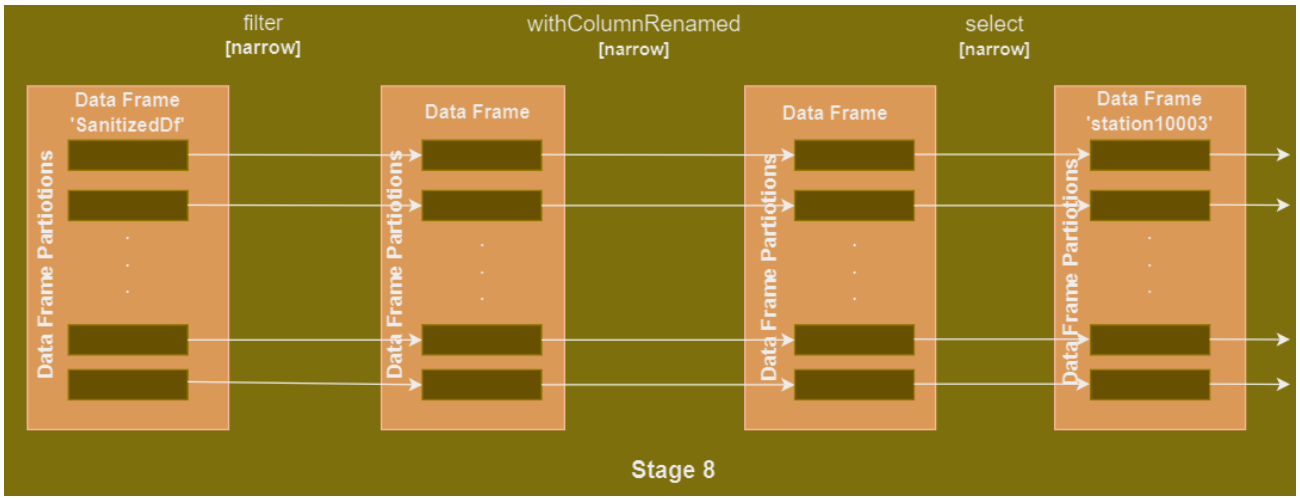


Figure 57: Stage 8

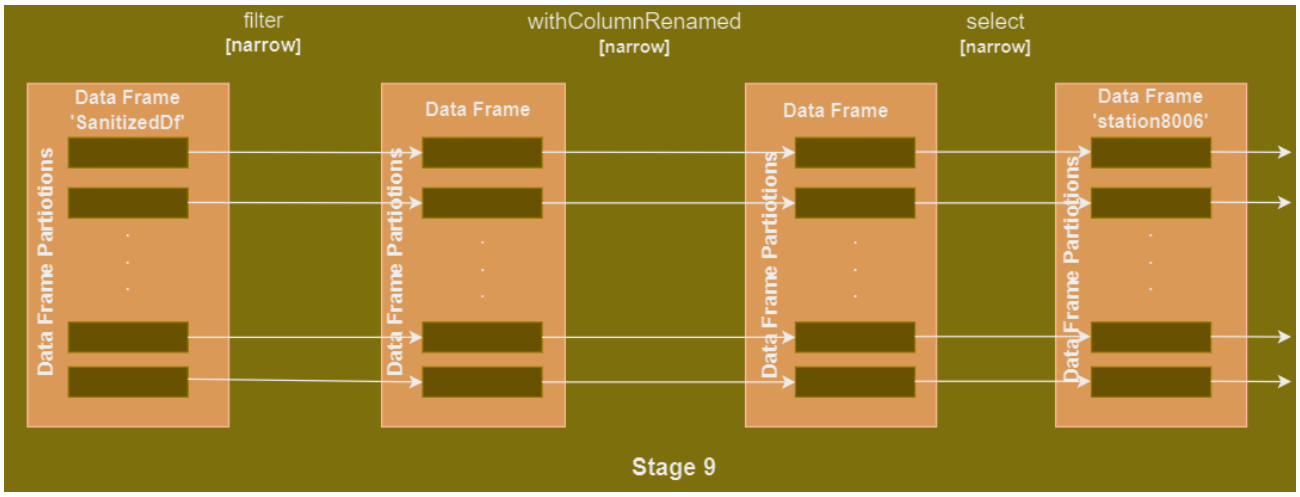


Figure 58: Stage 9

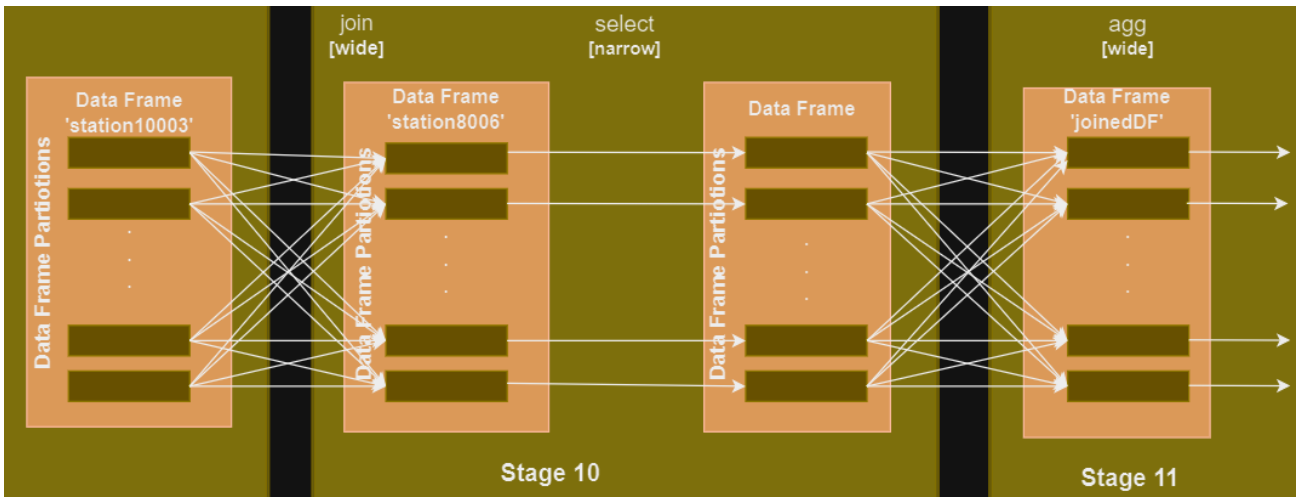


Figure 59: Stage 10 and 11

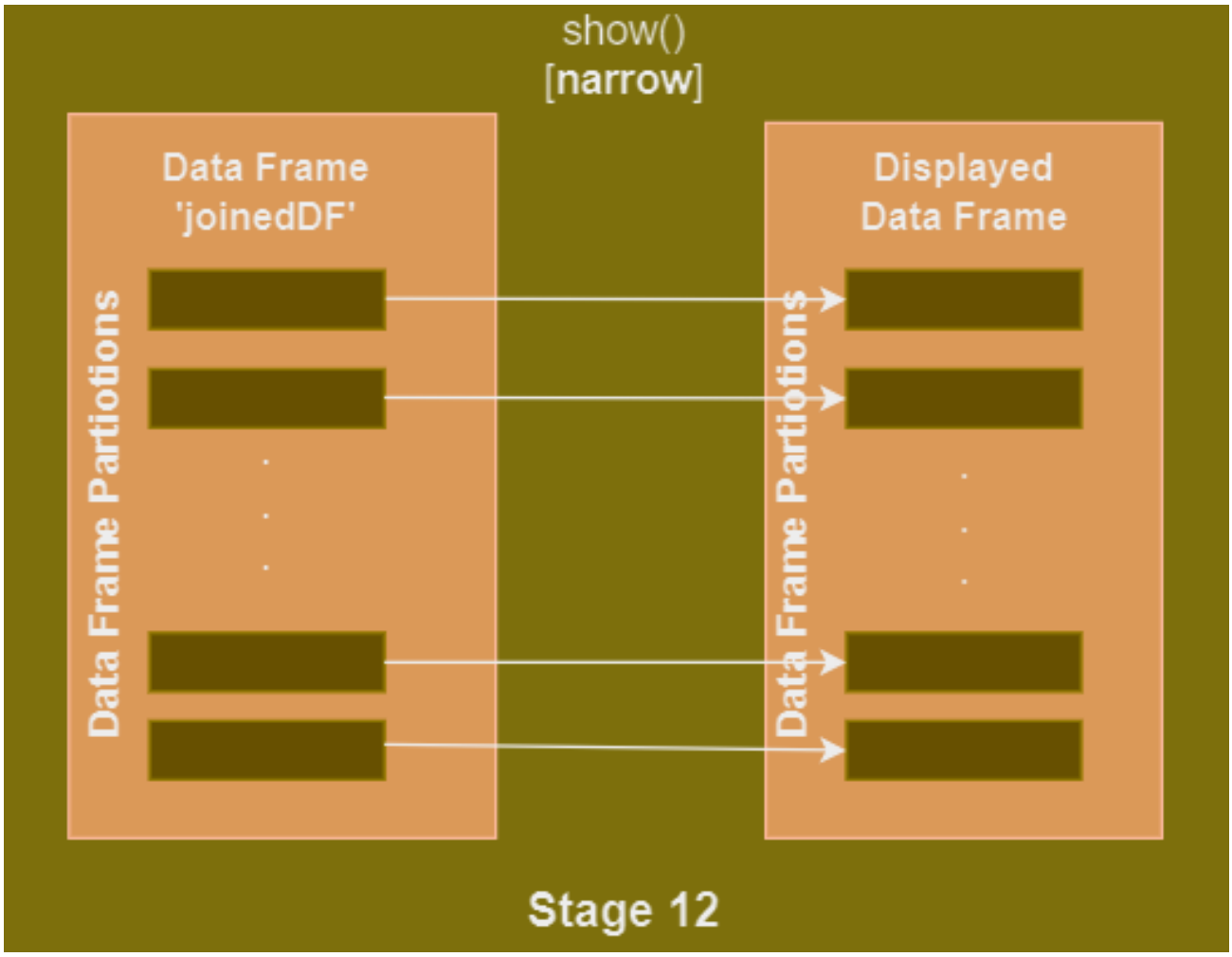


Figure 60: Stage 12 Of Logical Job 6

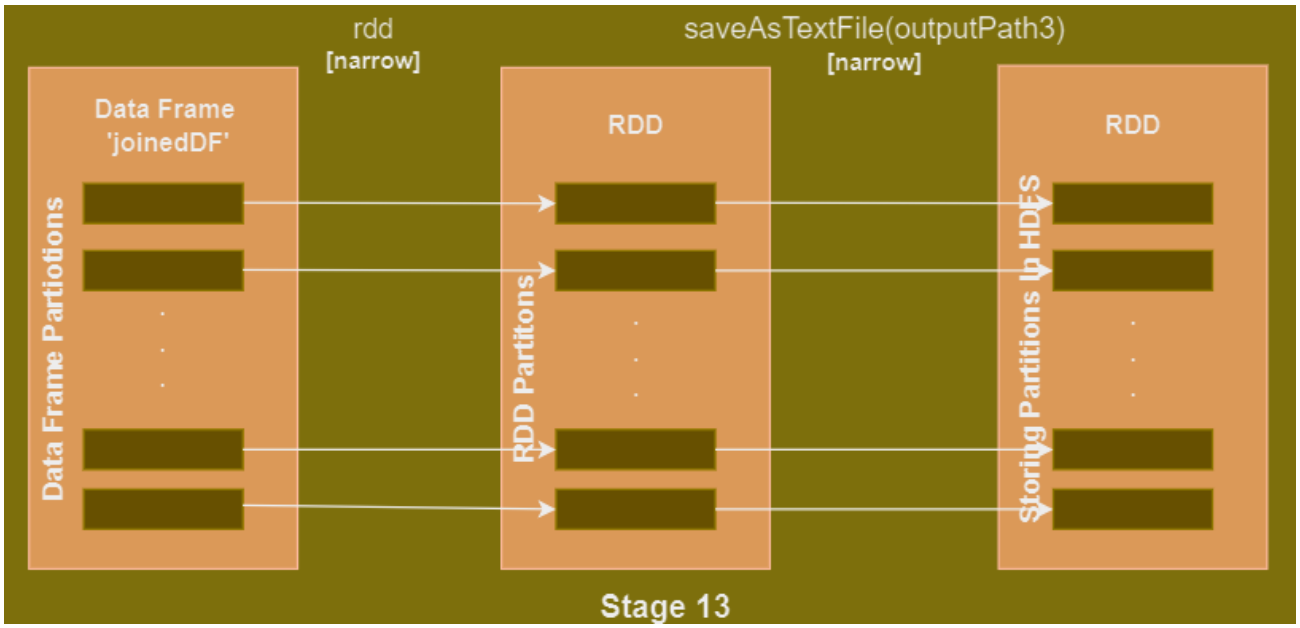


Figure 61: Stage 13 Of Logical Job 7

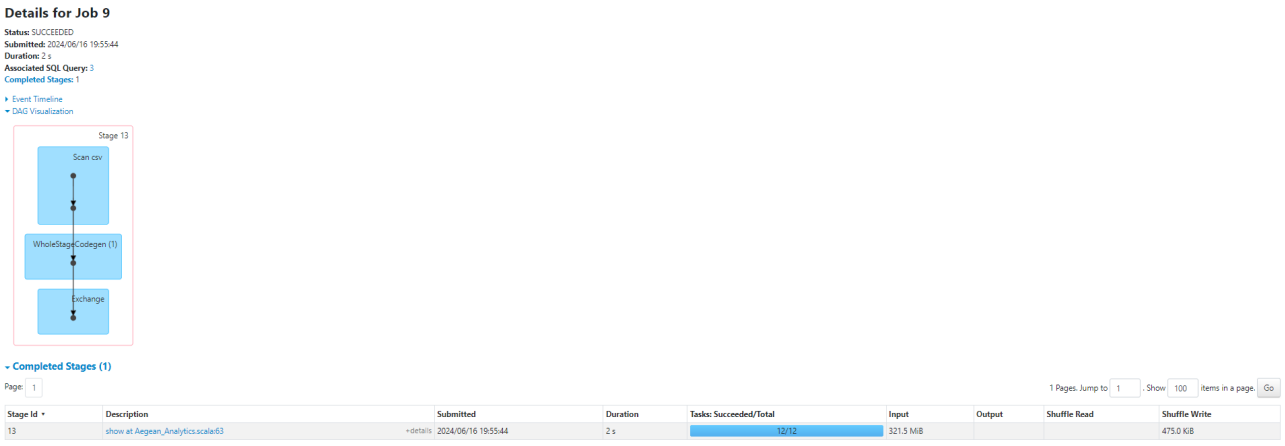


Figure 62: Physical Job 9

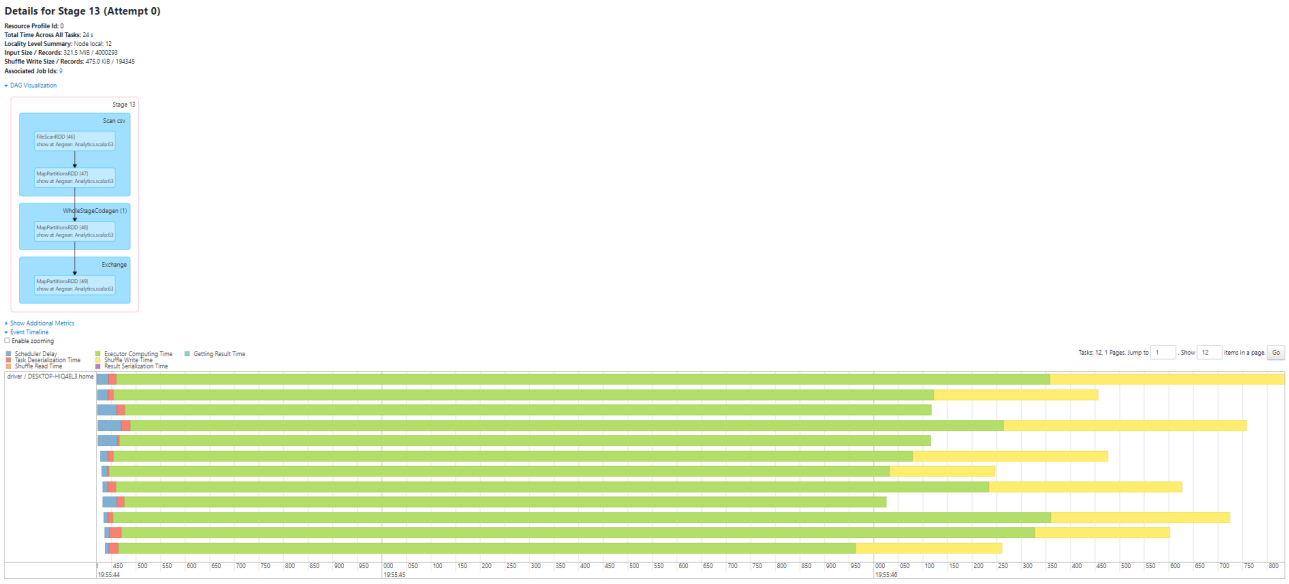


Figure 63: Stage 13 of Physical Job 9

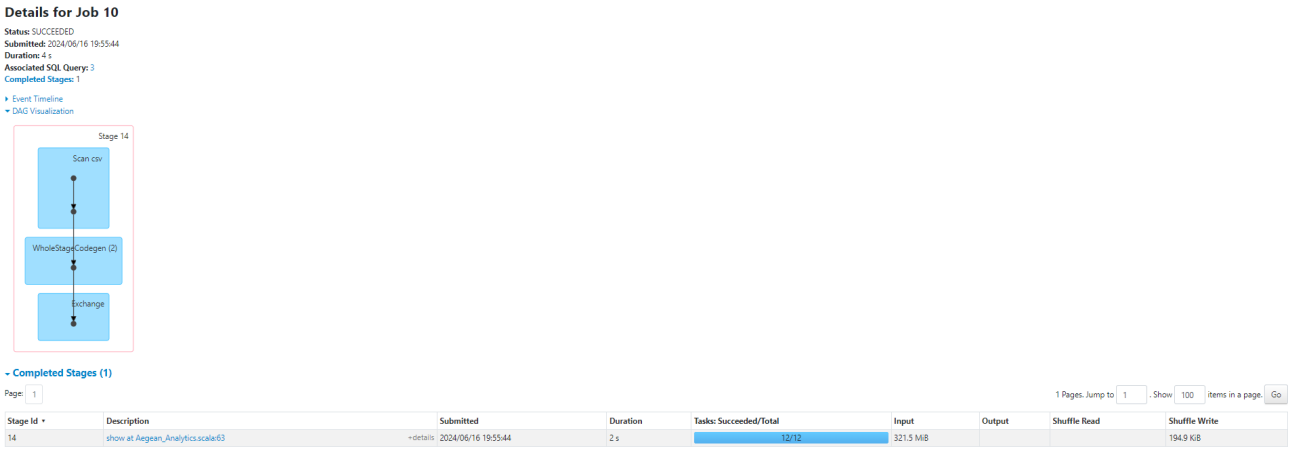


Figure 64: Physical Job 10

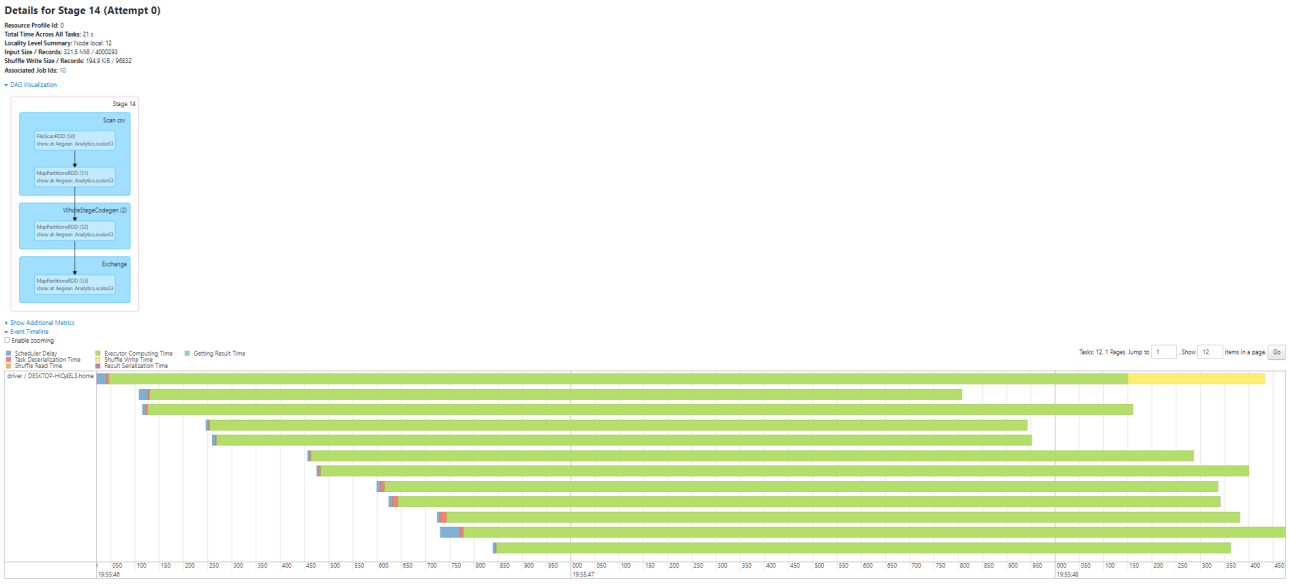


Figure 65: Stage 14 of Physical Job 10

