DATA3404: Data Science Platforms

Big Data Tuning

Group Assignment

Group Members:
NAME: Callum Van Den Hoek
SID: 460500911
NAME: Haodong Chen
SID: 450541320

TABLE OF CONTENTS

JOB DESIGN DOCUMENTATION

Task 1

Only the *flights* file will be accessed in Task 1.
A variable named *targetYear* with default value "1994" will save the user specified year. The program accepts an optional argument to specify the year.
3 fields (*date*, *airport_code* and *departure_time*) from the *flights_tiny* file will be loaded into a dataset. The program accepts an optional argument to specify which *flights* file to use.

Then the dataset will go through 5 steps to produce the result dataset:

1. First, use the **reduceGroup()** function and a **YearReducer** object in which all tuples with *date* that does not match the *targetYear* or if the *departure_time* is empty (canceled flight) will be filtered out. After this step, the *date* attribute will be removed, and the resulting dataset will only have 2 fields left: *airport_code* and *departure_time*.
2. Next, group the data set by the *airport_code* with the **groupBy** function.
3. Then for each group, apply the **reduceGroup** function with an **AirportCounter** object to produce a single tuple that stores the *airport_code* and the number of records for the each *airport_code* in the group.
4. Then, a dataset with all the *airport_code* and corresponding number of flights will be formed. Sort the dataset by the number of flights in descending order.
5. Use the **first()** function to get the top 3 results from the data set.

The result is then output to a file named 'result.txt'. The output filename can be specified via an optional argument.

Task 2

In Task 2, two files (*flights* and *airlines*) will be accessed.
A variable named *targetYear* with default value "1994" will save the year. The program accepts an optional argument to specify the year.
Dataset *airline* will store tuples with 3 fields (airline_code, airline_name, airline_country), using data from the *airlines* file.
Dataset *flights* will store tuples with 6 fields (*airline_code*, *flight_date*, *expect_depart_time*, *expect_arrive_time*, *actual_depart_time*, *actual_arrive_time*), using data from the *flights_tiny* file, as default. The program accepts an optional argument to specify which *flights* file to use.

The datasets will go through 6 steps to produce the result dataset:

1. First, use **reduceGroup()** function with a **USAirlineReducer** object to select all airlines from the United States in the *airline* dataset.
2. Then use the **reduceGroup()** function with a **YearDelayReducer** object alongside with the *targetYear* to:
    a. Filter out all the flights that's not in the *targetYear*;
    b. Filter out all the flights that's canceled (no departure/arrival time);
    c. Calculate the delay for each flight. Note that if a departure delay or arrival delay is less than 0 (earlier departure or arrival), the delay will be set to 0;
3. Next, join the result datasets from last two steps by *airline_code* with a **JoinALF** object.

4. Then, group the resulting dataset from last step by *airline_name*.
5. For each group calculate the average delay with **reduceGroup()** function and a **avgDelay** object.
6. The results from the previous step will form a single dataset that has all the US airlines with their average delay. Sort the dataset by the *airline_name* in ascending order to produce the final result.

The result is then output to a file named 'result.txt'. The output filename can be specified via an optional argument.

Task 3

For Task 3, three CSV files are accessed: *flights*, *airlines* and *aircrafts*. The program accepts an optional argument to specify which *flights* file to use.
From these files, the following initial data sets are created:
Airlines: A set of tuples with three fields – *carrier_code, name,* and *country.*
Flights: A set of tuples with two fields – *carrier_code* and *tail_number.*
Aircraft: A set of tuples with three fields – *tailnum, manufacturer* and *model.*

The next step is to reduce the *airlines* data set into one which contains only US airlines. This is achieved using a custom class named *USAirlineReducer*, which implements Flink's *GroupReduceFunction* interface. Within this class there is the *reduce* function, which takes an iterable set of Tuples (in this case, the *airlines* data set), iterates over each Tuple and adds each Tuple containing a US airline to the new data set which is output as *usAirline.*

The *usAirline* data set is then joined (using Flink's *join* transformation) to the *flights* data set using their *carrier_code* fields as the join condition. A new data set is then created by the class *JoinALF*, which implements Flink's *JoinFunction* interface, taking both data sets and outputting the joined data set *airlineTailNumbers*, which contains Tuples with the fields *name* from the *usAirline* data set and *tail_number* from the *flights* data set.

