Group 13

THE UNIVERSITY OF SYDNEY

School of Computer Science

ASSIGNMENT/PROJECT COVER SHEET - GROUP ASSESSMENT

Unit of Study: **DATA3404 Scalable Data Management**

Assignment Name: **Big Data Analysis Group Assignment (15%)**

Tutorial Time: **Thursday 12PM Tutorial 3**

Tutor Name: **Zachary Jin**

DECLARATION

I declare that I have read and understood the *University of Sydney Academic Dishonesty and Plagiarism in Coursework Policy*, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the *Academic Dishonesty and Plagiarism in Coursework Policy*, can lead to severe penalties as outlined under Chapter 8 of the *University of Sydney By-Law 1999* (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

Student Name	Student ID
Jacob Salway	480363228
Jonathan Then Tian Meng	490605077
Noy Hu	460043991

Task 1 – Top 3 Cessna Models

Job Design Documentation

Table Aircrafts and Table Flights are accessed for this task. There are four different datasets of Flights with "small", "medium", "large" and "massive" in the file name representing the size difference of these data. By changing the "size" variable, the dataset will automatically switch to the specific dataset.

Implementation:

1. Import the required functions from pyspark and spark Measure
2. Read in the Aircraft data and Flights data
3. Using Demo task providing function to clean column names of the two datasets
4. Filter Aircraft data to get all data with manufacturer == "CESSNA"
5. Using regexp_extract("model", \\d+, 0) to extra the (three) digits within the model using the regular expression and save it into new column named model_num
6. Concatenate the column manufacture and model_num to save it into new column named model_name
7. Merge the processed dataset Airlines and Aircrafts dataframes on the tail_number
8. Output should be groupby the model_name and count of each model_name appears, then sorted it in decent order
9. The first three data will become the final output represent the top-3 Cessna models
10. Write to DBFS file store with \t separator and UTF-8 encoding.

Optimisation/tuning

Refer to the task 1 table in appendix, there is a performance evaluation of the implementations for 4 different data sizes. Including executor Run Time, executor Cpu Time, shuffle Bytes Written and elapsed time.

In order to optimize the performance evaluation, I choose to performing the following steps to shorten the running time:

1. Filtering the CESSNA manufacture before joins
2. Only select talinum and model_name in the aircrafts data

By filtering the specific CESSNA manufacture we need in aircrafts table, it excludes the redundancy data in the merge table, which requires considerably high processing time. Only joining the dataset with only talinum and model_name in the aircrafts also helps reduce the space we use.

However, as you can see from the performance comparison in the appendix, these two methods only seem to make a slight difference. The running time after tuning still has slight advantages. This may mainly be caused by the Catalyst Optimiser within the Databricks system, which already did most of the optimization. As a result, the untuned method also seems to have a satisfactory evaluation.

Task 2 – Average Departure Delay

Job Design Documentation

There are two data tables that are accessed for this task, Airlines and Flights. For the Flights data, there are four data sets available. Changing the 'size' variable with String data type to "small", "medium", "large" or "massive" will substitute different numbers of rows in the Flights data. A variable called 'year' is also used to store the desired user-specified year.

Implementation:

1. Import the required functions from pyspark
2. Read in the Airlines and Flights data
3. Clean the column names of the two datasets using the function provided in the Demo Task
4. Merge the Airlines and Flights dataframes on the *carrier_code*
5. Filter airlines that are of United States origin
6. Split the *flight_date* and retrieve the year
7. Filter the rows to get all data points with the required year
8. Filter out cancelled flights by identifying NULLs in the *actual_departure_time* and *actual_arrival_time* columns
9. Convert the *scheduled_depature_time* and actual_departure_time into minutes and save it into new columns *scheduled_min* and *actual_min*
10. Find the time difference (in minutes) by taking the *actual_min* minus the scheduled_min and create a new column, 'delay', to store the result. There are some flights where the scheduled departure time is late at night and the actual departure time crosses over to the next day.