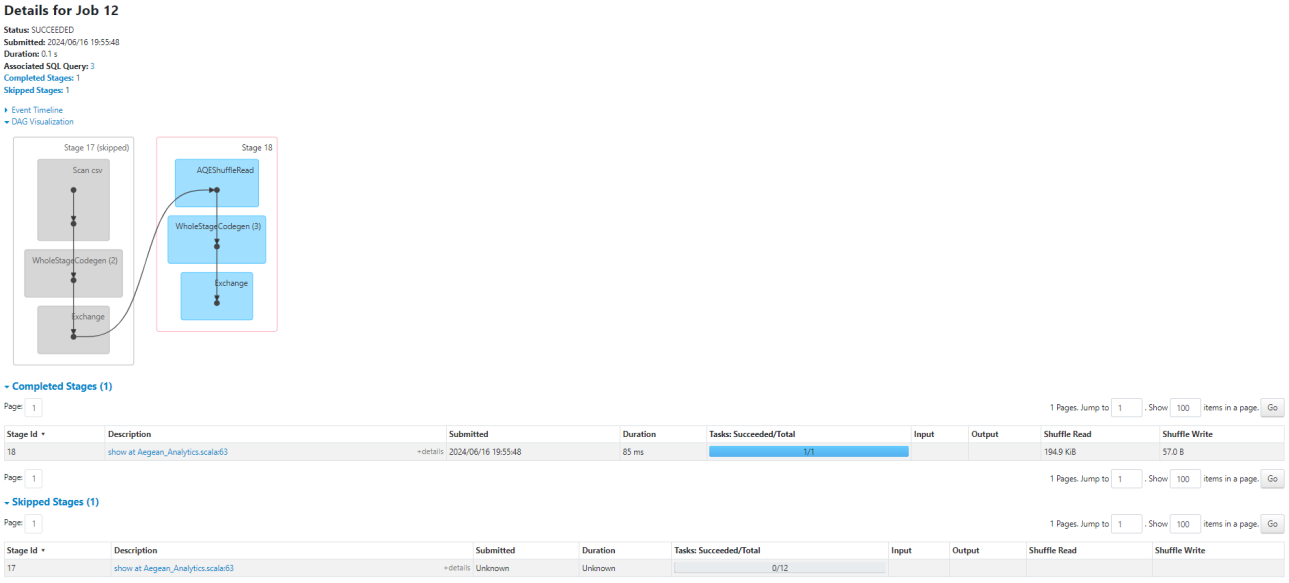


Figure 66: Physical Job 12

Figure 67: Stage 18 of Physical Job 12

Job 12 is associated with Stage 17 that is skipped and with Stage 18.

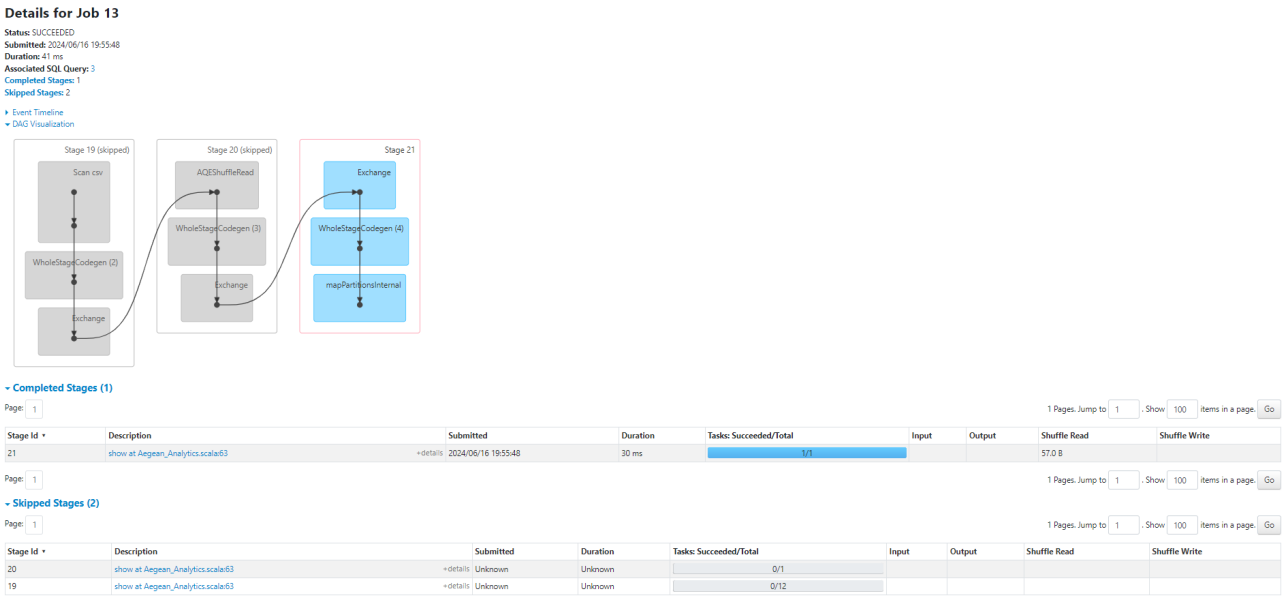


Figure 68: Physical Job 13

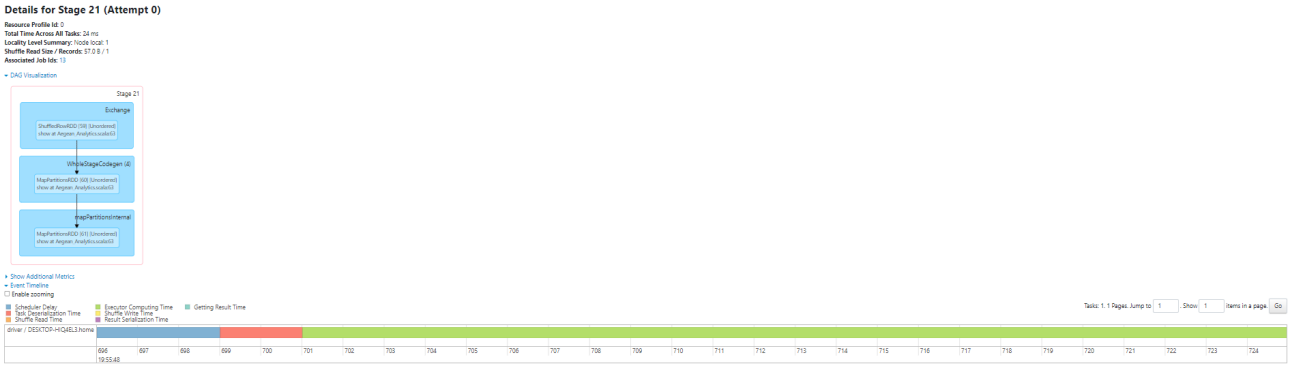


Figure 69: Stage 21 of Physical Job 13

All the above physical jobs in Spark are related to the execution of the show method as outlined in question 3. Physical Job 9 is associated with stage 13, physical Job 10 is associated with Stage 14, physical Job 12 is associated with Stage 17 that is skipped and with Stage 18 and physical Job 13 is associated with Stages 19 and 20 that are skipped and with Stage 21.

