BIG DATS PROGRAMING

ASSIGNMENT-2

1 To implement the PageRank algorithm using map reduce in hadoop, given adjacency list and initial page rank values in two different text files. The primary step here is to read the two text files and split them, then following the map reduce process for page rank algorithm implementation. But this process contains of reading two text files lead to more number of mappers and its very computationally expensive. To overcome this in this assignment Distributed cache of hadoop is used so that one of the files is cached and the other is used as input in here. The Intial Page rank values file is cached and Adjacency list file is used as input file.

The overall design flow of map reduce for Page Rank is shown in below:

The Page rank algorithm is a iterative algorithm and In each iteration rank values are computes and updated with previous values until the change in values from present and previous iteration is around 10^-6, i.e. the values are very similar with minute error.

Based on the given graph the rank values are computed using the standard page rank formula. And the generalized version for computing ranks is shown below.

$$\mathbf{r} = c\mathbf{P}\top\mathbf{r} + (1 - c)/n$$

In each iteration we take the updated page rank value file and upload it into the cache and also the rank values are outputted into new file. This type of iteration is also called as power iteration

$$\mathbf{r}^t = c \cdot \mathbf{P}^\top \cdot \mathbf{r}^{t-1} + (1 - \mathrm{c})\mathbf{1}/n$$

r_i^t : PageRank value of node i at iteration t

$$r_1^t = c \cdot r_4^{t-1} \cdot p_{4,1} + (1 - c)/5$$
$$r_2^t = c \cdot r_1^{t-1} \cdot p_{1,2} + (1 - c)/5$$
$$r_3^t = c \cdot r_2^{t-1} \cdot p_{2,3} + c \cdot r_5^{t-1} \cdot p_{5,3} + (1 - c)/5$$
$$r_4^t = c \cdot r_2^{t-1} \cdot p_{2,4} + c \cdot r_3^{t-1} \cdot p_{3,4} + (1 - c)/5$$
$$r_5^t = c \cdot r_4^{t-1} \cdot p_{4,5} + (1 - c)/5$$

1.1 Design of Mapper, This takes the input adjacency list row and also the Page Rank values, But as mentioned above, The page rank values are stored in Distributed cache and not given as input file path like adjacency list.

Let's see how the Page rank rank values are used in Mapper from Distributed cache, The pagerank.txt file is stored in distributed cache, This file is used in mapper class in such a way the values of PageRank are stored in a Hash Map.

Distributed cache, The hdfs file system sends small text files to each datanodes while running map reduce job through distributed cache. And these files are deleted after the job is completed from data nodes. This usage will decrease the time taken by reading file each and every time from hdfs.

