Big Data Programming Assignment-2 Report

1. Explanation of the source code

Solution:

```
package HadoopExamples;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;
import java.util.HashMap;
import java.util.Map;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class PageRank {

  public static class PowerIterationMapper
     extends Mapper<Object, Text, IntWritable, DoubleWritable>{

          // The PageRank Values of all the nodes; the PageRank vector
          private Map<Integer, Double> vPRValues = new HashMap<Integer, Double>();
          // The variables for this node and its out-neighbor nodes
          private Integer nThisNodeIndex = 0;
          private IntWritable nNeighborNodeIndex = new IntWritable();
          private Double dThisNodePRValue = 0.0;
          private Integer nThisNodeOutDegree = 0;
          private DoubleWritable dThisNodePassingValue = new DoubleWritable();

          @Override
          protected void setup(
          Mapper<Object, Text, IntWritable, DoubleWritable>.Context context)
          throws IOException, InterruptedException {

          /* Here we read all our page rank values into a HashMap just to use it later in the program*/

          if (context.getCacheFiles() != null
          && context.getCacheFiles().length > 0) {
```

```java
			URI[] cacheFiles = context.getCacheFiles();
			String sCacheFileName = cacheFiles[0].toString();
			System.out.println(sCacheFileName);
			FileSystem aFileSystem = FileSystem.get(context.getConfiguration());
			Path aPath = new Path(sCacheFileName);
		BufferedReader br = new BufferedReader(new InputStreamReader(aFileSystem.open(aPath)));
			String line;
			System.out.println("PR Values");
			// Read the PageRank values of all nodes in this iteration.
			while ((line = br.readLine()) != null) {
				// process the line.
				Integer nOneNodeIndex = 0;
				Double  dOneNodePRValue = 0.0;
				StringTokenizer itr = new StringTokenizer(line);
				nOneNodeIndex = Integer.parseInt(itr.nextToken());
				dOneNodePRValue = Double.parseDouble(itr.nextToken());
				vPRValues.put(nOneNodeIndex, dOneNodePRValue);
				System.out.println(nOneNodeIndex + " " + dOneNodePRValue);
			}
		}
		super.setup(context);
	}

	public void map(Object key, Text value, Context context
		  ) throws IOException, InterruptedException {
		// You need to complete this function.
		// Every line of the input file is a map task
		StringTokenizer itr = new StringTokenizer(value.toString());
		nThisNodeOutDegree = itr.countTokens() - 1;
		nThisNodeIndex = Integer.parseInt(itr.nextToken());
		dThisNodePRValue = vPRValues.get(nThisNodeIndex);

		/*
		Approach:
		---------
		1. For every node of the graph representing the links between pages,
		seperate out the node and their neighbours.
		2. For every neighbour, calculate the pagerank value.
		3. Calculation of pagerank value:
			a. Pagerank value of the in link multiplied with the transition value gives
			   the pagerank value for the node
			b. The transition value is the average of the values of the weights of all the neighbours.
			c. Our consideration is that all the weights of the neighbors is 1.
			d. Thus the transition value becomes 1/n where, n is the number of neighbors.
			4. Output of map fnction is the node index and PR value of that node
		*/

		while (itr.hasMoreTokens()) {
			nNeighborNodeIndex.set(Integer.parseInt(itr.nextToken()));
			Double resultValue = dThisNodePRValue * 1 / nThisNodeOutDegree;
			dThisNodePassingValue.set(resultValue);
			context.write(nNeighborNodeIndex, dThisNodePassingValue);
		}
```

```java
  }
}

public static class PowerIterationReducer
    extends Reducer<IntWritable,DoubleWritable,IntWritable,DoubleWritable> {
  private DoubleWritable dNewPRValue = new DoubleWritable();

        // The PageRank Values of all the nodes; the PageRank vector
        private Map<Integer, Double> vPRValues = new HashMap<Integer, Double>();
        private Integer nNumOfNodes = 0;
        private Double decayFactor = 0.85;

        /* Why did we write the setup function again?
                -> to maintain the hashmap for the reduce task.
          Why do we need a hashmap for the reduce task?
                -> in the calculation of the aggregation of PR values, we introduce a decay factor
                        it is used to spread the decay factor among all the in links
                Why do we need a decay factor?
                -> Decay factor concept is something like damping the value of a link. Meaning that
                when we have many infinite number of navigations to get to the resultant link, which if
                in turn points to the same starting node, this is a way of specifying that the importance
                of that link drops as we navigate infinitely so that we end up with at least minimal value
                instead of having an indeterminate value in case of infinite links.
        */

  @Override
  protected void setup(
                Reducer<IntWritable,DoubleWritable,IntWritable,DoubleWritable>.Context context)
      throws IOException, InterruptedException {
    if (context.getCacheFiles() != null
          && context.getCacheFiles().length > 0) {
        URI[] cacheFiles = context.getCacheFiles();
        String sCacheFileName = cacheFiles[0].toString();
        FileSystem aFileSystem = FileSystem.get(context.getConfiguration());