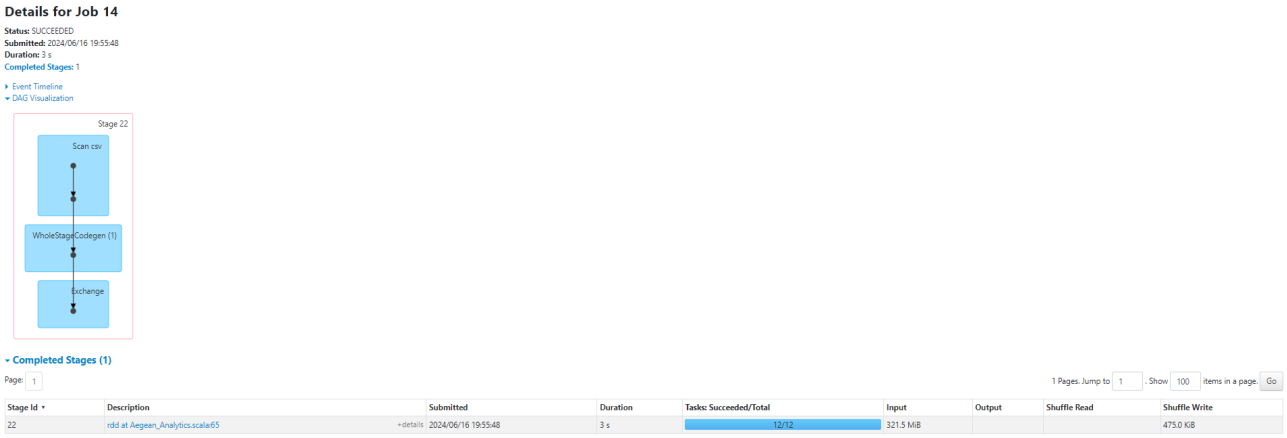


Figure 70: Physical Job 14

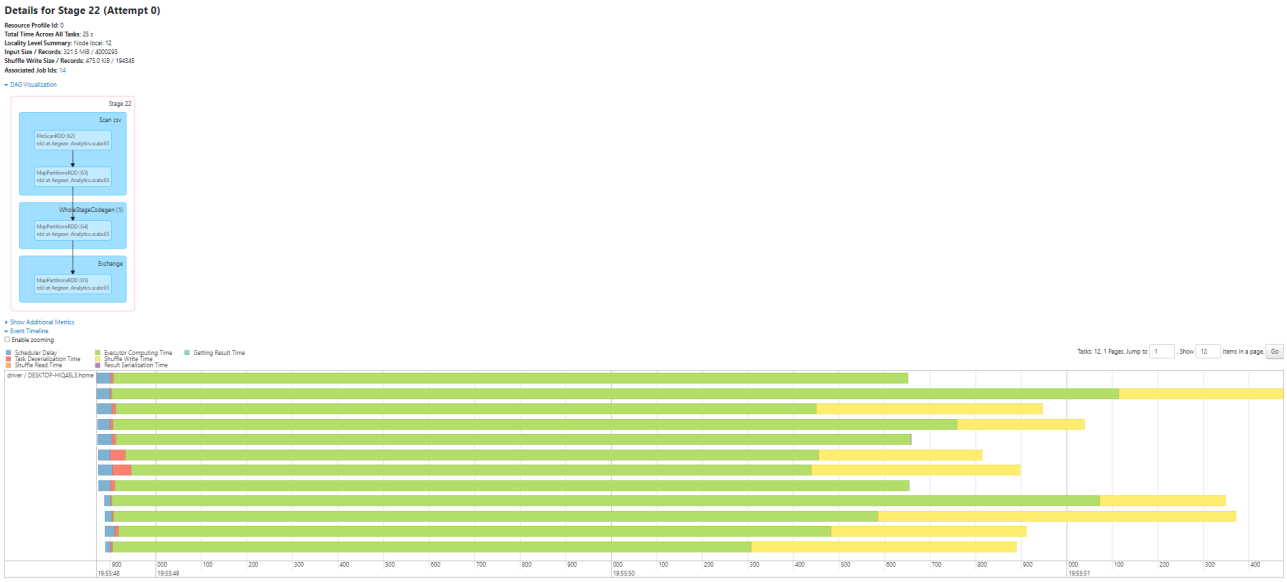


Figure 71: Stage 22 of Physical Job 14

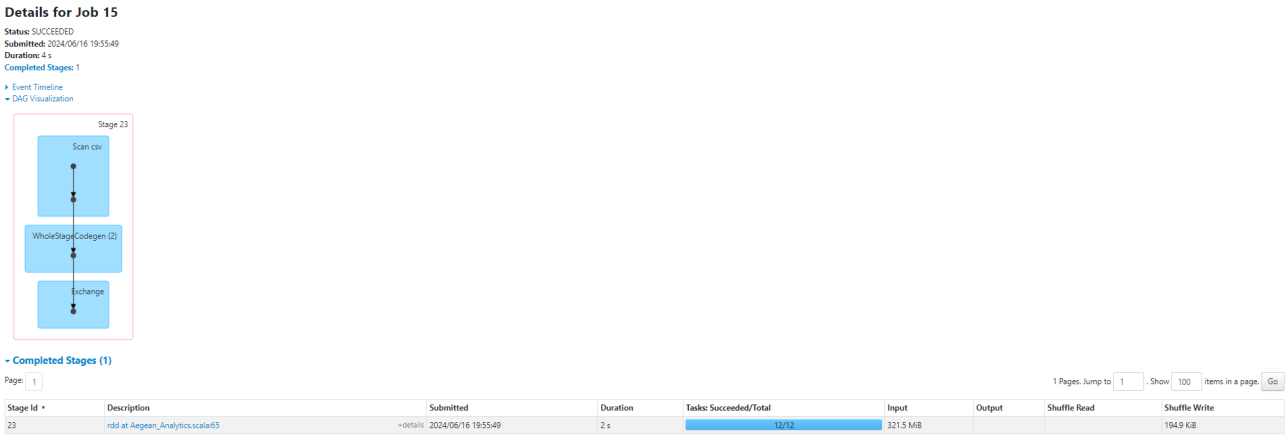


Figure 72: Physical Job 15

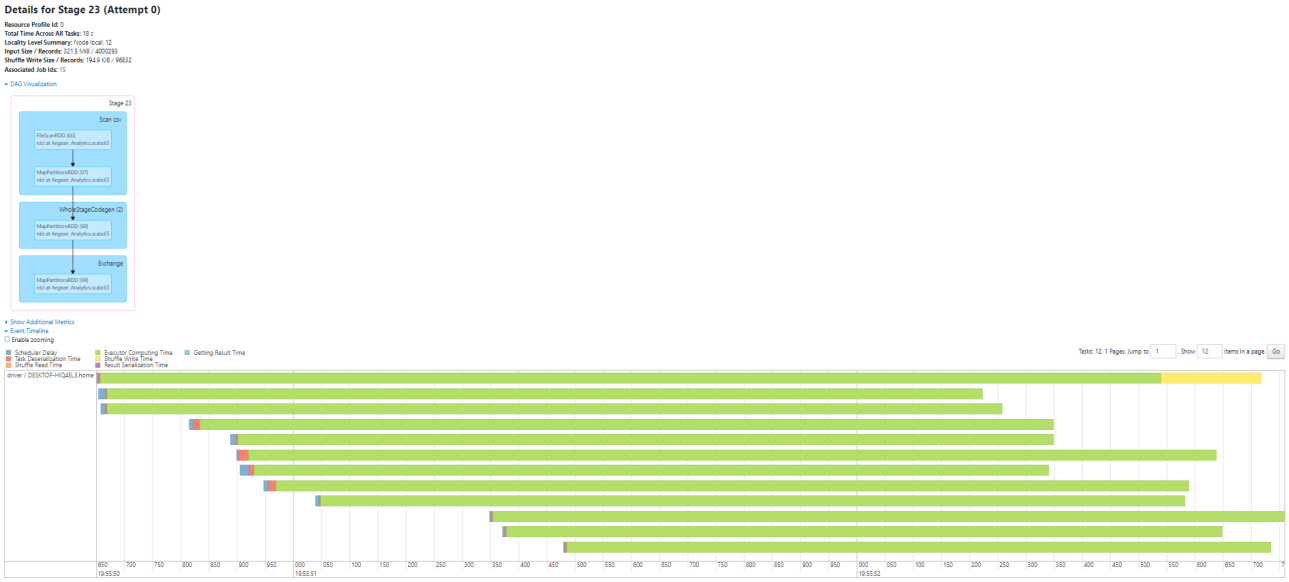


Figure 73: Stage 23 of Physical Job 15

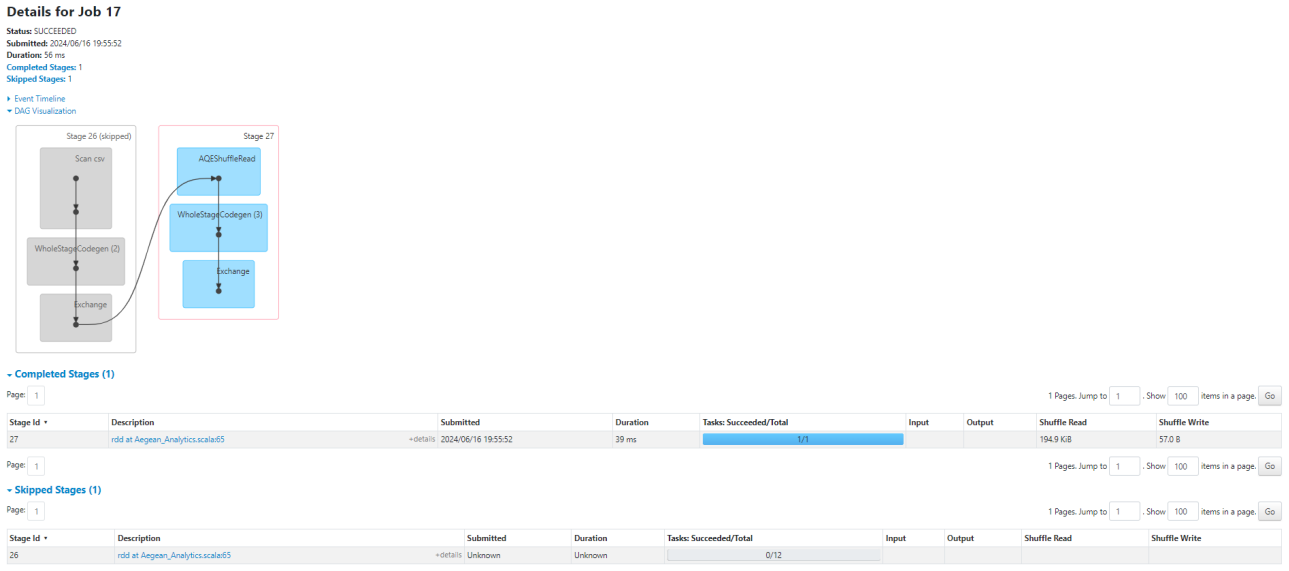


Figure 74: Physical Job 17

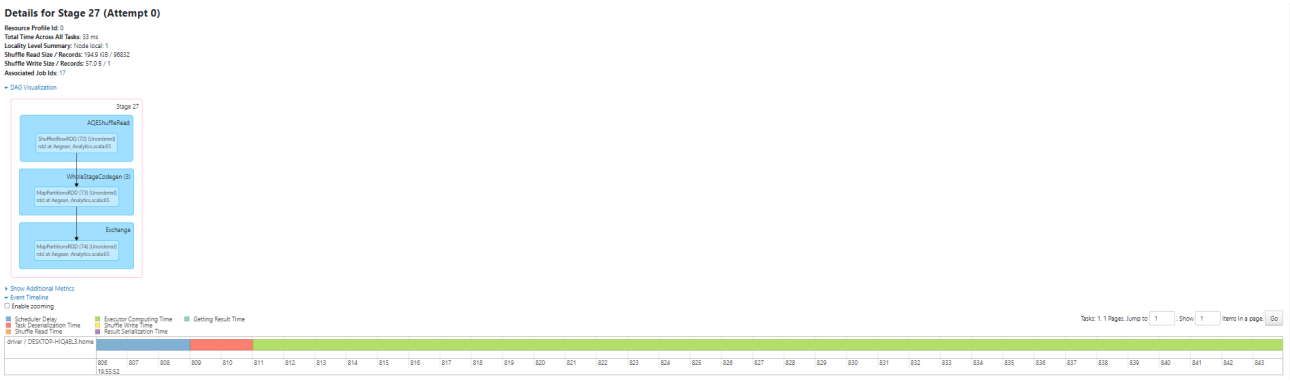


Figure 75: Stage 27 of Physical Job 17

Physical Jobs 14,15 and 17 are about transforming the result DataFrame of Question 3 to RDD and saving it. Physical Job 14 is associated with Stage 22, physical Job 15 is associated with Stage 23 and Physical Job 17 is associated with Stage 27.

