DATA3404: Data Science Platforms

Big Data Tuning

Group Assignment

May 2019

Group Members:

Boris Margovski 460361334
Karan Goda 460496371

Job Design Documentation

Task 1

CSV tables for flights and aircrafts are read into DataSet tuples "*flights*" (1 column) and "*aircrafts*" (3 columns) respectively. The column read from the flights table is "*tail_number*" and the columns from the aircrafts table is "*tailnum*", "*manufacturer*" and "*model*".

The aircrafts dataset is filtered to include models manufactured by Cessna, which is implemented using the *FilterFunction*. The output is written to the dataset "*filtaircrafts*".

The flights joins with the filtered aircrafts on the condition that tail numbers are matching in both datasets, with the model number of the aircraft stored in a single tupled dataset "*joined_results*". "*Joined_results*" is then transformed to a two-tupled dataset using an implemented *FlatMapFunction*, by mapping every string X in "*joined_results*" to <first three characters of X, 1>. Let Y be the first three characters of X, hence <Y, 1>. The transformation is stored in the variable "*total_results*".

After the transformation, "*total_results*" is grouped by the first three characters of each model aircraft. The total sum of matches for an element of the first column of "*total_results*" is found by aggregating the second column of "*total_results*".

Then "*total_results*" is sorted, using *sortPartition*(1, Order.DESCENDING), in descending order based on the second column since the second column is the aggregate value for the number of Y in "*joined_results*". The .first(3) function implements the LIMIT function by storing the first 3 results from the transformed "*joined_results*" to "*total_results*".

Finally, "*total_results*" is written to a text file using the *writeAsFormattedText* function.

Task 2

CSV tables for flights and airlines are read into DataSet tuples "*flights*" (5 columns) and "*airlines*" (3 columns) respectively. The column read from the flights table is "*carrier_code*", "*flight_date*", "*expect_depart_time*", "*actual_depart_time*" and "*actual_arrive_time*" and the columns from the aircrafts table is "*tailnum*", "*manufacturer*" and "*model*".

The first step is to target a two-tupled dataset *usAirlines* and filter the airlines dataset so that *usAirlines* contains airlines exclusive to the US. The 2-column tuple "*flightDelays*" is reduced from flights to get all flight delays from all airlines of United States. This is implemented via the *usAirlinesReducer* function which applies the United States filter.

The "*yearAndDelayReducer*" function implements a filter that removes all flights that didn't occur in the year specified by the user. Furthermore, flights with empty strings in arrival, departure or scheduled time are also filtered out.

The third filter is if the delay is equal to or less than 0. If the time the flight left is earlier than the scheduled time, then there is no delay and thus flights associated with this condition are filtered out. Therefore, the *yearAndDelayReducer* outputs the "*flightDelays*" tuple with two columns: the airline code and the total delay.

We need to join results for all flight delays per airline on the "*airline_code*" column and transform into a 3-column tuple output to include the joined airline code, the airline name with the flight delay. This is done by implementing *JoinAirLinesFlights*.

The 3-tuple "*airlineFlightDelays*" is grouped by airline names and then transforming to a 5-tupled "result" dataset to include the average delays, the minimum and maximum time. This is implemented by the "*avgDelay*" which iterates through records of flights to find the maximum and minimum delay and output in minute(s).

The 5-tuple dataset "*result*" is outputted to a text file.

Task 3

CSV tables for airlines, flights and aircrafts are read into DataSet tuples "*airlines*" (3 columns), "*flights*" (2 columns) and "*aircraftDetails*" (3 columns). The columns read from the airlines table include "*carrier_code*", "*name*" and "*country*". The columns read from the flights table include "*carrier_code*" and "*tail_number*". The columns read from the aircrafts table include "*tailnum*", "*manufacturer*" and "*model*". The country typed in by the user is stored in the "*country*" variable. The default country is "United States" before taking in user input for the country.

countryAirlineReducer reduces the airlines dataset to a new dataset exclusive to airlines located in the country. The next step is to join the filtered airlines with the flights based on the airline_code key. *AirlineFlightJoin* is applied to the joining two-tuple pair by returning a two-tuple of the airline name and the tail number that used the airline, with the results stored in dataset *airlinesAndTailNumbers*.

