**Assignment 4**

**PART 1. Read the following paper (attached), and write a short summary/report.**

Google has, over the past few years, built out a massively scalable infrastructure for its search engine and other applications, including Google Maps, Google Earth, GMail, Google Finance, and Google Apps. Google’s approach was to solve the problem at every level of the application stack. The goal was to build a scalable infrastructure for parallel processing of large amounts of data. Google therefore created a full mechanism that included a distributed file system, a column-family-oriented data store, a distributed coordination system, and a MapReduce-based parallel algorithm execution environment. Graciously enough, Google published and presented a series of papers explaining some of the key pieces of its infrastructure. The most important of these publications are as follows. Read the following papers, and write a short summary/report for each paper.

[http://static.googleusercontent.com/media/research.google.com/en/us/archive/gfs-sosp2003.pdf (Links to an external site.)](http://static.googleusercontent.com/media/research.google.com/en/us/archive/gfs-sosp2003.pdf)

[http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf (Links to an external site.)](http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf)

[http://static.googleusercontent.com/media/research.google.com/en/us/archive/bigtable-osdi06.pdf (Links to an external site.)](http://static.googleusercontent.com/media/research.google.com/en/us/archive/bigtable-osdi06.pdf)

[http://static.googleusercontent.com/media/research.google.com/en/us/archive/chubby-osdi06.pdf (Links to an external site.)](http://static.googleusercontent.com/media/research.google.com/en/us/archive/chubby-osdi06.pdf)

1. The Google File System

This paper introduces the progress of Google designing and implementing the Google File System, which shares many of the same goals as previous distributed file systems.

Google tried different ways in designing this file system. First, component failures are the norm rather than the exception. Second, files are huge by traditional standards. Third, most files are mutated by appending new data rather than overwriting existing data. Fourth, co-designing the applications and the file system API benefits the overall system by increasing flexibility.

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. It provides fault tolerance by constant monitoring, replicating crucial data and fast and automatic recovery. GFS also delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. GFS has met Google’s storage needs as the storage platform for research and development as well as production data processing

1. MapReduce: Simplified Data Processing on Large Clusters

This paper introduces the implement and usage of MapReduce. MapReduce is a programming model and an associated implementation for processing and generating large data sets.

The main content of this paper includes: the basic programming model; an implementation of the MapReduce interface tailored towards our cluster-based computing environment; several refinements of the programming model; performance measurements of implementation for a variety of tasks; the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system.

From this paper, I learn that: restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant; network bandwidth is a scarce resource; , redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

1. Bigtable: A Distributed Storage System for Structured Data

This paper introduces the design and implement of Bigtable, which is a distributed storage system for managing structured data. Bigtable provides a different interface than other systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage.

The main content of this paper includes: the data model; client API; the underlying Google infrastructure on which Bigtable depends; the fundamentals of the Bigtable implementation; some of the refinements to improve Bigtable’s performance.

From this paper, I learn that: large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols; it is important to delay adding new features until it is clear how the new features will be used; proper system-level monitoring is very important; the simple designs is very valuable.

1. The Chubby lock service for loosely-coupled distributed systems

This paper describe design and implement of Chubby lock service. The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment.

The author of the opening article mainly introduces Chubby, Google’s internal distributed lock service, which can provide coarse-grained lock services in a distributed environment, and can achieve synchronization, storage of metadata, and topology in a loosely coupled distributed system composed of large-scale small machines Or configuration information and other functions.

In the second part, the author described why they chose to do a centralized lock service. Developers use local clients to access the lock service instead of creating a Paxos algorithm library for developers to use. The author stated that this reason is not only determined by the developer's needs, but also because the central lock service can reduce the number of services that the application depends on. Such a design requires the client to be able to actively discover the data changes of the central service. At the same time, the client needs to be able to cache information to reduce the pressure on the server. Of course, some access restrictions are also required.

Next, the author describes why they chose to provide coarse-grained lock services instead of fine-grained, because this can make the lock holding time longer and reduce server pressure; fine-grained lock services will be unavailable even in small-scale Cause a large number of clients to stop working; application developers can implement fine-grained locks within the business themselves.

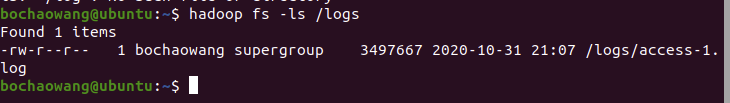
**PART 2 – Programming Assignment**

Copy the attached ‘access.log’ file into HDFS under /logs directory.

1. Import file to Hadoop:

hadoop fs -appendToFile access-1.log /logs/access-1.log

import result:



Using the access.log file stored in HDFS, implement MapReduce in Hadoop to find the number of times each IP accessed the website.

Mapper:

1. **import** org.apache.hadoop.io.IntWritable;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Mapper;
4. **import** java.io.IOException;
6. **public** **class** LogMapper **extends** Mapper<Object, Text, Text, IntWritable> {
8. **public** **void** map (Object key,Text value, Context context)
9. **throws** IOException, InterruptedException{
10. String[] inputSplit = value.toString().split(" ");
11. String url = inputSplit[0];
12. IntWritable one = **new** IntWritable(1);
14. context.write(**new** Text(url),one);
15. }
16. }

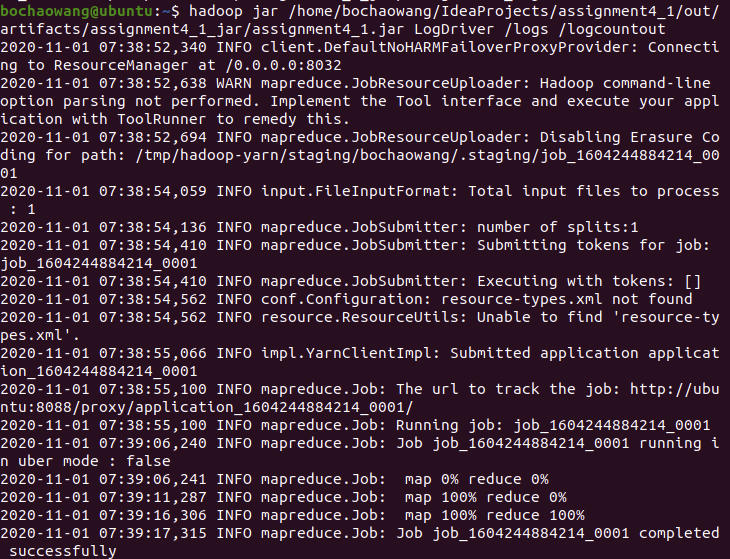
Reducer:

1. **import** org.apache.hadoop.io.IntWritable;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Reducer;
5. **import** java.io.IOException;
6. **public** **class** LogReducer **extends** Reducer<Text, IntWritable, Text, IntWritable>{
8. **protected** **void** reduce(Text key, Iterable<IntWritable> values, Context context)
9. **throws** IOException,
10. InterruptedException {
11. **int** urlcount = 0;
12. **for**(IntWritable val : values)
13. {
14. urlcount+=val.get();
15. }
16. context.write(key, **new** IntWritable(urlcount));
17. }
18. }