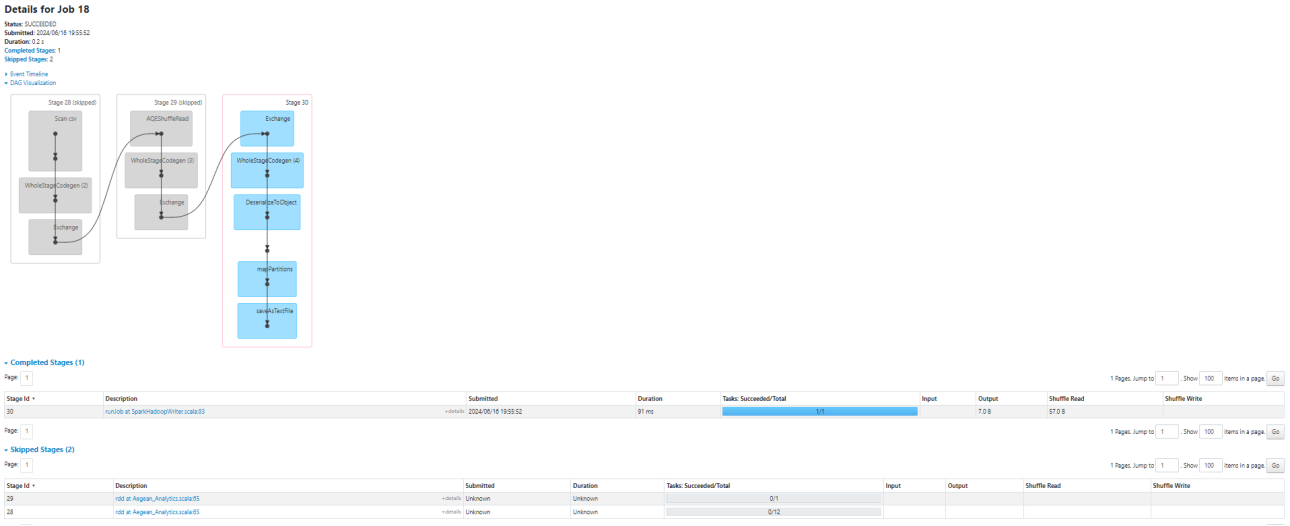


Figure 76: Physical Job 18

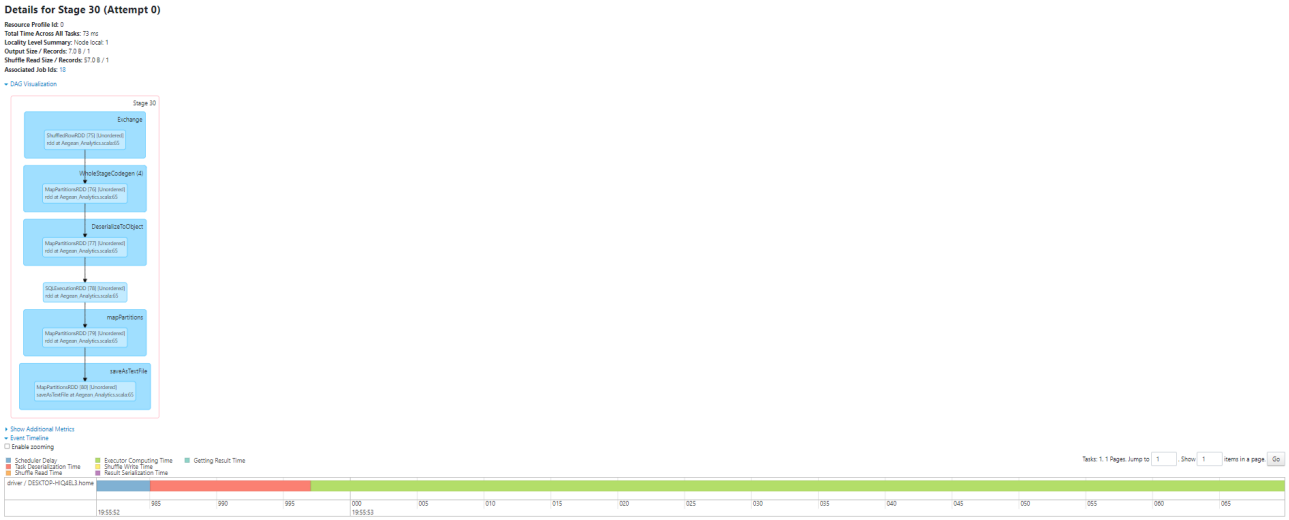


Figure 77: Stage 30 of Physical Job 18

Physical Job 18 is associated with Stage 28 and 29 that are skipped and with Stage 30. This Job is about writing our result to an output file.