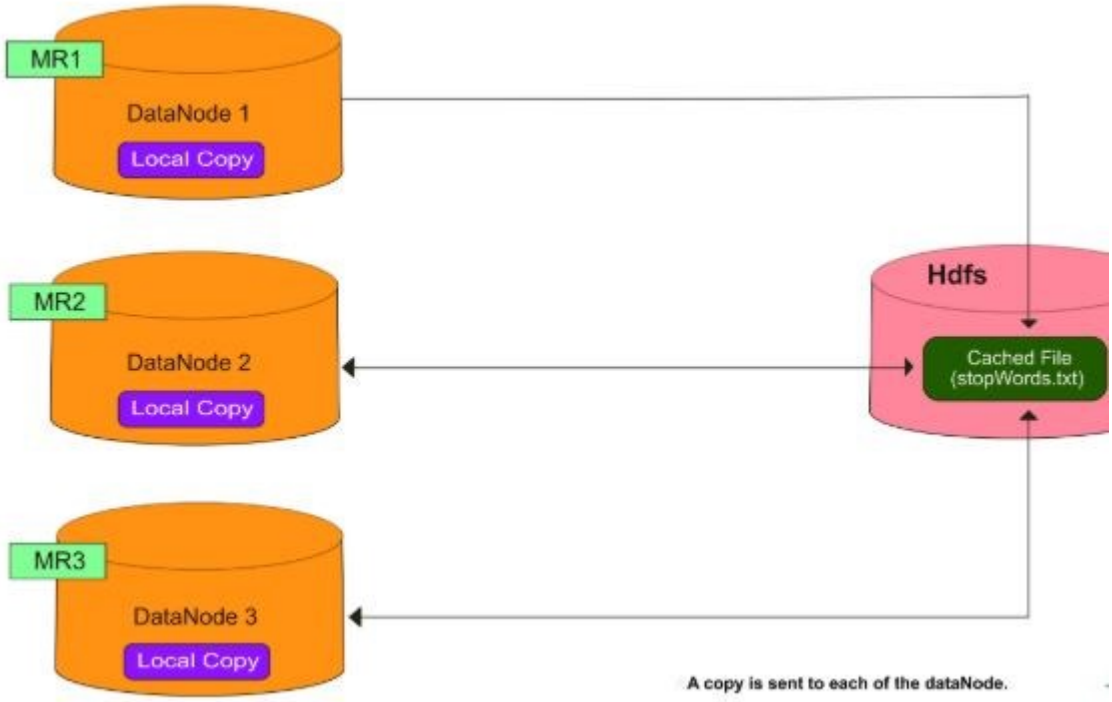

Fig: Showing the working of Distributed cache

Before Mapper design, the data from distributed cache is loaded and stored in a Hash Map, The Hashmap consists of key as nth node index and values as the rank of that node. This can be achieved by overriding the setup method, And this process follow as shown below.

As a initial step read the cached files into URI array and check whether there are any files, If there are any files create File system object and pass configuration object into it, File system is a abstract class of fairly generic file system, After this open the file using File system object and convert the byte stream into character with input stream reader and store it as buffer reader. As next step check this buffered for not empty and read line by line until its empty, Using string tokenize each line is tokenized and the first token is used as key i.e nth node index and the value as PageRank until all the buffer reader is empty. The output here is the Hash Map with key as nth node index and PageRank value of it.

The Hash table created above is used for fetching page rank values based on the adjacent list node index values. As shown below.

Hash Map:

Key	value
1	0.2
2	0.2
3	0.2
4	0.2
5	0.2

Adjacency List:

I	j
1	[2]
2	[3, 4]
3	[4]
4	[1, 5]
5	[3]

As show above the Mapper would read each line of adjacency list file and in this line is in <Text value> and the key be object, From the Object the value is converted into string and string tokenize it. After this the first value of tokenizer is used as key to hash map for getting the page rank value (from above I is the key for hashmap). Once we got the key next all tokens are taken into loop and compute rank*transition probability of the nodes and output it with <token, rank*transition probability> the transition probability is calculates using len(tokens)-1

Output of mapper for line 2 of adjacency list is shown below

<3, hashmap[2]*1/len(tokenizers-1)>

<4, hashmap[2]*1/len(tokenizers-1)>

In above 3, 4 represents nodes and in a similar for all nodes its calculated.

1.2 Reducer:

After Shuffling and sorting the input for reducer is <node number, list of values>. For example

<1,[0.25]>

<3, [0.2, 0.1]> etc…..

In the reducer For each of this input we compute the final using formula mentioned above and the output is written as <node number, rank>

And this process is continued until 30 iterations in the main/driver class of map reduce process.

1.3 I haven't used a combiner in this process, as the data and graph is small it doesn't have that great effect on the network congestion. And if we use combiner in this problem it will be after mapper output and before shuffle and sorting so it does not have any effect. And also the combiner is nothing but the reducer and one can use same reducer as combiner in many cases but it fails if the reducer that is implemented here for a single mapper in this case doesn't act as combiner. May be one can write separate class for reducer that act as combiner and further more customized class for reducer. This Combiner system may not work for multiple mapper if the same reducer is used.

