Programming 3 Assignment Report

Student Name: Dazhi Li

Student NETID: dazhili

Student ID#: 2400330

# 1. Documentation

This part briefly describes how I implement my Inversed Indexes program by MapReduce.

All programs were run on cssmpi8h(master)~cssmpi11h

## MapReduce Key Value Transforms

The most important part to achieve the Inversed Indexes algorithm is mapping key and value correctly on mapper and reducer.

The input format of the text files is TextInputFormat, which will read lines by lines of those files and assign an automatically generated key with a value, context of per line. Then, I used that input as my mapper’s input. My mapper will reformat those key values. The mapper is going to read the keywords that we are searching for. If a word matches with the keyword, the mapper will generate a new key, value pair. The key is the keyword while the value is the filename and a number of 1 with space between them. Those new key, value pairs will be the input of the reducer. The reducer is going to reduce the sum of the input. It utilizes a hashmap to cumulative adding number of keyword occurrence to the same filename. The whole key, value transforms is shown in fig1.

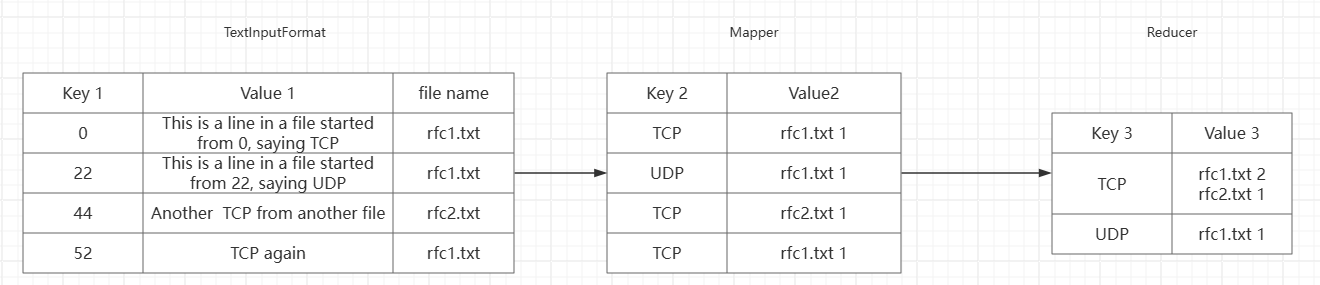


Fig1. Key Value Transformation Examples

## MapReduce layers

What connect a whole MapReduce program is the layers layout. In my program, this layout is the same with the normal MapReduce procedure, as shown in fig2. The shuffle stage is marked as dash line for not being customized and explicitly mentioned in my source code, but it does work in any MapReduce programs.



Fig2. MapReduce running layers

# 2.Source Code

All resource codes could be accessed in the zip file.

**InvertedIndexes.java**

import java.io.IOException;

import java.util.\*;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

import org.apache.hadoop.util.\*;

public class InvertedIndexes {

    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {

    // Get Job configuration for accessing parameters

    JobConf conf;

    public void configure( JobConf job ) {

        this.conf = job;

    }

    public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {

        // retrieve # keywords from JobConf

        int argc = Integer.parseInt( conf.get( "argc" ) );

        // parsing keyworods

        ArrayList<String> keywords = new ArrayList<String>();

        for (int i = 0; i < argc; i++) {

            keywords.add(conf.get("keyword" + i));

        }

        // get the current file name

        FileSplit fileSplit = ( FileSplit )reporter.getInputSplit( );

        String filename = "" + fileSplit.getPath( ).getName( );

        String line = value.toString();

        // read v1, which is the context of every line

        StringTokenizer tokenizer = new StringTokenizer(line);

        Text k2 = new Text();

        Text v2 = new Text();

        while (tokenizer.hasMoreTokens()) {

            String currToken = tokenizer.nextToken();

            for (String keyword : keywords) {

                // matching with keyword, if matched, add to the next k,v pair

                if (currToken.equals(keyword)) {

                    k2.set(keyword);

                    v2.set(filename + " " + 1);

                    output.collect(k2, v2);

                }

            }

        }

    }

    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, Text, Text, Text> {

        private HashMap<String, Integer> docContainer;

        private Text docListText;

        public void setup(Reducer.Context context) {

            docListText = new Text();

            docContainer = new HashMap<>();

        }

    public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {

        while (values.hasNext()) {

            String[] oneDoc = values.next().toString().split(" ");

            // check if this file already exist in the hash map, if not, create a kv pair. if already exist, add 1 up to the value