- **Question 4**

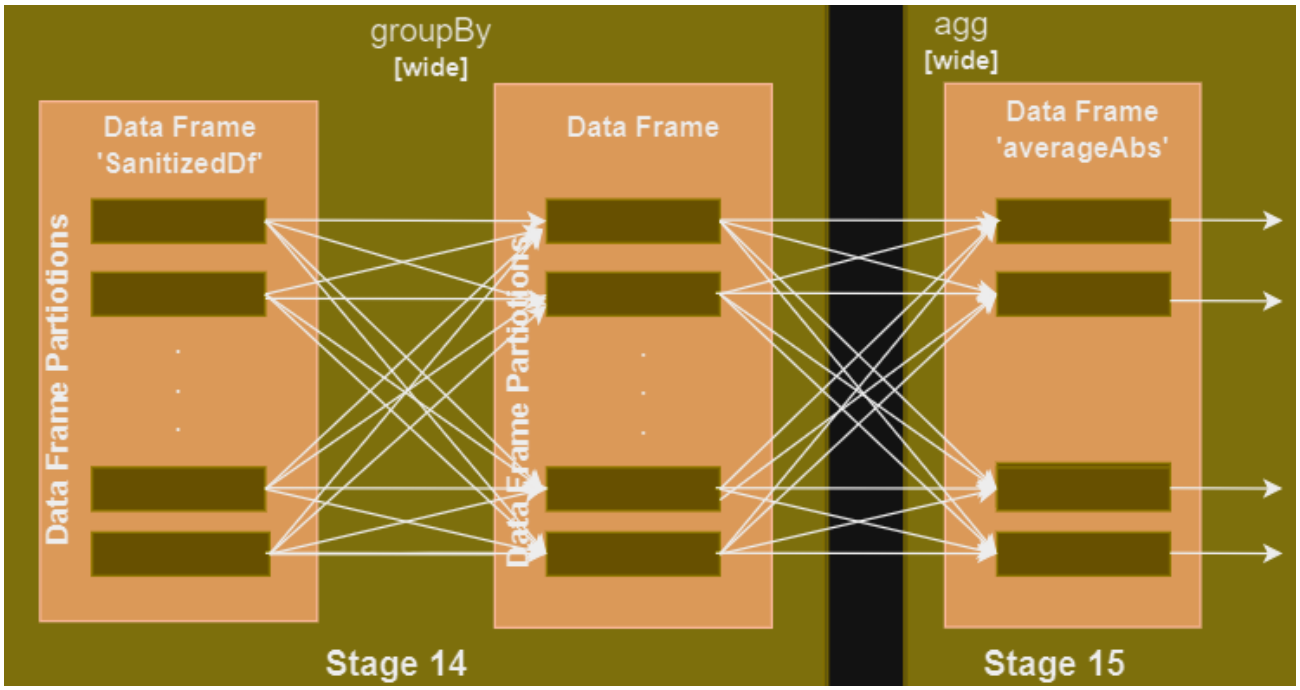


Figure 78: Stages 14 and 15

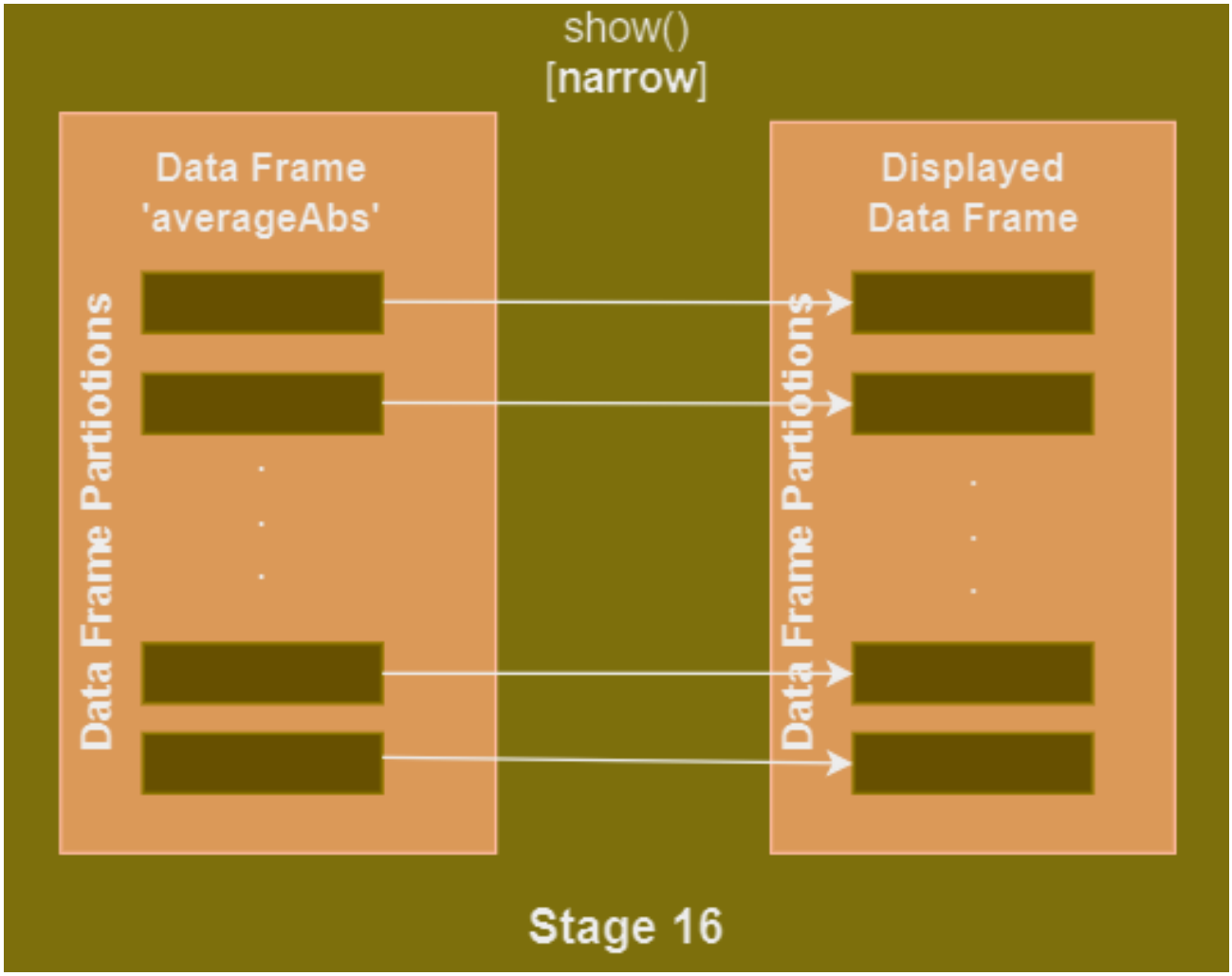


Figure 79: Stages 16 Of Logical Job 8

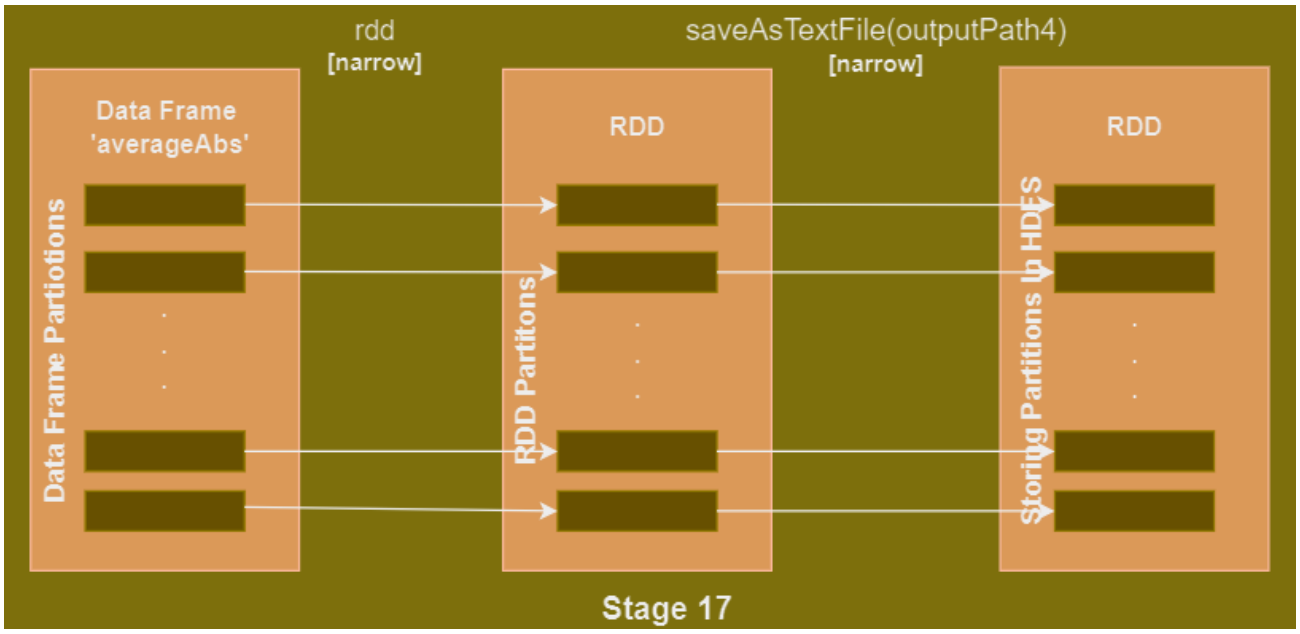


Figure 80: Stages 17 Of Logical Job 9

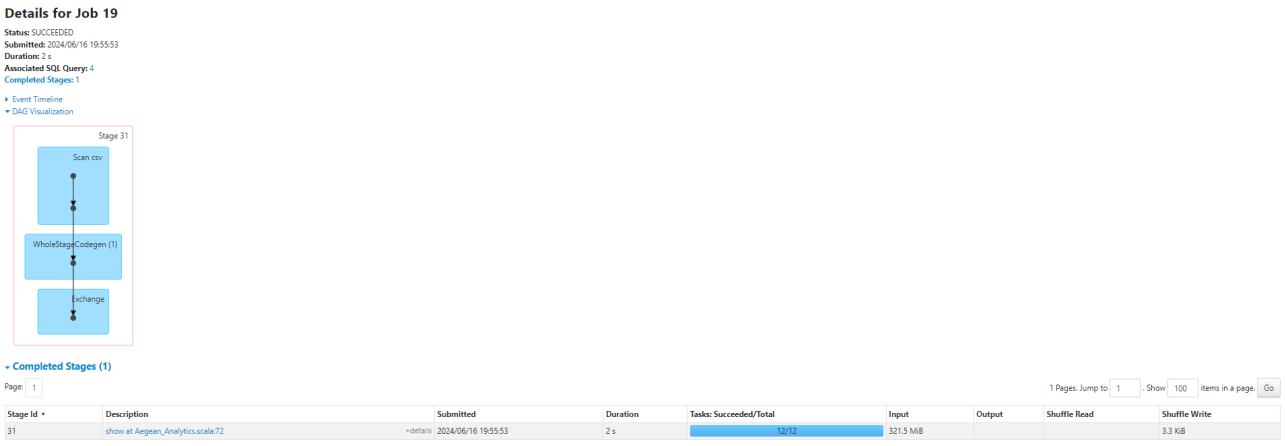


Figure 81: Physical Job 19

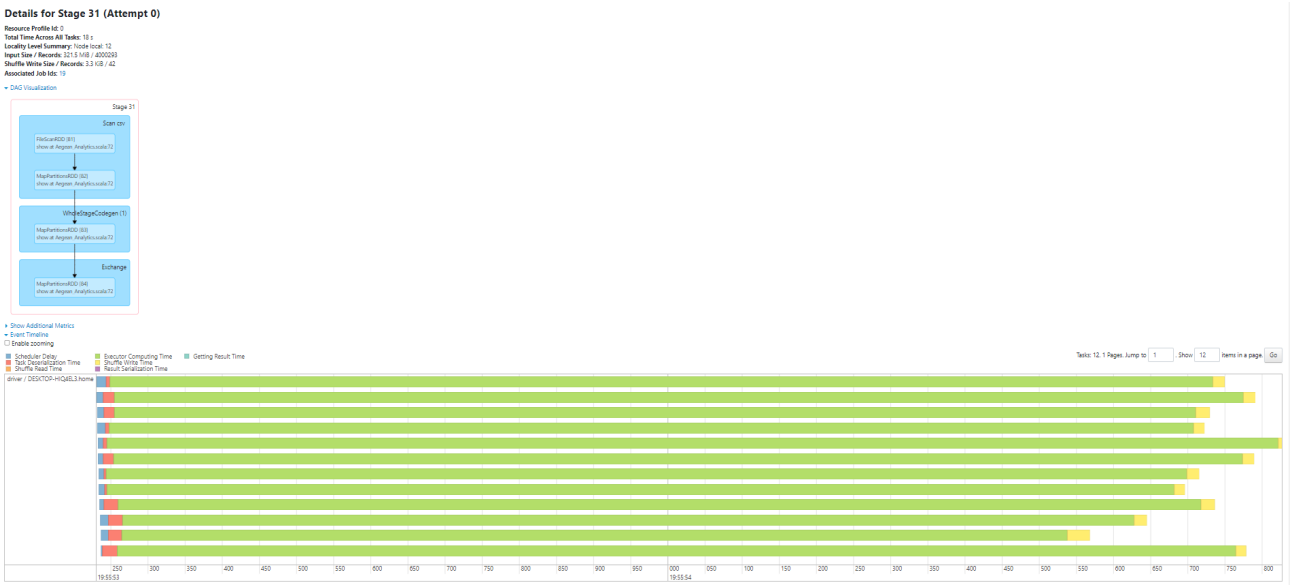


Figure 82: Stage 31 of Physical Job 19

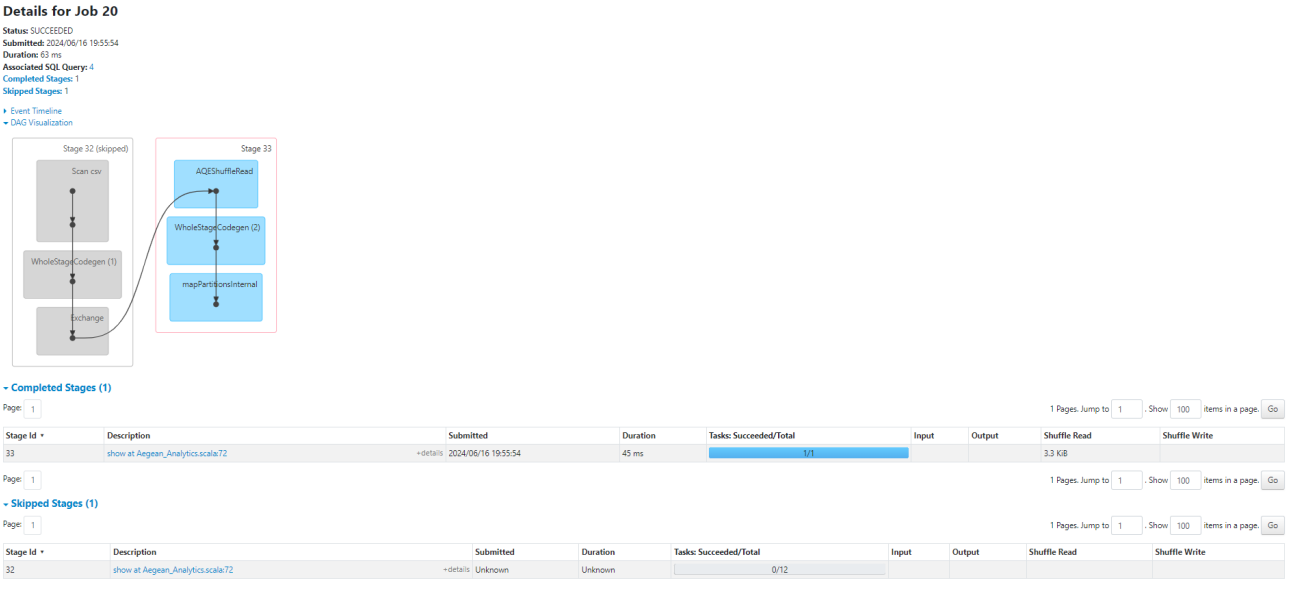


Figure 83: Physical Job 20

Figure 84: Stage 33 of Physical Job 20

Physical Jobs 19 and 20 are related to the execution of the show method as outlined in question 4. Job 19 is associated with Stage 31 and Job 20 is associated with Stage 32 that is skipped and with Stage 33.

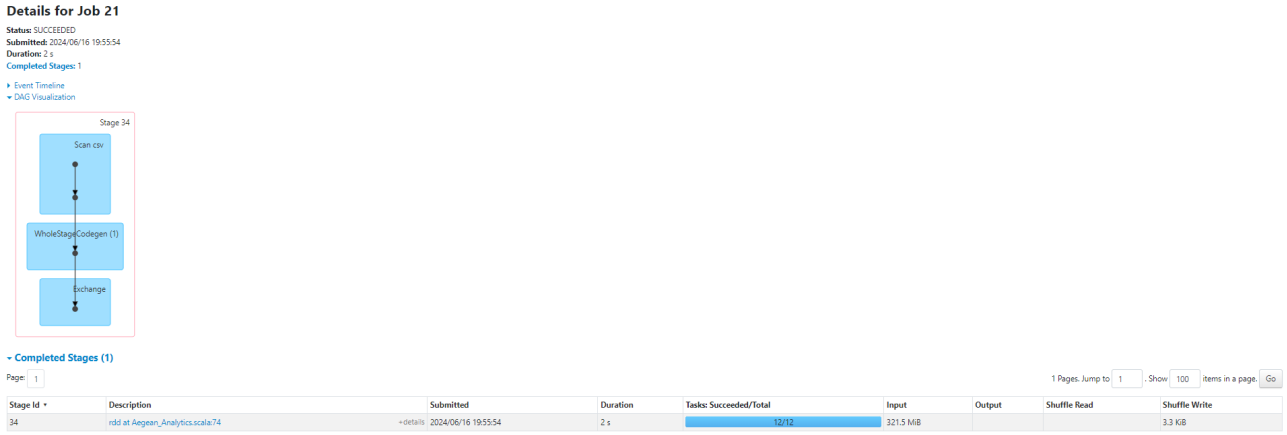


Figure 85: Physical Job 21

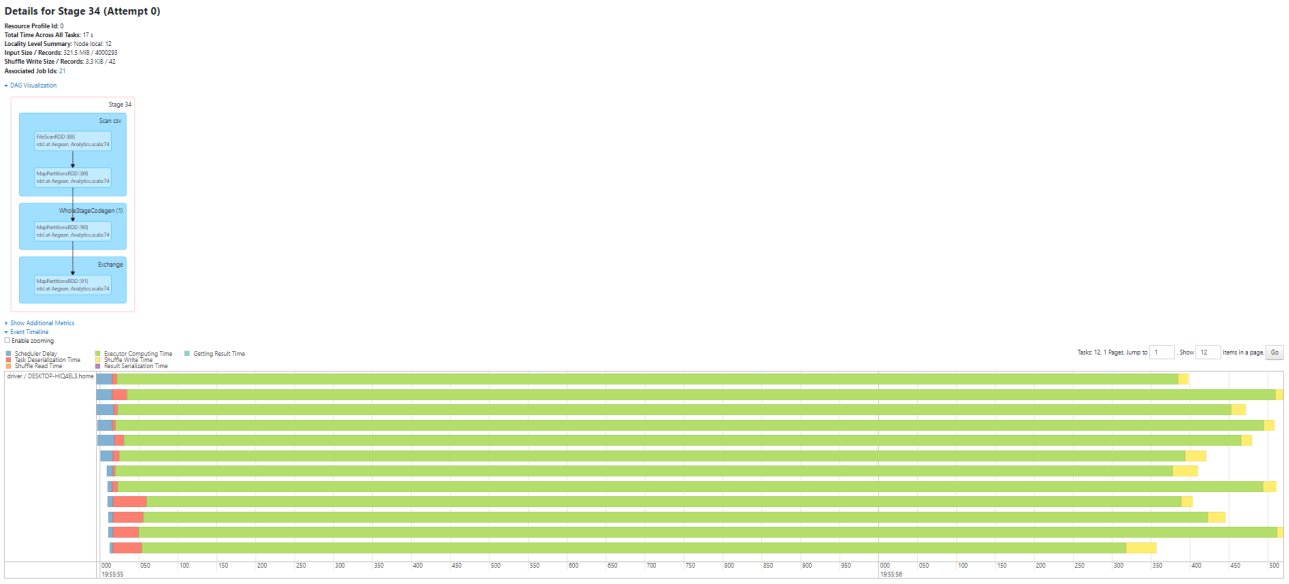


Figure 86: Stage 34 of Physical Job 21

Physical Job 21 is associated with Stage 34. This Job is about transforming the result DataFrame of Question 4 to RDD and saving it.