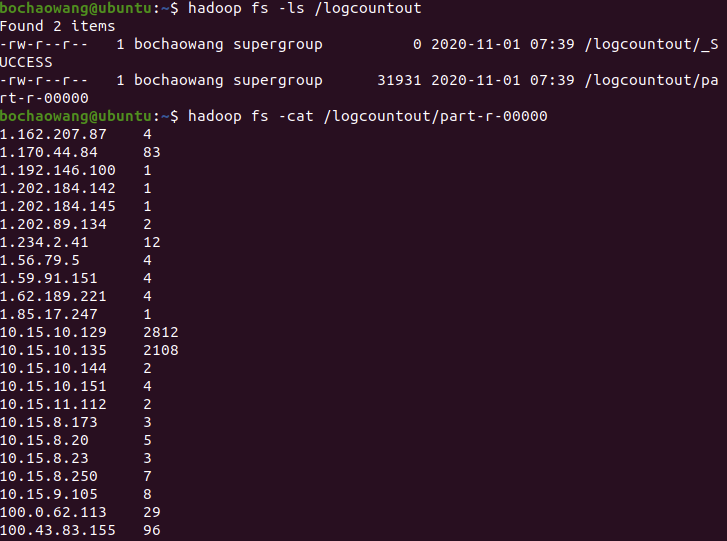
Driver:

1. **import** org.apache.hadoop.conf.Configuration;
2. **import** org.apache.hadoop.fs.FileSystem;
3. **import** org.apache.hadoop.fs.Path;
4. **import** org.apache.hadoop.io.IntWritable;
5. **import** org.apache.hadoop.io.Text;
6. **import** org.apache.hadoop.mapreduce.Job;
7. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
8. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
9. **import** java.io.IOException;
10. **public** **class** LogDriver {
11. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
12. Configuration conf = **new** Configuration();
13. FileSystem fs =FileSystem.get(conf);
14. **if**(fs.exists(**new** Path(args[1])))
15. {
16. fs.delete(**new** Path(args[1]), **true**);
17. }
18. Job job = Job.getInstance(conf);
19. job.setMapperClass(LogMapper.**class**);
20. job.setReducerClass(LogReducer.**class**);
21. job.setJarByClass(LogDriver.**class**);
22. job.setNumReduceTasks(1);
23. TextInputFormat.addInputPath(job, **new** Path(args[0]));
24. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
25. job.setMapOutputKeyClass(Text.**class**);
26. job.setMapOutputValueClass(IntWritable.**class**);
27. job.setOutputKeyClass(Text.**class**);
28. job.setOutputValueClass(IntWritable.**class**);
29. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
30. }
31. }

After build, use hadoop jar command to execute mapreduce:



Result:



**PART 3 – Programming Assignment**

Download and Copy all the files ([http://msis.neu.edu/nyse/ (Links to an external site.)](http://msis.neu.edu/nyse/)) (DailyPrices\_A to DailyPrices\_Z) to a folder in HDFS.

Use the command below to import all files:

FILES=NYSEr/NYSE\_\*.csv

for f in $FILES

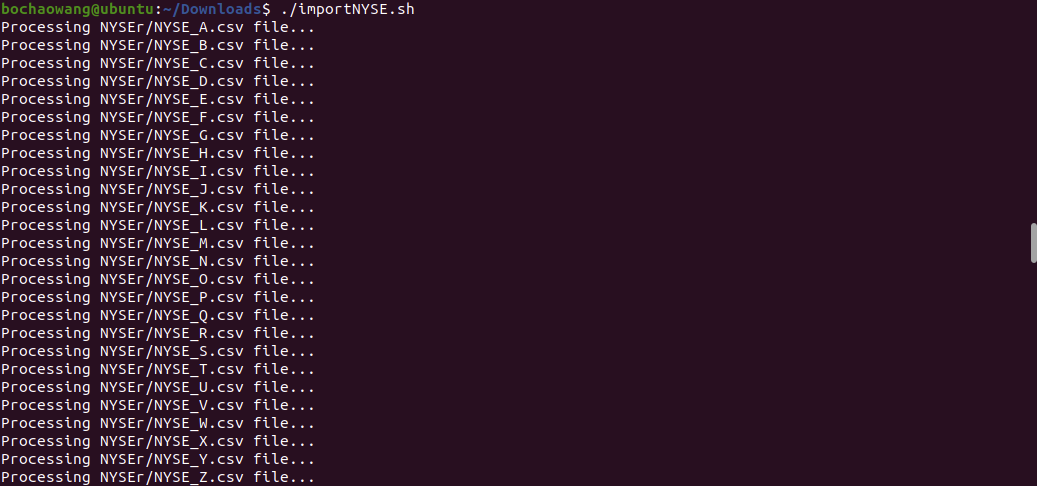
do

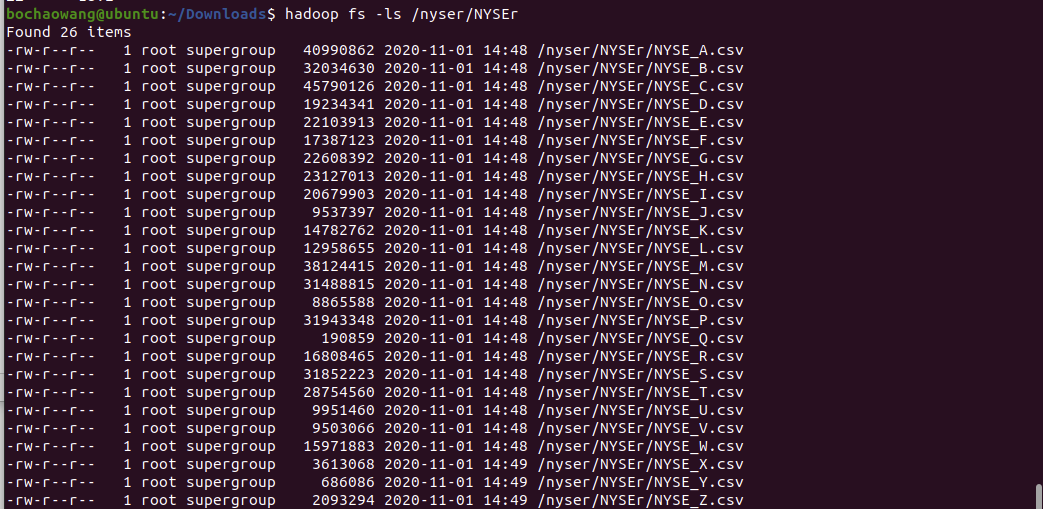
echo "Processing $f file..."

sudo /usr/local/bin/hadoop-3.3.0/bin/hadoop fs -appendToFile $f /nyser/$f

done

import result:





**PART 3.1 – Write a MapReduce to find the Max price of stock\_price\_high for each stock. Capture the running time programmatically (or manually using a wristwatch or smartphone).**

Maxtuple:

1. **import** org.apache.hadoop.io.Writable;
2. **import** java.io.DataInput;
3. **import** java.io.DataOutput;
4. **import** java.io.IOException;
6. **public** **class** MinMaxTuple **implements** Writable{
7. **private** Double max;
9. **public** MinMaxTuple() {
10. }

13. **public** **void** write(DataOutput out) **throws** IOException {
14. out.writeDouble(max);
15. }
17. **public** **void** readFields(DataInput in) **throws** IOException {
18. max = in.readDouble();
19. }

22. **public** Double getMax() {
23. **return** max;
24. }
26. **public** **void** setMax(Double max) {
27. **this**.max = max;
28. }

31. **public** String toString()
32. {
33. **return** max.toString();
34. }
35. }

Mapper:

1. **import** org.apache.hadoop.io.LongWritable;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Mapper;
5. **import** java.io.IOException;
6. **public** **class** NyseMapper **extends** Mapper<LongWritable, Text, Text, MinMaxTuple > {
7. MinMaxTuple minMaxTuple = **new** MinMaxTuple();
8. Text deptKey = **new** Text();
10. **protected** **void** map(LongWritable key, Text value, Context context)
11. **throws** IOException,
12. InterruptedException, IOException {
13. String line = value.toString();
14. String[] tokens = line.split(",",-1);
15. **double** high = 0.0;
17. **if**(tokens[4].length()>0){
18. high = Double.parseDouble(tokens[4]);
19. minMaxTuple.setMax(high);
20. deptKey.set(tokens[1]);
21. }
22. context.write(deptKey, minMaxTuple);
23. }
24. }

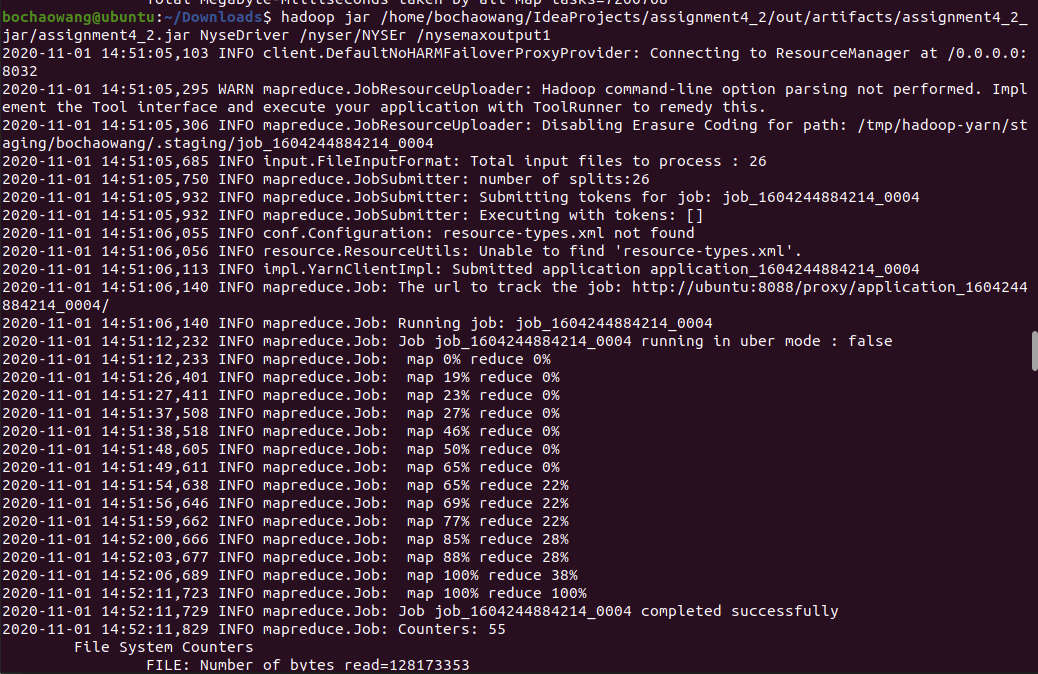
Reducer:

1. **import** org.apache.commons.math3.analysis.function.Min;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Reducer;
5. **import** java.io.IOException;
6. **import** java.util.ArrayList;
7. **import** java.util.Collections;
8. **import** java.util.List;
9. **public** **class** NyseReducer **extends** Reducer<Text, MinMaxTuple, Text, MinMaxTuple>{
10. **private** MinMaxTuple result= **new** MinMaxTuple();
11. **protected** **void** reduce(Text key, Iterable<MinMaxTuple> values, Context context)
12. **throws** IOException,
13. InterruptedException, IOException {
14. result.setMax(**null**);
15. List<Double> high\_prices = **new** ArrayList<Double>();
16. **for**(MinMaxTuple minMaxTuple : values)
17. {
18. high\_prices.add(minMaxTuple.getMax());
19. **if**(result.getMax() == **null** || minMaxTuple.getMax()>result.getMax()){
20. result.setMax(minMaxTuple.getMax());
21. }
22. }
23. context.write(key,result);
24. }
25. }

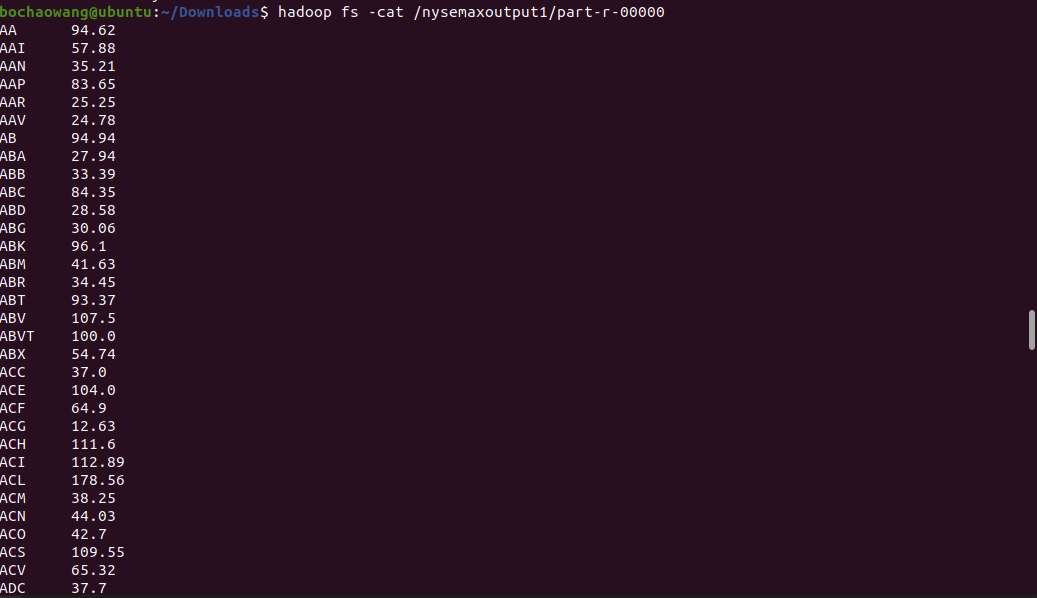
Driver:

1. **import** org.apache.hadoop.conf.Configuration;
2. **import** org.apache.hadoop.fs.FileSystem;
3. **import** org.apache.hadoop.fs.Path;
4. **import** org.apache.hadoop.io.Text;
5. **import** org.apache.hadoop.mapreduce.Job;
6. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
7. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
9. **import** java.io.IOException;
10. **public** **class** NyseDriver {
11. **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException, InterruptedException {
12. Configuration configuration = **new** Configuration();
13. Job job = Job.getInstance(configuration, "Summarization Pattern");
14. job.setJarByClass(NyseDriver.**class**);
16. job.setMapOutputKeyClass(Text.**class**);
17. job.setMapOutputValueClass(MinMaxTuple.**class**);
19. job.setInputFormatClass(TextInputFormat.**class**);
20. job.setOutputFormatClass(TextOutputFormat.**class**);
22. job.setOutputKeyClass(Text.**class**);
23. job.setOutputValueClass(MinMaxTuple.**class**);
25. job.setMapperClass(NyseMapper.**class**);
26. job.setReducerClass(NyseReducer.**class**);
28. TextInputFormat.addInputPath(job, **new** Path(args[0]));
29. Path outDir = **new** Path(args[1]);
30. TextOutputFormat.setOutputPath(job, outDir);
32. FileSystem fs = FileSystem.get(job.getConfiguration());
33. **if**(fs.exists(outDir)){
34. fs.delete(outDir, **true**);
35. }
36. job.waitForCompletion(**true**);
37. }
38. }

Run job:



Result:



Run time:

Begin at 2020-11-01 14:51:05,103

End at 2020-11-01 14:52:11,829

It takes 1:06,826 minute

**PART 3.2 – Write a Java Program to implement PutMerge as discussed in the class** to merge the NYSE files in a single file on HDFS. Now, repeat 4.1 on the single merged-file. Capture the running time.

Did MapReduce on a single file run faster than running MapReduce on a bunch of files?

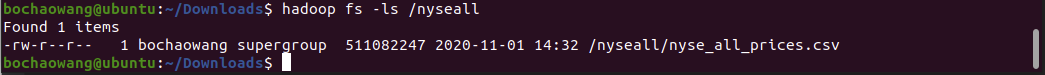
 PutMerge.java:

1. /\*
2. \* To change this license header, choose License Headers in Project Properties.
3. \* To change this template file, choose Tools | Templates
4. \* and open the template in the editor.
5. \*/
6. **package** com.mycompany.nysemerge;
8. **import** java.io.File;
9. **import** java.io.IOException;
10. **import** java.nio.charset.Charset;
11. **import** java.nio.file.Files;
12. **import** java.nio.file.Path;
13. **import** java.nio.file.Paths;
14. **import** java.util.ArrayList;
15. **import** java.util.Arrays;
16. **import** java.util.List;
18. /\*\*
19. \*
20. \* @author wangbaichao
21. \*/
22. **public** **class** PutMerge {
23. **public** **static** **void** main(String[] args) **throws** IOException {
24. List<Path> pathss = **new** ArrayList<>();
25. String path="/Users/wangbaichao/Downloads/NYSEr";
26. File file = **new** File(path);
27. File[] tempList = file.listFiles();
29. **for** (**int** i = 0; i < tempList.length; i++) {
30. **if** (tempList[i].isFile()) {
32. Path cur = Paths.get(tempList[i].toString());
33. pathss.add(cur);
34. }
36. }
37. List<String> mergedLines = getMergedLines(pathss);
38. Path target = Paths.get("/Users/wangbaichao/Downloads/NYSEall/ nyse\_all\_prices.csv ");
39. Files.write(target, mergedLines, Charset.forName("UTF-8"));
40. }
42. **private** **static** List<String> getMergedLines(List<Path> paths) **throws** IOException {
43. List<String> mergedLines = **new** ArrayList<> ();
44. **for** (Path p : paths){
45. List<String> lines = Files.readAllLines(p, Charset.forName("UTF-8"));
46. **if** (!lines.isEmpty()) {
47. **if** (mergedLines.isEmpty()) {
48. mergedLines.add(lines.get(0)); //add header only once
49. }
50. mergedLines.addAll(lines.subList(1, lines.size()));
51. }
52. }
53. **return** mergedLines;
54. }
55. }

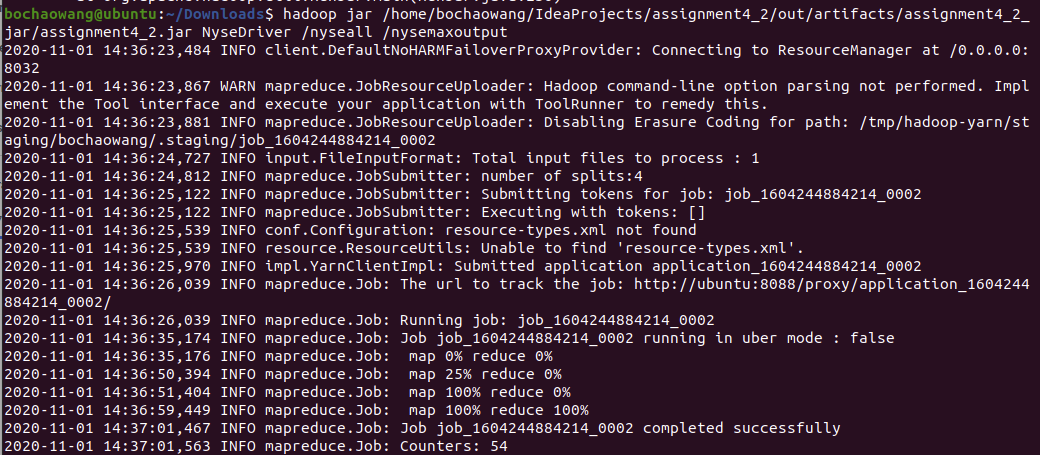
Import merged file to HDFS:

hadoop fs -appendToFile nyse\_all\_prices.csv /nyseall/nyse\_all\_prices.csv

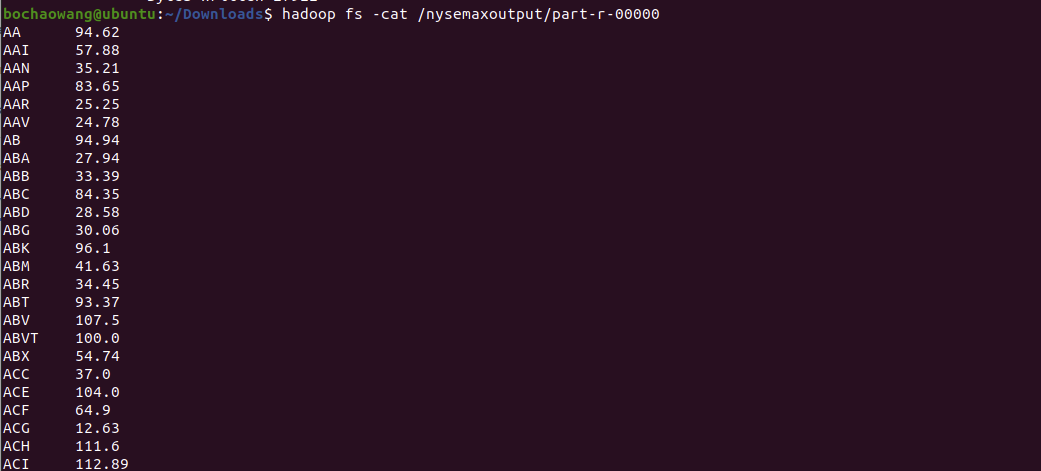
import result:



Run job:



Result:



Runtime:

Begin at : 2020-11-01 14:36:23,484

End at : 2020-11-01 14:37:01,563

It takes 38.081 seconds

It is faster than running job on multiple files.