    Eg. Scheduled Departure Time: 2359, Actual Departure Time: 0010
    Scheduled Departure Time (min): 1439, Actual Departure Time (min): 10
    Delay (min): -1429

    For these instances, we designed a threshold value of -720 minutes. If the delay is **less than** -720 minutes like in the example above, we will add 1440 minutes to simulate it as the next day. Otherwise, the delay is kept. The results are saved in a new column called *final_delay*.
11. Select the required columns to perform aggregation over, *name* and *final_delay*
12. Filter out rows with no/negative delays from *final_delay*
13. Group by and order by US airlines as well as aggregate the delay to determine the count, average, minimum and maximum delay for each US airline in the user-specified year.
14. Write to DBFS file store with \t separator and UTF-8 encoding.

Optimisation/tuning

A performance evaluation was performed on the implementation of Task 2 with the four varying dataset sizes. Refer to the Task 2 tables in the Appendix. Recorded in the table is the elapsed time, execution run time, execution CPU time and the number of shuffle bytes written.

For optimisations, we decided to project the required columns and filter out the number of rows before merging the two datasets. In the Airlines data, we filtered out the US airlines and projected only the *name* and *carrier_code*. As for the Flights data, we filtered out the year, filtered the cancelled flights and removed those flights which were not delayed. Only the *carrier_code* and *final_delay* columns were projected. This resulted in the two dataframes having only two columns each before merging together. The idea is to remove columns that are unnecessary in the computation of our result as these columns will use up additional memory, space and input/output, resulting in our query taking a longer time to process.

The two datasets were then joined through a broadcast join, with Airlines data being the smaller dataset. If the broadcasted relation is small enough, broadcast joins will be fast, as they require minimal data shuffling. This would ultimately speed up the entire query process.

There does not seem to be much of a difference between the tuned and untuned implementations. This could be attributed to DataFrame API using the catalyst optimizer which creates a query plan resulting in better performance even though the query hasn't been tuned. The Community Edition account on Databricks is also a limitation as this free account is only able to access 2 core and 1 Databricks Unit.

Task 3 – Most Popular Aircraft Types

Job Design Documentation

My Spark job was implemented with the Spark DataFrame API. I will describe the DataFrame code for the unoptimised version below and touch on the optimisations and their effects on the physical plan in the next section.

Logical:

1. **Extract:** Load the CSV files (`flights`, `airlines`, and `aircrafts`) from the DBFS file store. Any spaces in the column names are removed.
2. **Transform:**
    a. `Flights` is inner joined to `airlines` by `carrier_code`, which is then inner joined to `aircrafts` by `tail_number`.
    b. Filter by `airlines.country` == country (parameter of function)
    c. Using a regex, remove any characters from `aircrafts.model` that come after the first sequence of three digits. If no three digit sequence is encountered, do not modify the string **(assuming this is correct based on tutor's comments)**.
    d. Grouping by `carrier_code`, `manufacturer`, and `model`, count the number of distinct `flight_number`.
    e. Using a window function, partitioning by `carrier_code` and ordering by number of distinct `flight_number` in descending order, generate a rank and filter to rank less than five.
    f. Grouping by `carrier_code`, collect the concatenation of `manufacturer` and `model` into a list structure, then convert to a string and surround with square braces.
    g. Join `airlines` by `carrier_code`, then select `name` and the serialised list, then order by `name` in alphabetical order.
3. **Load:**
    a. Write to DBFS file store with \t separator and UTF-8 encoding.

I also wrote a mostly complete RDD version of this job, however it performed so poorly and the RDD tuple structure was much more annoying to code for.

Optimisation/tuning

Since the DataFrame API uses the Catalyst Optimiser to perform operations such as projections to remove unneeded columns and pushing down filter or join predicates. However, I still specified these optimisations manually in my optimised version by:

1. Moving country filtering before the joins
2. Only selecting `carrier_code`, `flight_number` and `tail_number` for the flights file

Since both the airlines and aircrafts files are relatively small, I cached these dataframes in the executor memory and provided broadcast join hints for these dataframes.

For the massive flights file, I also implemented a repartition step after all the joins to try to improve parallelism, however the job ran slower and had a much higher shuffle write size. Normally, properly partitioning data will improve performance by allowing more parallelism.