The results of all the tail numbers associated to an airline joins with all rows in the aircrafts table that have matching tail numbers. The output of this join is stored as a 4-tupled dataset *airlinesAndAircraftDetails* and implemented by *ATNumberAndAircraftJoin*, with each 4-tupled element consisting of the airline name, a tail number found at that airline name, the manufacturer and model for that tail number.

airlinesAndAircraftDetails has to then be grouped by the tail_number since the program is searching for the most common manufacturer and model in terms of number of flights, with the tail number being the key in both flights and aircraft tables. The aggregating of distinct tail numbers is done by *TailnumberCounter*, where the input is the grouped *airlinesAndAircraftDetails* and the output is a 5-tupled element, with the 5th attribute being the total count of tail numbers. The dataset result is stored to *aircraftUsedCount*. It is then sorted alphabetically on airline name in ascending order, and then sorted on tail number in descending order.

Reducing *TailnumberCounter* to only 5 elements for each airline is implemented using the *FiveMostUsedReducer* function. *TailnumberCounter* is iterated through until a counter reaches 5 or there are no more tail numbers to iterate through for an airline i.e. there are less than 5 tail numbers for an airline. The output is a 2-tupled element; the first column is the airline and the second column is a list of mostly used aircrafts represented as a manufacturer-model tuple. The results are stored to dataset *aircraftUsedCountFive*.

aircraftUsedCountFive is then written out to a text file with a list iterator used for the second column of *aircraftUsedCountFive* to write the model/manufacturer tuple i.e. AIRLINE NAME, [MODEL1, MODEL2, etc].

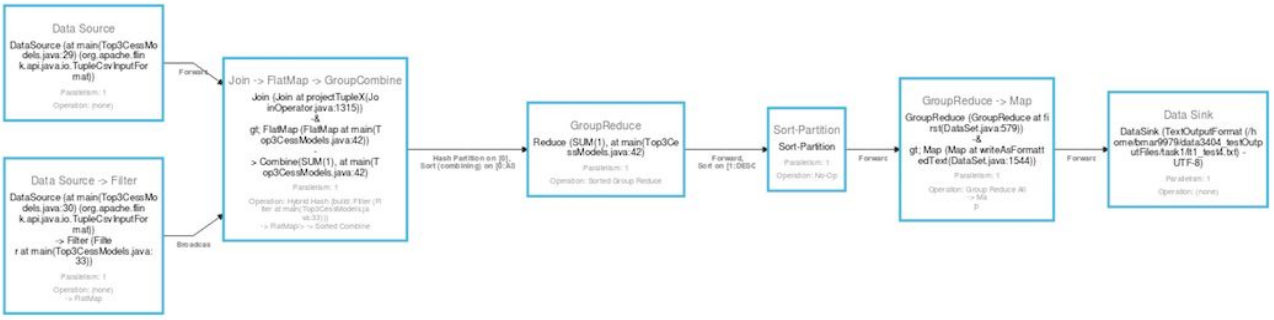
Tuning Decisions and Justifications

Task 1

Very minor tuning was applied since there is only 1 join required on two tables. The two main tuning methods are:

1. Filtering the aircrafts to only Cessna manufacturers, resulting in less runtime on the join
2. Loading the minimum number of columns from flights and aircrafts CSV table

Execution Plan For Task 1



Task 2

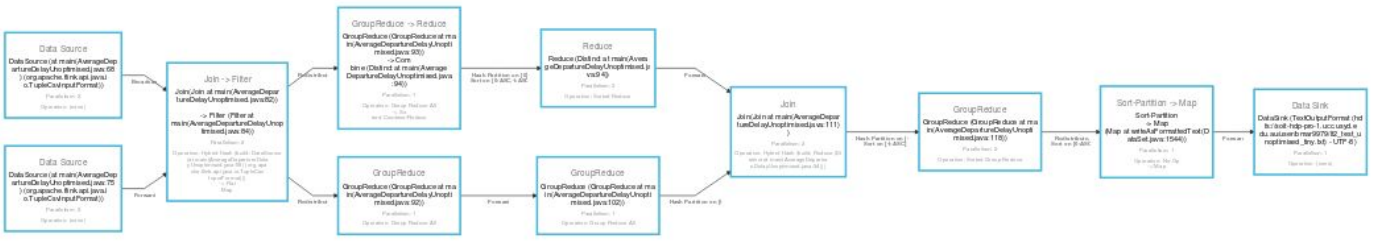
AverageDepartureDelayUnoptimised required a 7 tuple join on the airlines and flights table. This led to more I/O processes on fields that were not to be processed until the final result. In other words, more fields were unnecessarily loaded to memory resulting in poorer times. This is the case when filtering out airlines that weren't in the United States, since the same fields were rewritten to another 7 tuple element.

