Scalable Data Management

Practical Assignment

Big Data Tuning

Table of Contents

1. Job Design Documentation and Optimisation

1.1 Task 1: Top-3 Cessna Models

1.1.1 Unoptimised RDD API Implementation

1. Read in the Aircraft and Flight csv files as RDDs, remove rows with NULL.
2. The **map()** function is used to split the data on comma separators (“,”).
3. The entries where the manufacturer is Cessna are selected using the **FIlter()** function, flights where tail_number is an empty string are removed.
4. The **map()** function is used to reduce the dimensions of the aircraft RDD by taking only the tailnum and model columns. The same is done with Flights on tailnum and carrier_code.
5. By using a **regex expression**, the string representation for each model number is formatted to only have it’s first 3 digits using a **map()** function.
6. The **join()** function is used to combine the reduced aircraft and flight RDDs on tailnum=tail_number.
7. The **reduceByKey()** and **takeOrdered()** functions are used to group and order the results of our final joined RDD, and then print out the results

1.1.2 Optimised DataFrame API Implementation

1. Aircrafts and flights are cleaned and combined using the **join() (union)** function on tail number.
2. The **filter()** function is applied to only store those with “CESSNA” as a manufacturer.
3. Format the model name to only include 3 digits using the **withColumn()** function and using **regexp_extract()** to change the value of each instance.
4. Use the **groupBy() (model count)** and **orderBy() (descending)** to produce a DF with grouped models and their total flights.
5. Select the top three models using the **head(3)** function. This converts the DF into a new list, that is used to output the results.

1.1.3 Changes in execution plans

In our DAG visualisation of our unoptimised RDD implementation *(see appendix 3.1)*, the bottleneck is in the PairwiseRDD join which takes 1.9 minutes to complete. In comparison, our DF implementation avoids this by performing a broadcast join. On the large dataset, Spark broadcasts the smaller aircrafts DF, and avoids shuffling over the much larger flights DF that has upwards of 14 million rows. This avoids the very network-intensive task that could result in bottlenecking as observed in the RDD implementation, thus leading to faster execution times in general. Note also that the first HashAggregate speeds up the second after the exchange in DF.

1.1.4 Justification of Any Tuning Decisions

Converting task from RDD to DataFrame
RDDs are slower because there is no inbuilt optimisation engine available for them and RDDs do not have the ability to take advantage of Apache Sparks advanced optimisers such as the catalyst optimiser and Tungsten execution engine. The only method to optimise each RDD is by taking the attributes into account and projecting only the columns that are required. However, DataFrames have the ability to use the optimisation engines. The DataFrame task implementations use the catalyst tree transformation framework to form an optimized logical and physical plan.

Fiter and Project earlier

The Aircraft and Flight datasets had their dimensions reduced by selecting only the columns that are needed in the task. The manufacturer Cessna was filtered early straight after the join, which also reduced size and improved performance.

Broadcast Join

The Aircraft dataset is quite smaller than the Flights dataset, therefore the broadcast hash join was executed to split the data into buckets that are later merged. Since there is minimal data shuffling the join is faster.

1.2 Task 2: Average Departure Delay

1.2.1 Unoptimised RDD API Implementation

1. Read in the Airline and Flight csv files as RDDs, the **map()** function is used to split the data on comma separators (“,”).
2. The **filter()** function is used to keep flights within the United States in the Airline RDD.
3. Carrier and name columns are only taken to be projected using the **map()** function in the Flight RDD.
4. The **filter()** function is then used to take only rows with flight_date equaling the specified year, and to remove null values from the departure times in the Flight RDD.
5. Next a user-defined function called **convert_24_hr()** is used to add padding to times under 10 hours to allow for time calculation, for example 15 -> 0015. This function also converts 2400 into 0000.
6. Then the Flight RDD is reduced to keep only carrier_code, scheduled_depature_time, actual_departure_time using the **map()** function.
7. The UDF **compute_delay()** is used to convert the departure times into a datetime format, then delay is calculated in minutes as time_departed - time_scheduled assuming there are no delays earlier/later than 12 hours .
8. The **filter()** function is used to keep delay values that are greater than 0 in the Flight RDD.
9. The Flight RDD is then transformed using the **groupByKey()** function. This groups on carrier_code.
10. The Flight and Airline RDDs are combined using the **join()** function and the **map()** function is applied to organise the RDD as airline_name, num_delays, average_delays, min_delay, max_delay.
11. The RDD has the function **collect()** applied and the results are sorted using **sort()** and printed.

1.2.2 Optimised DataFrame API Implementation

1. Airlines and flights are cleaned and only columns that are required by task 2 are projected using the **select()** function.
2. The **filter()** function is applied to only include US airlines by filtering the country column.
3. To work with the dates in flight_date, **withColumn()** is used to iterate through them and the **cast()** function is applied to each instance to convert it from string to DateType.
4. The **filter()** and **between()** functions are used on the flight_date column to only keep instances between the specified year.
5. The **filter()** and **isNotNull()** functions are used to remove null values in actual and scheduled departure times.
6. The **withColumn()** function is used and calls the UDF **compute_delay(),** as well as creating a new column called delay that holds a delay integer value in minutes.
7. The user-defined function **compute_delay()** is used to calculate the difference in time between time_scheduled, it converts the time in a string and adds the zeros for times that are under 10 hours (145 -> 0145). Then converts it into minutes and returns the delay assuming there are no delays earlier/later than 12 hours.
8. The DataFrame is **filtered()** to only keep delayed flights (those with delays > 0)
9. The Airline and Flight datasets are combined using the **broadcast join()** function on carrier_code.
10. The new delayed DataFrame has the **groupBy() (airline name)** applied and **aggregates** calculated including count, mean, min and max delay.