After this, the newly created *airlineTailNumbers* data set is joined to the *aircraft* data set, using the same approach as the previous step. The join condition in this case is the *tail_number* field. The joined data set is input into the class *JoinALC* which is again similar to the previous step in that it creates a new data set *aircraftDetails*, this time containing Tuples with the fields *name* and *tail_number* from the *airlineTailNumbers* data set, as well as *manufacturer* and *model* from the *aircraft* data set.

The data set *aircraftDetails* then undergoes the following operations:
1. The Tuples are grouped by *tail_number*,
2. The groups are reduced using Flink's *reduceGroup* function, along with a custom class *AircraftCounter*, which implements the *GroupReduceFunction* interface. This class contains an overriding *reduce* function, in which each Tuple is iterated over and a counter is created for each unique tail number. This counter is then included in a new data set containing Tuples with the same four fields found in *aircraftDetails* as well as an additional field containing the count for each tail number. The decision was made to exclude null tail numbers.
3. The resultant data set is sorted alphabetically by airline name (ascending) using Flink's *sortPartition* function,

4. The data set is then sorted in descending order by tail number count, again using *sortPartition*.

The result of these operations is a data set *countResult*, which is sorted by airline, and for each airline sorted from most used to least used tail number.
The *countResult* data set is then reduced using Flink's *reduceGroup* function along with a custom class *AirlineReducer*. This class implements Flink's *GroupReduceFunction* interface and iterates over *countResult,* first creating a Tuple containing a string field for the airline name, and a new ArrayList. It then adds to the ArrayList the *manufacturer* and *model* fields of the first five Tuples (five most used tail numbers) from the given airline. Once five Tuples have been added, the Tuple containing the airline name and the ArrayList is added to the output data set and a new Tuple is created for the next airline, continuing until all the airlines have been iterated over. The final data set is output as *reduceResult.*
Finally, *reduceResult* is converted to a List and input into the custom function *saveResultsToFile*. This function uses methods from the java.io and FileUtils libraries to create an output file and write to it the data from *reduceResult*, formatting it according to the assignment brief. The data is also output to the Hadoop cluster via the *writeAsFormattedText()* function, with the default filename 'result.txt'. The filename can be specified via an optional argument.

Tuning Decisions and Justifications

Task 1

Task 1 only involves 1 set of data (from various flights files). It has no join operations and is not computationally heavy thus not much tuning can be applied in this task. However there are still some minor optimisation choices which been made, for example, at the beginning, filter the dataset by year in order to reduce the dataset size for later steps.

Task 2

For task 2, besides the minor optimisations, a major tuning decision made was to reduce the number of tuples before joining two datasets.

Before the tuning, when the *airline* dataset joins with the the *flights* dataset, they will produce a dataset storing tuples with 7 fields. This is not ideal because the size of the output dataset will grow exponentially when the size of input data increases.

After the tuning, the *flights* dataset will get reduced from a dataset storing tuples with 6 fields, into tuples with just 2 fields (*airline_code*, *delay*). And now when joining the two datasets, there is much less data to deal with for the **join()** function, and the size of the output tuple is also much smaller. Thus we should see some performance improvement.

We also tried the joinWithHuge()/joinWithTiny() functions when joining two datasets, but it didn't make much difference, therefore we just kept using the regular join function.