The changes made to optimise this scenario include:

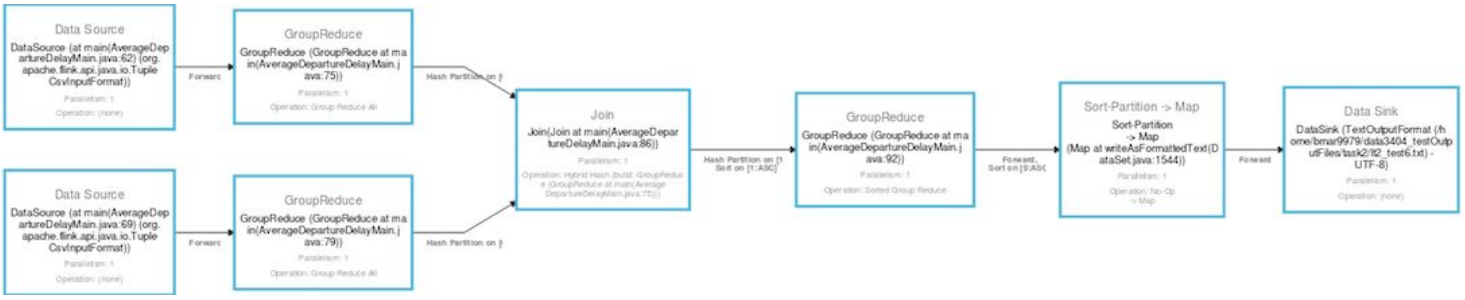
1. Reduce the number of columns written to a tuple to reduce join times. After loading the necessary columns from CSV files to airlines and flights table, apply the filters (*usAirlinesReducer* and *yearAndDelayReducer*) and write the results with the minimum number of columns required.
2. Execute a join function with as little number of columns as possible. This requires running the join function after filtering and not before. For the variable airlineFlightDelays in *AverageDepartureDelayMain*, a join function is made on two columns from both tables in comparison to seven columns in *AverageDepartureDelayUnoptimised*. This results in less data being written (two columns) and less I/O time taken.

Execution Plans For Task 2

Execution plan before optimising:



Execution plan after optimising

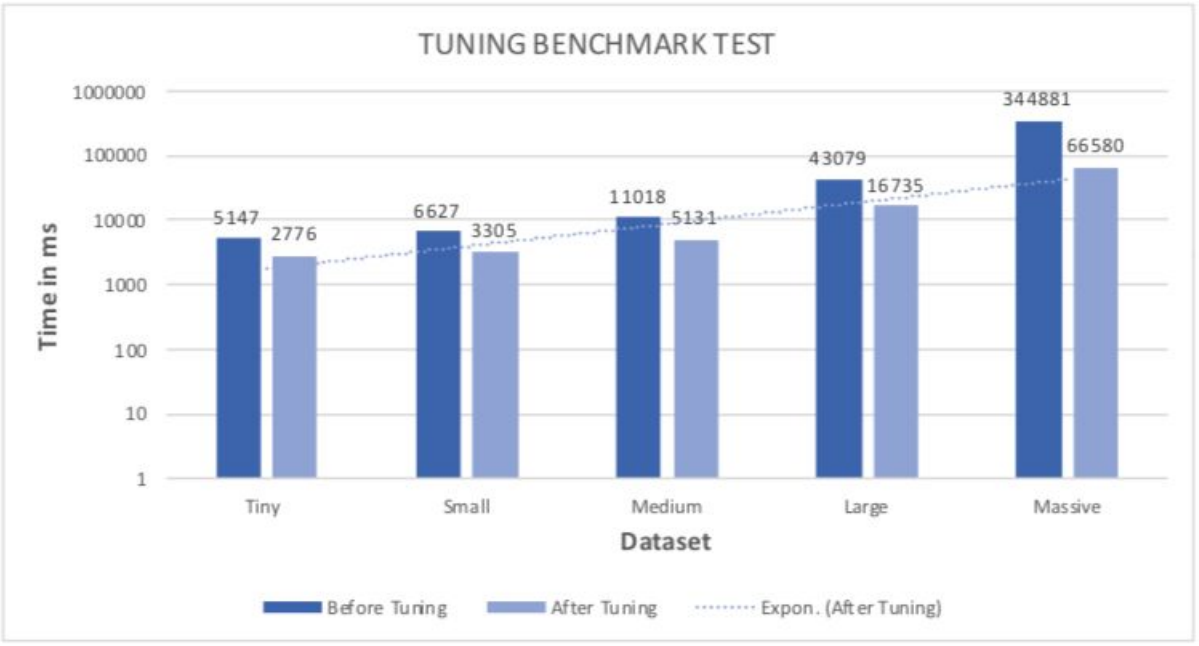


The changes before and after optimisation are very visible and we can see that there are less reductions involved once the task 2 query has been optimised.