1.2.3 Changes in execution plans

In our DAG visualisation of our unoptimised RDD implementation *(see appendix 3.2, Figure 3.2.1),* there is a bottleneck during the groupByKey transformation, taking 1.4 minutes to complete in the large dataset. This process requires shuffling through all flights to group on tail numbers. Additionally, in large DF, filtering by flights by year=2001, reduces the number of rows from upwards of 17 million to just less than 500000 - this is a reduction to less than 30% of the initial large flights dataset (note: this process is also similarly achieved in RDD prior). In suggestion of further improvements to optimization, a repartitionBy year in reading the large flights csv, would avoid reading in 17 million rows and thus reduce runtime from this step alone (this would be more apparent on large scale datasets). In addition, more attention could be made on optimising aggregate functions used.

1.2.4 Justification of Any Tuning Decisions

Converting task from RDD to DataFrame:

In theory, DataFrames ability to use the optimisation engines should improve performance. Nevertheless, this is not achieved in the performance evaluation section across all dataset sizes. This is further discussed in the performance evaluation section.

Join later and Broadcast Join

Similar to task one, the broadcast join is used to join the smaller airline file to the larger flight file, since there is minimal data shuffling the join is faster. The join is also done as the second last step before aggregating the output. The join was usually done as the first job, however, after searching for optimisations, executing the join only when needed did improve execution time, because the dataset dimensions are smaller during the transformations.

Fiter and Project earlier

The datasets are reduced as much as possible, because having a smaller projection can lead to faster transformations and loading. This means only selecting columns that are necessary for the task.

1.3 Task 3: Most Popular Aircraft Types

1.3.1 Unoptimised RDD API Implementation

1. The Flights, Airlines and Aircraft datasets are read in as RDDs and the **filter()** function is used to remove rows with NULL and empty strings.
2. The Airlines RDD is filtered by the user specified country using the **filter()** function and is mapped using the **map()** function to (carrier_code, (name,country).
3. Next the header is removed with **first()** and rows with null in the aircraft RDD are **filtered().**
4. The aircrafts RDD is projected with only the tailnum, manufacturer and model using the **map()** function. The flights RDD is also mapped to (carrier_code , tailnum).
5. The airline and flight RDDs are joined using the **join()** function as airline_count.
6. The airline_count RDD is mapped to (tailnum, (name, carrier_code)) using the **map()** function.
7. The function **reduceByKey()** is applied to airline_count to calculate the sum in the lambda function.
8. Format the model name to only include 3 digits using the **map()** and **regexp_extract()** to change the value of each instance on the aircraft RDD.
9. The aircraft and airline count RDDs are joined using the **join()** function.
10. The **reduceByKey()** function is used to calculate the total count of flights per (model, manufacturer) for the airlines.
11. The combined RDD is then sorted using **sortBy()** and **sortByKey().**
12. Lastly, the results are printed to the correct format.

1.3.2 Optimised DataFrame API Implementation

1. The Flights, Airlines and Aircraft datasets are read in, and are cleaned.
2. Airline and flight are combined using the **join()** function on carrier_code.
3. The **drop()** function is used to drop airlines carrier_code to remove the duplicate and the remaining data has the function **filter()** applied to project only rows with country equaling to the user specified country.
4. This DataFrame is then transformed using the **groupBy()** function on carrier_code, name, country and tail_number with **count()**. This is done to get the flight count of different airlines in the specified countries.
5. Format the model name to only include 3 digits using the **withColumn()** function and using **regexp_extract()** to change the value of each instance on the aircraft DataFrame.
6. The previously transformed airline count DataFrame is combined with newly formatted aircraft DataFrame by using the **broadcast join()** function on tail_number.
7. This DataFrame has the **groupBy** function applied on name, manufacturer, and model_name, with the **sum()** function on the count column. This provides the total count of flights per (model, manufacturer) for the airlines.
8. The null values are removed from model_name and manufacturer by using the **filter()** and **isNotNull()** functions.
9. A **window** function is required to use the **rank()** function to determine the ranking of the models. The window function is used on the **partitionBy()** function**.**
10. The **partitionBy()** function is used to partition the DataFrame on the models, and it is ordered on the most used models by applying the functions **orderBy()** and **desc().**
11. The airlines are then ranked using **rank()** and **over()** to rank the airlines over the window. The **filter()** function is used to get the top 5, and the **sort()** function is used to sort the ranks from largest to smallest. Lastly, the output is printed.

1.3.3 Changes in execution plans

In referring to the DAG Visualisations of our unoptimised RDD implementation *(see Appendix 3.1)*, the bottleneck is in the PairwiseRDD join which takes 2.4 minutes to complete. This is a result of shuffle operations in Spark, and thus, it is recommended to keep shuffle operations at minimum by filtering earlier. This is attempted in our steps to filter and project early, however, to best optimise joins - we rewrite our task in DF to make use of broadcast join to avoid sending all the data of the large DF over the network, as join is a network-intensive operation.

In comparison, in our optimised DF implementation this particular stage containing the join of interest is split into 8 completed tasks with a median of 48s to complete each. Additionally, Spark's catalyst optimizer chooses a broadcast join *(refer to Appendix 3.3, Figure 3.3.3)* - broadcasting the smaller DF to cluster executors and evaluating the join condition with partitions of each executor to the larger DF. Thus, all the data in the small DF is sent to all nodes in the cluster, preparing a join without shuffling data in the larger DF. On reflection, by considering the size of intermediary results - this broadcast could be further optimised by reducing the number of rows before this join (i.e. filtering country).

1.3.4 Justification of Any Tuning Decisions

Converting task from RDD to DataFrame
DF use of Spark's catalyst optimizer is shown to greatly reduce the execution time in this task *(see Appendix 3.3, Figure 3.4.3)* when processing large datasets. Interestingly, DF is slower than RDD on the small dataset, however, as discussed within the performance evaluation - this is likely attributed to the overhead costs of setting up the DF named-column structure (although minimal) and in our own implementation of DF not optimally reducing the size of intermediary results between each transformation step.