**PART 4 – Programming Assignment**

Write one MapReduce program using each of the classes that extend FileInputFormat<k,v>

(CombineFileInputFormat, FixedLengthInputFormat, KeyValueTextInputFormat, NLineInputFormat, SequenceFileInputFormat, TextInputFormat)

http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/lib/input/FileInputFormat.html

You could use any input file of your choice. The size of the input files is not important. The MR programs could simply do counting, or any other analysis you choose.

1) CombineFileInputFormat

this method is to solve small files. So I split part of nyse\_all\_prices.csv to 28 files and put them into hdfs

mapper:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.\*;
3. **import** org.apache.hadoop.fs.\*;
4. **import** org.apache.hadoop.io.\*;
5. **import** org.apache.hadoop.mapreduce.lib.input.\*;
6. **import** org.apache.hadoop.mapreduce.\*;
7. **import** org.apache.hadoop.mapreduce.lib.output.\*;
8. **import** org.apache.hadoop.util.\*;
9. **public** **class** CombineMapper **extends** Mapper<Object, Text, Text, IntWritable >{
10. **public** **void** map (Object key,Text value, Context context)
11. **throws** IOException, InterruptedException{
12. String[] inputSplit = value.toString().split(",");
13. String symble = inputSplit[1];
14. IntWritable one = **new** IntWritable(1);
16. context.write(**new** Text(symble),one);
17. }
18. }

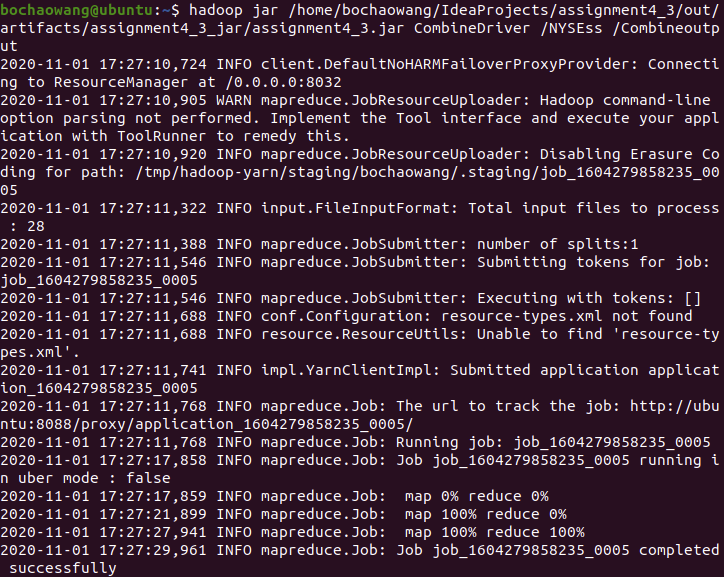
Reducer:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.\*;
3. **import** org.apache.hadoop.fs.\*;
4. **import** org.apache.hadoop.io.\*;
5. **import** org.apache.hadoop.mapreduce.lib.input.\*;
6. **import** org.apache.hadoop.mapreduce.\*;
7. **import** org.apache.hadoop.mapreduce.lib.output.\*;
8. **import** org.apache.hadoop.util.\*;
9. **public** **class** CombineReducer **extends** Reducer<Text, IntWritable, Text, IntWritable>{
11. **protected** **void** reduce(Text key, Iterable<IntWritable> values, Context context)
12. **throws** IOException,
13. InterruptedException {
14. **int** symblecount = 0;
15. **for**(IntWritable val : values)
16. {
17. symblecount+=val.get();
18. }
19. context.write(key, **new** IntWritable(symblecount));
20. }
21. }

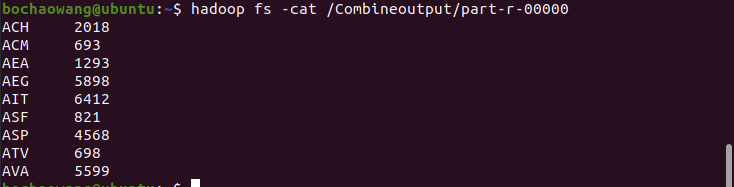
Driver:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.\*;
3. **import** org.apache.hadoop.fs.\*;
4. **import** org.apache.hadoop.io.\*;
5. **import** org.apache.hadoop.mapreduce.lib.input.\*;
6. **import** org.apache.hadoop.mapreduce.\*;
7. **import** org.apache.hadoop.mapreduce.lib.output.\*;
8. **import** org.apache.hadoop.util.\*;
9. **public** **class** CombineDriver {
10. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
11. Configuration conf = **new** Configuration();
12. FileSystem fs =FileSystem.get(conf);
13. **if**(fs.exists(**new** Path(args[1])))
14. {
15. fs.delete(**new** Path(args[1]), **true**);
16. }
17. Job job = Job.getInstance(conf);
18. job.setInputFormatClass(CombineTextInputFormat.**class**); // specify the input format
20. job.setMapperClass(CombineMapper.**class**);
21. job.setReducerClass(CombineReducer.**class**);
22. job.setJarByClass(CombineDriver.**class**);
23. job.setNumReduceTasks(1);
24. TextInputFormat.addInputPath(job, **new** Path(args[0]));
25. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
26. job.setMapOutputKeyClass(Text.**class**);
27. job.setMapOutputValueClass(IntWritable.**class**);
28. job.setOutputKeyClass(Text.**class**);
29. job.setOutputValueClass(IntWritable.**class**);
30. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
31. }
32. }

Run job:



Result:



Runtime:

Begin: 2020-11-01 17:27:10,724

End: 2020-11-01 17:27:30,038

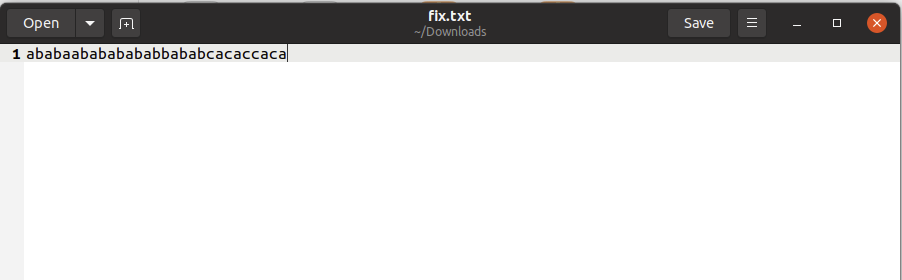
It takes 19.314 seconds

If donot use CombineFileInputFormat, it take more than 2 minites

1. FixedLengthInputFormat

FixedLengthInputFormat is an input format used to read input files which contain fixed length records.