Execution Plans

Image 2.2.1: Execution plan before Tuning

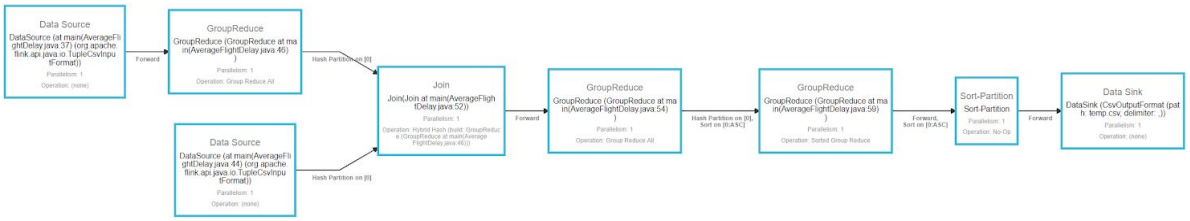
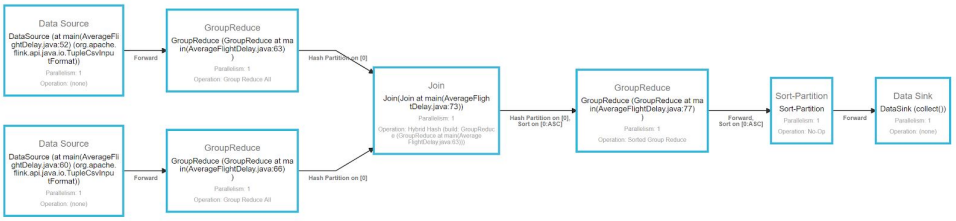


Image 2.2.2: Execution plan after tuning

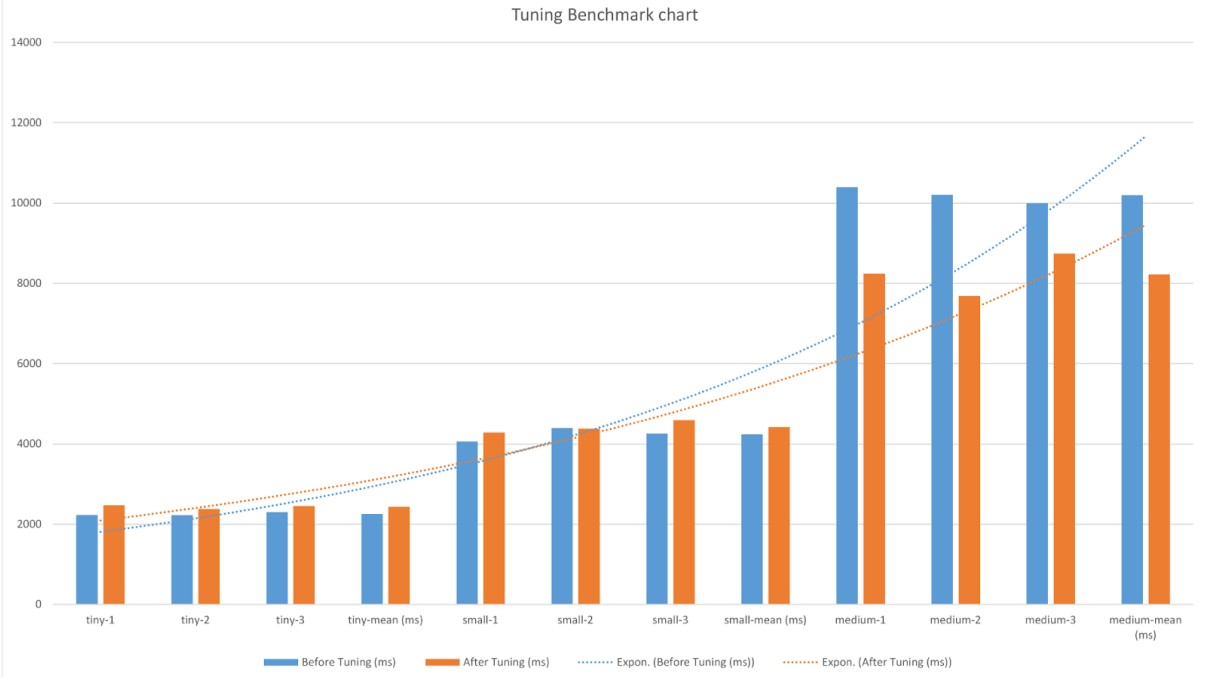


As shown in the two execution plans, the change is very obvious, the group reduction process that happened after the joining has been moved to before the joining operation.