Filtering and projection early
The DataFrames are reduced for smaller projections, leading to faster transformations and loading. Nevertheless, further filtering could be made to assist broadcast joins and overall task executions at each stage. This means considering the size of intermediary results at each transformation, and only selecting columns and filtering out rows that are necessary for the task. Modifying the implementation based on Spark's execution plan can assist in harnessing the appropriate number of threads and cores in the cluster for the task.

Broadcast Join
In joining aircraft_model and airlines_count we specify a broadcast join, and broadcast on the smaller DF aircraft_model DF. Thus, we avoid shuffling the larger DF, leading to less shuffling, and overall faster runtimes. This also reduces the chance of bottlenecks occurring in the network which inevitably slow down execution if not avoided.

2. Performance Evaluation

Results Table 2.1, and graphs 3.4.1, 3.4.2 and 3.4.3 in the appendix, show the average runtimes of each task on different dataset sizes, for each task recorded. The runtimes used were an average across execution times for each task implementation in table 2.1 below. Notably, across all tasks, our unoptimised implementations in RDD ran faster than all optimised versions in DF on the smaller datasets. This is due to RDD performing better on smaller datasets that can fit into memory - as opposed to DF which has slower performance rates at first due to initial overhead costs in organizing named columns similar to a relational database table.

Nevertheless, as the size of the datasets increase do our DF implementations begin to show it's benefit by taking advantage of Spark's catalyst optimizer in optimizing a query plan - resulting in a reduced rate of growth in runtimes. The initial overhead costs in setting up the DF structure is thus minimized by the optimizer's query plan in executing step-by-step transformations as we scale-out the size of the dataset. This is especially apparent in Task 3 *(see graph 3.4.3),* where our optimised DF implementation's rate of increase in runtime is lesser than that of the other tasks. This provides evidence to support that this query may also have very good scalability.

Against our expectations, our Task 2 optimised DF implementation ran slower on all dataset sizes in comparison to our RDD implementation. However, this may be unique to our own implementation of DF for Task 2 and not a reflection of the performance of DF in comparison to RDD. Upon reflection, we recommend improvements to take advantage of Spark's catalyst optimizer by being wary of the size of intermediary results between each transformation. That is, by minimising the outputs of each transformation to only the features that are required as inputs for the next transformation and overall task computation, we can reduce the multiprocessing load and make better use out of the number of threads and cores in our cluster at each step.

Thus, although DF is able to optimize query plans, we should not lazily rely on this and instead endeavour to make further optimisations by giving Spark's optimizer the necessary guidance.

Table 2.1: Execution time of the different tasks in RDD and DF on varying size datasets

Size	Task 1 RDD	Task 1 DF	Task 2 RDD	Task 2 DF	Task 3 RDD*	Task 3 DF	Task 3 DF opt
Small	2.34s 2.25s 2.03s 2.04s 2.14s	3.48s 3.42s 3.07s 2.65s 3.59s	2.25s 2.24s 1.84s 1.85s 1.74s	5.02s 4.19s 4.39s 3.82s 4.39s	4.42s 3.96s 4.17s 3.97s 4.17s	6.33s 5.61s 5.81s 5.94s 5.40s	5.82s 5.70s 5.51s 5.95s 5.74s
Average	**2.16s**	**3.24s**	**1.98s**	**4.36s**	**4.14s**	**5.82s**	**5.74s**
Medium	12.30s 12.71s 12.62s 12.16s	12.59s 12.53s 12.15s 11.92s	9.49s 8.94s 9.02s 9.11s	15.16s 16.20s 15.26s 15.28s	19.34s 18.78s 18.61s 18.97s	15.93s 16.35s 16.87s 15.62s	16.00s 17.01s 16.61s 16.21s
Average	**12.45s**	**12.30s**	**9.14s**	**15.48s**	**18.92s**	**16.19s**	**16.46**
Large	2.00m 2.00m 2.00m	1.48m 1.45m 1.46m	1.42m 1.46m 1.41m	1.90m 1.88m 1.89m	2.96m 3.04m 2.98m	1.65m 1.66m 1.64m	1.66m 1.64m 1.61m
Average	**2.00m**	**1.46m**	**1.43m**	**1.89m**	**2.99m**	**1.65m**	**1.64m**
Massive	15.96m 15.88m	11.59m 11.53m	11.42m 11.36m	14.90m 14.72m	32.53m 31.57m	12.15m 12.01m	12.95m 12.67m
Average	**15.92m**	**11.56m**	**11.39m**	**14.81m**	**32.05m**	**12.08m**	**12.81m**