            // oneDoc[0] : filename

            // oneDoc[1] : 1

            if (docContainer.containsKey(oneDoc[0])) {

                Integer sum= docContainer.get(oneDoc[0]);

                sum += 1;

                docContainer.replace(oneDoc[0], sum);

            } else {

                docContainer.put(oneDoc[0], 1);

            }

        }

        // go though all the hashmap and concatenate all the kv to single string

        String singleLine = "";

        for (String fileName : docContainer.keySet()) {

            singleLine += fileName + " " + docContainer.get(fileName) + " ";

        }

        docListText.set(singleLine);

        output.collect(key, docListText);

    }

    }

    public static void main(String[] args) throws Exception {

    // input format:

    // hadoop jar invertedindexes.jar InvertedIndexes input output keyword1 keyword2 ...

    JobConf conf = new JobConf(InvertedIndexes.class);

    conf.setJobName("InvertedIndexes");

    conf.setInputFormat(TextInputFormat.class);

    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));

    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setMapperClass(Map.class);

    conf.setMapOutputKeyClass(Text.class);

    conf.setMapOutputValueClass(Text.class);

    conf.setCombinerClass(Reduce.class);

    conf.setReducerClass(Reduce.class);

    conf.setOutputKeyClass(Text.class);

    conf.setOutputValueClass(Text.class);

    conf.set( "argc", String.valueOf( args.length - 2 ) ); // argc maintains #keywords

    for ( int i = 0; i < args.length - 2; i++ )

        conf.set( "keyword" + i, args[i + 2] ); //keyword1, keyword2

    JobClient.runJob(conf);

    }

}

# 3.Execution Result

## 3.1 Additional Work on Blood Dataset

I did the additional work on doing the same Inversed Indexes program on blood dataset. The 4 node computation result is shown in fig3. The 1 node computation result is shown in fig 4.

It took 1hrs, 28 mins, 15 secs = 5,295 secs --- 4 node computation (cssmpi8h ~ cssmpi11h)

It took 4hr, 51 mins, 10 secs = 17,470 secs --- 1 node computation (cssmpi8h)