2. The Experimentation results are shown below.

1 0.15555492363830498

2 0.16223821815273626

3 0.23119216214996707

4 0.29545977242068655

5 0.15555492363830498

2.1

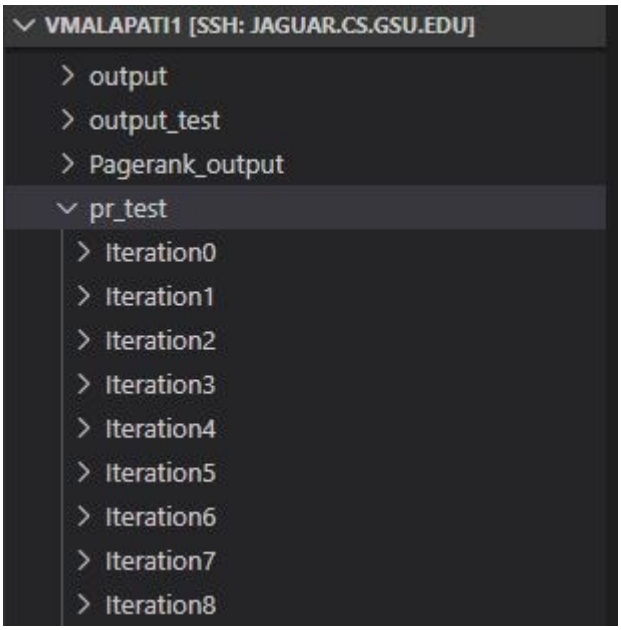

Fig: Showing output files created after 30 iterations

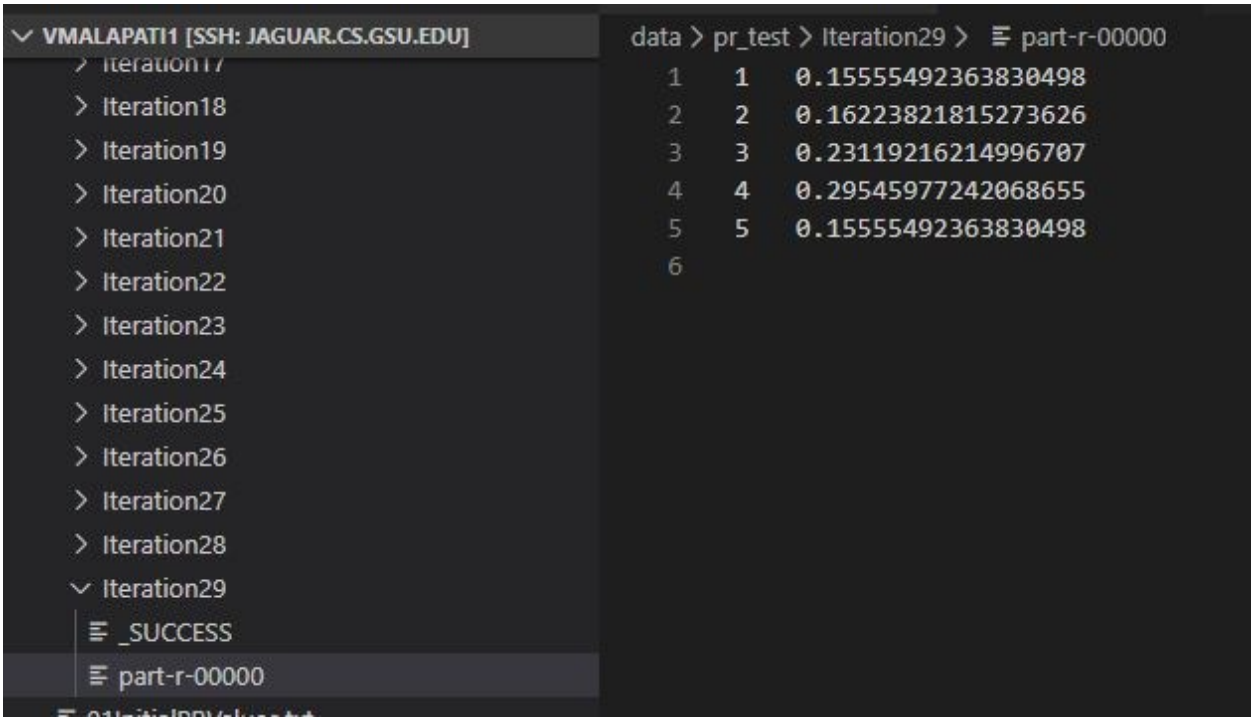

Fig: showing result after 30 iterations

The results are as shown above and these results are from the 30th iteration. The error in results from 29th iteration to 30th iteration is nearly 10^(-4). Since there is very minute change in results we stopped iterations at this point.

The results are as equal to the results that are given in the class.