3. Appendix

3.1 Task 1 Figures

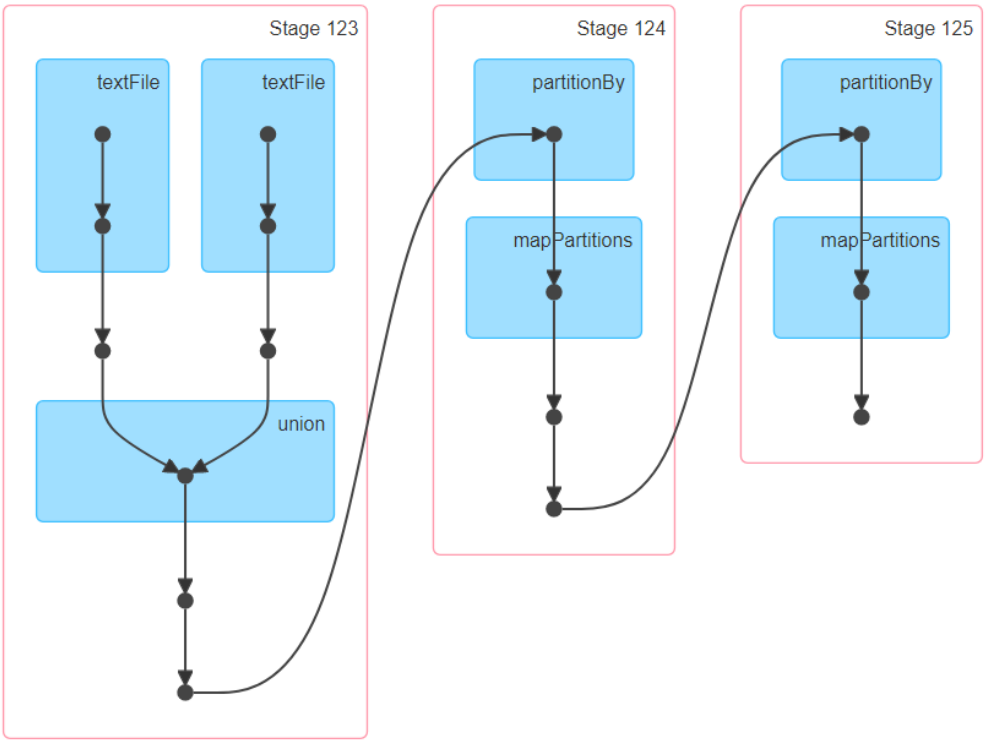

Figure 3.1.1 RDD DAG Visualisation for Task 1

▾Completed Stages (3)

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

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
60	2963530365940293614	stagemetrics.begin() task_1_rdd(spark, f"{dbfs... takeOrdered at <command-3137801540294900>:24 +details	2021/06/03 08:29:38	0.3 s	18/18			862.0 B	
59	2963530365940293614	stagemetrics.begin() task_1_rdd(spark, f"{dbfs... reduceByKey at <command-3137801540294900>:24 +details	2021/06/03 08:29:31	7 s	18/18			1874.8 KiB	862.0 B
58	2963530365940293614	stagemetrics.begin() task_1_rdd(spark, f"{dbfs... join at <command-3137801540294900>:21 +details	2021/06/03 08:27:37	1.9 min	18/18				1874.8 KiB