However with only the single two core driver that Databricks provides, I suspect this benefit is moot and the exchange needed to shuffle the data provides no practical benefit while slowing down the execution time.Sadly, the optimisations did not have any groundbreaking effects on the query since most of the optimisations were already done by the Catalyst Optimiser. The main difference was that the airlines and aircraft files were read as an InMemoryTableScan rather than a scan of the CSV file, including for the airlines join near the end of the plan because of the caching (see Figure 3 and 4).

The joins were still completed with broadcast exchanges and hash joins since the airlines and aircraft tables are below the Spark auto-broadcast threshold.

Performance Evaluation

Tables and charts for performance related data in appendix.

For each data size, I measured the executor CPU time, executor run time, garbage collection time and shuffle bytes written. From Figure 1, it is obvious that that optimisations I made had little effect for small, medium, and large flight sizes, and because of the extra shuffle needed to repartition the dataset, actually made it slower for massive.

From Figure 2, it also appears that the job scales linearly with regards to flights row count vs executor CPU time. The job is most definitely bottlenecked by compute rather than shuffle/IO or memory. Garbage collection time was also not a bottleneck for this job (see Table 1).

Appendix

Task 1

small (untuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	3649	5642	1593	1279
2	3362	5232	1484	1279
3	3515	5874	1460	1279
4	3449	5614	1582	1279
5	3877	5944	1512	1279
average	3570.4	5661.2	1526.2	1279

small (tuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	3503	5231	1302	1279
2	3166	4783	1475	1279
3	3045	5134	1453	1279
4	3013	4871	1426	1279
5	2839	4521	1430	1279
average	3113.2	4908	1417.2	1279

medium (untuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	14644	90439	11268	2788
2	12066	80213	11167	2788
3	12486	81228	11681	2788
4	13121	86965	11312	2788
5	11737	77864	11299	2788
average	12810.8	83341.8	11345.4	2788

medium (tuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	12387	78815	11301	2788
2	11677	77164	11173	2788
3	12344	78606	11707	2788
4	11940	78964	11288	2788
5	11731	77765	11169	2788
average	12015.8	78262.8	11327.6	2788

medium (untuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	14644	90439	11268	2788
2	12066	80213	11167	2788
3	12486	81228	11681	2788
4	13121	86965	11312	2788
5	11737	77864	11299	2788
average	12810.8	83341.8	11345.4	2788

medium (tuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	12387	78815	11301	2788
2	11677	77164	11173	2788
3	12344	78606	11707	2788
4	11940	78964	11288	2788
5	11731	77765	11169	2788
average	12015.8	78262.8	11327.6	2788

medium (untuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	14644	90439	11268	2788
2	12066	80213	11167	2788
3	12486	81228	11681	2788
4	13121	86965	11312	2788
5	11737	77864	11299	2788
average	12810.8	83341.8	11345.4	2788

medium (tuned)	elapsed time	executor run time	executor cpu time	shuffle bytes written
1	12387	78815	11301	2788
2	11677	77164	11173	2788
3	12344	78606	11707	2788
4	11940	78964	11288	2788
5	11731	77765	11169	2788
average	12015.8	78262.8	11327.6	2788

Group 13

Task 2

Task 2	Elapsed Time (ms)			
	Small	Medium	Large	Massive
1	4247	14741	111591	839633
2	3862	15094	109251	837174
3	3737	15083	109631	835575
4	4456	15212	111614	832415
5	4545	14843	109488	834568
Average Before Tuning	**4169.4**	**14994.6**	**110315**	**835873**

Task 2	Executor RunTime (ms)			
	Small	Medium	Large	Massive
1	5592	92525	841797	6492603
2	5290	95062	824176	6476325
3	4779	93173	828210	6465949
4	5068	94662	840980	6443261
5	5713	94026	825002	6456587
Average Before Tuning	**5288.4**	**93889.6**	**832033**	**6466945**

Task 2	Elapsed Time (ms)			
	Small	Medium	Large	Massive
1	4179	14439	112283	837643
2	3877	14694	109789	835914
3	4011	14825	109249	834357
4	4297	14981	110643	828515
5	4246	14741	110929	832484
Average After Tuning	**4122**	**14736**	**110578.6**	**833782.6**