Performance Improvement

Task 2 Benchmarks Table

Task 2	Year Selected is 2007				
(Using average ms)	Tiny	Small	Medium	Large	Massive
Before Tuning	5147	6627	11018	43079	344881
After Tuning	2776	3305	5131	16735	66580



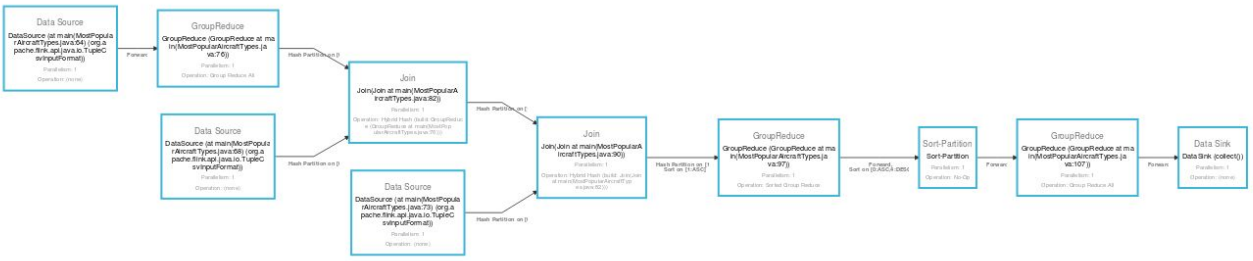
As we can see from the logarithmic scaled chart, there is an extremely large increase in the unoptimised query's time depending on file size as compared to the optimised task 2 query. This is most likely due to the join factor and reduction that task 2 efficiently does as compared to task 2 unoptimised which would increase the time to join.

Task 3

The main changes made to optimise performance was:

1. Reducing the number of columns required when executing a join function. Since the country has already been chosen via user input, there is no need to keep the "*country*" column of the airlines table when targeting variable "*countryAirlineReducer*".
2. Placing the join execution after the filtering out of airlines not associated with input country is done i.e. the joining of all airlines based on a country with the flights associated to the airline. This resulted in a more compact dataset stored in *countryAirline.* This was done to avoid computing a top 5 calculation on countries that was not chosen by the user resulting in an exponential improvement in runtime speed.
3. Since the only details that we need per flight is the tail number (as we are outputting results based on manufacturer and model of aircraft), the tail number is the only column written to "*airlinesAndTailNumbers*" as it is the only field required to make future joins from the "flights" table, resulting in smaller I/O time required.