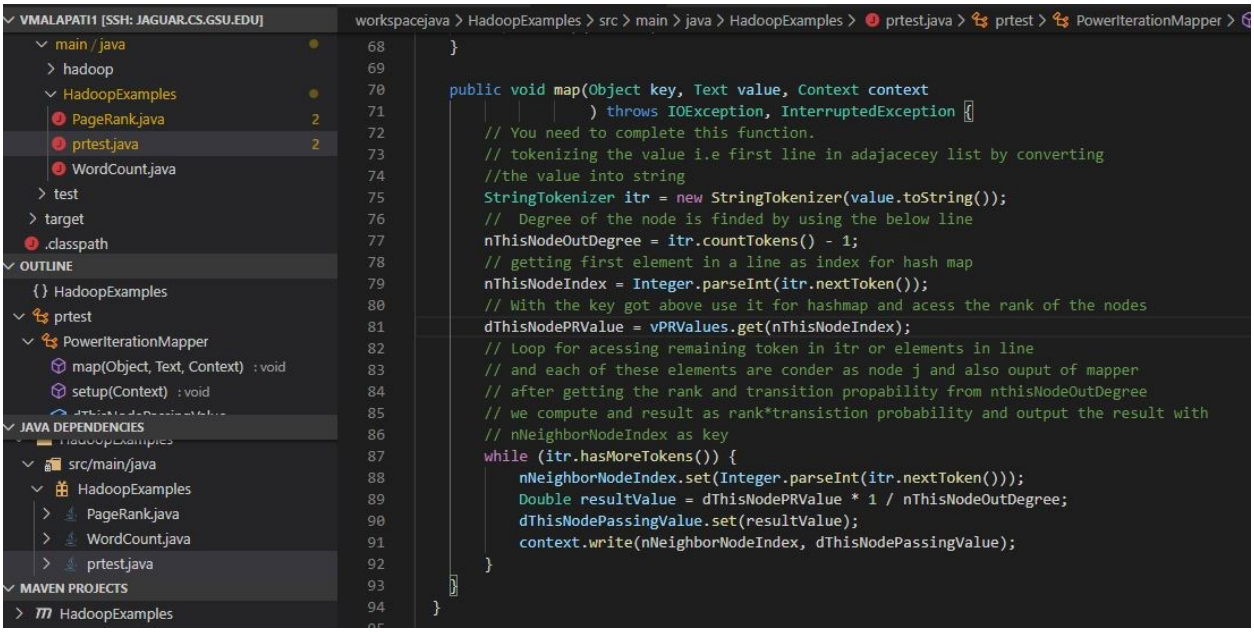

Fig: Showing the mapper code

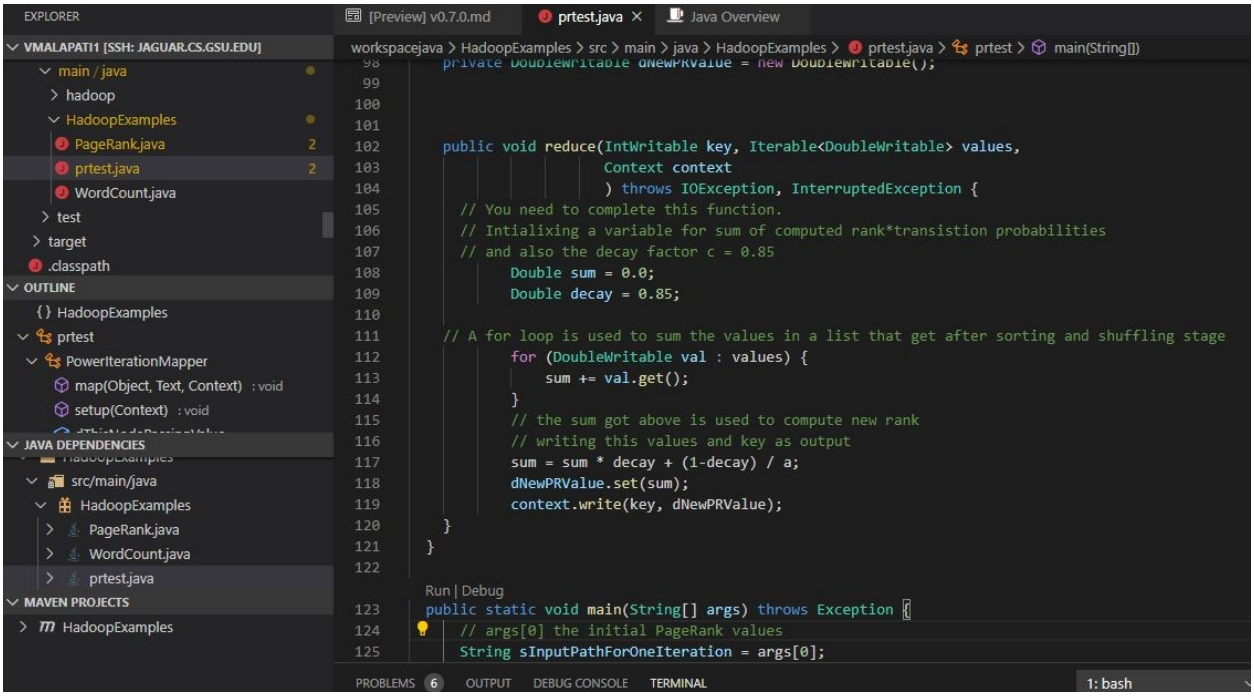