I create 1 txt file like this:



It is 30 bytes

Use command below to set record length:

FixedLengthInputFormat.setRecordLength(conf, 5);

Mapper:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.\*;
3. **import** org.apache.hadoop.fs.\*;
4. **import** org.apache.hadoop.io.\*;
5. **import** org.apache.hadoop.mapreduce.lib.input.\*;
6. **import** org.apache.hadoop.mapreduce.\*;
7. **import** org.apache.hadoop.mapreduce.lib.output.\*;
8. **import** org.apache.hadoop.util.\*;
9. **public** **class** CMapper **extends** Mapper<Object, BytesWritable, Text, IntWritable >{
10. **private** Text word = **new** Text();
12. @Override
13. **public** **void** map(Object key, BytesWritable value, Context context) **throws** IOException, InterruptedException {
14. word.set(value.getBytes());
15. IntWritable ONE = **new** IntWritable(1);
16. context.write(word, ONE);
17. }
18. }

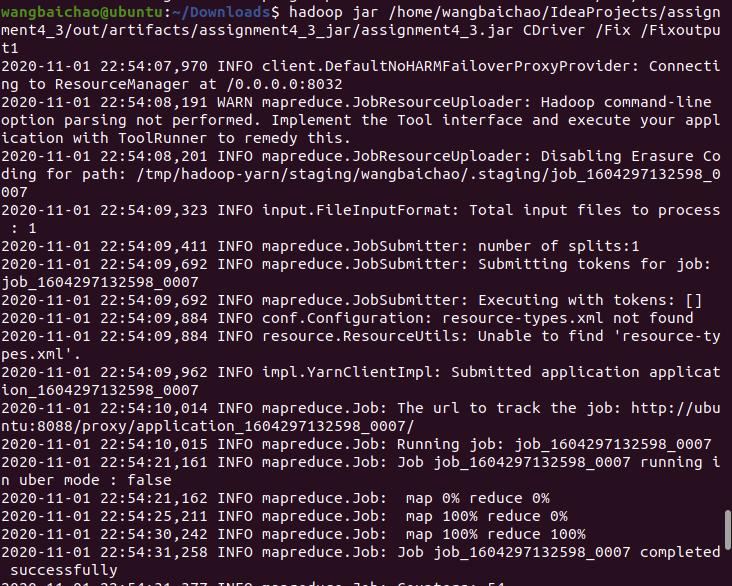
Reducer:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.\*;
3. **import** org.apache.hadoop.fs.\*;
4. **import** org.apache.hadoop.io.\*;
5. **import** org.apache.hadoop.mapreduce.lib.input.\*;
6. **import** org.apache.hadoop.mapreduce.\*;
7. **import** org.apache.hadoop.mapreduce.lib.output.\*;
8. **import** org.apache.hadoop.util.\*;
9. **public** **class** CReducer **extends** Reducer<Text, IntWritable, Text, IntWritable>{
11. **protected** **void** reduce(Text key, Iterable<IntWritable> values, Context context)
12. **throws** IOException,
13. InterruptedException {
14. **int** symblecount = 0;
15. **for**(IntWritable val : values)
16. {
17. symblecount+=val.get();
18. }
19. context.write(key, **new** IntWritable(symblecount));
20. }
21. }

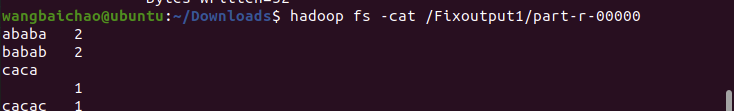
Driver:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.\*;
3. **import** org.apache.hadoop.fs.\*;
4. **import** org.apache.hadoop.io.\*;
5. **import** org.apache.hadoop.mapreduce.lib.input.\*;
6. **import** org.apache.hadoop.mapreduce.\*;
7. **import** org.apache.hadoop.mapreduce.lib.output.\*;
8. **import** org.apache.hadoop.util.\*;
9. **public** **class** CDriver {
10. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
11. Configuration conf = **new** Configuration();
12. FileSystem fs =FileSystem.get(conf);
13. **if**(fs.exists(**new** Path(args[1])))
14. {
15. fs.delete(**new** Path(args[1]), **true**);
16. }
17. FixedLengthInputFormat.setRecordLength(conf, 5);
19. Job job = Job.getInstance(conf);
20. job.setInputFormatClass(FixedLengthInputFormat.**class**); // specify the input format
22. job.setMapperClass(CMapper.**class**);
23. job.setReducerClass(CReducer.**class**);
24. job.setJarByClass(CDriver.**class**);
25. TextInputFormat.addInputPath(job, **new** Path(args[0]));
26. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
27. job.setMapOutputKeyClass(Text.**class**);
28. job.setMapOutputValueClass(IntWritable.**class**);
29. job.setOutputKeyClass(Text.**class**);
30. job.setOutputValueClass(IntWritable.**class**);
31. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
32. }
33. }

Run job:



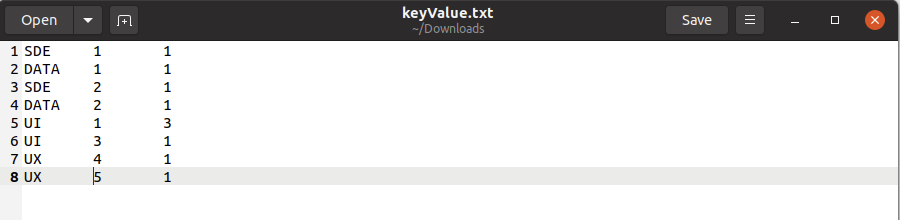
Result:



1. KeyValueTextInputFormat

KeyValueTextINputFormat is an InputFormat for plain text files. Files are broken into lines. Either line feed or carriage-return is used to signal the end of a line. Each line is divided into key and value parts by a separator byte. If no such a byte exists, the key will be the entire line and value will be empty. The separator byte can be specified in config file under the attribute name mapreduce.input.keyvaluelinerecordreader.key.value.separator. The default is the tab character ('\t').

I creat a txt like this. Use tab to split.



