Report for Assignment 3 – CSE 587
Compute the volatility of stocks in NASDAQ using PIG and HIVE and compare with Map-Reduce
Submitted by Rahul Derashri (50135205)

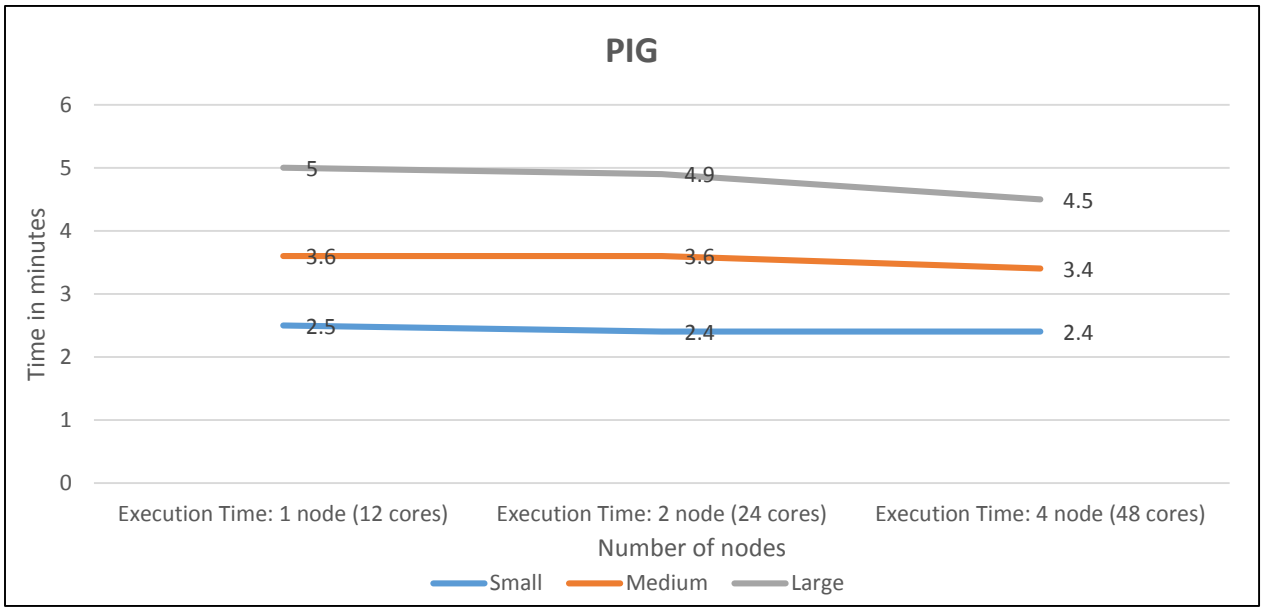
PIG:

I started the code keeping in mind the usage of UDF's. So initially implemented using multiple Java UDF's to compute and get the top 10 and bottom 10 volatility stocks. When ran on CCR it was taking 8-9 minutes for small data set. But as I was using many UDF's, I changed the code to just use just 2 Java UDF in all. The 1st UDF take the grouped data according to the stock name and return back the volatility of stocks and the 2nd UDF just take the sorted data according to volatility and produce top 10 and lowest 10 volatility stocks. The final implementation explained above is taking just 2.5 minutes for the small dataset.

The main problem occurred during working on PIG was to figure out the issues in UDF. The first UDF I wrote for the assignment was giving some null initialization issue which I was able to figure out later. I even tried to write the UDF using Accumulator interface but later changed to the basic one using EvalFunc.

The timings for PIG implementation is:

PIG			
Problem Size	Execution Time: 1 node (12 cores)	Execution Time: 2 node (24 cores)	Execution Time: 4 node (48 cores)
Small	2.5 min	2.4 min	2.4 min
Medium	3.6 min	3.6 min	3.4 min
Large	5 min	4.9 min	4.5 min