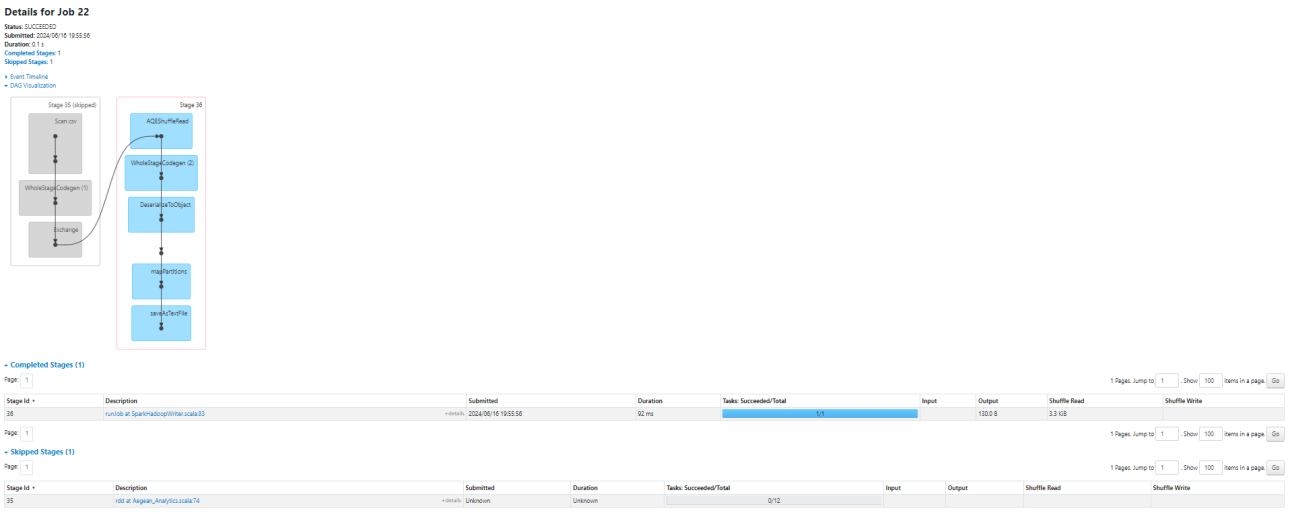


Figure 87: Physical Job 22

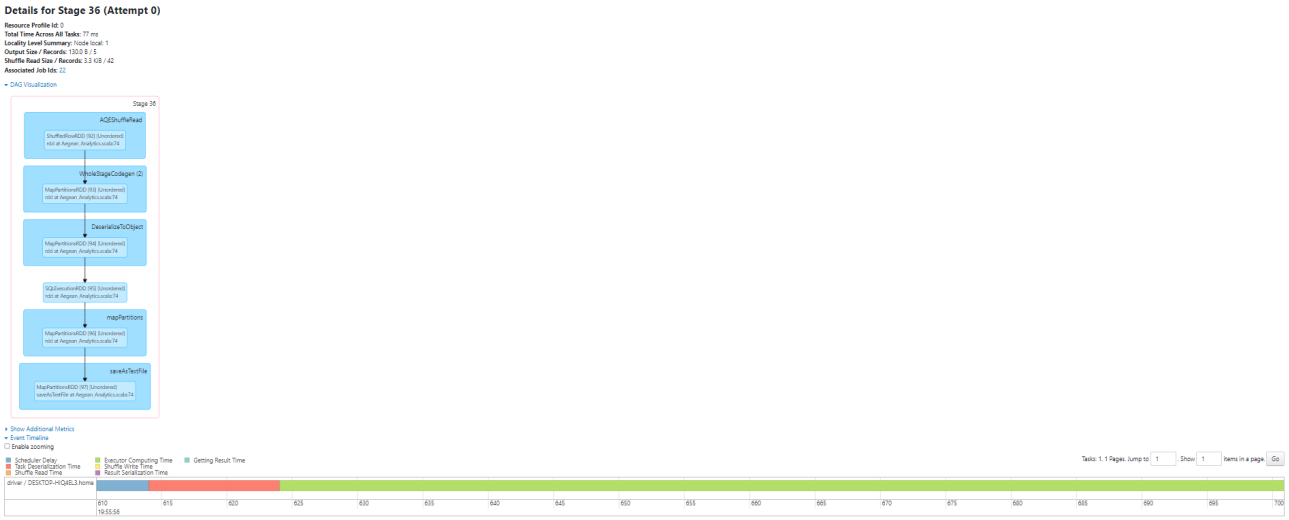


Figure 88: Stage 36 of Physical Job 22

Physical Job 22 is associated with Stage 35 that is skipped and with Stage 36. This Job is about writing our result to an output file.