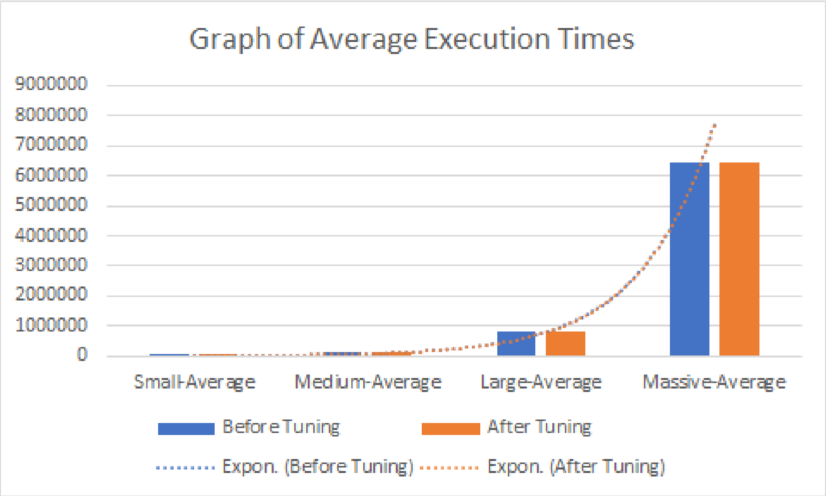
Task 2	Executor RunTime (ms)			
	Small	Medium	Large	Massive
1	5457	91694	846412	6474278
2	4778	92508	830333	6475542
3	4944	93163	824457	6453702
4	5422	93956	835418	6410211
5	5618	92593	838104	6438693
Average After Tuning	**5243.8**	**92782.8**	**834944.8**	**6450485**

Task 2	Executor CpuTime (ms)			
	Small	Medium	Large	Massive
1	1511	14749	143116	1115851
2	1498	14847	142015	1115923
3	1589	14744	143350	1113687
4	1500	14848	144514	1111871
5	1553	14780	143047	1115654
Average Before Tuning	**1530.2**	**14793.6**	**143208.4**	**1114597**

Task 2	Shuffle Bytes Written (Bytes)			
	Small	Medium	Large	Massive
1	1694	1715	1726	1751
2	1694	1715	1726	1751
3	1694	1715	1726	1751
4	1694	1715	1726	1751
5	1694	1715	1726	1751
Average Before Tuning	**1694**	**1715**	**1726**	**1751**