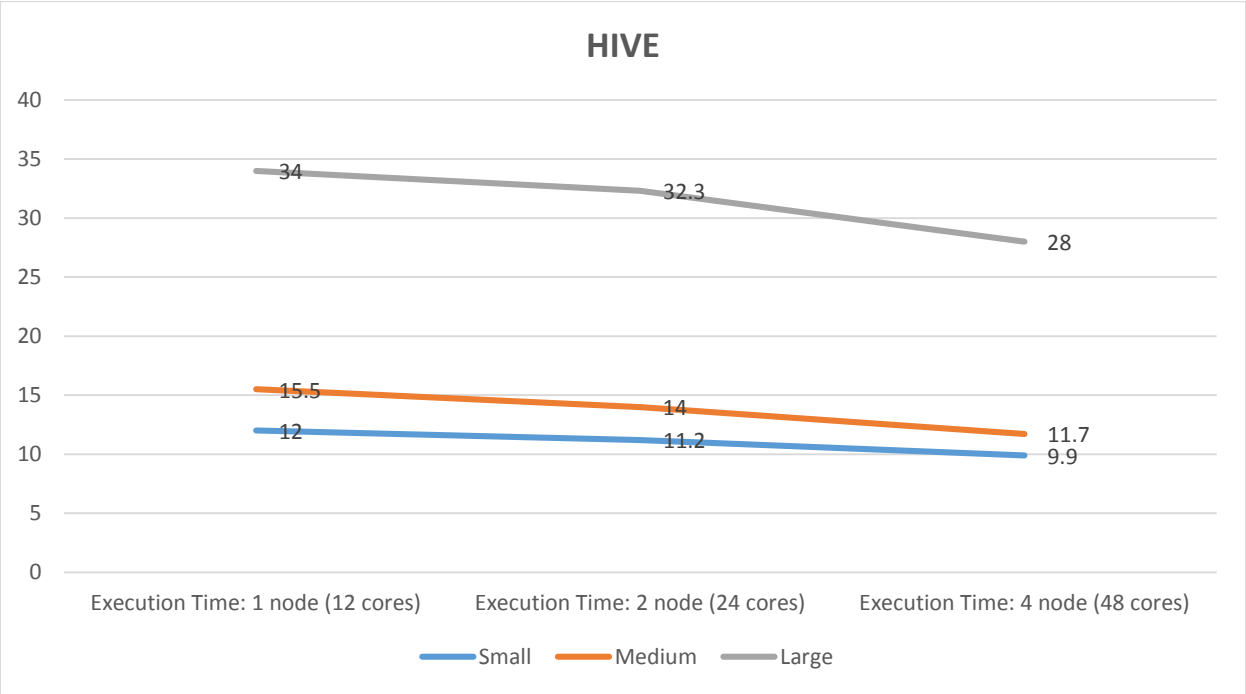
HIVE:

The implementation is done completely using hive queries. Here I have not used any UDF's to perform any part of the implementation. The initial code was just implemented in 3 hours but later while running the jobs on CCR, figured it out that the code in not optimized properly. The code was working properly for small dataset in about 10-12 minutes but it was getting stuck at the query where JOIN is performed between two tables (First date and last date data of the month) for calculating Rate of Return for medium and large dataset. Then tried to improve the code by minimizing the number JOIN's and minimizing the number of columns to join. But still no success. Then decided to rewrite the code. Again implemented the new code with NO JOIN's at all. Used single table to store the first and last date of the month concatenated with the Adj_Close values so that no need to join to fetch corresponding values from different tables to get the Adj_Close.

The main problem encountered during the complete implementation was the optimizing JOIN operations. Otherwise the coding and understanding was easy as compared to PIG.

The timings for PIG implementation is:

HIVE			
Problem Size	Execution Time: 1 node (12 cores)	Execution Time: 2 node (24 cores)	Execution Time: 4 node (48 cores)
Small	12	11.2	9.9
Medium	15.5	14	11.7
Large	34	32.3	28