Figure 3.1.2 RDD Completed Stages for Task 1

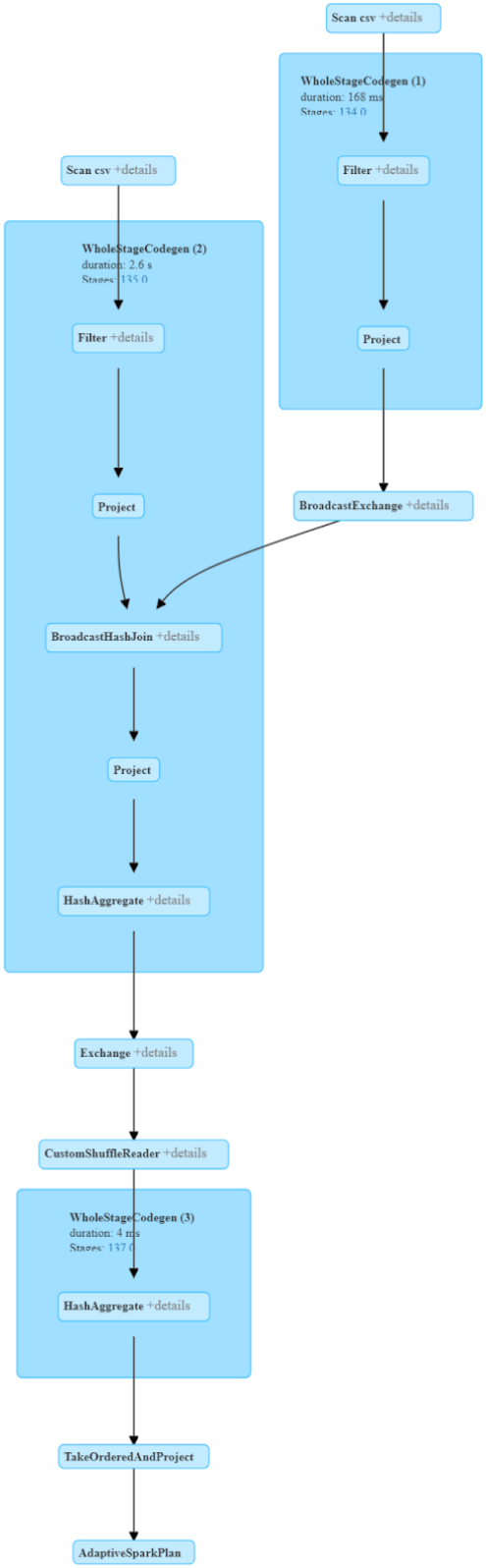

Figure 3.1.3 DF Query Execution Plan for Task 1

3.2 Task 2 Figures

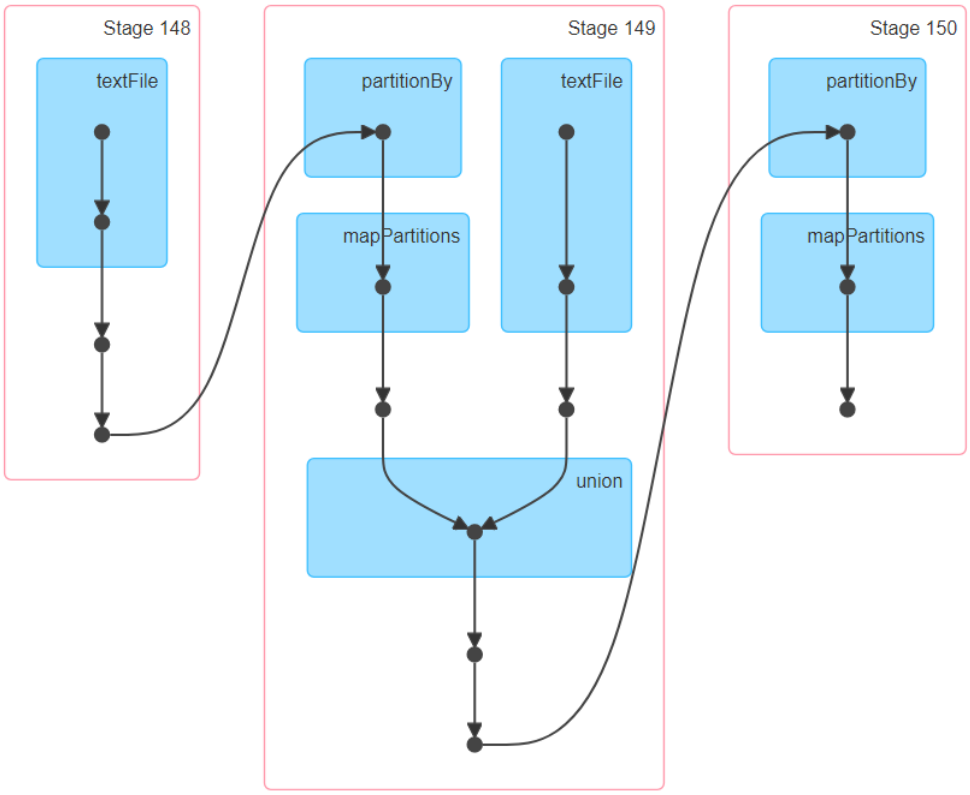

Figure 3.2.1 RDD DAG Visualisation for Task 2

Completed Stages (3)

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

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
60	2963530365940293614	stagemetrics.begin() task_1_rdd(spark, f"{dbfs... takeOrdered at <command-3137801540294900>:24 +details	2021/06/03 08:29:38	0.3 s	18/18			862.0 B	
59	2963530365940293614	stagemetrics.begin() task_1_rdd(spark, f"{dbfs... reduceByKey at <command-3137801540294900>:24 +details	2021/06/03 08:29:31	7 s	18/18			1874.8 KiB	862.0 B
58	2963530365940293614	stagemetrics.begin() task_1_rdd(spark, f"{dbfs... join at <command-3137801540294900>:21 +details	2021/06/03 08:27:37	1.9 min	18/18				1874.8 KiB

Figure 3.2.2 RDD Completed Stages for Task 2

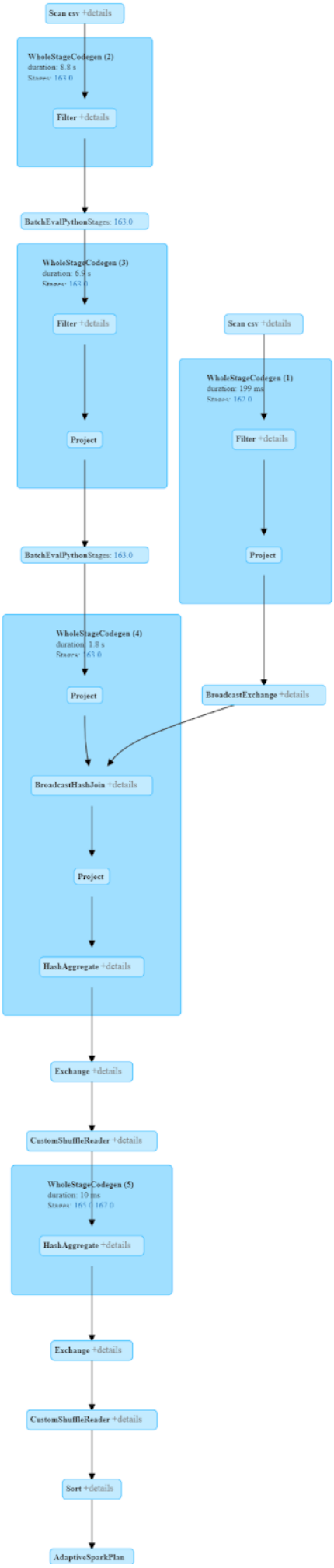

Figure 3.2.3 DF Query Execution Plan for Task 2

3.3 Task 3 Figures

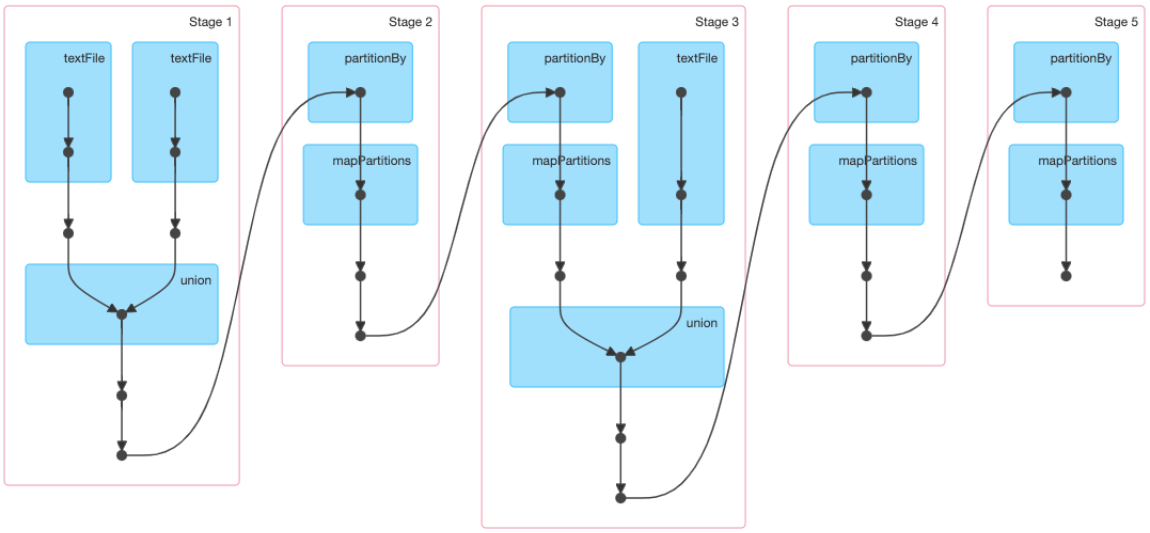

Figure 3.3.1 RDD DAG Visualisation Task 1 (on Large Dataset)