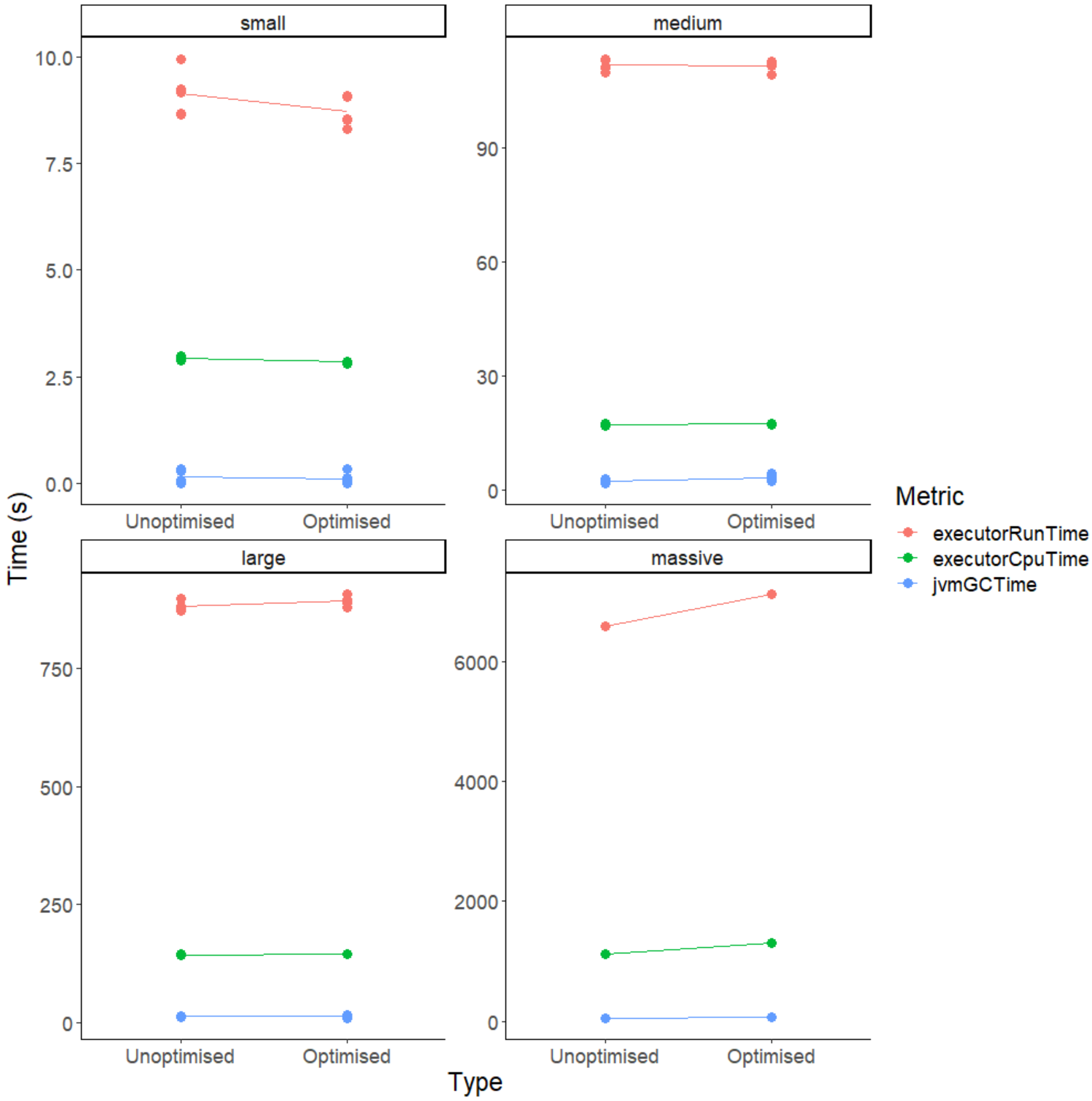
Task 2	Executor CpuTime (ms)			
	Small	Medium	Large	Massive
1	1534	14717	144930	1117164
2	1500	14701	143793	1116104
3	1529	14701	142273	1113652
4	1584	14769	144481	1113128
5	1481	14702	142782	1112198
Average After Tuning	**1525.6**	**14718**	**143651.8**	**1114449**

Task 2	Shuffle Bytes Written (Bytes)			
	Small	Medium	Large	Massive
1	1694	1715	1726	1751
2	1694	1715	1726	1751
3	1694	1715	1726	1751
4	1694	1715	1726	1751
5	1694	1715	1726	1751
Average After Tuning	**1694**	**1715**	**1726**	**1751**