        Path aPath = new Path(sCacheFileName);
        BufferedReader br = new BufferedReader(new InputStreamReader(aFileSystem.open(aPath)));
                        String line;
                        while ((line = br.readLine()) != null) {
                // process the line.
                Integer nOneNodeIndex = 0;
                Double  dOneNodePRValue = 0.0;
                StringTokenizer itr = new StringTokenizer(line);
                nOneNodeIndex = Integer.parseInt(itr.nextToken());
                dOneNodePRValue = Double.parseDouble(itr.nextToken());
                                vPRValues.put(nOneNodeIndex, dOneNodePRValue);
                        }
        nNumOfNodes = vPRValues.size();
    }
    super.setup(context);
  }
```

```java
            /*
            Approach:
            ---------
            1. The input of reducer function is key - node index Value - page rank value
            2. We aggregate page ranks of all the ndoes having same index.
            3. we also add the decay factor to dampen the infinite navigations
            4. Output is a key - representing the node index, Value - representing aggregated PR
            */

  public void reduce(IntWritable key, Iterable<DoubleWritable> values,
                 Context context
                 ) throws IOException, InterruptedException {
   // You need to complete this function.
                              Double sum = 0.0;
                              for (DoubleWritable val : values) {
                                        sum += val.get();
                              }
                              sum = sum * decayFactor + (1.0 - decayFactor) / nNumOfNodes;
                              dNewPRValue.set(sum);
                              context.write(key, dNewPRValue);
  }
}

public static void main(String[] args) throws Exception {
          // args[0] the initial PageRank values
          String sInputPathForOneIteration = args[0];
          // args[1] the input file containing the adjacency list of the graph
          String sInputAdjacencyList = args[1];
          // args[2] Output path
          String sExpPath = args[2];
          String sOutputFilenameForPreviousIteration = "";
          // args[3] number of iterations
          Integer nNumOfTotalIterations = Integer.parseInt(args[3]);
          for (Integer nIdxOfIteration = 0;
                    nIdxOfIteration < nNumOfTotalIterations; nIdxOfIteration++){
                    System.out.println("Iteration: " + nIdxOfIteration);

                    /* The configuration object in hadoop is much like system properties in java.
                       They provide global parameters which help you to configure your job and the
                       hadoop cluster.
                   */

                    Configuration conf = new Configuration();

                       /*The Job class allows the user to configure the job, submit it, control its execution,
                       and query the state.
                       The set methods only work until the job is submitted, afterwards they will throw an
                       IllegalStateException.*/

                    Job job = Job.getInstance(conf, "Power Iteration Method");

                    job.setJarByClass(PageRank.class);//configuring pagerank job
                    job.setMapperClass(PowerIterationMapper.class);//configuring the mapper job
```

```
                job.setReducerClass(PowerIterationReducer.class);//configuring the reducer job

                /* the input output specification is IntWritable and DoubleWritable respectively
                this is to take the input as index of node and output the decimal page rank value
                using the power iteration method*/

                job.setOutputKeyClass(IntWritable.class);
                job.setOutputValueClass(DoubleWritable.class);

                if (nIdxOfIteration > 0) { // In the Iteration 2, 3, 4, ...,
                // the output of the previous iteration => the input of this iteration
                        sInputPathForOneIteration = sOutputFilenameForPreviousIteration;
                }

                //adding the initial pagerank values to the cache to read it intermittently
                job.addCacheFile(new Path(sInputPathForOneIteration).toUri());

                FileInputFormat.addInputPath(job, new Path(sInputAdjacencyList));
                // Change the output directory
                String sOutputPath = sExpPath + "/Iteration" +

               nIdxOfIteration.toString() + "/";

               /* On the successful completion of a job, the MapReduce runtime creates a
               _SUCCESS file in the output directory.
               This may be useful for applications that need to see if a result set is complete just by
               inspecting HDFS. (MAPREDUCE-947)*/

                String sOutputFilename = sOutputPath + "part-r-00000";
                sOutputFilenameForPreviousIteration = sOutputFilename;

                FileOutputFormat.setOutputPath(job, new Path(sOutputPath));
                if (nIdxOfIteration < nNumOfTotalIterations - 1) {
                        job.waitForCompletion(true);
                } else {
                        System.exit(job.waitForCompletion(true) ? 0 : 1);
                }
        }
 }
}
```