Fig: Showing reducer code

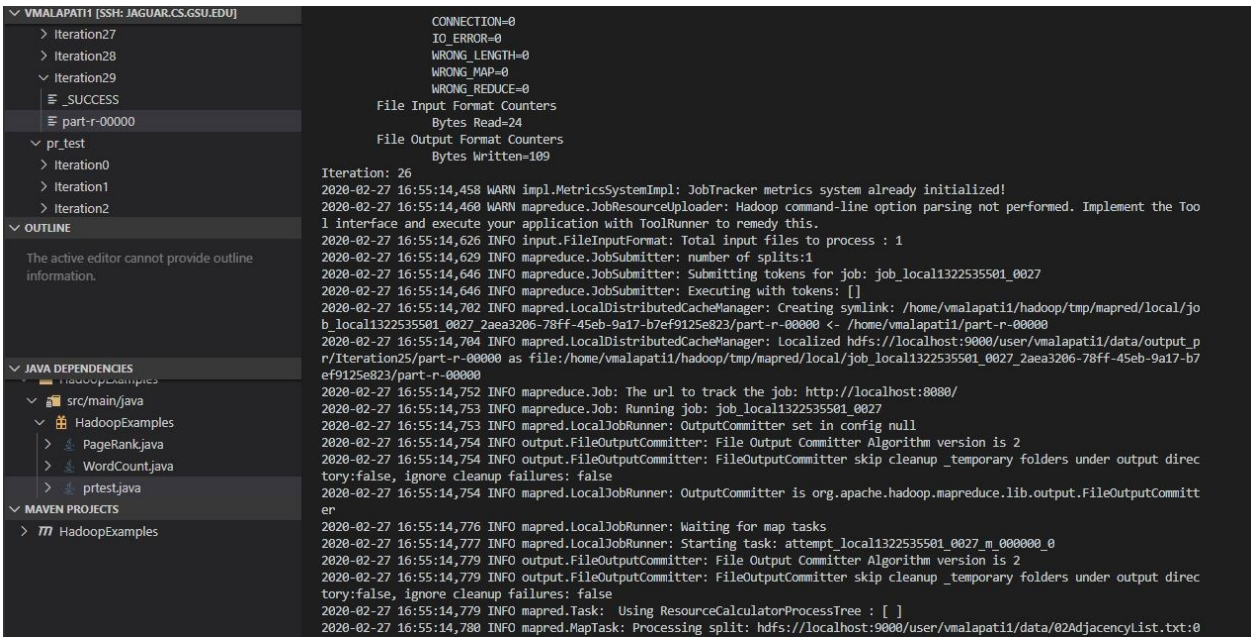

Fig: Showing the printed screen at 26th iteration during execution of the code