Execution Plan For Task 3

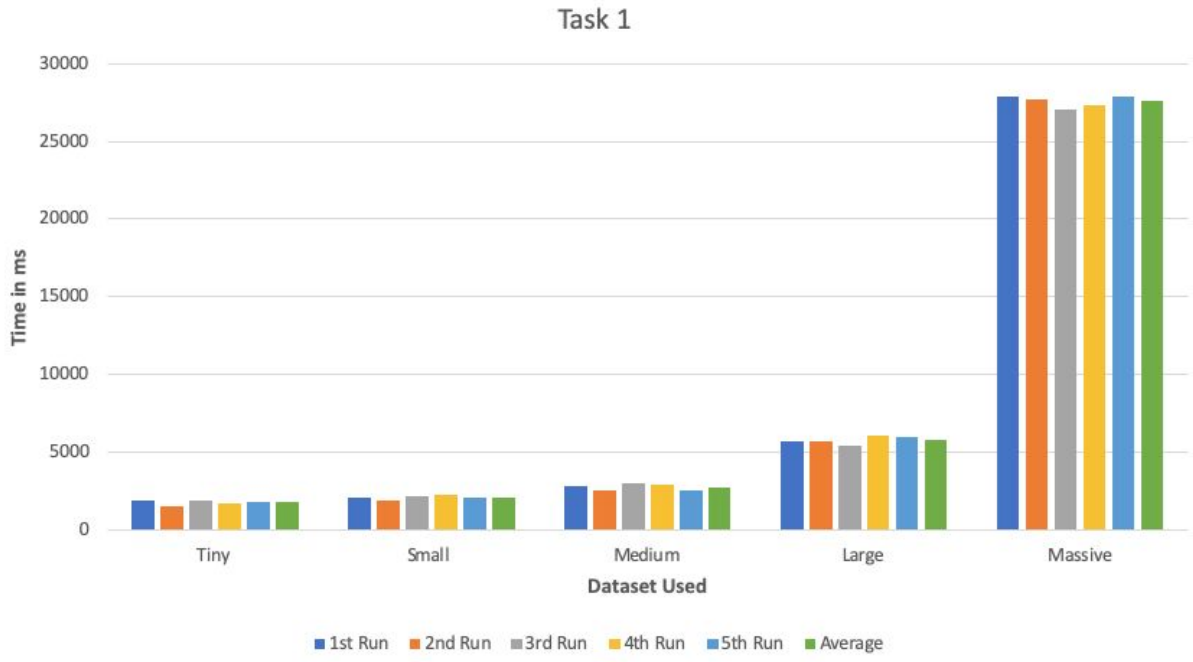


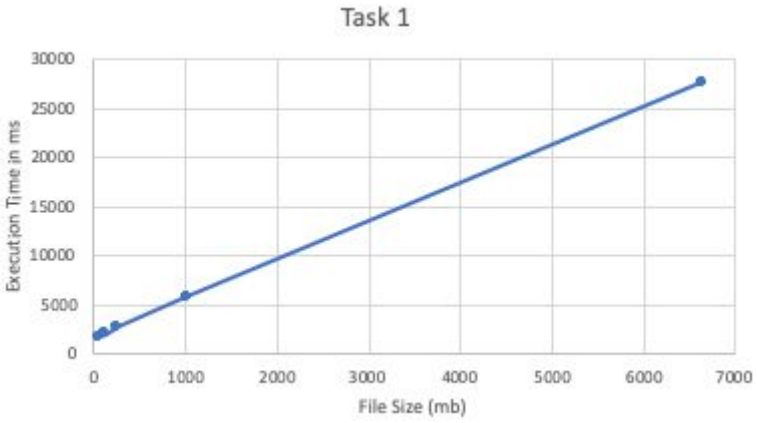
Performance Evaluation

Task 1

Very little difference in speed was seen between tiny, small and medium data sizes. Assume the average speed between the 5 runs for each data size is within the range of quickest to slowest. Therefore, the largest data size took twice as long to process as the other three data sizes.

Task 1	Tiny	Small	Medium	Large	Massive
1st Run	1838	2042	2803	5711	27855
2nd Run	1504	1856	2472	5645	27696
3rd Run	1899	2134	2958	5404	27102
4th Run	1664	2259	2868	6057	27343
5th Run	1741	2011	2556	5986	27898
Average	1729	2060	2731	5760	27578