1.1 How is the mapper function defined? What kind of intermediate results are generated?

Solution:

```
public void map(Object key, Text value, Context context
          ) throws IOException, InterruptedException {
               // You need to complete this function.
               // Every line of the input file is a map task
               StringTokenizer itr = new StringTokenizer(value.toString());
               nThisNodeOutDegree = itr.countTokens() - 1;
```

```
		nThisNodeIndex = Integer.parseInt(itr.nextToken());
		dThisNodePRValue = vPRValues.get(nThisNodeIndex);

		/*
		Approach:
		---------
		1. For every node of the graph representing the links between pages,
		seperate out the node and their neighbours.
		2. For every neighbour, calculate the pagerank value.
		3. Calculation of pagerank value:
			a. Pagerank value of the in link multiplied with the transition value gives
			the pagerank value for the node
			b. The transition value is the average of the values of the weights of all the
			neighbours.
			c. Our consideration is that all the weights of the neighbors is 1.
			d. Thus the transition value becomes 1/n where, n is the number of
			neighbors.
		4. Output of map fnction is the node index and PR value of that node

		*/

		while (itr.hasMoreTokens()) {
			nNeighborNodeIndex.set(Integer.parseInt(itr.nextToken()));
			Double resultValue = dThisNodePRValue * 1 / nThisNodeOutDegree;
			dThisNodePassingValue.set(resultValue);
			context.write(nNeighborNodeIndex, dThisNodePassingValue);
		}
	}
```

Explanation:

The mapper function is defined as follows:

1. First the input being the adjacency list, the node at first index of every line specifies the node of the graph, and the rest of the elements of line from adjacency list represents its neighbors.
2. Since the adjacency list is read line by line, the output is also associated line by line. Map function outputs, for every line of the adjacency list, the page rank value associated with a single neighbor as <key, value> -> <neighborNodeIndex, calculatedPageRankValue>
3. Likewise, Page Rank value is calculated for all the other neighbors. And this process repeats for all the other node indices. At this point we have <key, value> pairs. Keys may be same, but the values are different(In a way that they are dependent on the in node link rather than that of the neighbor node under consideration)

1.2 How is the Reducer function defined? How do you aggregate the intermediate results and get the final output?

Solution:

```
public void reduce(IntWritable key, Iterable<DoubleWritable> values,
                Context context
                ) throws IOException, InterruptedException {
   // You need to complete this function.
                        Double sum = 0.0;
                        for (DoubleWritable val : values) {
                                sum += val.get();
                        }
                        sum = sum * decayFactor + (1.0 - decayFactor) / nNumOfNodes;
                        dNewPRValue.set(sum);
                        context.write(key, dNewPRValue);
  }
```

Explanation:

The Reducer function is defined as follows:

1. The input for reducer is the output of mapper function. The values of pairs having same keys is summed
2. The aggregated values give the page rank value of the node for that iteration of the power method.