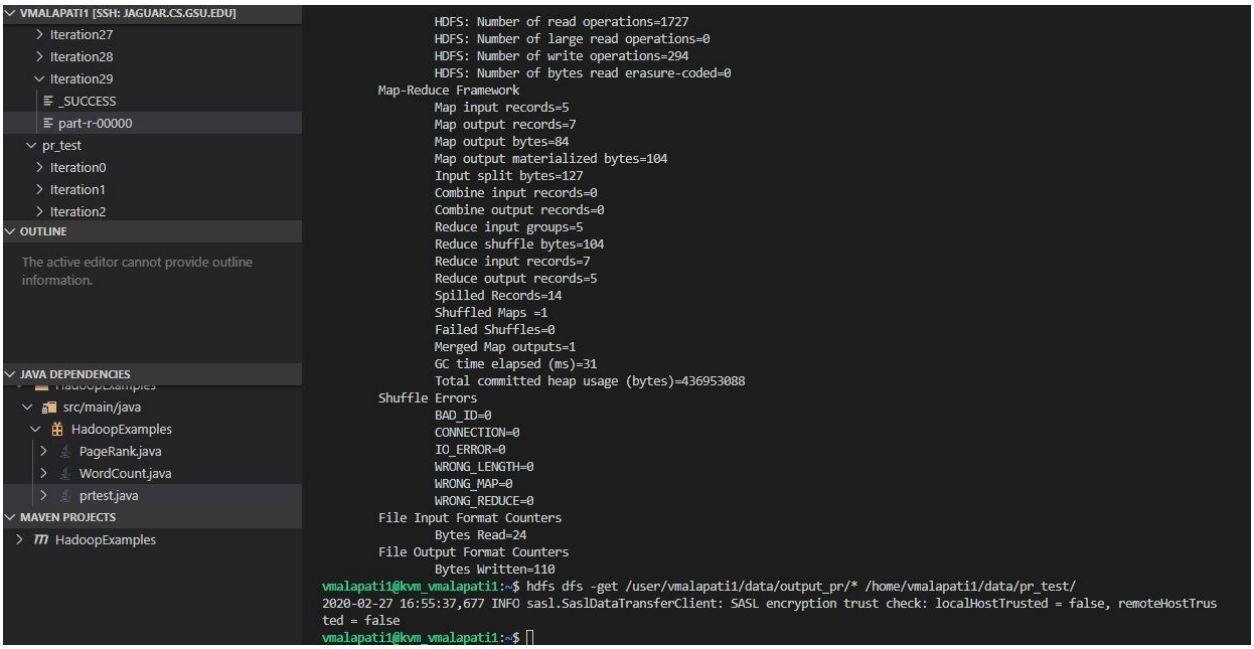

Fig: showing the get usage for getting files in hdfs into local system