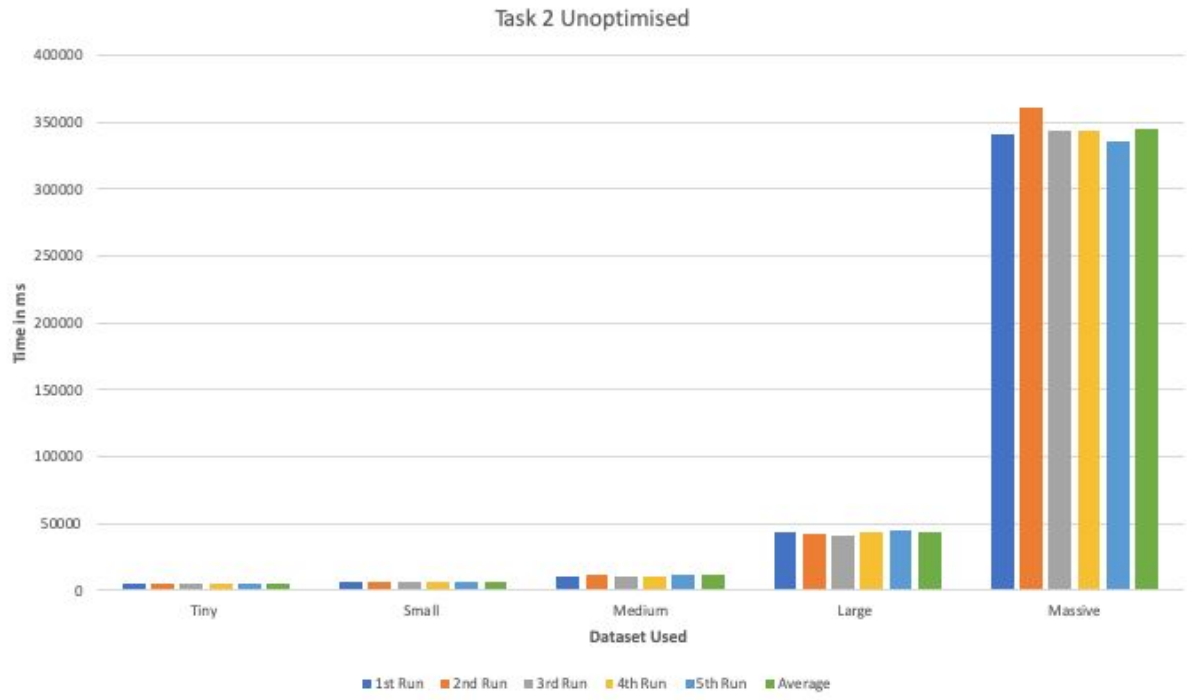


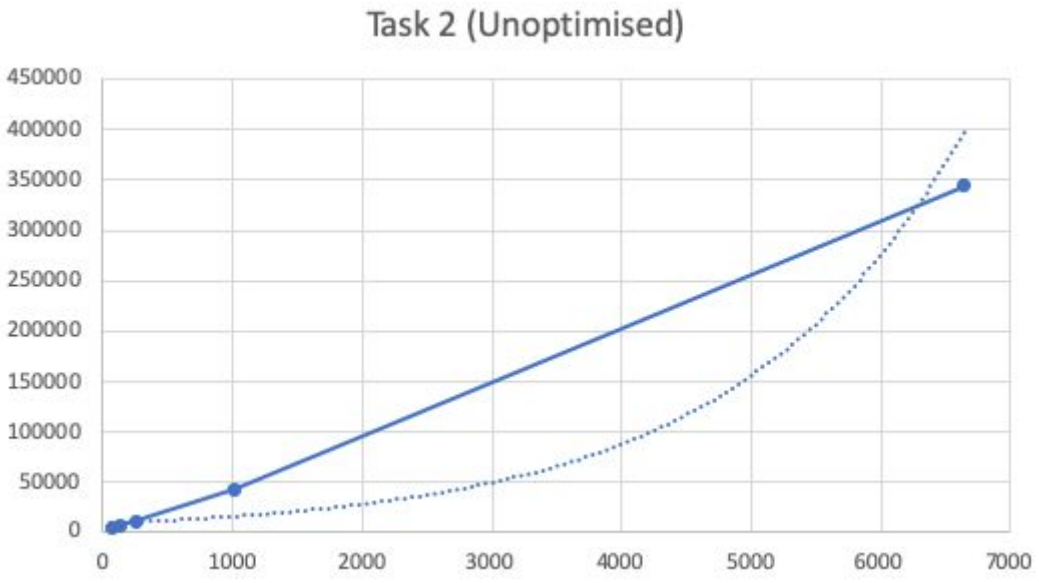
Observation: The program runs quite efficiently with the time scaling linearly with the input data size.

Task 2 Unoptimised

The difference in speed between tiny and medium datasets was just over two times. The difference between medium, large and massive is visible (massive takes 10 times as much as large and 30 times as much as medium), showing an exponential impact the datasets are having on runtime.

Task 2 Unoptimised	Tiny	Small	Medium	Large	Massive
1st Run	4467	6622	10782	43447	340770
2nd Run	5152	6537	11250	42504	360593
3rd Run	5386	6850	10879	41203	343019
4th Run	5611	6442	10675	43675	344192
5th Run	5123	6685	11504	44568	335831
Average	5147	6627	11018	43079	344881