1.3. Do you use combiner function? Why and why not?

Solution:

1. A Combiner, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.
2. The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large, and the data transferred to the reduce task is high.
3. The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as **key-value collection** pairs.
4. The following key-value pair is the input taken from the Map phase.
    a. <1, $PR_{2,1}$>, <2, $PR_{4,2}$>, <1, $PR_{4,1}$>, <2, $PR_{3,2}$>, <1, $PR_{6,1}$>,
5. ***There was no use of a combiner in the assignment.*** This is because that the size of data is small. When the data is huge then it makes proper sense to use a combiner to reduce the network overload.

2. Experimental Results

After 1st iteration

1	0.115
2	0.2
3	0.28500000000000003
4	0.28500000000000003
5	0.115

After 10th iteration

1	0.1529082053474905
2	0.1625278811189075
3	0.23304687344035746
4	0.29860883474575384
5	0.1529082053474905

After 20th iteration

1	0.1554953495156925
2	0.16211327565713446
3	0.23115960666009716
4	0.2957364186513832
5	0.1554953495156925

After 30th iteration

1	0.15555492363830498
2	0.16223821815273626
3	0.23119216214996707
4	0.29545977242068655
5	0.15555492363830498

2.1 Screenshots of the key steps. For example, the screenshot for the outputs in the terminal when you run "Hadoop jar YourJarFile" command. It will demonstrate that your program has no bug.

Solution:

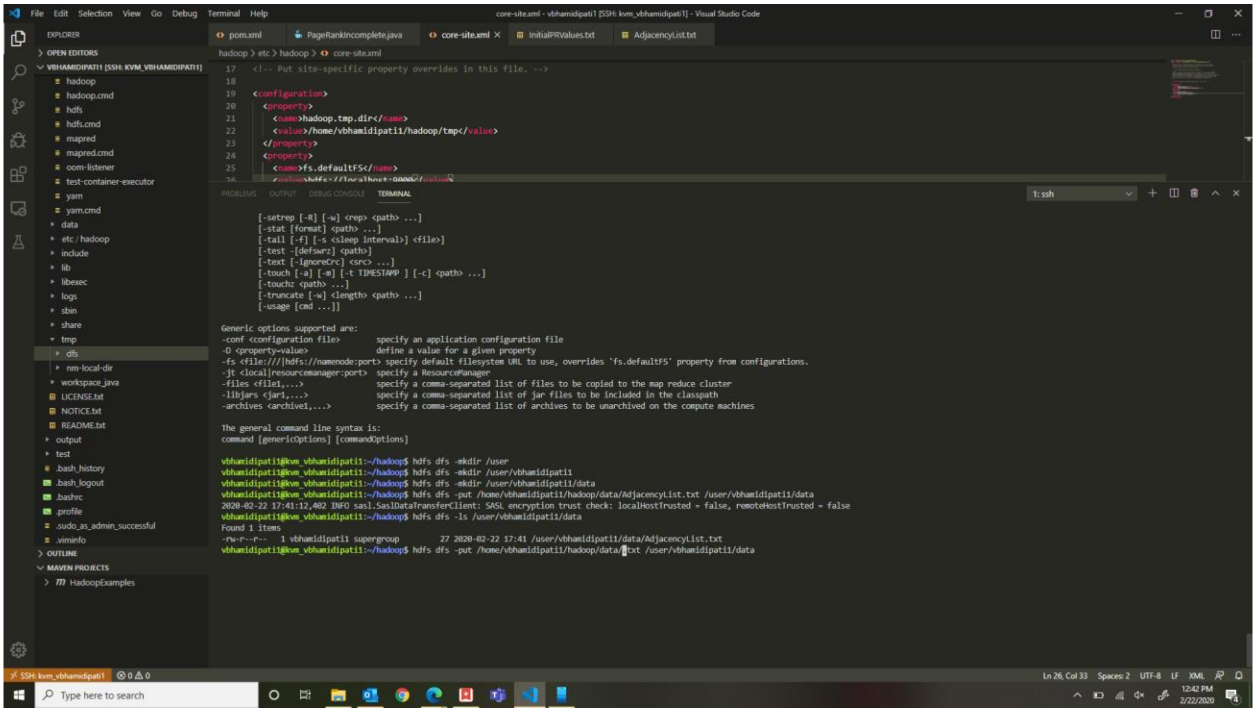

Fig 1: Pushing the input files onto the cloud

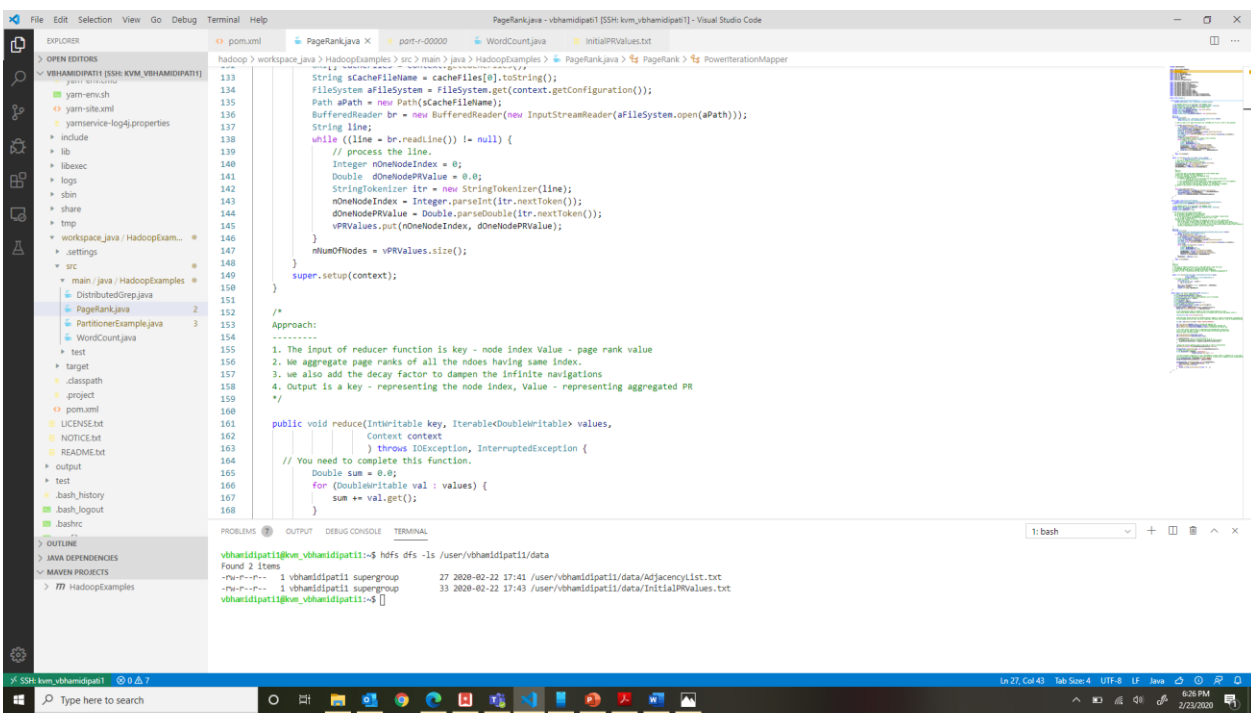

Fig 2: Displaying that the file upload to HDFS was successful

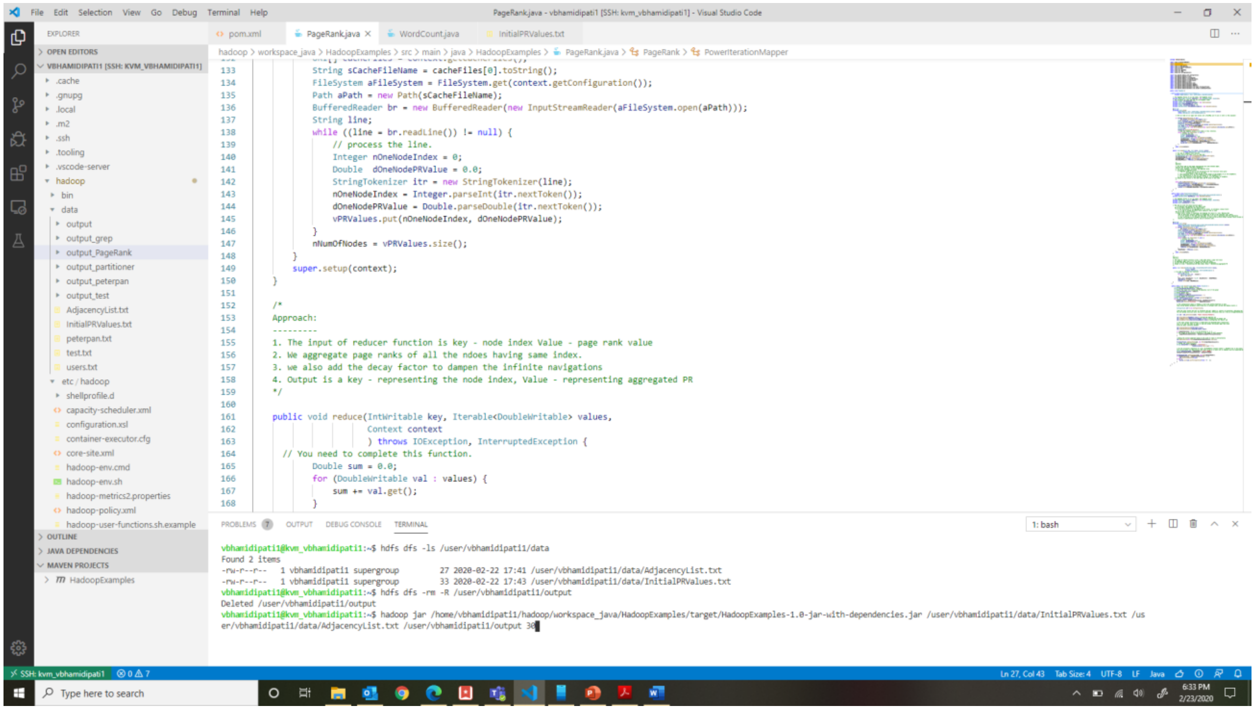