The performance improvement is 17,470/5,295 = 3.3

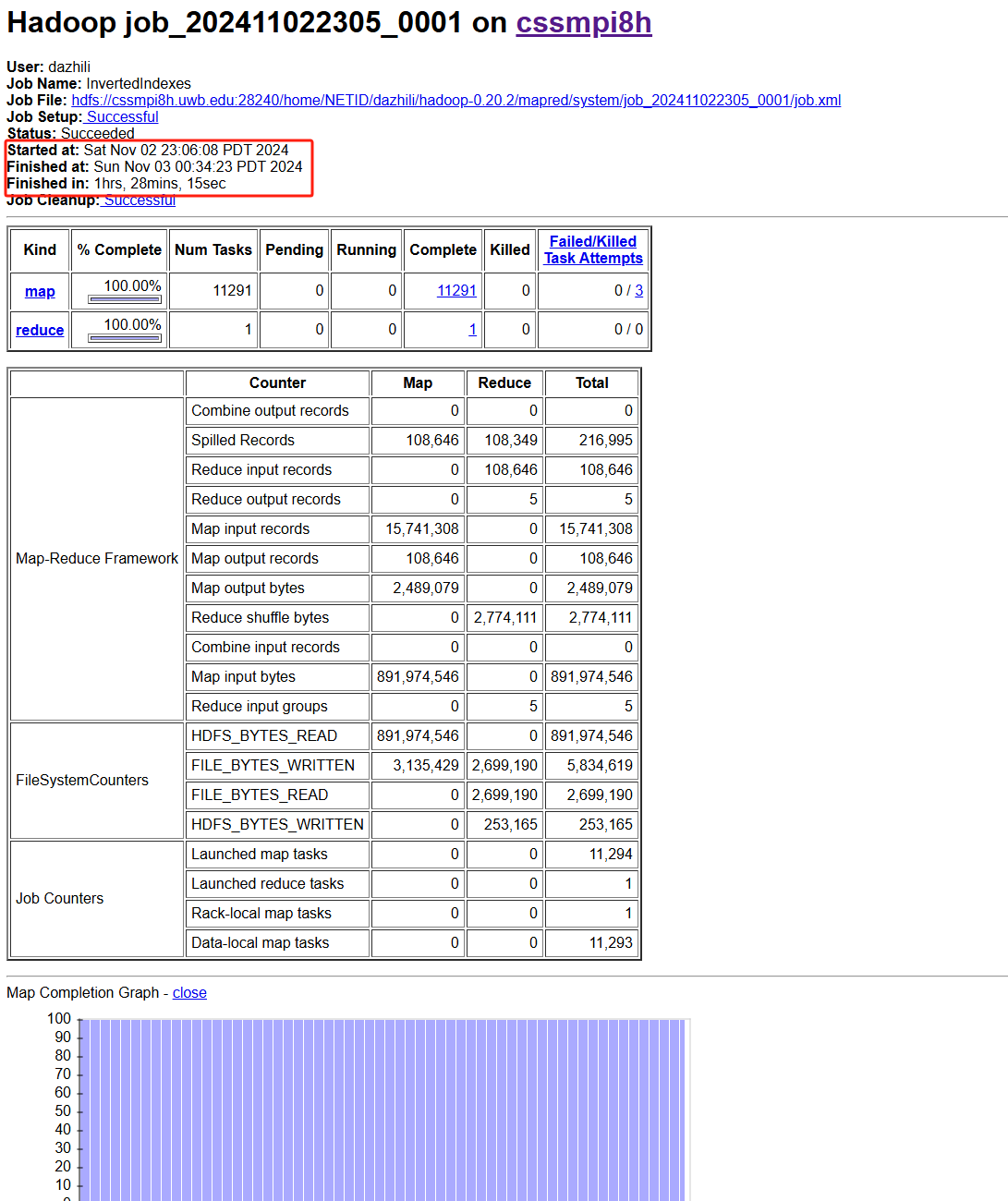


Fig3. Blood dataset running result – 4 node computation

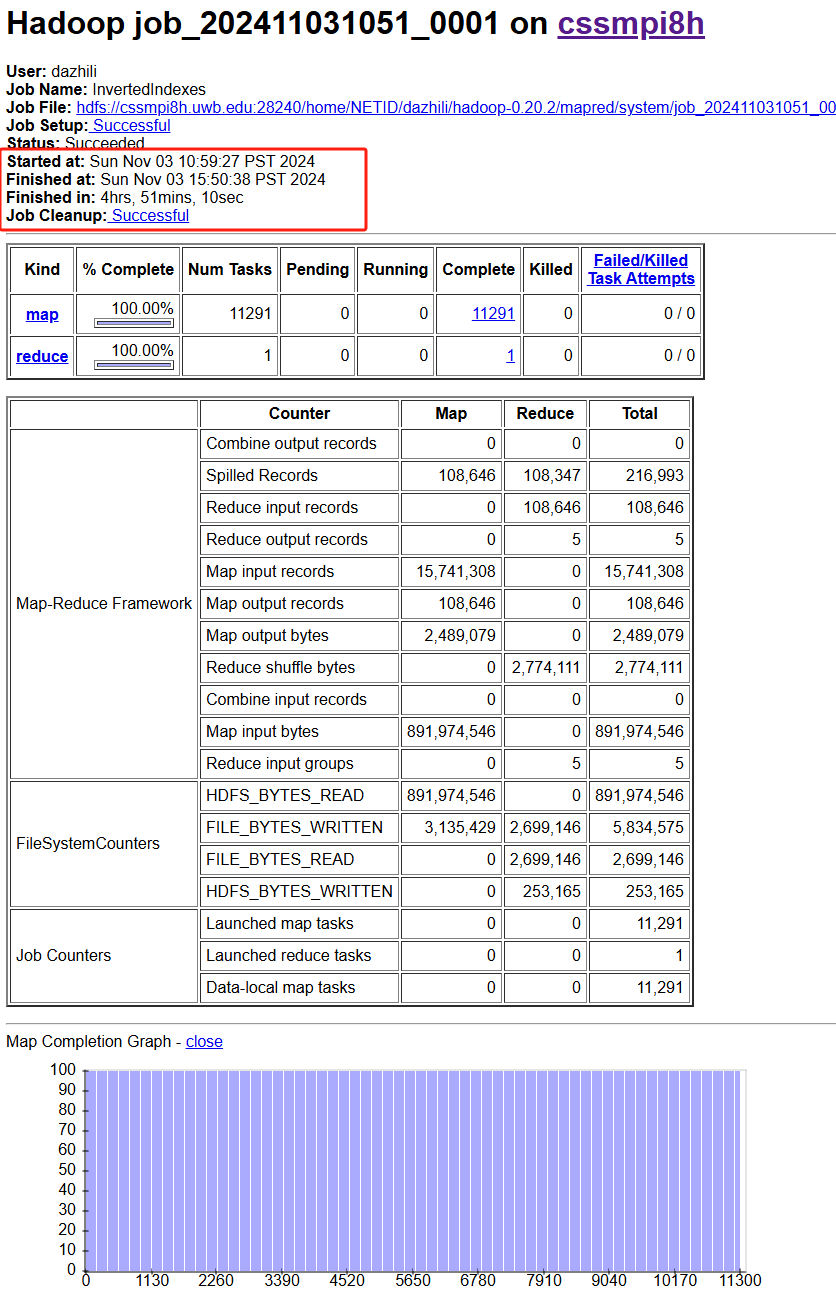


Fig4. Blood dataset running result – 1 node computation

## 3.2 Standard Result on RFC Dataset

It took 58 secs --- 4 node computation (cssmpi8h) shown in fig5.

It took 4 mins, 4 secs = 244 secs --- 1 node computation (cssmpi8h) shown in fig6.

The performance improvement is 244/58= 4.2