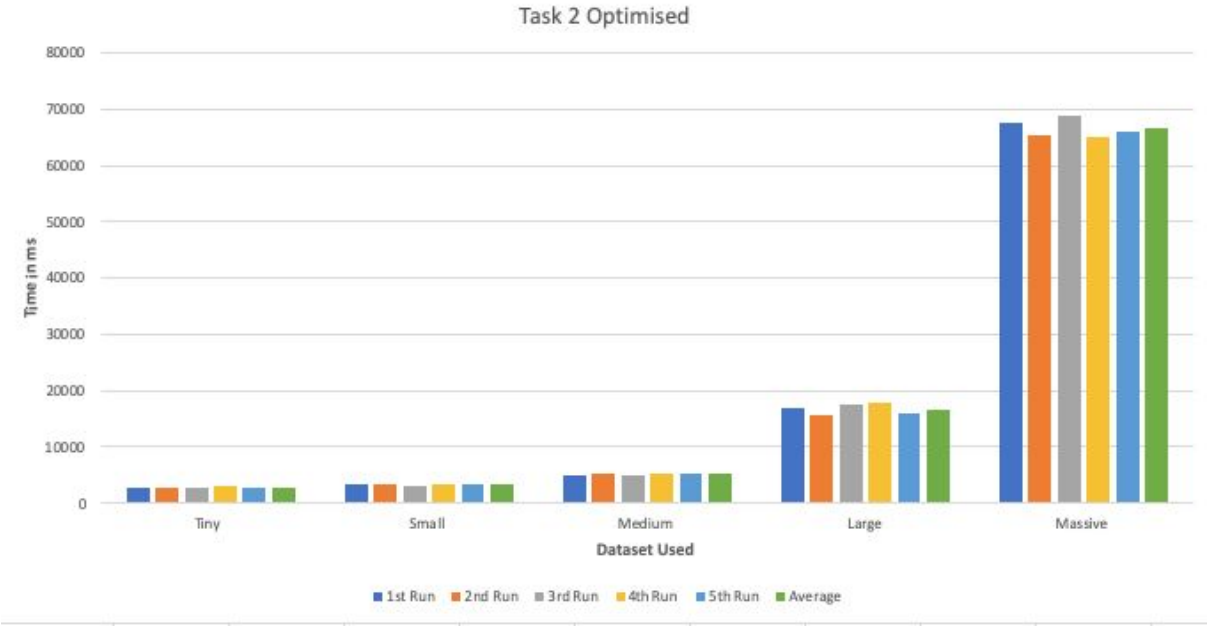


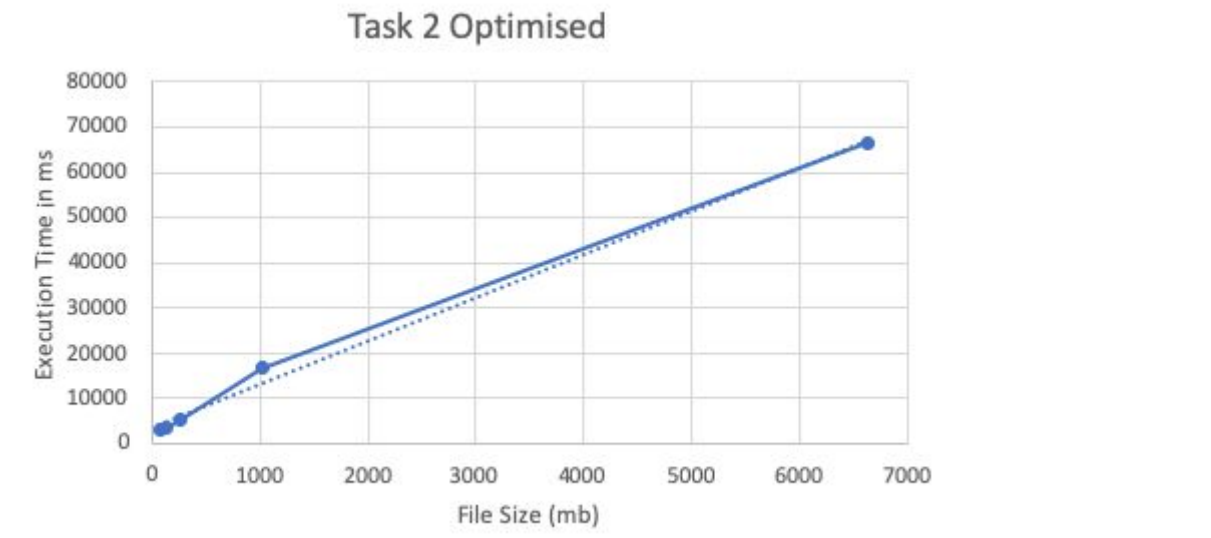
Observation: The program is extremely inefficient with the time scaling exponentially with the input data size.

Task 2 Optimised

The difference in speed between tiny and small datasets was non-existent, while medium had taken twice as much time as tiny. Large datasets took three times as much time as medium on average.

Task 2 Optimised	Tiny	Small	Medium	Large	Massive
1st Run	2694	3236	5026	16813	67551
2nd Run	2887	3408	5333	15581	65430
3rd Run	2593	3215	4869	17472	68786
4th Run	2911	3288	5177	17969	65010
5th Run	2799	3378	5253	15841	66123
Average	2776	3305	5131	16735	66580