Fig. 3: Running the Hadoop jar command

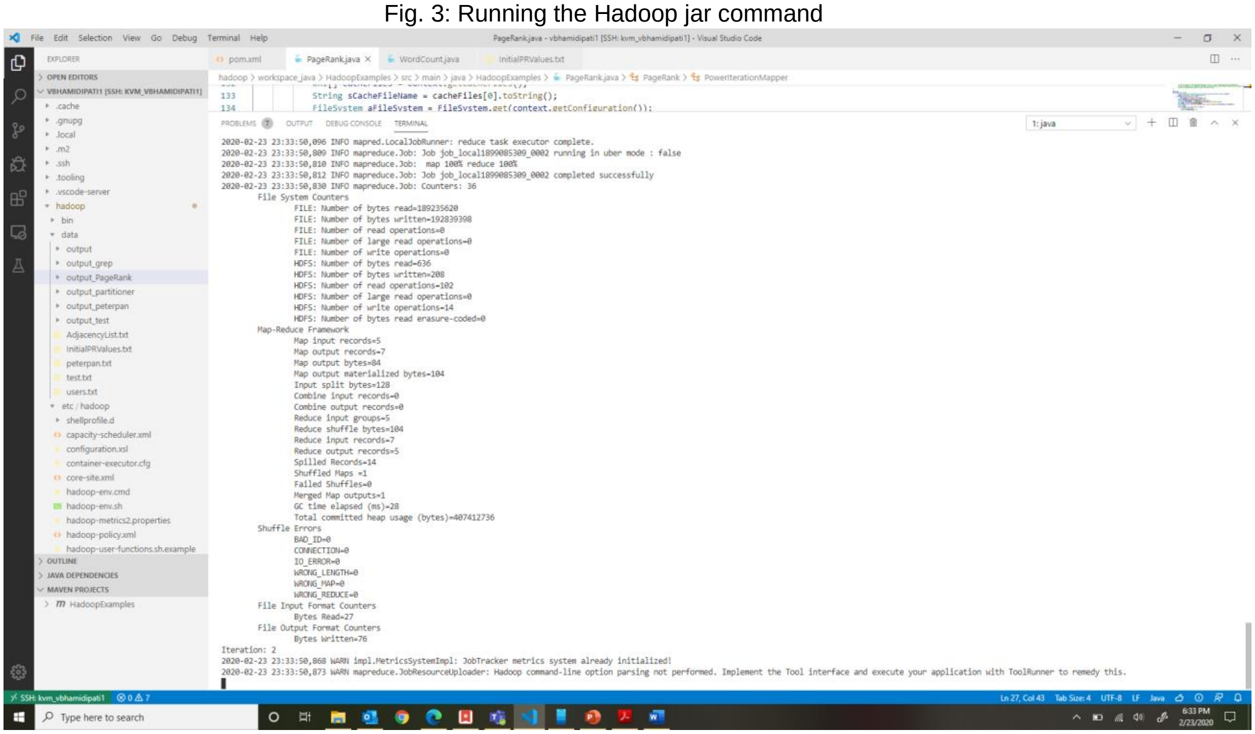

Fig. 4: Map Reduce Job running at iteration -2

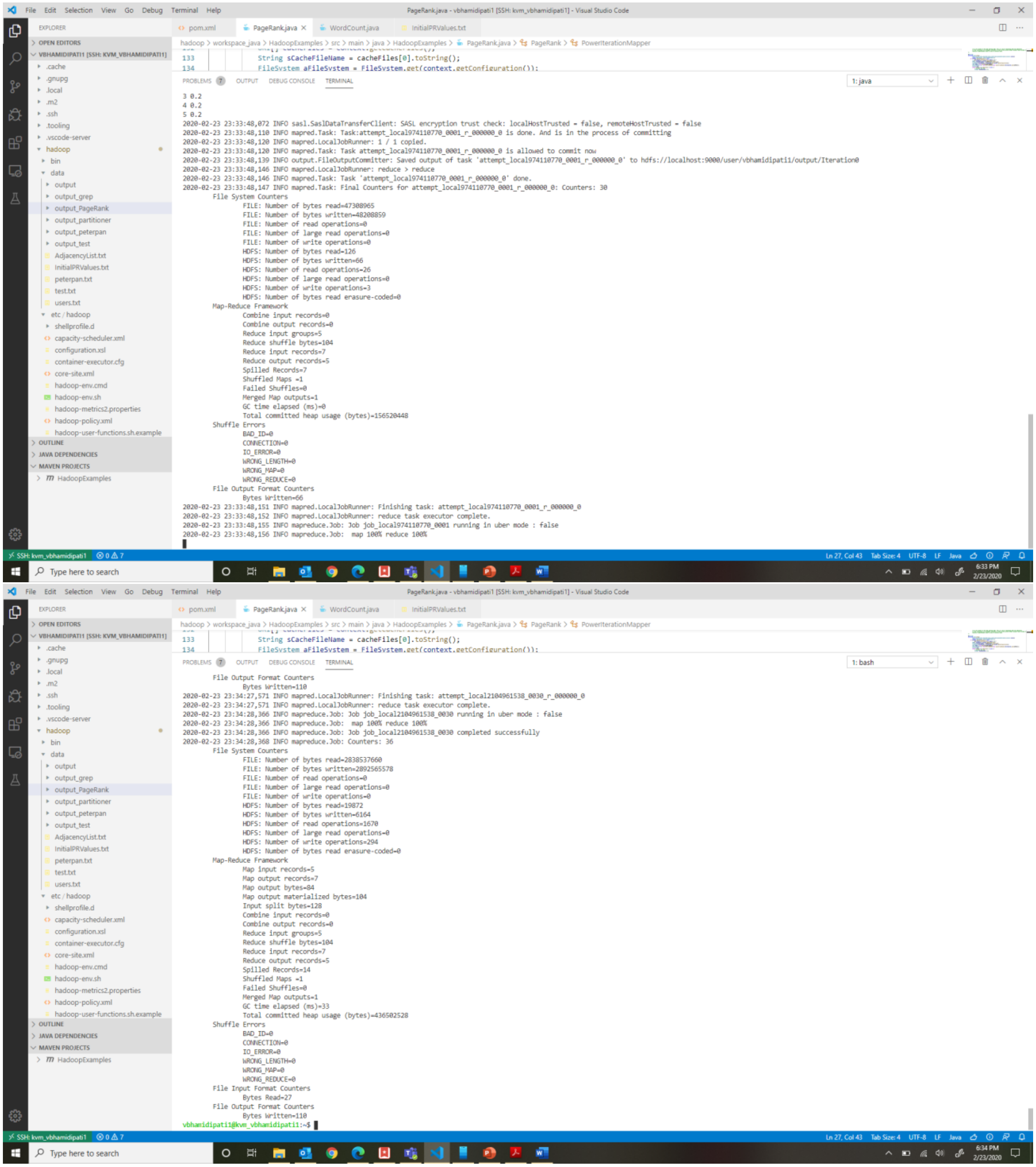

Fig. 5: Complete Map reduce job for Page Rank Evaluation