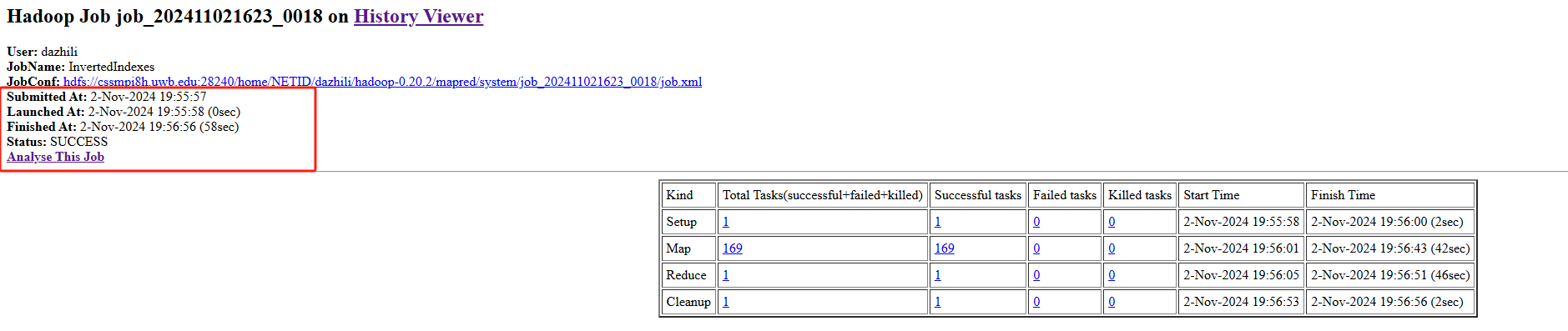


Fig5. RFC dataset running result – 4 node computation

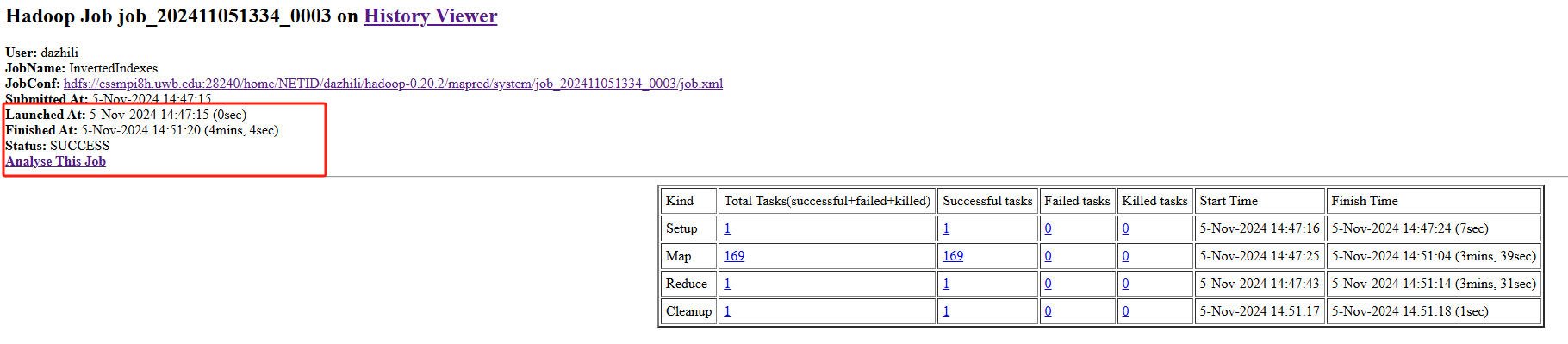


Fig6. RFC dataset running result – 1 node computation

## 3.3 Correct Execution

The Inversed Indexes program generates different output each time. However, the total number of txt files is fixed each time. And by comparing specific file keywork occurrence time, I could prove that this program runs totally correct as the professor’s example. The correct answer was shown in fig7.