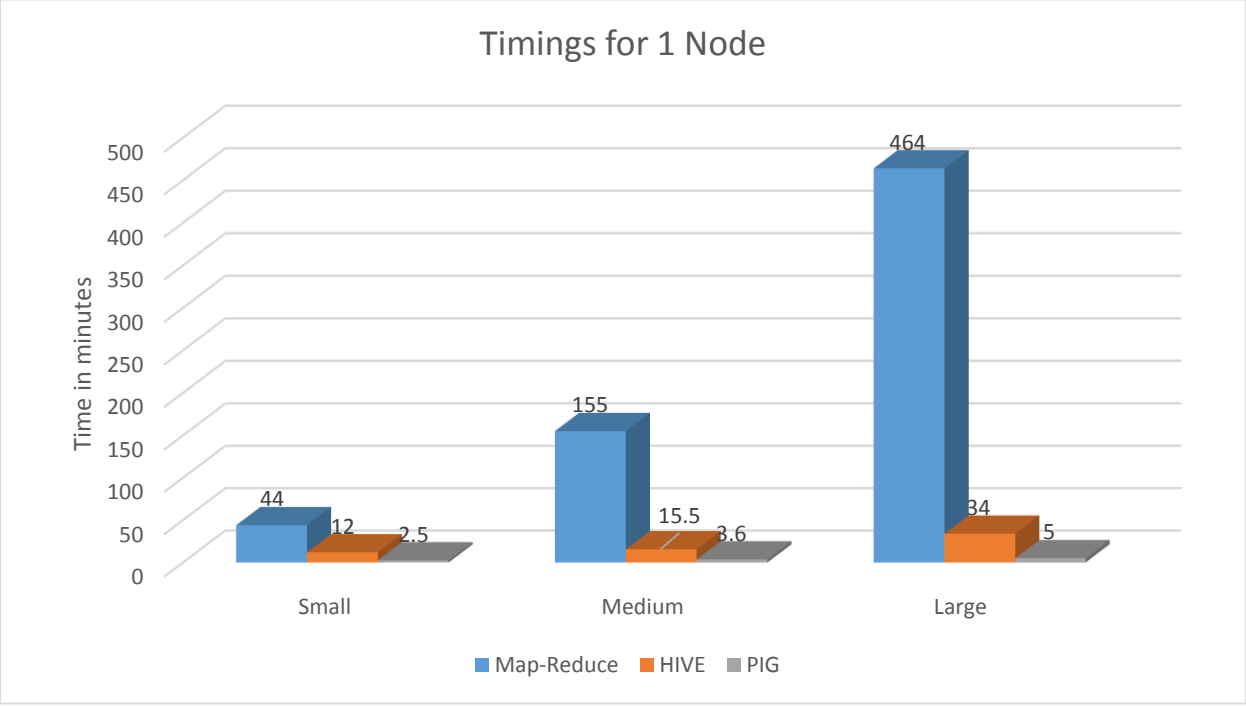
Comparison of Map-Reduce v/s HIVE v/s PIG:

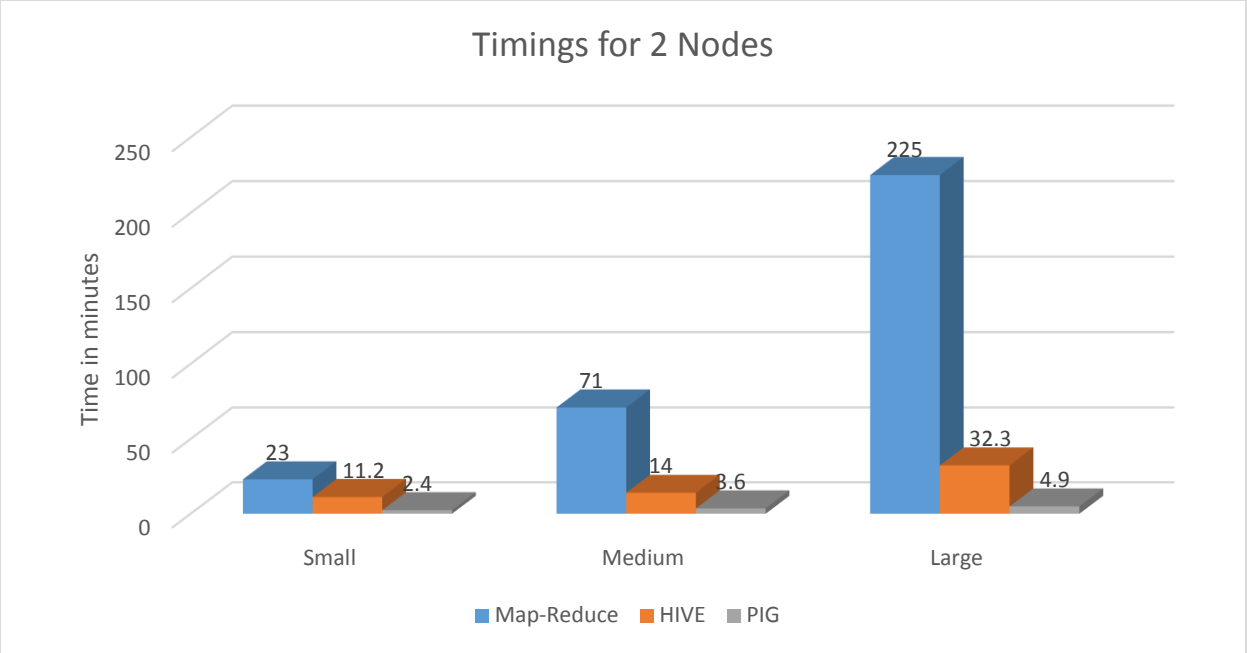
According to the first project the timings for the Map-Reduce job for the same calculation were:

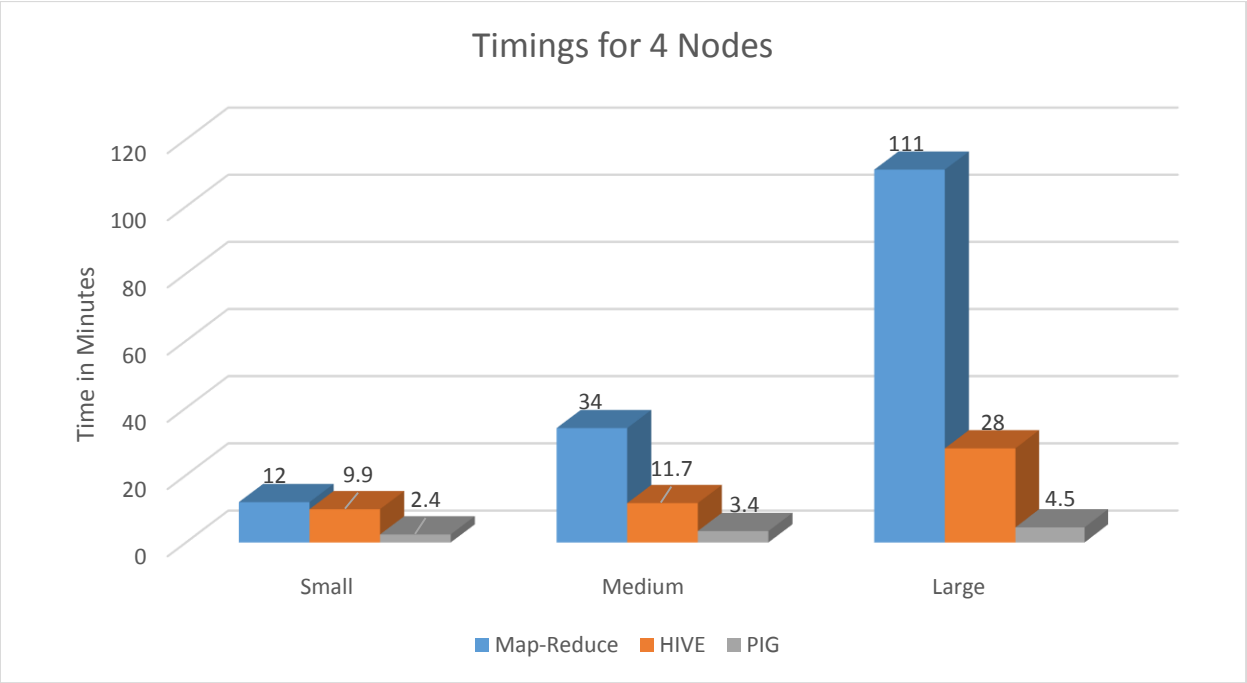
Map-Reduce			
Problem Size	**Execution Time: 1 node (12 cores)**	**Execution Time: 2 node (24 cores)**	**Execution Time: 4 node (48 cores)**
Small	44 min	23 min	12 min
Medium	155 min	71 min	34 min
Large	464 min	225 min	111 min

According to the timings tables of all three we can say that PIG is the fastest and the hive is the nearest to PIG but Map-Reduce is far more time consuming than others. But let's not forget that the implementation also matters. As the Map-Reduce is implemented almost the same way as PIG Java UDF there is huge plus provided by PIG so we can surely say that PIG is better among all but that also internally uses Mappers and Reducers. HIVE on the other is implemented completely using queries supported by HIVE. It was work of complete data –table manipulations with some of the internal function of HIVE such as `STDDEV_SAMP`.