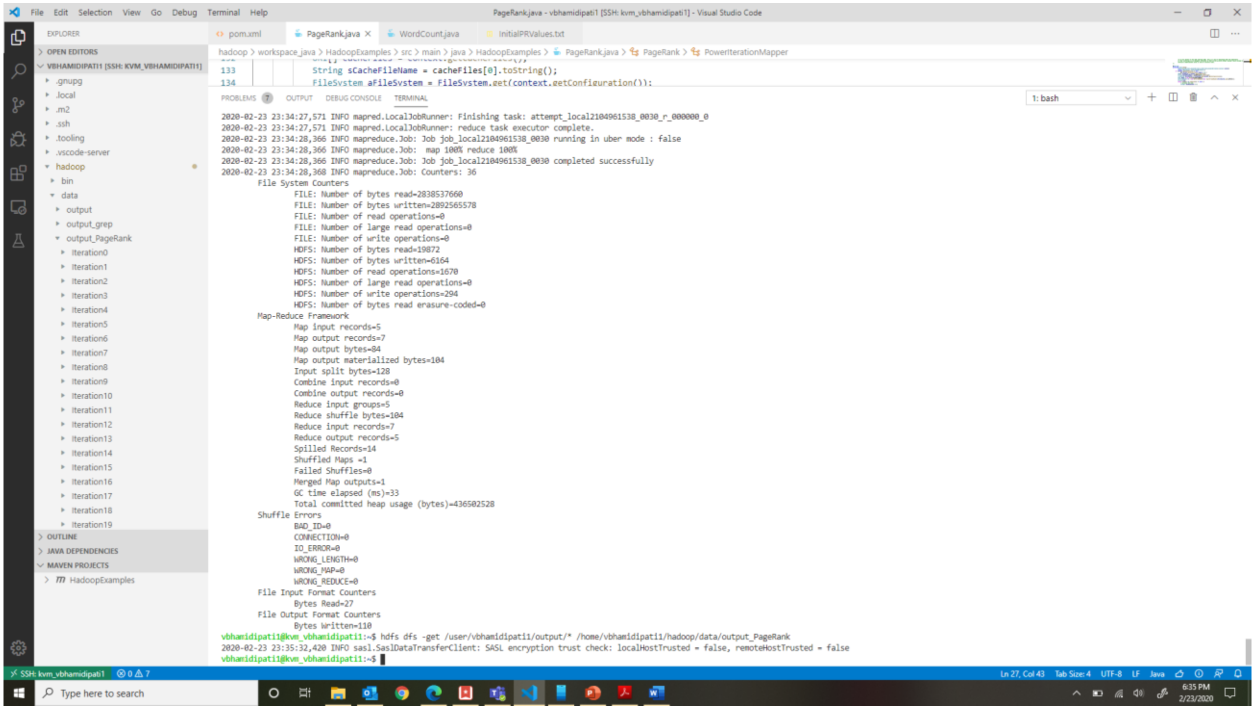

Fig. 6: Making a local copy of the output in the KVM

2.2 Explain your results. Does your implementation give the exact PageRank values? How large are the errors?

Solution:

After 30th iteration

1	0.15555492363830498
2	0.16223821815273626
3	0.23119216214996707
4	0.29545977242068655
5	0.15555492363830498

Note: *The implementation gave exact page rank values as was expected. Errors are computer to be minimal.*