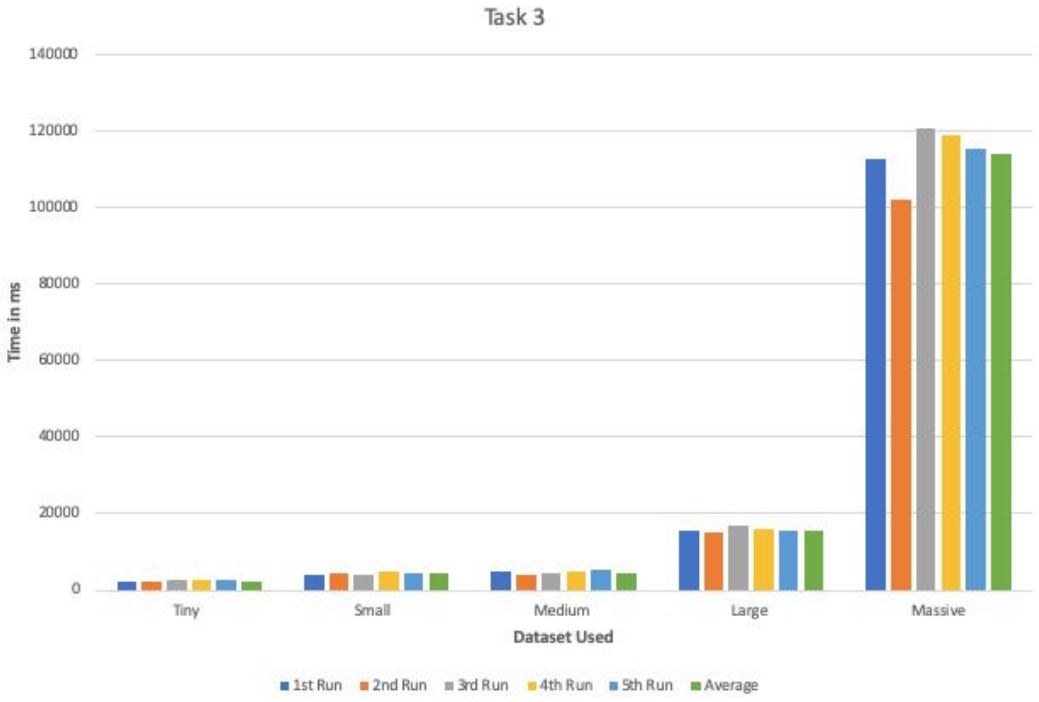


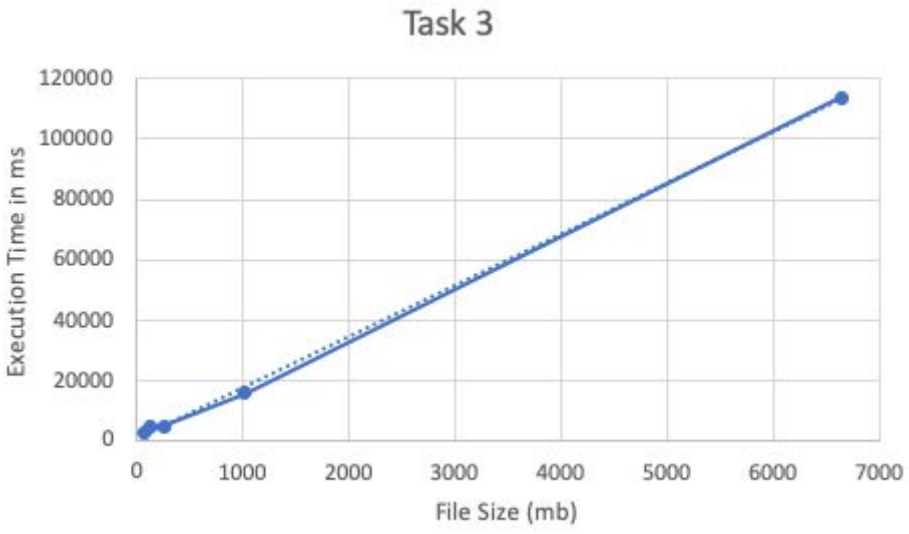
Observation: The program runs quite efficiently with the time scaling linearly with the input data size.

Task 3

The difference in speed between tiny and small datasets was just over two times, with no difference between small and medium datasets. Massive dataset takes 10 times as much as large and 30 times as much as medium, which is also shown in the performance output of Task 2 unoptimised.

Task 3	Tiny	Small	Medium	Large	Massive
1st Run	2273	4135	4801	15317	112648
2nd Run	2100	4566	4122	14881	102019
3rd Run	2454	4213	4356	16750	120596
4th Run	2686	4757	4675	15891	118675
5th Run	2509	4398	5122	15737	115492
Average	2404	4413	4615	15715	113886





Observation: The program runs quite efficiently with the time scaling linearly with the input data size.

Appendix

The HDFS location of the output files for our assignment.

Task 1:
/user/bmar9979/task1/
Task 2 (Unoptimised):
/user/bmar9979/
Task 2 (Optimised):
/user/bmar9979/task2/
Task 3:
/user/bmar9979/task3/

The execution plans and runtime excels can be seen in more detail in the assignment zip attachment.