The time comparisons graphs for 1 Node, 2 Nodes and 4 Nodes are as follow:




Timings for 2 Nodes
Time in minutes
250
200
150
100
50
0
23
11.2
2.4
71
14
8.6
225
32.3
4.9
Small
Medium
Large
Map-Reduce
HIVE
PIG


Timings for 4 Nodes
Time in Minutes
120
100
80
60
40
20
0
12
9.9
2.4
34
11.7
3.4
111
28
4.5
Small
Medium
Large
Map-Reduce
HIVE
PIG

Conclusion from My Implementation:

The implementation and optimization plays important role in minimizing timings. For example minimizing the number of JOIN's reduces the complexity and increases proper utilization of resources because of which the HIVE code started to work for even Large and Medium datasets in my case. From the above timings we can say that PIG and HIVE are much faster than Map-Reduce eventhough they internally uses the same.

Results for three are coming same:

Highest 10 for Small Dataset		Lowest 10 for Small Dataset	
ACST.csv	9.271589761859985	VCSH.csv	0.004637760185632481
NETE.csv	5.396253961502245	AXPWW.csv	0.0044388372955839524
XGTI.csv	4.542344311472957	CADT.csv	0.004156636196742422
TNXP.csv	3.248332196781867	SKOR.csv	0.003948740216629902
EGLE.csv	3.0222065379013143	AGZD.csv	0.003938593878697365
PTCT.csv	1.8462537015817715	TRTLU.csv	0.003478105118698644
GOGO.csv	1.7793421751861378	MBSD.csv	0.0025000459104618897
MEILW.csv	1.7188134065765308	VGSH.csv	0.0013014906189562883
ROIQW.csv	1.3965320839591489	GAINO.csv	5.650074160499658E-4
CFRXZ.csv	1.0792682449689408	LDRI.csv	5.149336582098077E-4

Highest 10 for Medium Dataset		Lowest 10 for Medium Dataset	
ACST-3.csv	9.271589761859985	MBSD-1.csv	0.0025000459104618897
ACST-2.csv	9.271589761859985	VGSH-2.csv	0.0013014906189562883
ACST-1.csv	9.271589761859985	VGSH-3.csv	0.0013014906189562883
NETE-3.csv	5.396253961502245	VGSH-1.csv	0.0013014906189562883
NETE-1.csv	5.396253961502245	GAINO-3.csv	5.650074160499658E-4
NETE-2.csv	5.396253961502245	GAINO-1.csv	5.650074160499658E-4
XGTI-1.csv	4.542344311472957	GAINO-2.csv	5.650074160499658E-4
XGTI-3.csv	4.542344311472957	LDRI-2.csv	5.149336582098077E-4
XGTI-2.csv	4.542344311472957	LDRI-1.csv	5.149336582098077E-4
TNXP-1.csv	3.248332196781867	LDRI-3.csv	5.149336582098077E-4

Highest 10 for Large Dataset		Lowest 10 for Large Dataset	
ACST-6.csv	9.271589761859985	LDRI-7.csv	5.149336582098077E-4
ACST-5.csv	9.271589761859985	LDRI-10.csv	5.149336582098077E-4
ACST-4.csv	9.271589761859985	LDRI-6.csv	5.149336582098077E-4
ACST-3.csv	9.271589761859985	LDRI-5.csv	5.149336582098077E-4
ACST-2.csv	9.271589761859985	LDRI-4.csv	5.149336582098077E-4
ACST-1.csv	9.271589761859985	LDRI-3.csv	5.149336582098077E-4
ACST-10.csv	9.271589761859985	LDRI-2.csv	5.149336582098077E-4
ACST-9.csv	9.271589761859985	LDRI-1.csv	5.149336582098077E-4
ACST-8.csv	9.271589761859985	LDRI-8.csv	5.149336582098077E-4
ACST-7.csv	9.271589761859985	LDRI-9.csv	5.149336582098077E-4