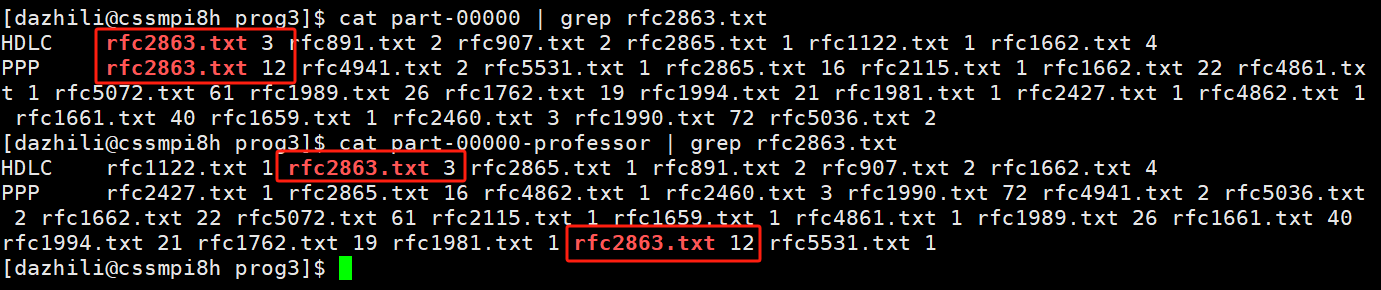
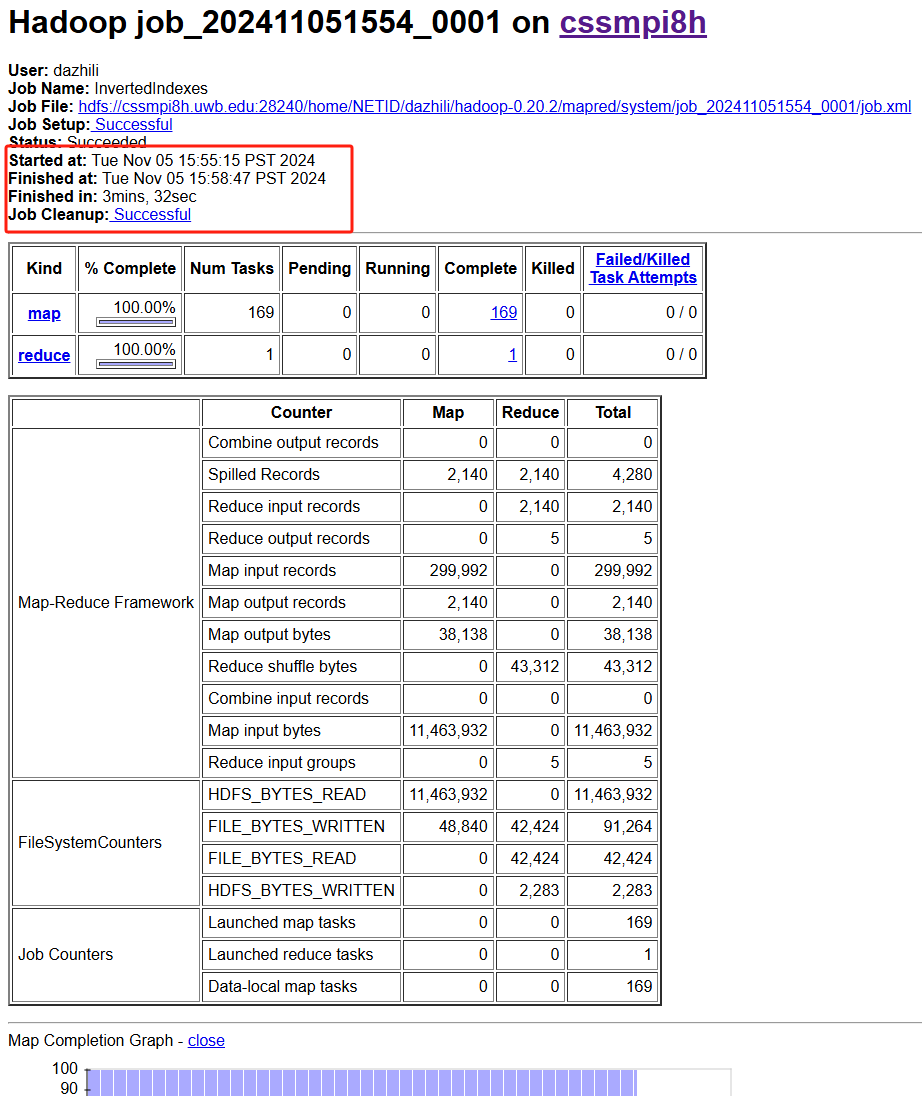
