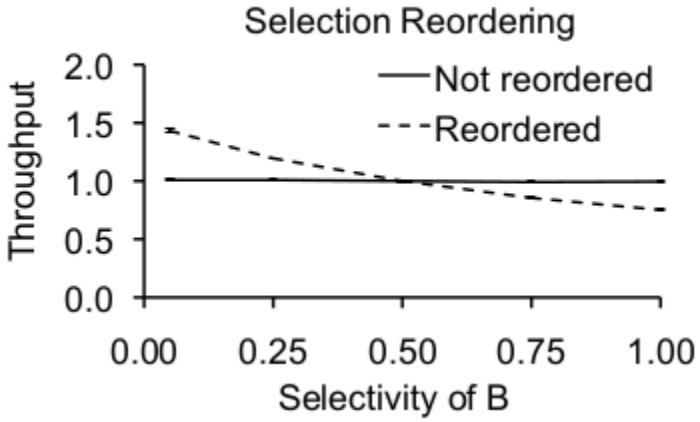
Homework 2

Meet Desai
mpd2155

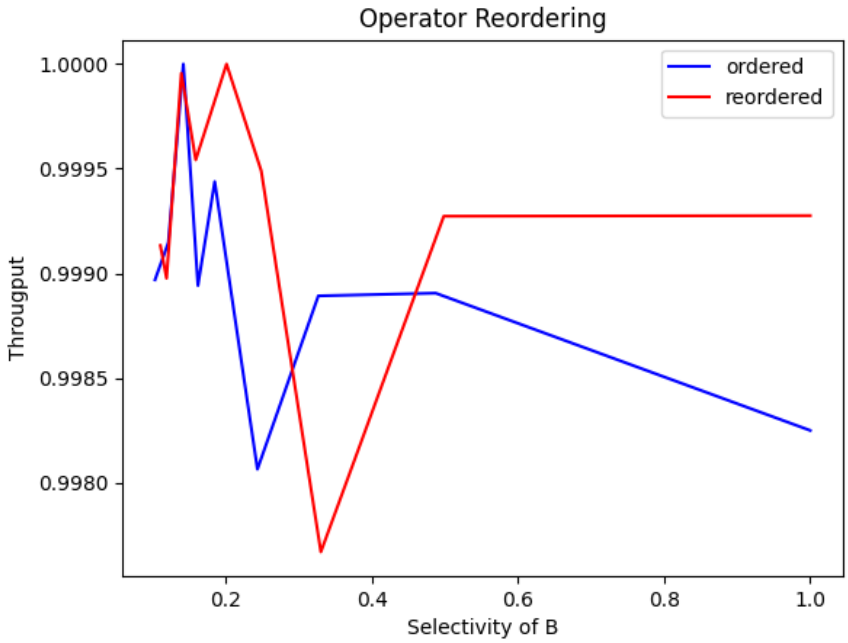
All the codes can be executed using spark-submit and they save the graph automatically in the same folder. Also, numpy and pyspark are the dependencies that are needed to run the codes.

Operator reordering:

For operator reordering, I generated data using the random function in python. I generated 100000 numbers and then ran them through 2 different filters. One with selectivity 0.5 and other with variable selectivity. Selectivity is the ratio between the output tuples and the input tuples. While throughput is the number of input tuples divided by the total time of the process. The graph I was trying to reproduce was:



I tried many different things, such as, taking average over 20 different iterations, generating 100000 random integers for every iteration, using sleep function to add a fixed process time, normalising throughput, but I was not able to reproduce the graph. I always got graphs like this:



These graphs always seemed random. I thought that it might be due to the fact that I am running the whole process on my local machine and due to other processes running in the background the load division might be uneven. To test this I tried to get the process time of individual operators and found this for selectivity of B = 1. We need to subtract 0.05 seconds to get the process time of an operator as that is the fixed process time added using sleep function in python.

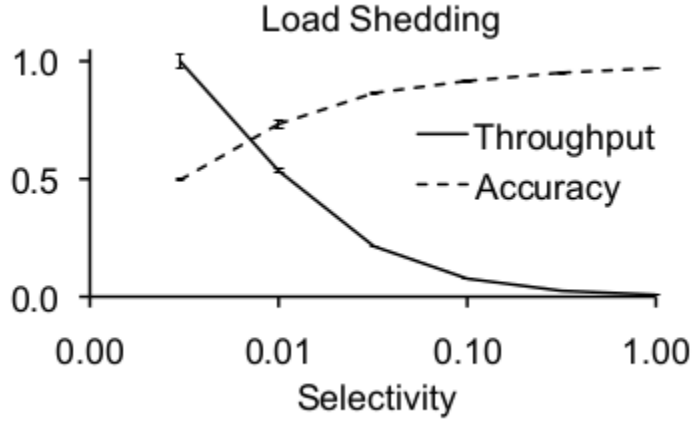
```
A ordered --> 0.0526430606842041
B ordered --> 0.050377607345581055
Selectivity of B --> 1.0
B reordered --> 0.05029654502868652
A reordered --> 0.05013275146484375
Selectivity of B --> 1.0
```