Performance Improvement

Image 2.2.3 Benchmarks:

Year: 1994												
Dataset	flights_tiny.csv				flights_small.csv				flights_medium.csv			
Iteration	tiny-1	tiny-2	tiny-3	tiny-mean (ms)	small-1	small-2	small-3	small-mean (ms)	medium-1	medium-2	medium-3	medium-mean (ms)
Before Tuning (ms)	2230	2222	2298	2250.00	4059	4397	4256	4237.33	10394	10204	9994	10197.33
After Tuning (ms)	2471	2377	2450	2432.67	4283	4378	4589	4416.67	8240	7682	8740	8220.67



As shown in the graph, the difference is very clear: with the increasing size of input data, the execution time before tuning slows down further and further when compared with the one after tuning. But an unexpected discovery is that when the input data size is small, the execution time before tuning is shorter than after tuning. This might due to the join function being able to deal with small datasets efficiently, and reducing the dataset beforehand will just increase the overhead. Therefore for optimal performance, a file size threshold can be

set, to choose which version to use. But in general, the one after tuning would be a better choice.

Task 3

For Task 3, there was a similar decision made to alter the data source retrieved from the source *flights* file. Prior to tuning, a data set would be created from the specified *flights* CSV file, with the fields *flight_id, carrier_code* and *tail_number*. The decision was made to omit the *flight_id* field, reducing the amount of fields in each tuple to 2 and thus reducing the amount of data to include.

Additionally, a change was made to the ordering of the steps in the execution plan. Prior to tuning, the program would run each operation on the data set of all flights, until the final step where it would reduce the data to only those flights by US airlines. To improve this, the data is reduced to only US airlines at the beginning of the program, and then the subsequent operations are carried out on this reduced data set. This has the effect of vastly reducing the amount of tuples that the program had to carry out operations on. In addition, each tuple for the data set would have one less field - as the *country* field is no longer required after the US airline reducer step is carried out.

See appendix for execution plans, benchmarks and graphs.

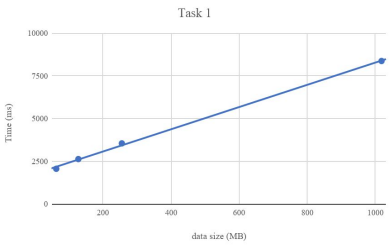
The difference is minor but still noticeable. As with the results of the tuning for Task 2, it can be seen that, as the input data size increases, the execution time increases less post-tuning than pre-tuning. Again, also as with Task 2, the pre-tuning program processes smaller data sets faster than the post-tuning program. This could be due to the way that some of the .join() and .groupBy() operations handle small data sets. Overall, the post-tune program would be more ideal for most conditions as the data sets will generally be larger in this type of program.