- **Question 5**

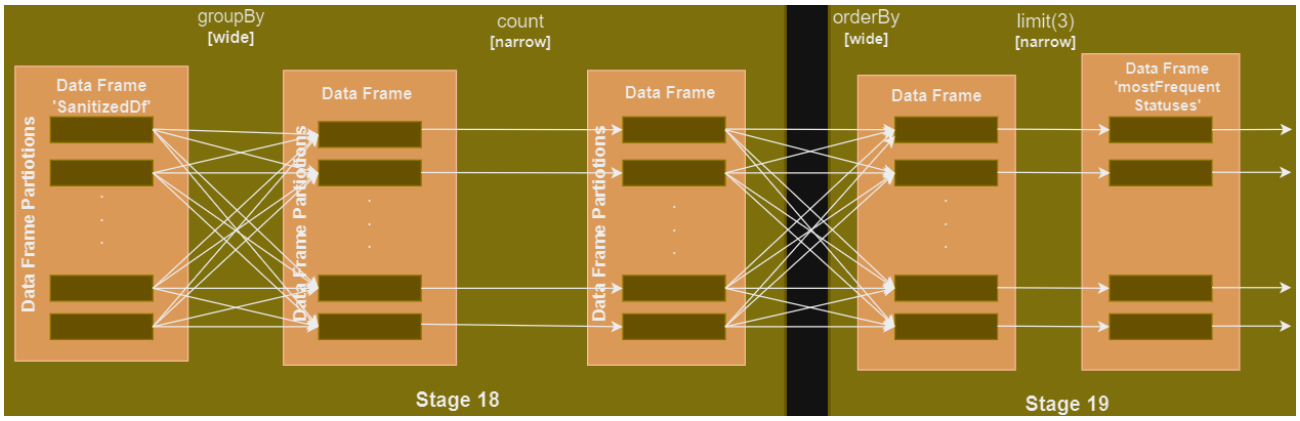


Figure 89: Stages 18 and 19

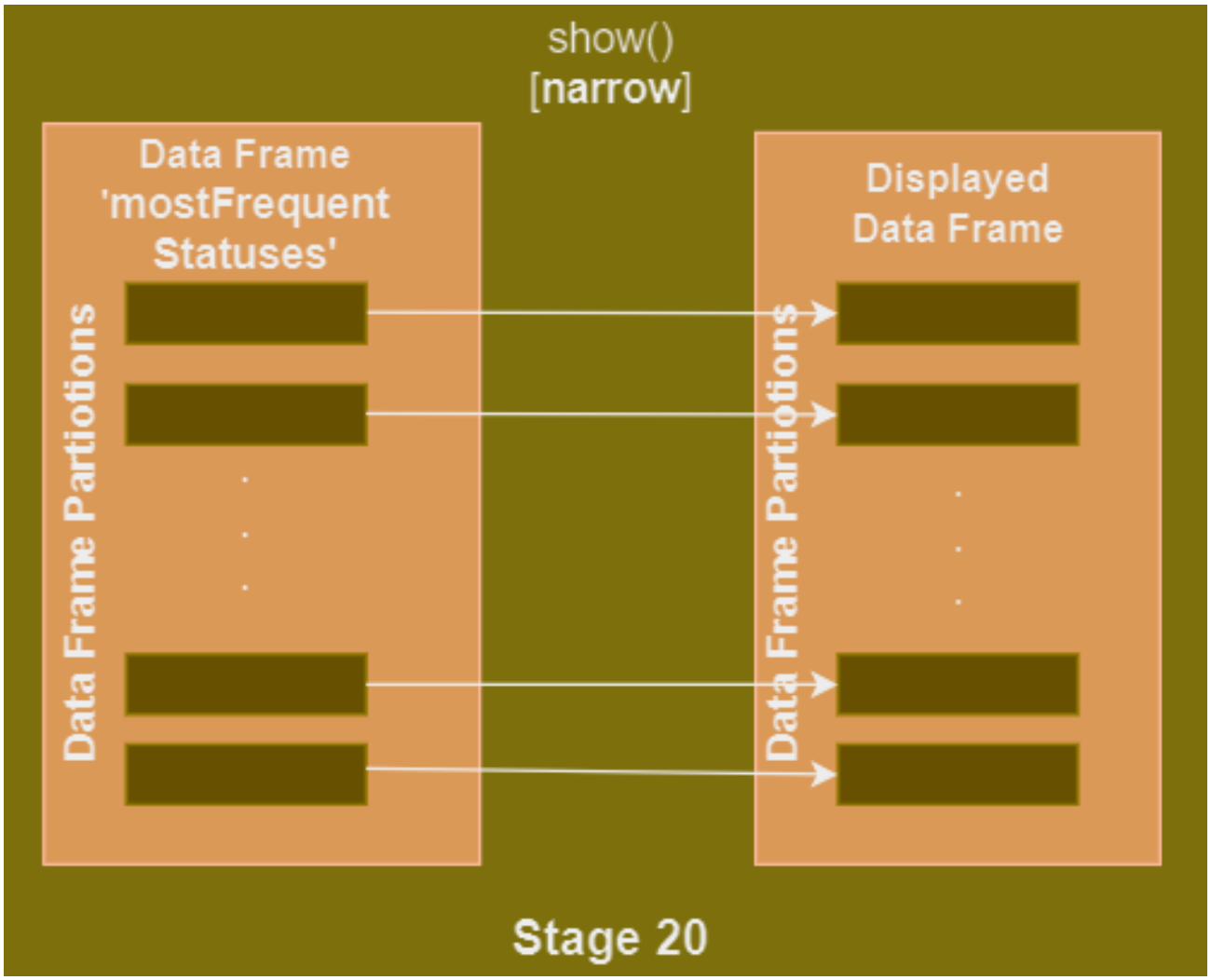


Figure 90: Stages 20 Of Logical Job 10

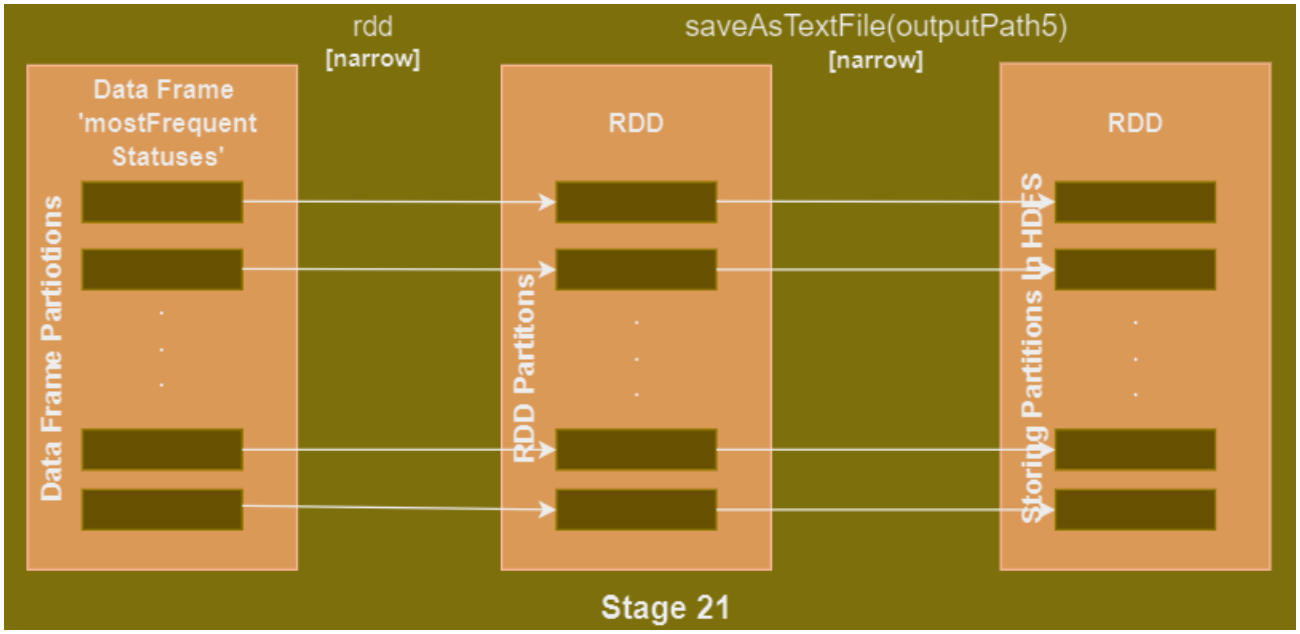


Figure 91: Stages 21 Of Logical Job 11

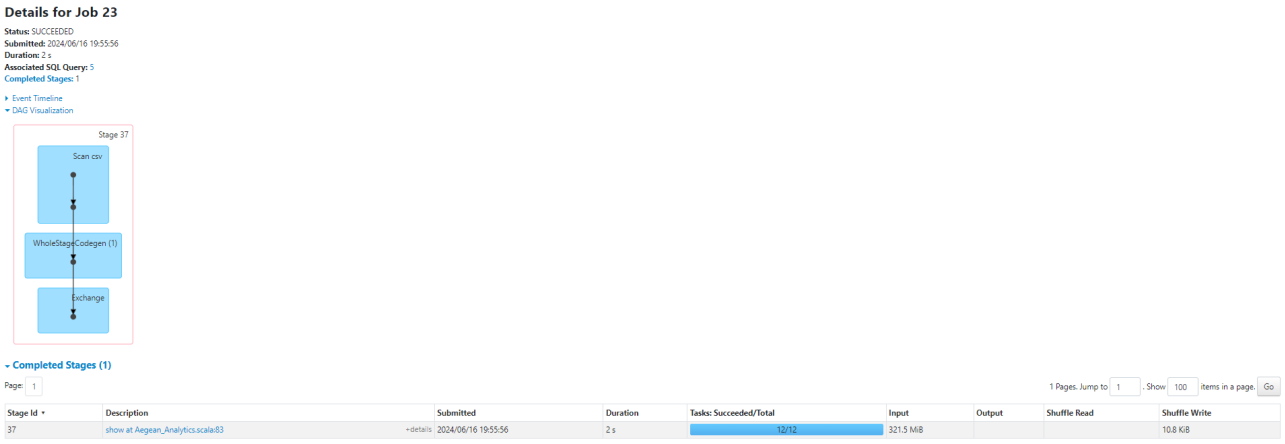


Figure 92: Physical Job 23

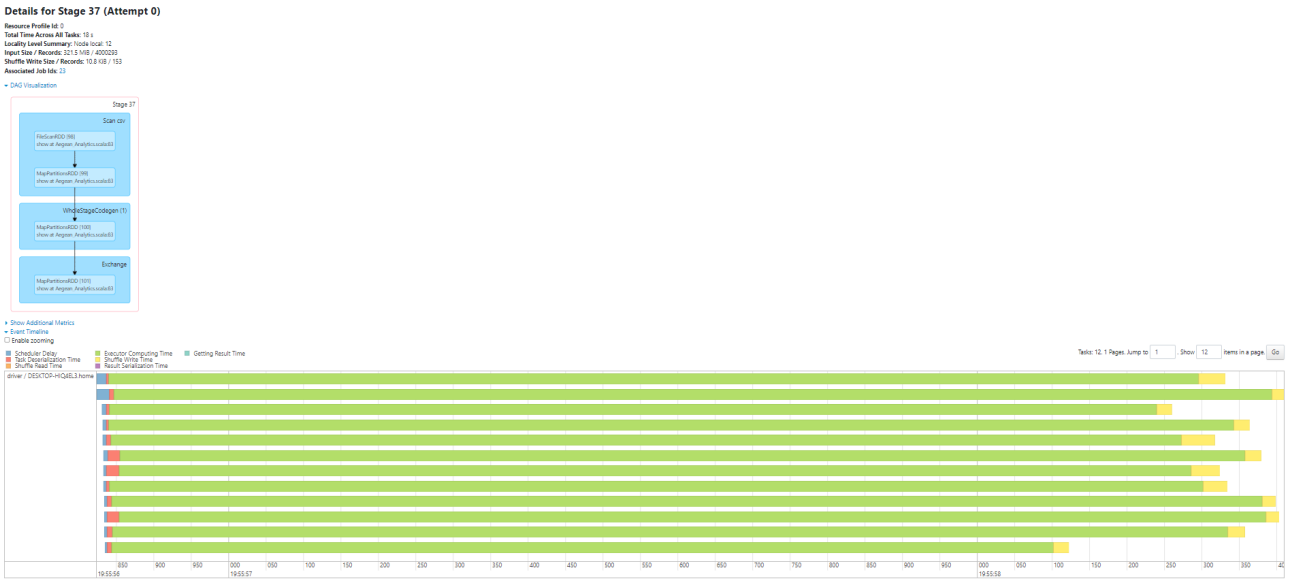


Figure 93: Stage 37 of Physical Job 23

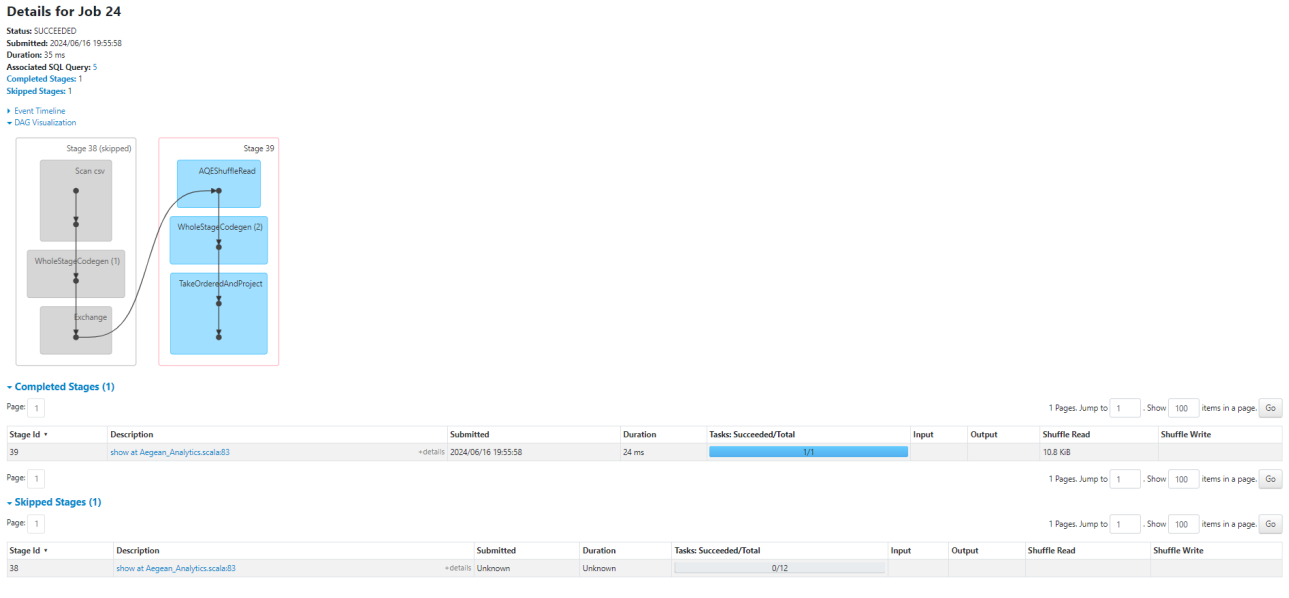


Figure 94: Physical Job 24

Figure 95: Stage 39 of Physical Job 24

Physical Jobs 23 and 24 are about the execution of the show method as outlined in question 5. Job 23 is associated with stage 37, while Job 24 is associated with Stage 38 that is skipped and with Stage 39.

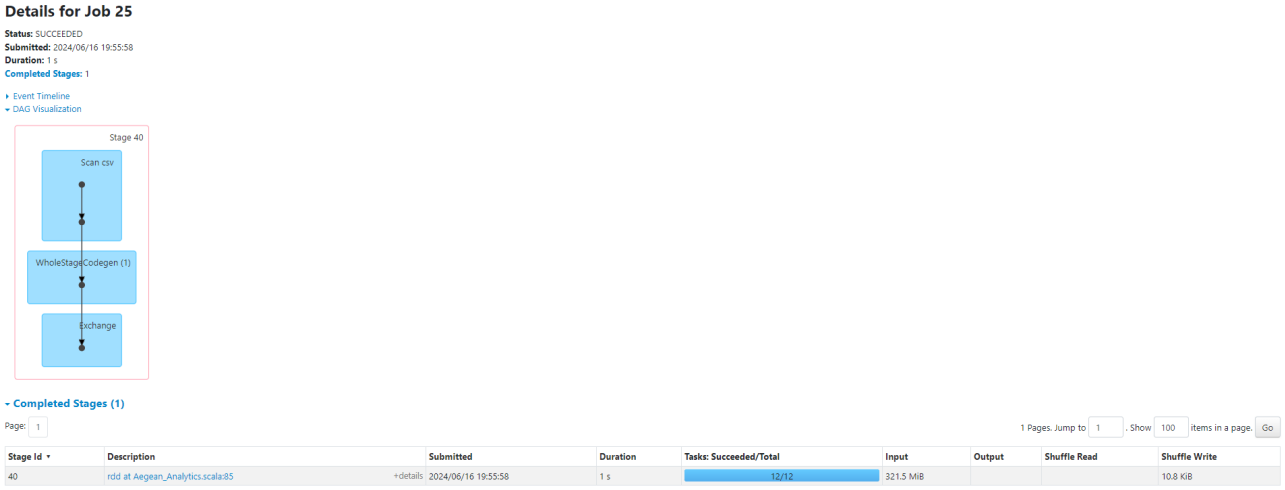


Figure 96: Physical Job 25

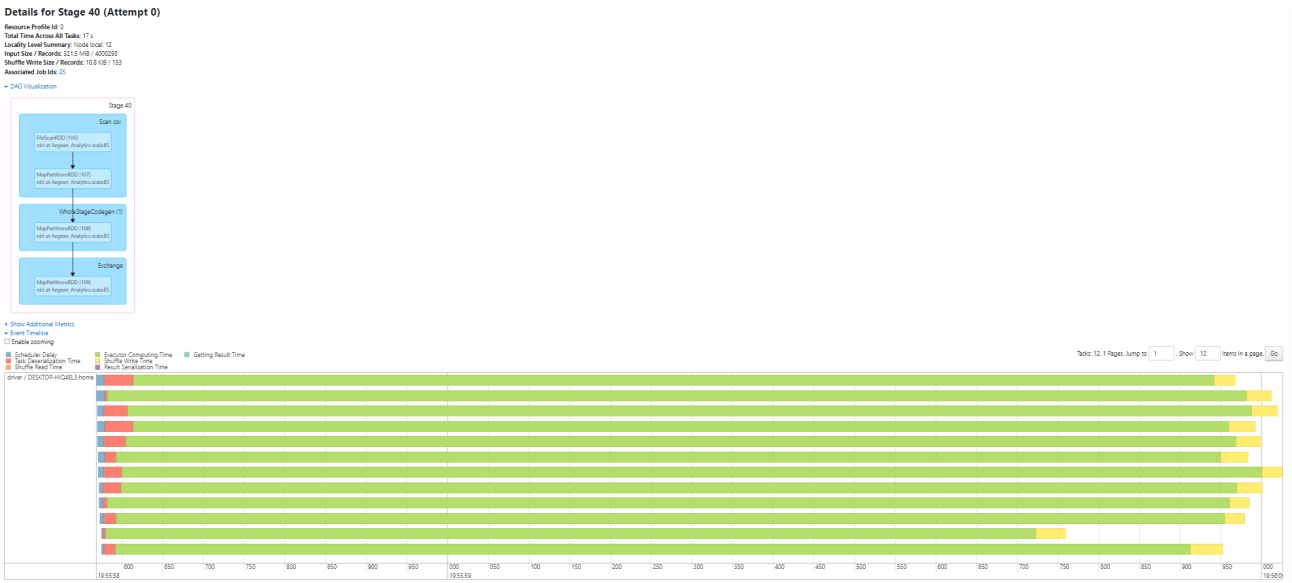


Figure 97: Stage 40 of Physical Job 25

Physical Job 25 is about transforming the result DataFrame of Question 5 to RDD and saving it.

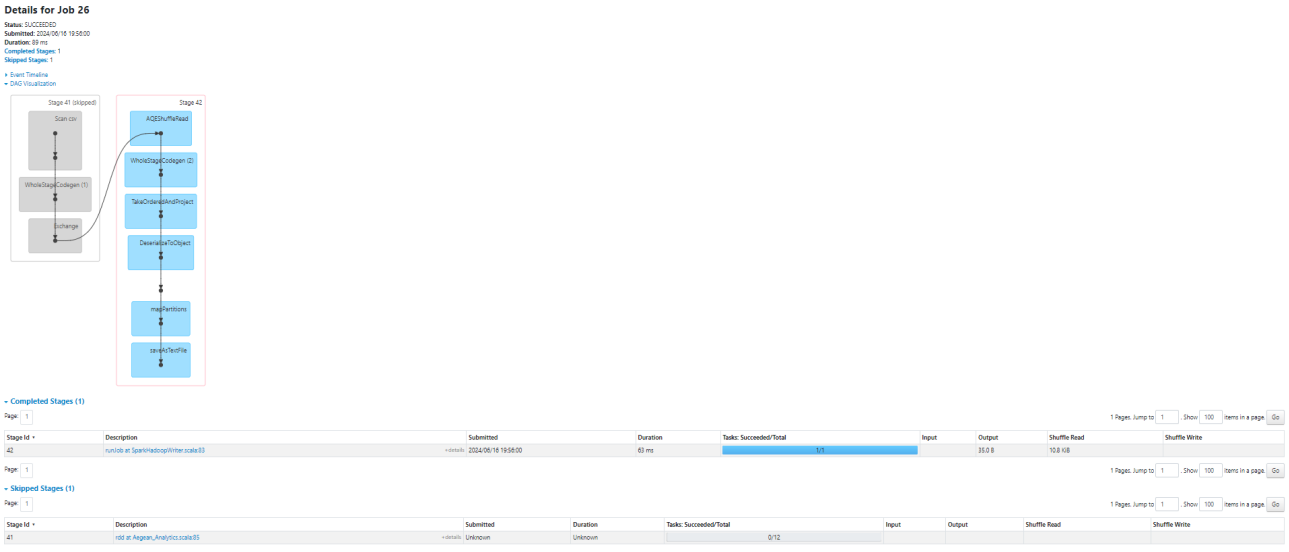


Figure 98: Physical Job 26

Figure 99: Stage 42 of Physical Job 26

Physical Job 26 is associated with Stage 41 that is skipped and with Stage 42. This Job is about writing our result to an output file.