Task 3

Figure 1



Group 13

Figure 2

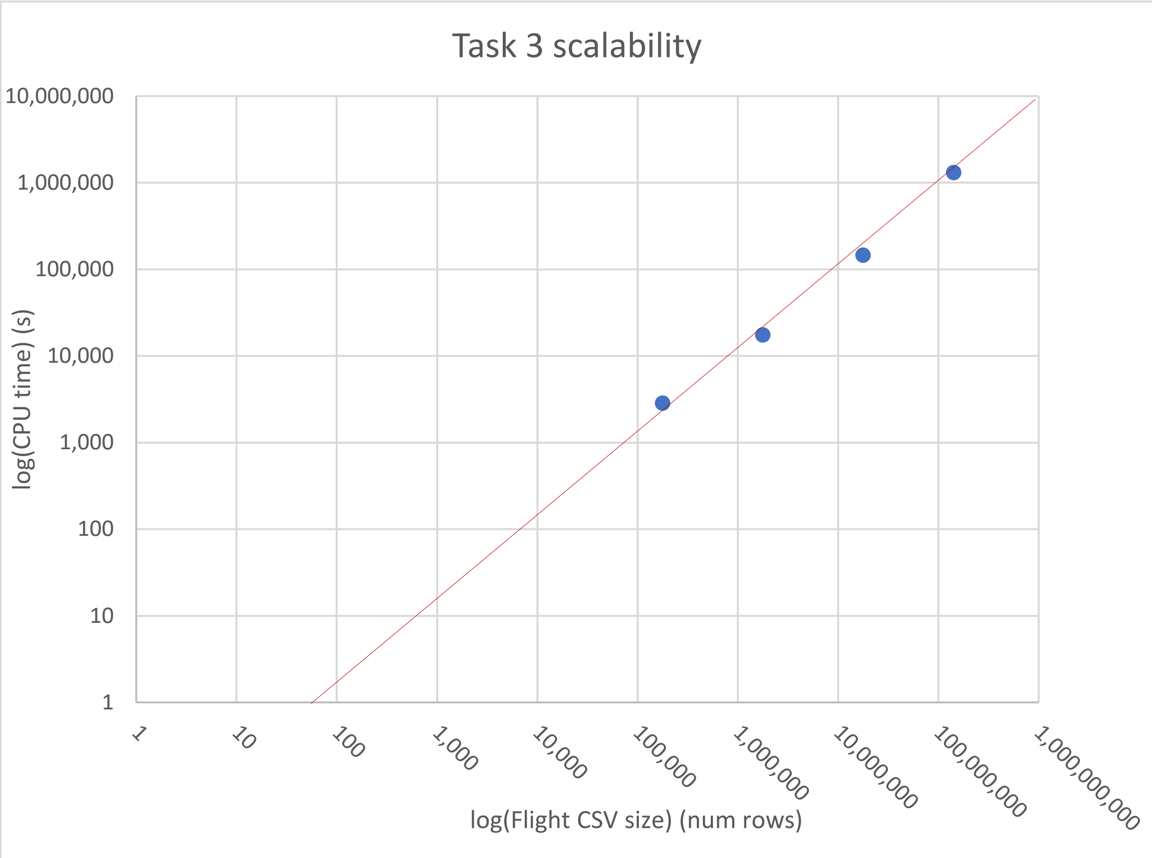


Figure 3

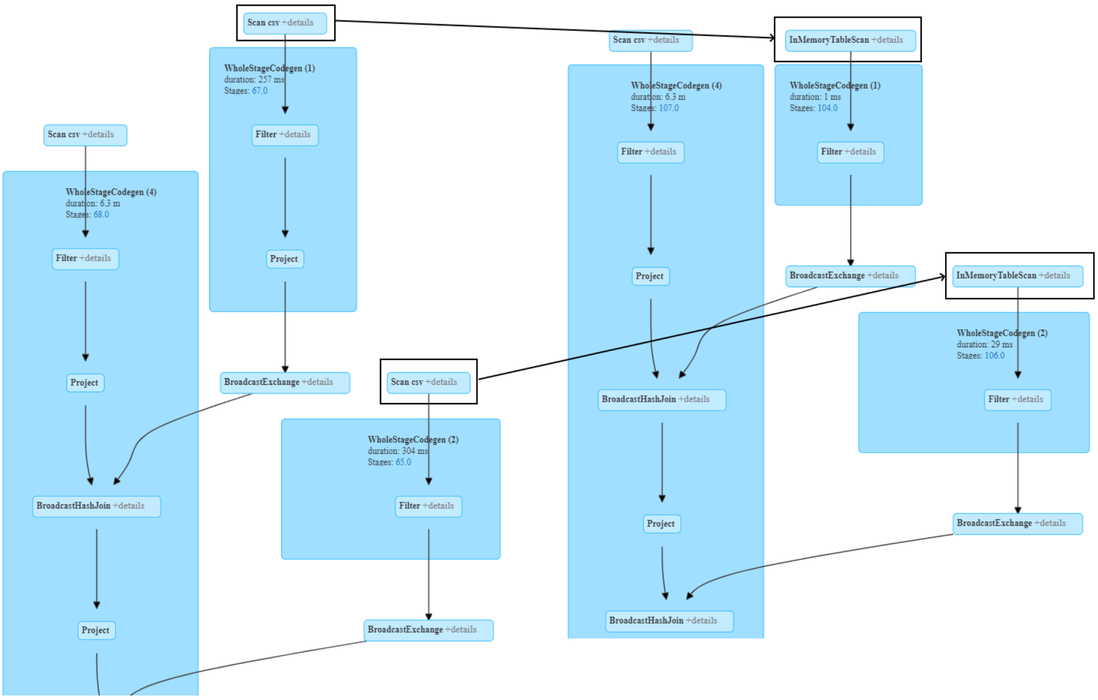


Figure 4

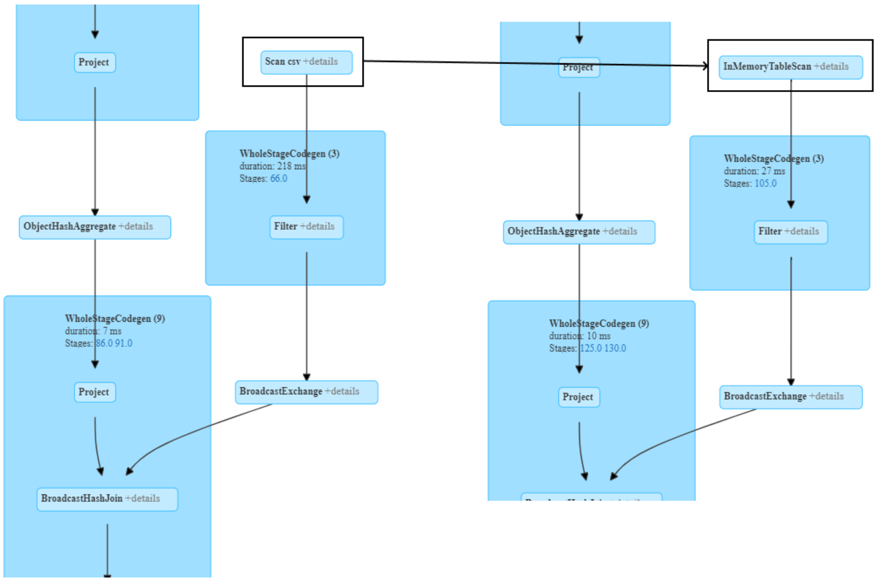


Table 1

Type	Size	executorRunTime	executorCpuTime	shuffleBytesWritten	jvmGCTime
Optimised	small	9073	2804	977746	0
Optimised	small	8529	2845	977746	130
Optimised	small	8319	2861	977746	65
Optimised	small	9098	2858	977746	0
Optimised	small	8556	2853	977746	327
Optimised	medium	112741	17408	4052678	3578
Optimised	medium	111613	17225	4052678	4352
Optimised	medium	112378	17622	4052678	3101
Optimised	medium	109284	17195	4052678	2459
Optimised	medium	111519	17287	4052678	2311
Optimised	large	896380	145248	9477850	16433
Optimised	large	907874	144121	9477850	12515
Optimised	large	891059	144758	9477850	13193
Optimised	large	880076	144445	9477850	10912
Optimised	large	887300	144518	9477850	9270
Optimised	massive	7134639	1300458	4.06E+08	61534
Unoptimised	small	9958	2910	1160092	333
Unoptimised	small	8647	2881	1160092	0
Unoptimised	small	9241	2989	1160092	69
Unoptimised	small	8665	2939	1160092	0
Unoptimised	small	9157	2924	1160092	291
Unoptimised	medium	111073	17122	4870666	2325
Unoptimised	medium	111415	17198	4870666	2824
Unoptimised	medium	109943	17040	4870666	1988
Unoptimised	medium	113444	17354	4870666	2205
Unoptimised	medium	112922	17339	4870666	1814
Unoptimised	large	897239	143089	11054495	13740
Unoptimised	large	875955	142652	11054495	11403
Unoptimised	large	880250	144216	11054495	12605
Unoptimised	large	880058	144070	11054495	12656
Unoptimised	large	872349	143605	11054495	11571
Unoptimised	massive	6599061	1125506	26065152	52977