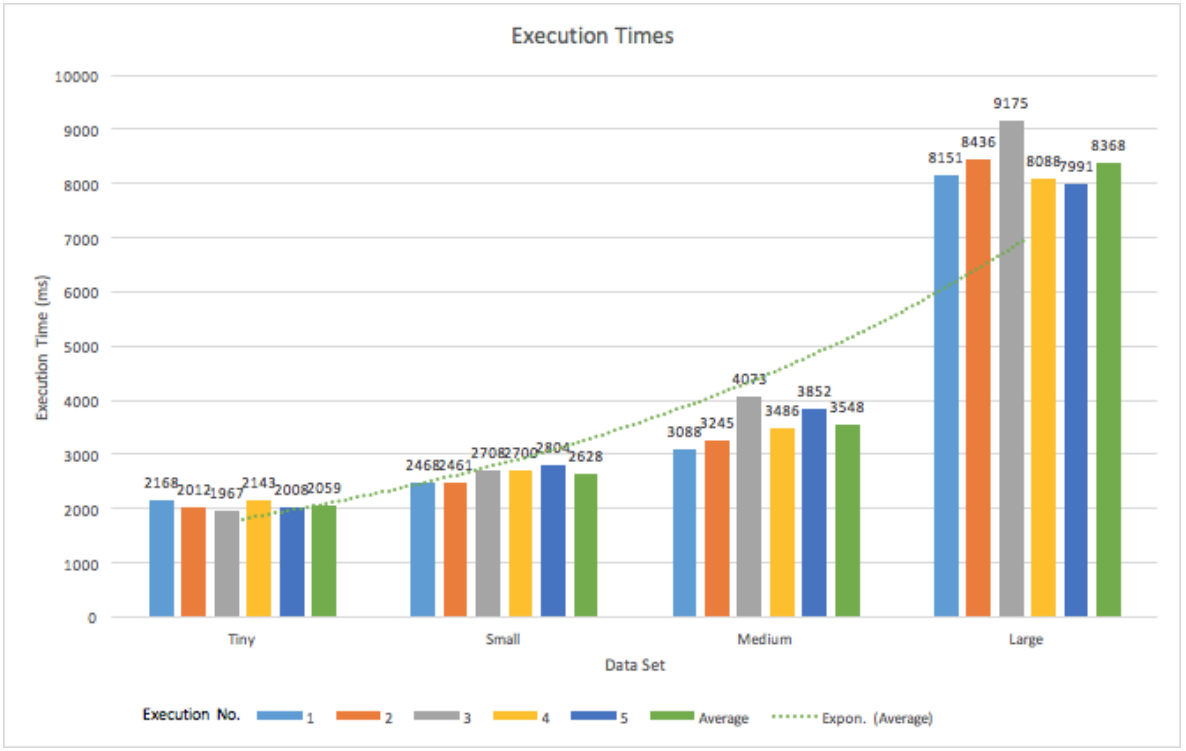
Performance Evaluation

To evaluate the performance of each program, the program was run five times for each data set, using 1994 as the input year. All times are in milliseconds.

Task 1

TASK 1	Tiny	Small	Medium	Large
1	2168	2468	3088	8151
2	2012	2461	3245	8436
3	1967	2708	4073	9175
4	2143	2700	3486	8088
5	2008	2804	3852	7991
Average	**2059**	**2628**	**3548**	**8368**