Additionally, we have the following physical Jobs in Spark.

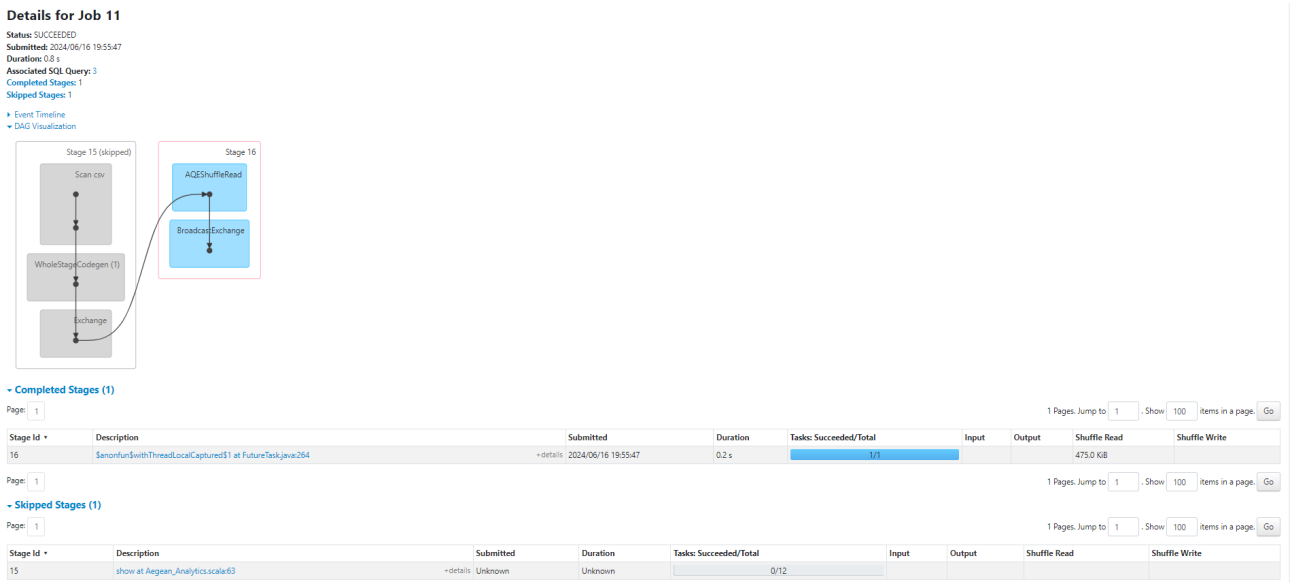


Figure 100: Physical Job 11

Figure 101: Stage 16 of Physical Job 11

Physical Job 11 is associated with stage 15 that is skipped and with stage 16.

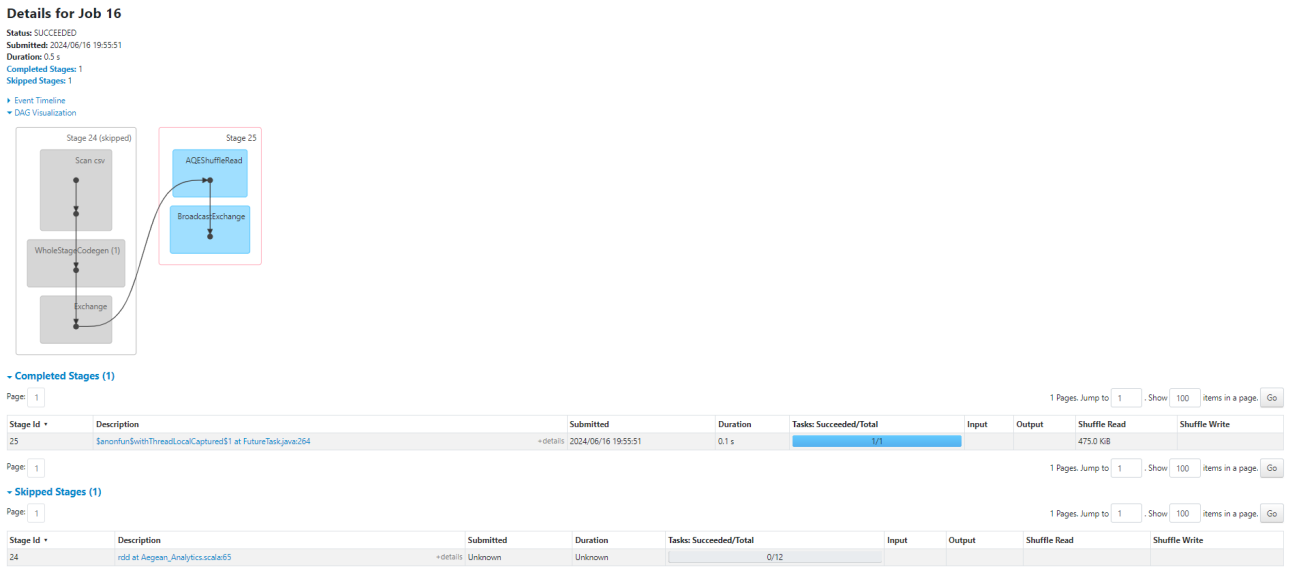


Figure 102: Physical Job 16

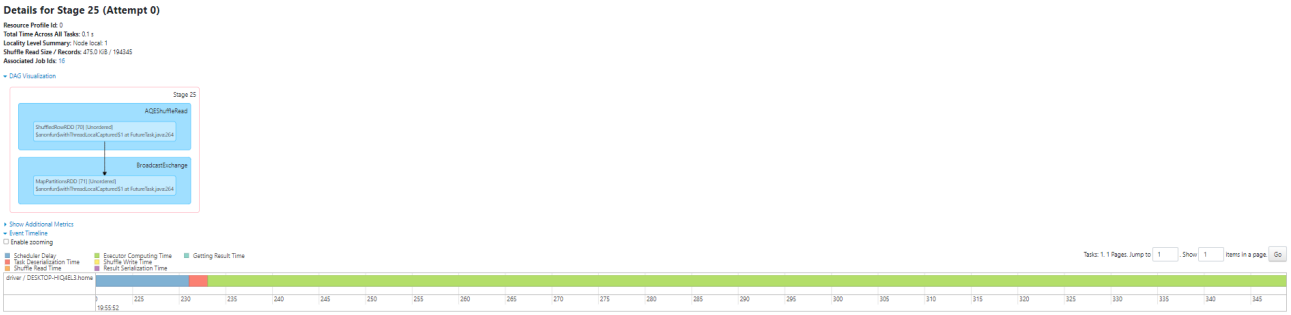
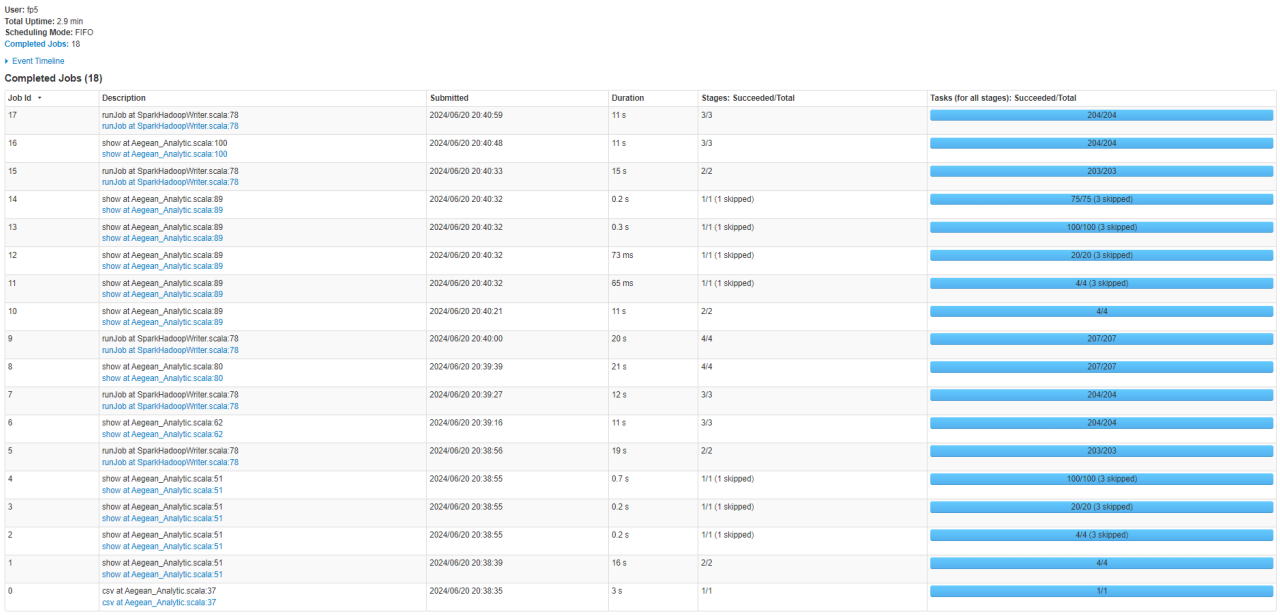


Figure 103: Stage 25 of Physical Job 16

Physical Job 16 is associated with Stage 24 that is skipped and with Stage 25.
This kind of Physical Jobs $anonfun$withThreadLocalCaptured$1 at FutureTaskjava:264 indicates that an anonymous function is executed in a way that keeps track of certain variables specific to each thread. This is common when Spark runs tasks in parallel or uses multiple threads.

2.4 Comparison of Execution Environments

All of the physical jobs and their associated stages mentioned above are part of the result from local execution. When we tried to run the program on cluster, we noticed some differences.



User: fp5
Total Uptime: 2.9 min
Scheduling Mode: FIFO
Completed Jobs: 18

▸ Event Timeline

Completed Jobs (18)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
17	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2024/06/20 20:40:59	11 s	3/3	204/204
16	show at Aegean_Analytic.scala:100 show at Aegean_Analytic.scala:100	2024/06/20 20:40:48	11 s	3/3	204/204
15	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2024/06/20 20:40:33	15 s	2/2	203/203
14	show at Aegean_Analytic.scala:89 show at Aegean_Analytic.scala:89	2024/06/20 20:40:32	0.2 s	1/1 (1 skipped)	75/75 (3 skipped)
13	show at Aegean_Analytic.scala:89 show at Aegean_Analytic.scala:89	2024/06/20 20:40:32	0.3 s	1/1 (1 skipped)	100/100 (3 skipped)
12	show at Aegean_Analytic.scala:89 show at Aegean_Analytic.scala:89	2024/06/20 20:40:32	73 ms	1/1 (1 skipped)	20/20 (3 skipped)
11	show at Aegean_Analytic.scala:89 show at Aegean_Analytic.scala:89	2024/06/20 20:40:32	65 ms	1/1 (1 skipped)	4/4 (3 skipped)
10	show at Aegean_Analytic.scala:89 show at Aegean_Analytic.scala:89	2024/06/20 20:40:21	11 s	2/2	4/4
9	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2024/06/20 20:40:00	20 s	4/4	207/207
8	show at Aegean_Analytic.scala:80 show at Aegean_Analytic.scala:80	2024/06/20 20:39:39	21 s	4/4	207/207
7	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2024/06/20 20:39:27	12 s	3/3	204/204
6	show at Aegean_Analytic.scala:62 show at Aegean_Analytic.scala:62	2024/06/20 20:39:16	11 s	3/3	204/204
5	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2024/06/20 20:38:56	19 s	2/2	203/203
4	show at Aegean_Analytic.scala:51 show at Aegean_Analytic.scala:51	2024/06/20 20:38:55	0.7 s	1/1 (1 skipped)	100/100 (3 skipped)
3	show at Aegean_Analytic.scala:51 show at Aegean_Analytic.scala:51	2024/06/20 20:38:55	0.2 s	1/1 (1 skipped)	20/20 (3 skipped)
2	show at Aegean_Analytic.scala:51 show at Aegean_Analytic.scala:51	2024/06/20 20:38:55	0.2 s	1/1 (1 skipped)	4/4 (3 skipped)
1	show at Aegean_Analytic.scala:51 show at Aegean_Analytic.scala:51	2024/06/20 20:38:39	16 s	2/2	4/4
0	csv at Aegean_Analytic.scala:37 csv at Aegean_Analytic.scala:37	2024/06/20 20:38:35	3 s	1/1	1/1

Figure 104: Physical Jobs On Cluster

Above, we see the completed jobs on the cluster. We observed that the number of jobs executed on the cluster is less than the number of jobs during local execution. Additionally, each job on the cluster has up to 3 completed stages, whereas during local execution, each job typically had only 1 completed stage. Furthermore, on the cluster, each stage can run up to 200 tasks, while in local execution, the maximum number of tasks per stage was only 12. These differences between cluster and local execution are expected. On the cluster, Spark performs more advanced optimizations and can skip redundant stages more effectively. This occurs because the cluster environment has more executors available, allowing for greater parallelism and more efficient task distribution compared to local execution, which has fewer executors and limited resources.