Completed Stages (5)

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

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	6458086796325186295	stagemetrics.begin() task_3_rdd(spark, f"{dbfs... sortBy at <command-3919452579252838>:75 +details	2021/06/03 06:40:05	0.5 s	20/20			113.9 KiB	
4	6458086796325186295	stagemetrics.begin() task_3_rdd(spark, f"{dbfs... reduceByKey at <command-3919452579252838>:69 +details	2021/06/03 06:40:04	1 s	20/20			399.6 KiB	113.9 KiB
3	6458086796325186295	stagemetrics.begin() task_3_rdd(spark, f"{dbfs... join at <command-3919452579252838>:63 +details	2021/06/03 06:40:03	2 s	20/20			237.1 KiB	399.6 KiB
2	6458086796325186295	stagemetrics.begin() task_3_rdd(spark, f"{dbfs... reduceByKey at <command-3919452579252838>:55 +details	2021/06/03 06:39:16	47 s	18/18			47.2 MiB	237.1 KiB
1	6458086796325186295	stagemetrics.begin() task_3_rdd(spark, f"{dbfs... join at <command-3919452579252838>:48 +details	2021/06/03 06:36:51	2.4 min	18/18				47.2 MiB

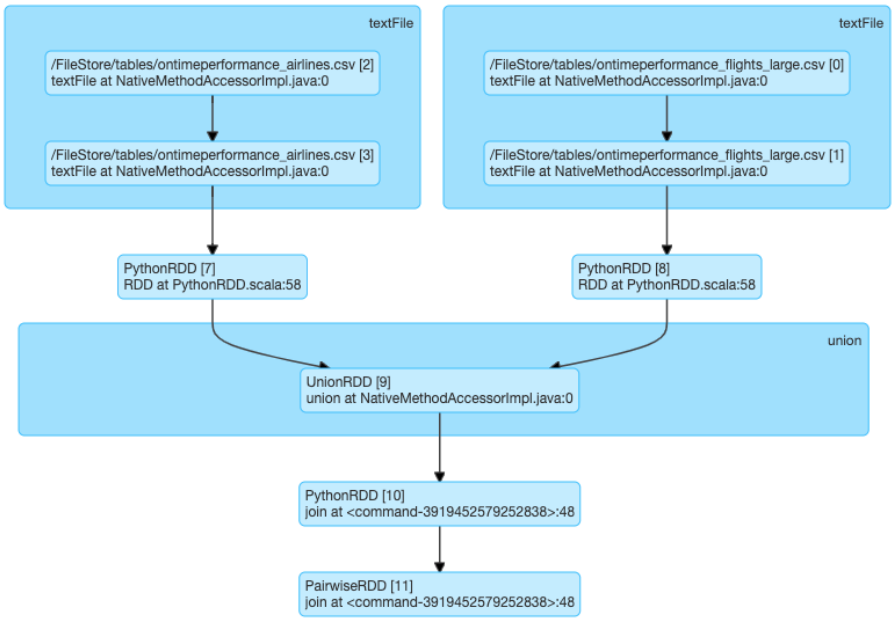