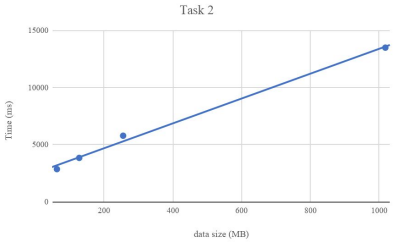


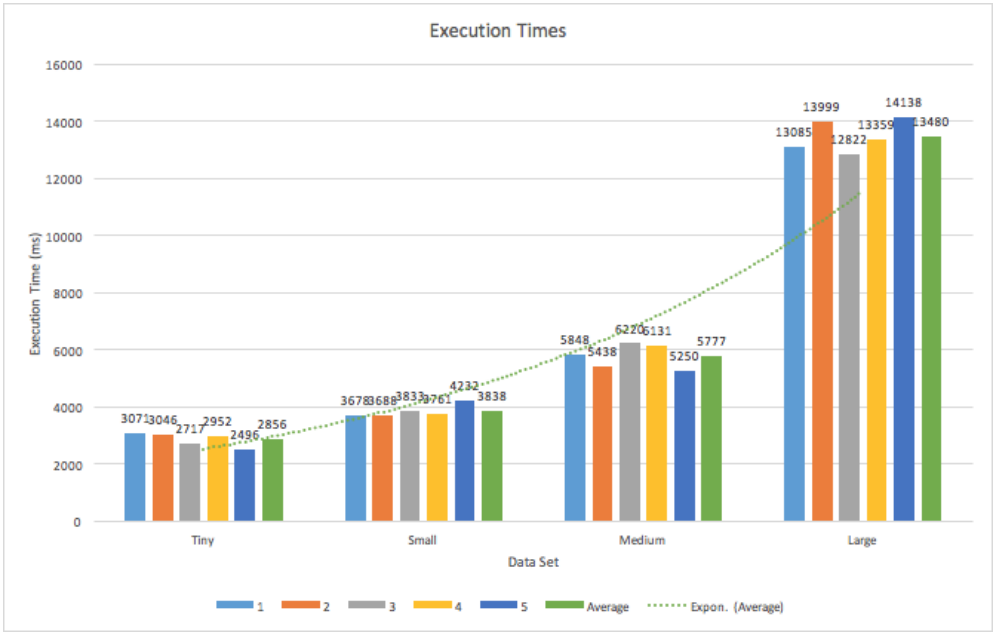


Explanations:
As can been seen in the *time-data graph* the program runs relatively efficiently - execution time scales linearly with the input data size.

Task 2

TASK 2	Tiny	Small	Medium	Large
1	3071	3678	5848	13085
2	3046	3688	5438	13999
3	2717	3833	6220	12822
4	2952	3761	6131	13359
5	2496	4232	5250	14138
Average	**2856**	**3838**	**5777**	**13480**