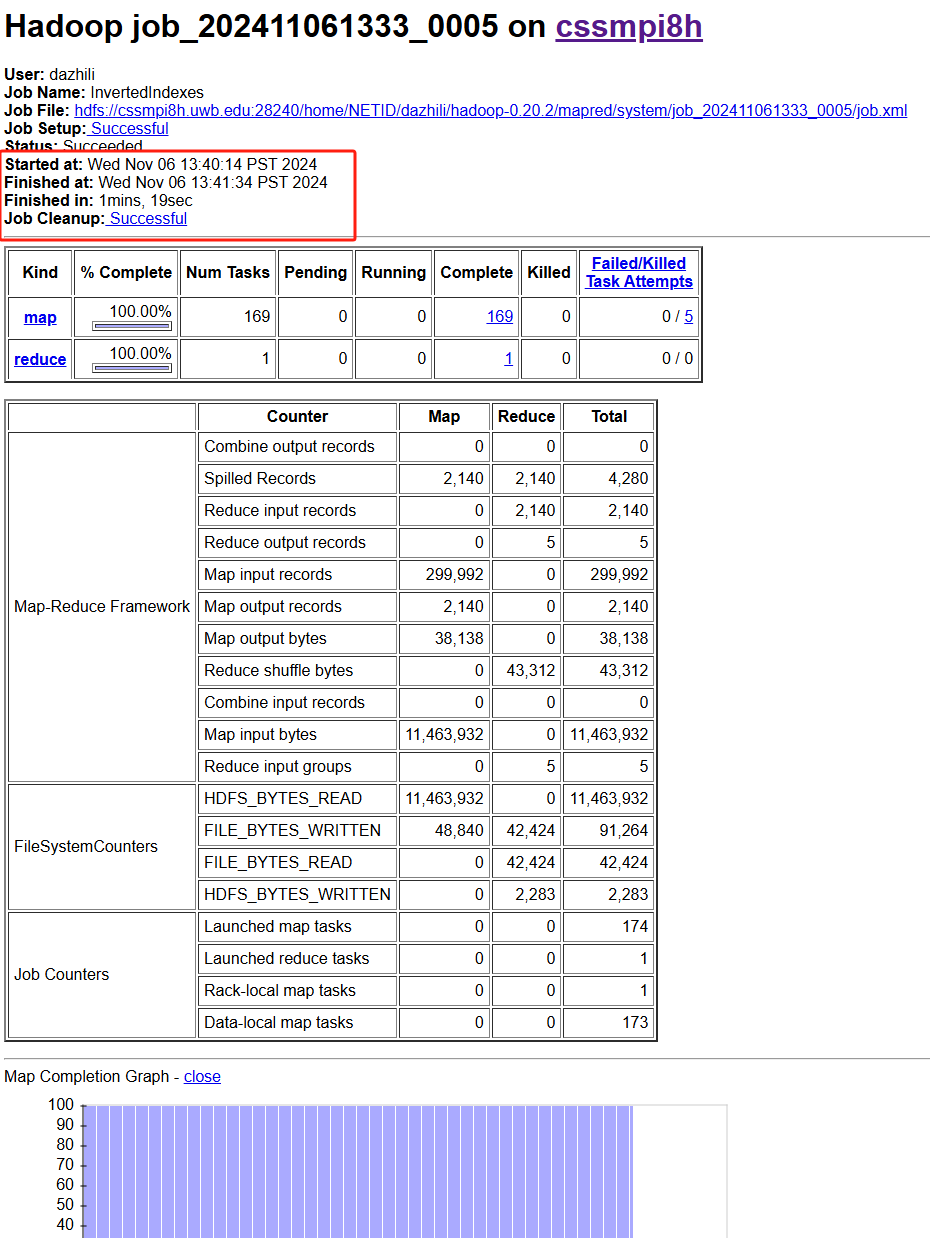