Figure 3.3.2 RDD Completed Stages for Task 3

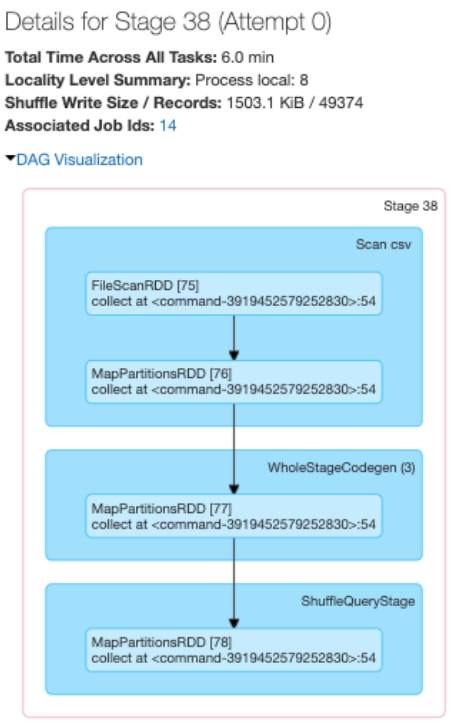

Figure 3.3.2 RDD Completed Stages For Task 3

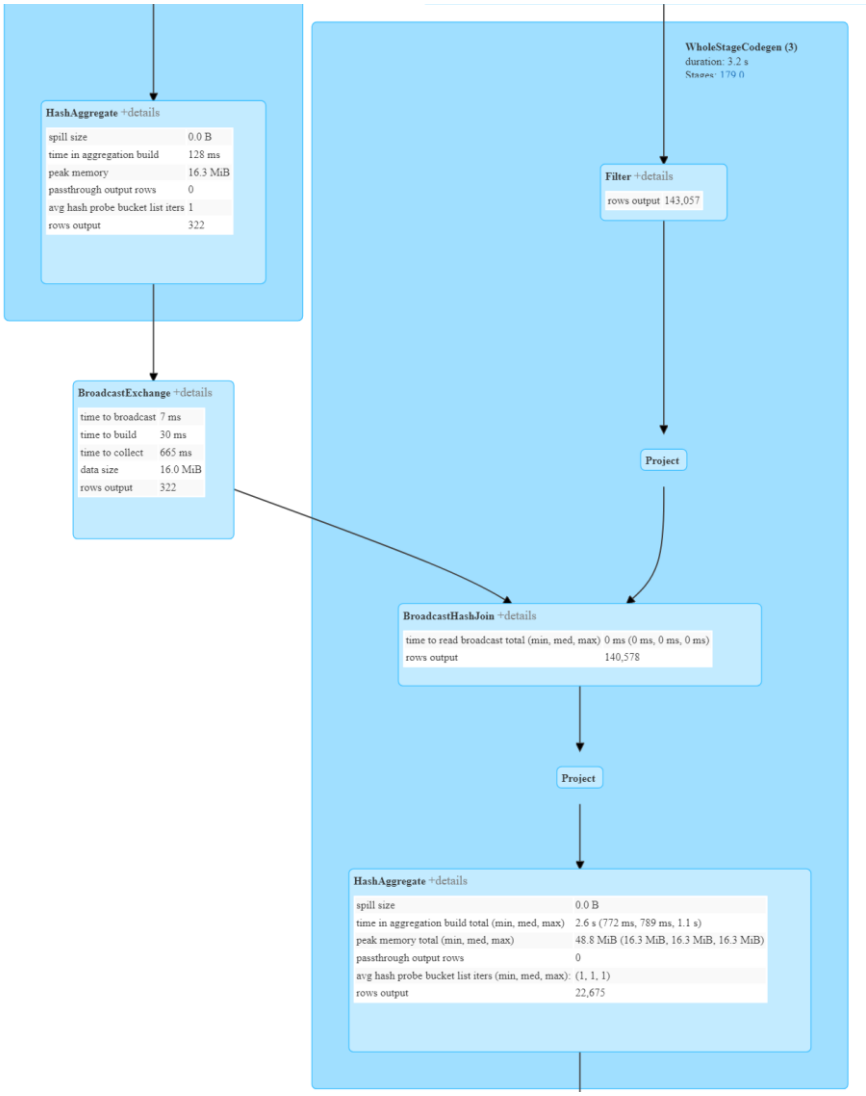

Figure 3.3.3 DF Query Execution Plan (join of interest) for Task 3

3.4 Performance Evaluation Graphs

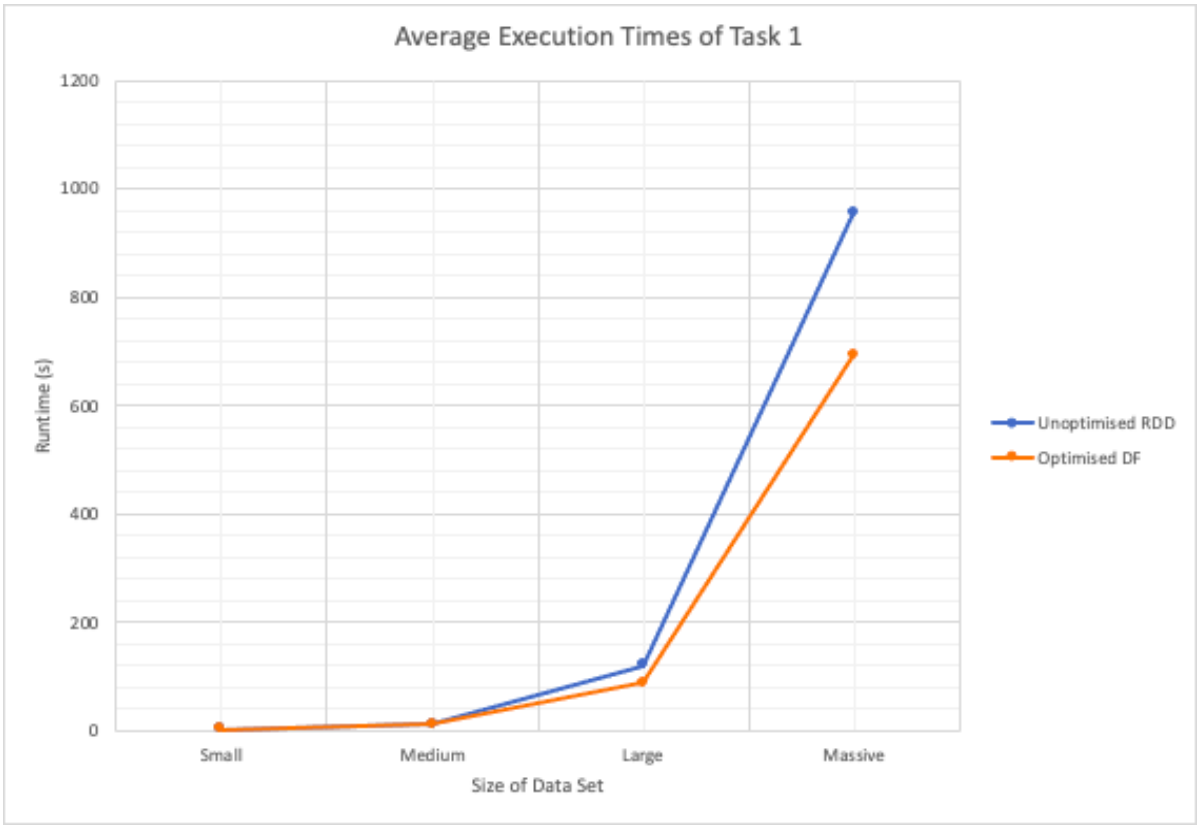

Figure 3.4.1 Average Execution Times of Task 1

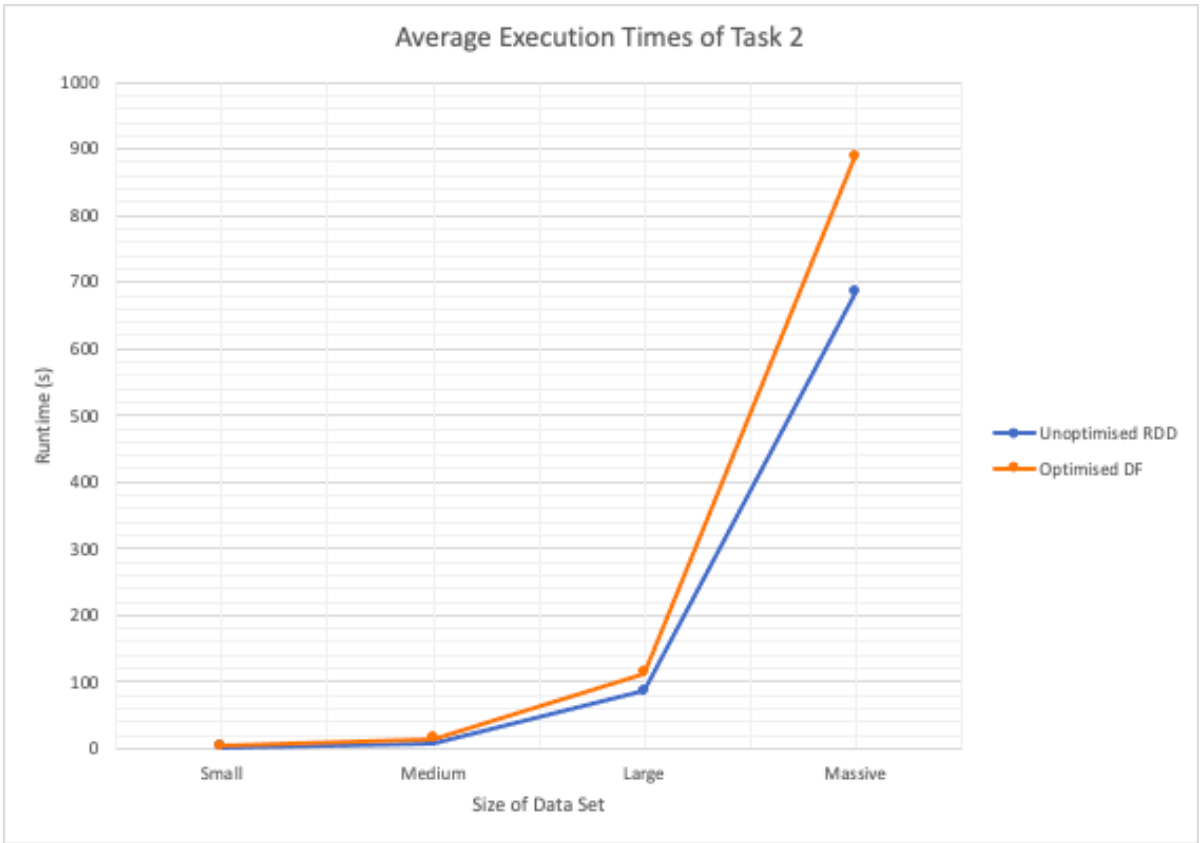

Figure 3.4.2 Average Execution Times of Task 2

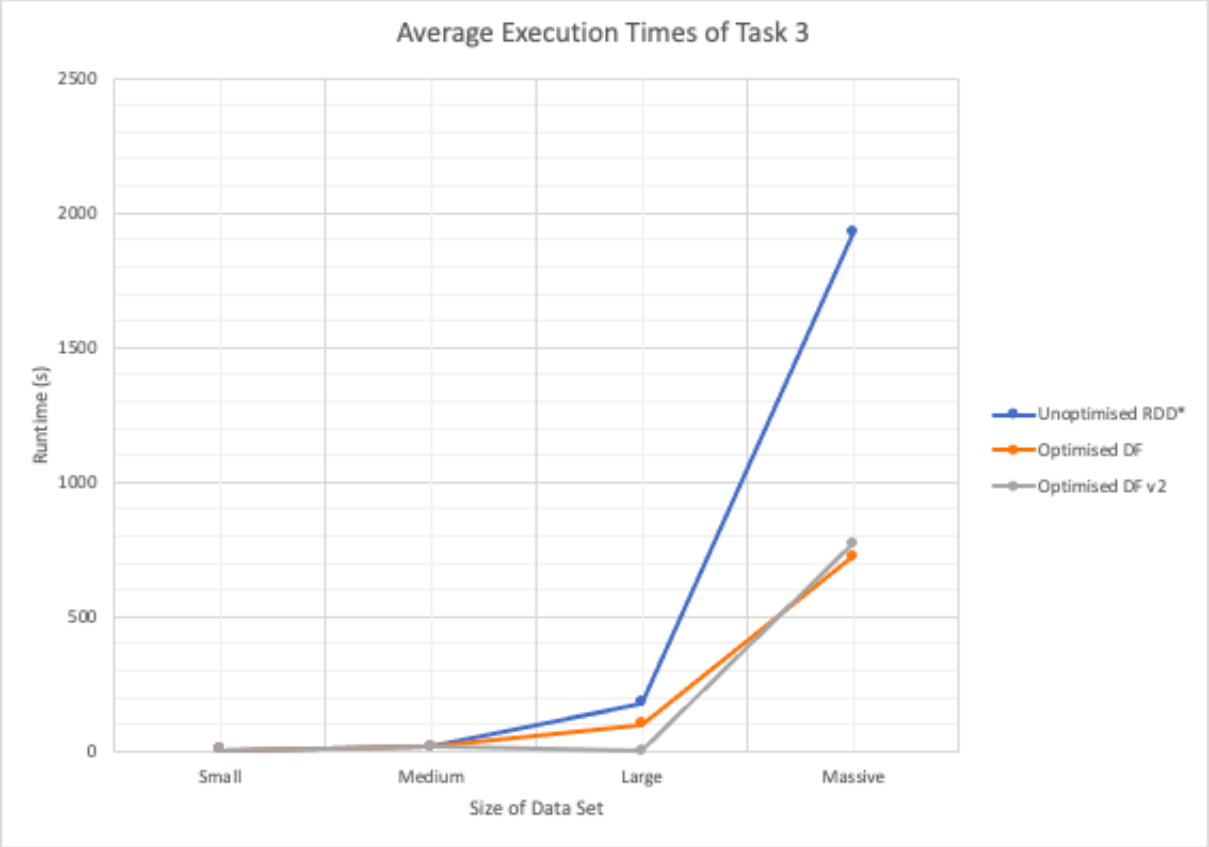

Figure 3.4.3 Average Execution Times of Task 3