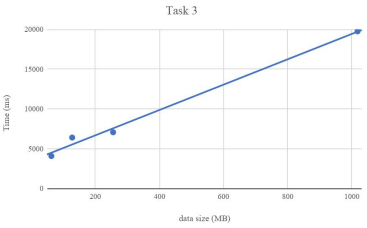


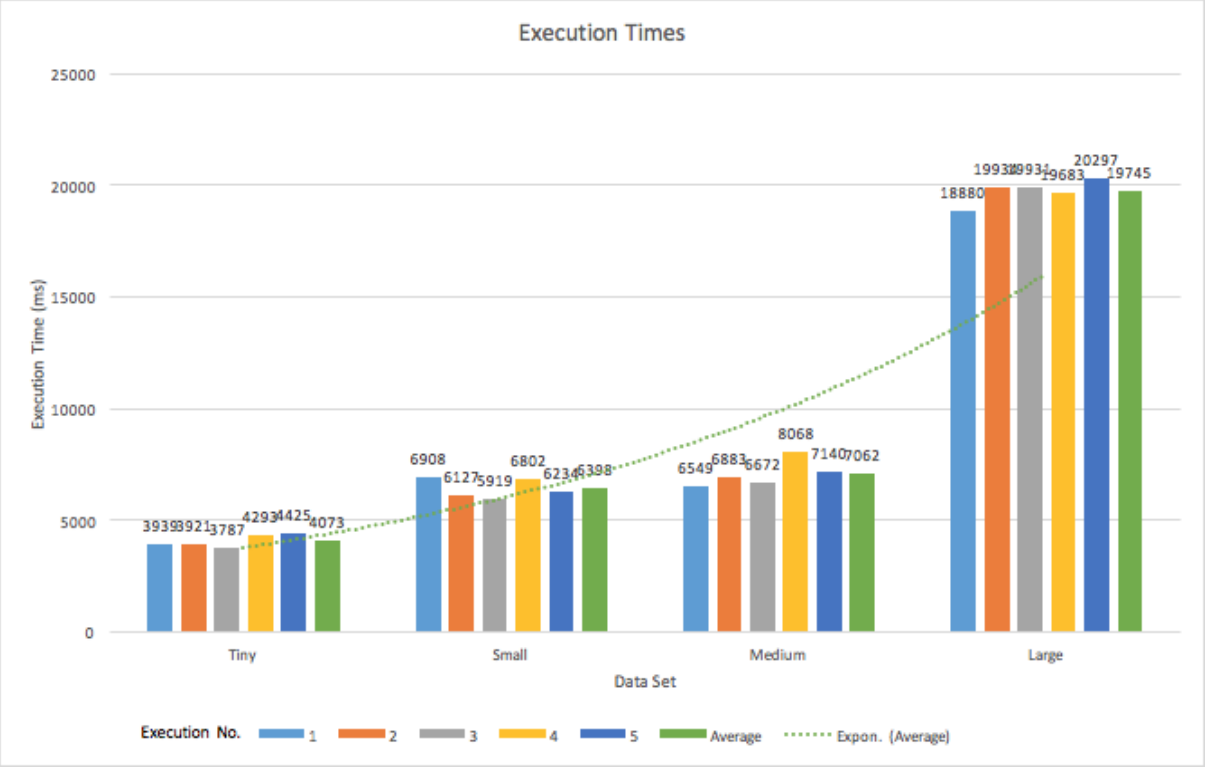


Explanations:
Similar to Task 1, the execution time scales linearly with the input data size.

Task 3

TASK 3	Tiny	Small	Medium	Large
1	3939	6908	6549	18880
2	3921	6127	6883	19934
3	3787	5919	6672	19931
4	4293	6802	8068	19683
5	4425	6234	7140	20297
Average	**4073**	**6398**	**7062**	**19745**





Explanations:
Again, Task 3 scales in a linear fashion according to input data size.

APPENDIX

HDFS location of output file locations

Task 1 (year 1994):
/user/hche8927/output-t1/
Task 2 (year 1994):
/user/hche8927/output-t2/
Task 3 (year 1994):
/user/cvan8308/output-t3/

Execution Plans, Benchmarks & Graphs for Task 3

Execution Plans
Image 2.3.1 Execution Plan Before Tuning:

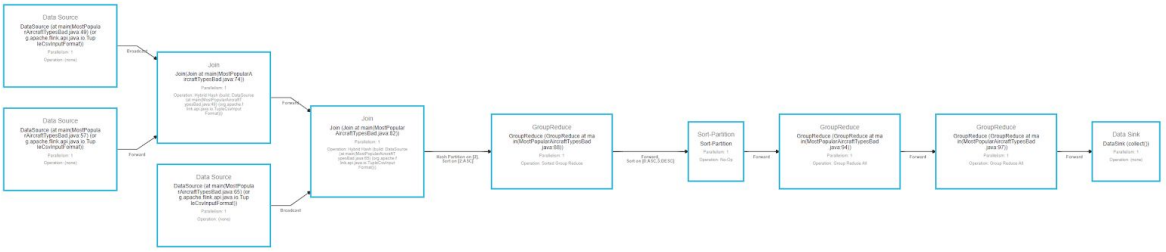
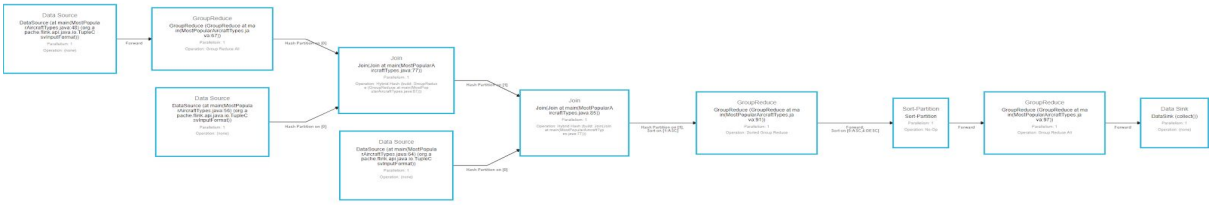


Image 2.3.2 Execution Plan After Tuning:



Performance Improvement

Image 2.3.3 Benchmarks:

	Tiny-1	Tiny-2	Tiny-3	Tiny - Average	Small-1	Small-2	Small-3	Small-Average
Before Tuning	3412	3528	3853	3597	4777	5027	5003	4935
After Tuning	3939	3921	3787	3882	6908	6127	5919	6318
	Medium-1	Medium-2	Medium-3	Medium-Average	Large-1	Large-2	Large-3	Large-Average
Before Tuning	7940	8138	7805	7961	20962	21108	23196	21755
After Tuning	6549	6883	6672	6701	18880	19934	19931	19581