Fig7. Proof of correct execution





# 4.Discussions

## 4.1 File distribution over a cluster system

In MPI, if a file need to be shared with other nodes, we have to manually called MPI communication functions like MPI\_Send or MPI\_Bcast to explicitly share this message with other nodes. In this case, file were usually manipulated in the application layer by programmers.

In Hadoop File System, things are totally different. Hadoop will abstract a file system to share raw files within a cluster. User only need to put their file on the abstract file system and Hadoop will handle the rest of it. Moreover, Hadoop is extremely good at handing a large amount of files, and those management is totally transparent to the user.

## 4.2 Collective/Reductive operation to create inverted indexing

In MPI, if I want to achieve Inversed Indexes, this is more difficult than in MapReduce. I need a master node to scatter files/lines to each node to go though every keyword and count for it. After that, each node should send message back to the master node. Mater node then scatter reduce work to each MPI machine to reduce the result on the same keyword. Finally, all the nodes send information back to the master and master should work on its own to get a final result. This communication is quite tedious.

In MapReduce, I only need to define a mapper and a reducer which supply functions on data processing. MapReduce will split tasks to each slave machine. This way is much easier than MPI. It is also worthy to mention that tasks split in MPI is totally operated by the programmer, which could result in unequal distribution of tasks.

## 4.3 Amount of boilerplate code

In MPI, the amount of boilerplate code I think is much larger than that in MapReduce. This is because in MPI I have to make a lot of collective communication code. Also, the manually data partitioning part is quite heavy.

In MapReduce, since MapReduce does many things automatically like tasks split and fault tolerance, the amount of boilerplate code is much less. However, it does have some boilerplate code like job configuration part.

## 4.4 Anticipated execution performance

In my opinion, MPI will outperform MapReduce in most cases if performance is the only metric. MPI manipulate data in memory while MapReduce relies on the disk, which is a huge overhead for data exchanging. In our wordcount example, MPI will have better performance than MapReduce. However, when the data amount goes to a large scale, MapReduce may perform better than MPI.

## 4.5 Fault tolerance; recovery from a crash

If I am using MPI to do the Inversed Indexes computation, I am going to monitor all the works distributed on the master node. If any crash happens to the slave node, e.g. long time without receiving message from the slave, the master node should re-assign this task to another slave node. The crashed node will not be used for computation until the connection is back.

# 5.LAB3 Details

## 5.1 Implementation

As the implementation is already finished, no source code will be provided. However, all files are zipped in the attachment.

## 5.2 Execution Result

1) A snapshot of a MapReduce execution on your account: bin/hadoop jar .... as shown in fig8.