Ideally, it is assumed that the cost of both the operators is the same and only the selectivity is varied but in practice it can be seen that that is clearly not the case. Since, selectivity of B is one, Theoretically, the process time for operator A should be the same in both the cases because it has to

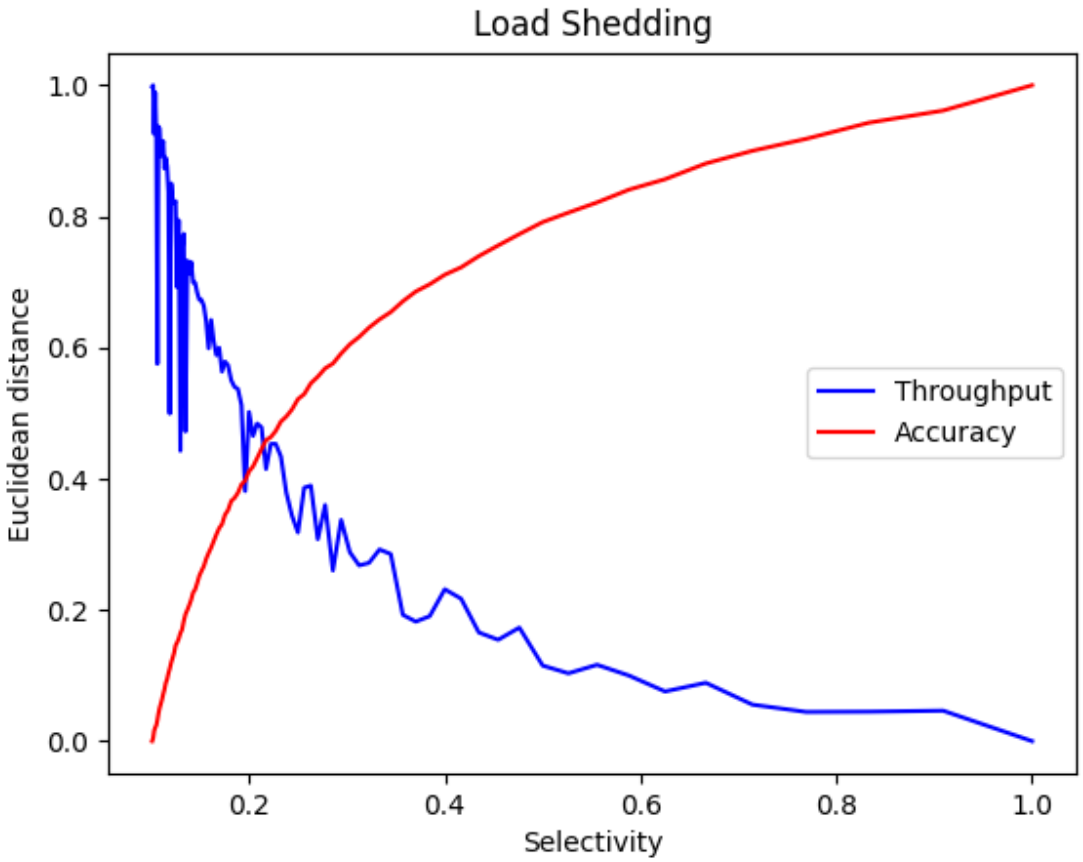
process the same number of tuples but as seen here it is more than 10 times for ordered operation as compared to unordered operation. This kind of behavior accounts for the seemingly random graphs that I have received.

Load Shedding:

For load shedding, I generated data using the random function in python. For dynamic loads. I started by generating 10000 tuples and then increased this load by a factor of 1.1, 1.2, 1.3 etc. For the load shedding part, everytime the tuples increased from 10000. The selectivity will go down from one such that there will always be 10000 tuples for the operator to process. The selectivity is 10000 divided by the number of input tuples. The process time is the time taken to complete the whole process, i.e., for load shedder to calculate selectivity and then removing the extra tuples till the operator finishes mapping on the tuples. The throughput is the number of input tuples divided by the process time. To calculate the euclidean distance between the original and the reduced tuples, I used numpy. The graph I was trying to reproduce was:



I was able to reproduce the graph as follows:

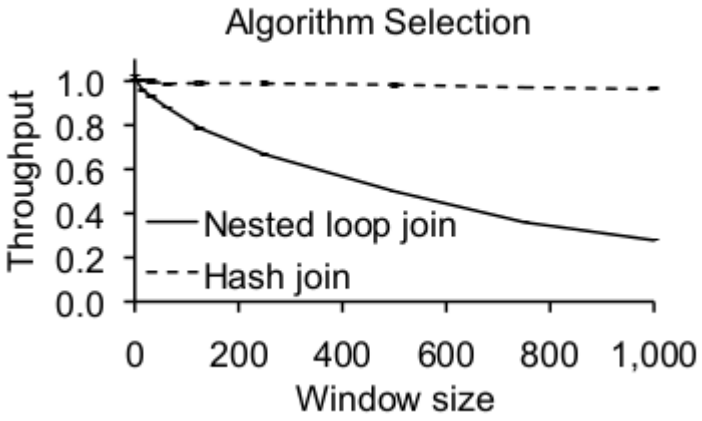


I have normalised the throughput and accuracy so as to get them between 0 and 1, so that they can be seen on the graph together.

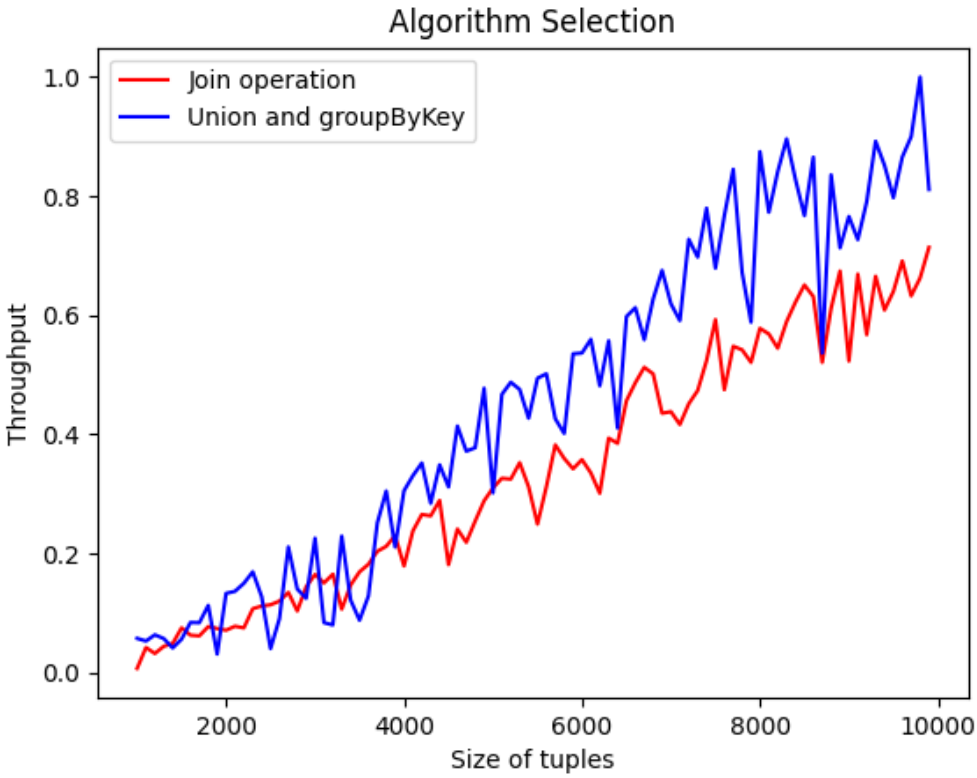
Algorithm Selection:

For Algorithm Selection, I generated data using the random function in python. I created 2 separate RDD with a random key between 1 and 100 and a value for that key between 1 and 5. I used 2 different algorithms to 'join' these two RDDs. One function was directly to use join, while the other was to take a 'union' of the 2 RDD and then use 'groupByKey' to get the same output as that of join function. Then I varied the size of tuples from 1000 to 10000 and compared the throughput between those two. The process time was the time needed to complete a process, join in one case and union plus groupByKey in the other. The throughput is the number of

input tuples divided by the process time. I also normalised the throughput. The graph I was trying to reproduce was :



I received a graph as follows:



Although this graph is not the same, The algorithms are different, i.e., the one used in the paper and one that I used. Also, this graph makes sense as my throughput is directly proportional to the input tuples, which means it will increase as input tuples increase. The advantage of this optimization is to prove that one algorithm is better than the other in terms of throughput, which can be clearly seen in this graph as the union plus groupByKey outperforms join operation almost every time.

Thank you.