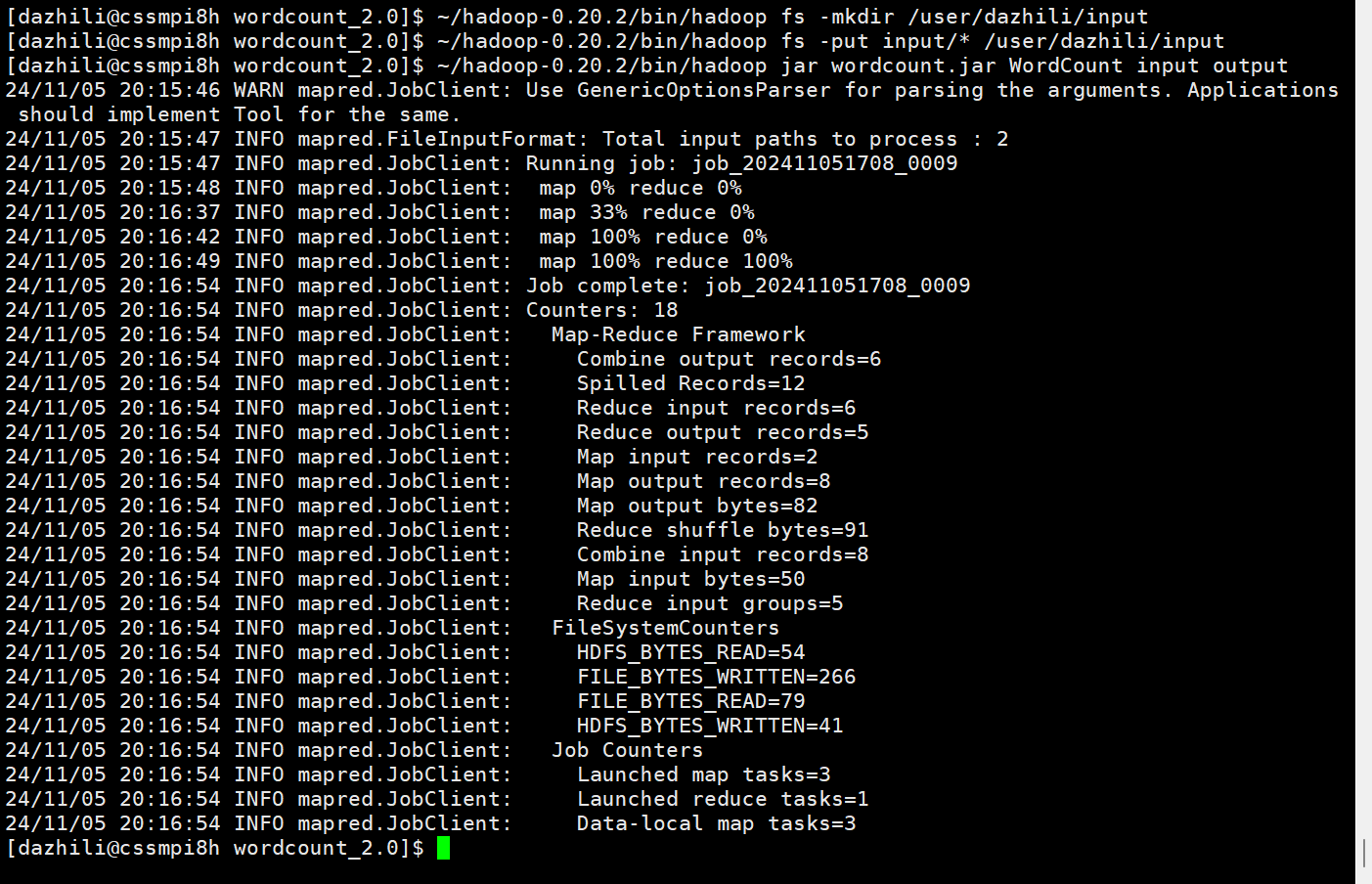


Fig8. Execution result for lab3

2) A snapshot of your /user/yourAccount/output as shown in fig9.

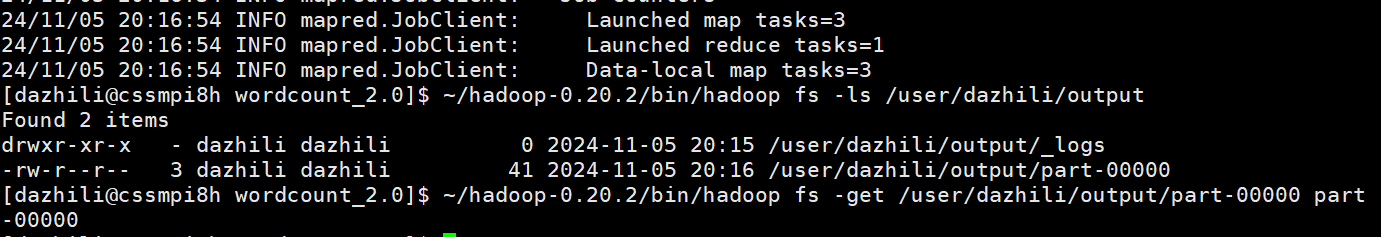


Fig9. Output directory preview

3) A file of part-00000 as shown in fig10.

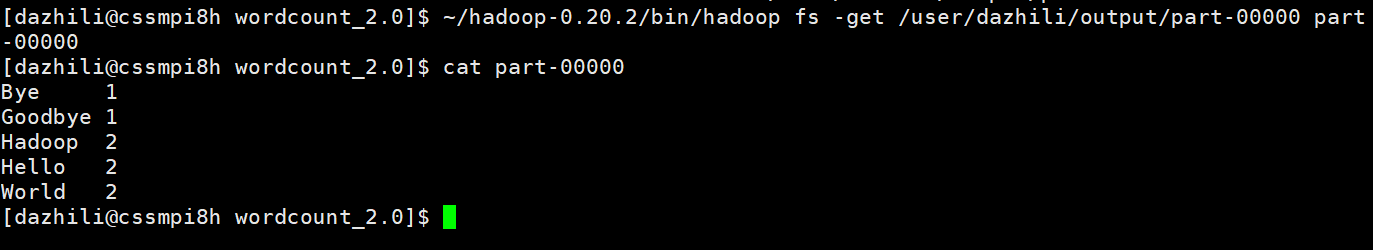


Fig10. Part-00000 result for lab3

# 6.Summary

In this report I briefly documented the Inversed Indexes program and how I implement it with MapReduce. After experiencing both MPI and MapReduce usage, I discussed about the difference between these two in some perspectives.