Mapper:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.io.IntWritable;
4. **import** org.apache.hadoop.mapreduce.Mapper;
6. **public** **class** KMapper **extends** Mapper<Text, Text, Text, IntWritable> {
7. @Override
8. **public** **void** map(Text key, Text value, Context context) **throws** IOException, InterruptedException {
9. Text word = **new** Text();
10. word.set(key);
11. IntWritable ONE = **new** IntWritable(1);
12. context.write(word, ONE);
13. }
14. }

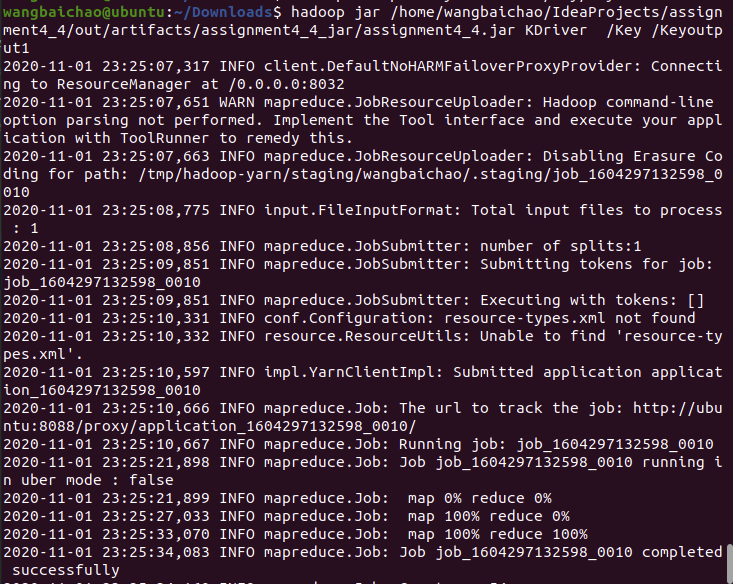
Reducer:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.io.IntWritable;
4. **import** org.apache.hadoop.mapreduce.Reducer;
6. **public** **class** KReducer **extends** Reducer<Text, IntWritable, Text, IntWritable> {
8. @Override
9. **public** **void** reduce(Text key, Iterable<IntWritable> values, Context context)
10. **throws** IOException, InterruptedException {
11. **int** counter = 0;
12. **for** (IntWritable i : values) {
13. counter += i.get();
14. }
15. context.write(key, **new** IntWritable(counter));
16. }
17. }

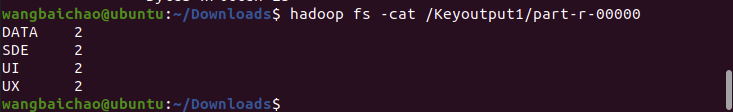
Driver:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.conf.Configuration;
3. **import** org.apache.hadoop.fs.FileSystem;
4. **import** org.apache.hadoop.fs.Path;
5. **import** org.apache.hadoop.io.Text;
6. **import** org.apache.hadoop.io.IntWritable;
7. **import** org.apache.hadoop.mapreduce.Job;
8. **import** org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
9. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
10. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
11. **public** **class** KDriver {
13. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
14. Configuration conf = **new** Configuration();
15. FileSystem fs =FileSystem.get(conf);
16. **if**(fs.exists(**new** Path(args[1])))
17. {
18. fs.delete(**new** Path(args[1]), **true**);
19. }
20. Job job = Job.getInstance(conf);
21. job.setJarByClass(KDriver.**class**);
22. job.setInputFormatClass(KeyValueTextInputFormat.**class**);
24. job.setMapperClass(KMapper.**class**);
25. job.setReducerClass(KReducer.**class**);
26. job.setJarByClass(KDriver.**class**);
27. TextInputFormat.addInputPath(job, **new** Path(args[0]));
28. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
29. job.setMapOutputKeyClass(Text.**class**);
30. job.setMapOutputValueClass(IntWritable.**class**);
31. job.setOutputKeyClass(Text.**class**);
32. job.setOutputValueClass(IntWritable.**class**);
33. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
34. }
35. }

Run job:



Result:



1. NLineInputFormat

NLineInputFormat which splits N lines of input as one split. In many "pleasantly" parallel applications, each process/mapper processes the same input file (s), but with computations are controlled by different parameters.(Referred to as "parameter sweeps"). One way to achieve this is to specify a set of parameters (one set per line) as input in a control file (which is the input path to the map-reduce application, whereas the input dataset is specified via a config variable in JobConf.). The NLineInputFormat can be used in such applications, that splits the input file such that by default, one line is fed as a value to one map task, and a key is an offset. i.e. (k,v) is (LongWritable, Text). The location hints will span the whole mapred cluster.

I use access-1.log as input, cut it to 35000 lines.

I set number of line per split to 350

So there should be 100 jobs

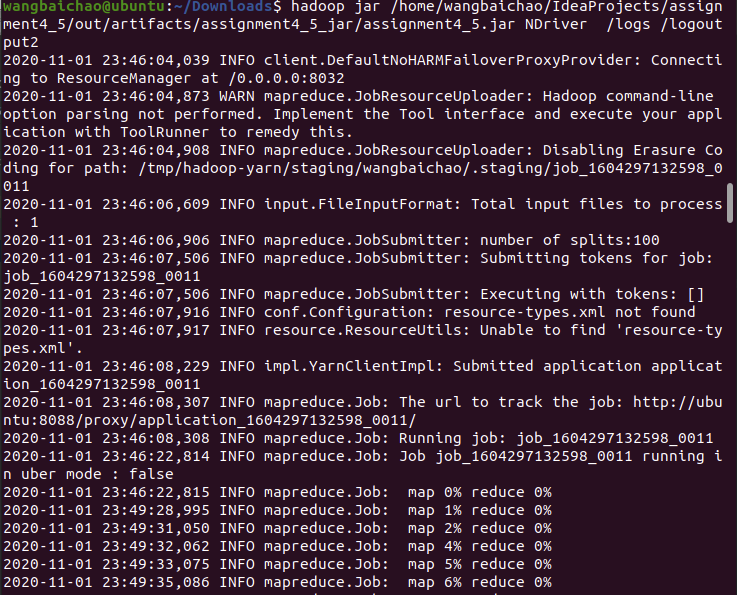
Mapper is same as LogMapper;

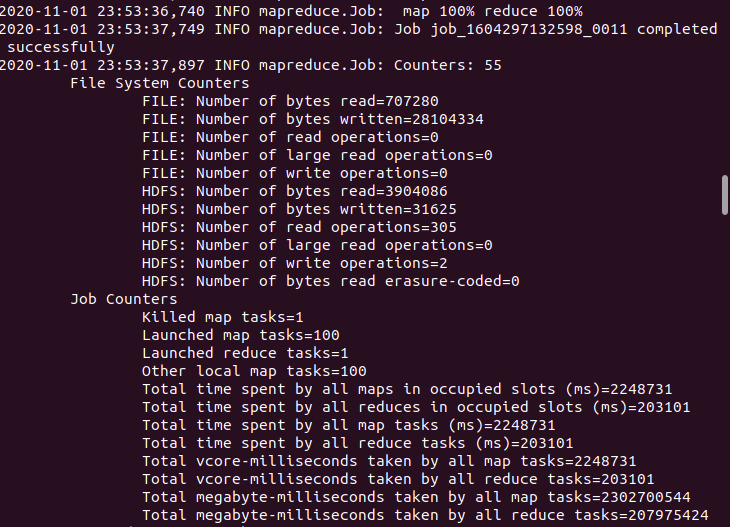
Reducer is same as LogReducer;

Driver:

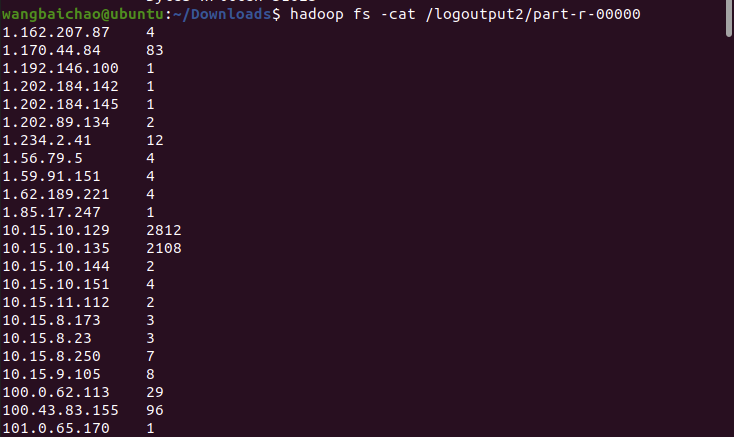
1. **import** org.apache.hadoop.conf.Configuration;
2. **import** org.apache.hadoop.fs.FileSystem;
3. **import** org.apache.hadoop.fs.Path;
4. **import** org.apache.hadoop.io.IntWritable;
5. **import** org.apache.hadoop.io.Text;
6. **import** org.apache.hadoop.mapreduce.Job;
7. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
8. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
9. **import** java.io.IOException;
10. **import** org.apache.hadoop.mapreduce.lib.input.\*;
11. **public** **class** NDriver {
12. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
13. Configuration conf = **new** Configuration();
14. FileSystem fs =FileSystem.get(conf);
15. **if**(fs.exists(**new** Path(args[1])))
16. {
17. fs.delete(**new** Path(args[1]), **true**);
18. }
20. Job job = Job.getInstance(conf);
21. job.setInputFormatClass(NLineInputFormat.**class**);
22. NLineInputFormat.setNumLinesPerSplit(job,350);
24. job.setMapperClass(NMapper.**class**);
25. job.setReducerClass(NReducer.**class**);
26. job.setJarByClass(NDriver.**class**);
27. TextInputFormat.addInputPath(job, **new** Path(args[0]));
28. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
29. job.setMapOutputKeyClass(Text.**class**);
30. job.setMapOutputValueClass(IntWritable.**class**);
31. job.setOutputKeyClass(Text.**class**);
32. job.setOutputValueClass(IntWritable.**class**);
33. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
34. }
35. }

Run job:





Result:

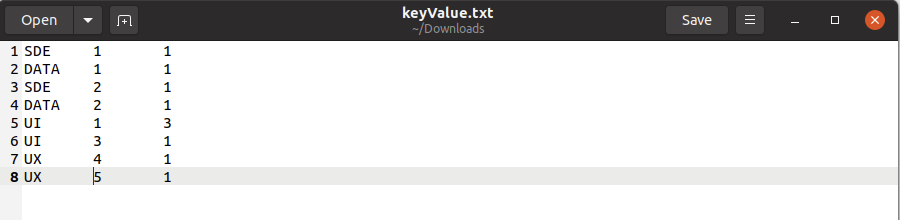


1. SequenceFileInputFormat

SequenceFiles are flat files consisting of binary key/value pairs. SequenceFile provides SequenceFile.Writer, SequenceFile.Reader and SequenceFile.Sorter classes for writing, reading and sorting respectively.

First, I convert a txt to SequenceFile

Original txt:



Mapper:

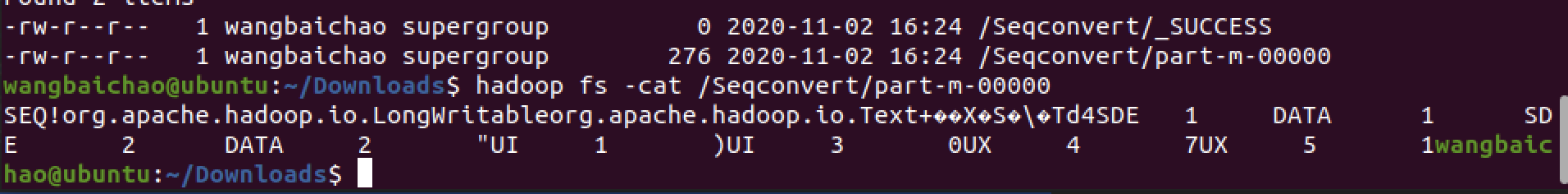
1. **import** java.io.IOException;
2. **import** org.apache.hadoop.io.LongWritable;
3. **import** org.apache.hadoop.io.Text;
4. **import** org.apache.hadoop.mapreduce.Mapper;

7. **public** **class** FormatConverterMapper **extends**
8. Mapper<LongWritable, Text, LongWritable, Text> {
10. @Override
11. **public** **void** map(LongWritable key, Text value, Context context)
12. **throws** IOException, InterruptedException {
13. context.write(key, value);
14. }
15. }

Driver:

1. **import** org.apache.hadoop.fs.Path;
2. **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
3. **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
4. **import** org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
5. **import** org.apache.hadoop.mapreduce.Job;
6. **import** org.apache.hadoop.conf.Configured;
7. **import** org.apache.hadoop.conf.Configuration;
8. **import** org.apache.hadoop.util.Tool;
9. **import** org.apache.hadoop.util.ToolRunner;
11. **public** **class** FormatConverterTextToSequenceDriver **extends** Configured **implements** Tool {
13. @Override
14. **public** **int** run(String[] args) **throws** Exception {
16. **if** (args.length != 2) {
17. System.out.printf("Two parameters are required for FormatConverterTextToSequenceDriver-<input dir> <output dir>\n");
18. **return** -1;
19. }
21. Job job = **new** Job(getConf());
22. job.setJarByClass(FormatConverterTextToSequenceDriver.**class**);
23. job.setJobName("Create Sequence File, from text file");
25. FileInputFormat.setInputPaths(job, **new** Path(args[0]));
26. FileOutputFormat.setOutputPath(job, **new** Path(args[1]));
28. job.setMapperClass(FormatConverterMapper.**class**);
29. job.setOutputFormatClass(SequenceFileOutputFormat.**class**);
30. job.setNumReduceTasks(0);
32. **boolean** success = job.waitForCompletion(**true**);
33. **return** success ? 0 : 1;
34. }
36. **public** **static** **void** main(String[] args) **throws** Exception {
37. **int** exitCode = ToolRunner.run(**new** Configuration(), **new** FormatConverterTextToSequenceDriver(), args);
38. System.exit(exitCode);
39. }
40. }

After convert:



Then write a mapreduce to count on this sequence file

Mapper:

1. **import** org.apache.hadoop.io.IntWritable;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Mapper;
4. **import** java.io.IOException;
6. **public** **class** SMapper **extends** Mapper<Object, Text, Text, IntWritable> {
8. **public** **void** map (Object key,Text value, Context context)
9. **throws** IOException, InterruptedException{
10. String[] inputSplit = value.toString().split("\t");
11. String pos = inputSplit[0];
12. IntWritable one = **new** IntWritable(1);
14. context.write(**new** Text(pos),one);
15. }
16. }

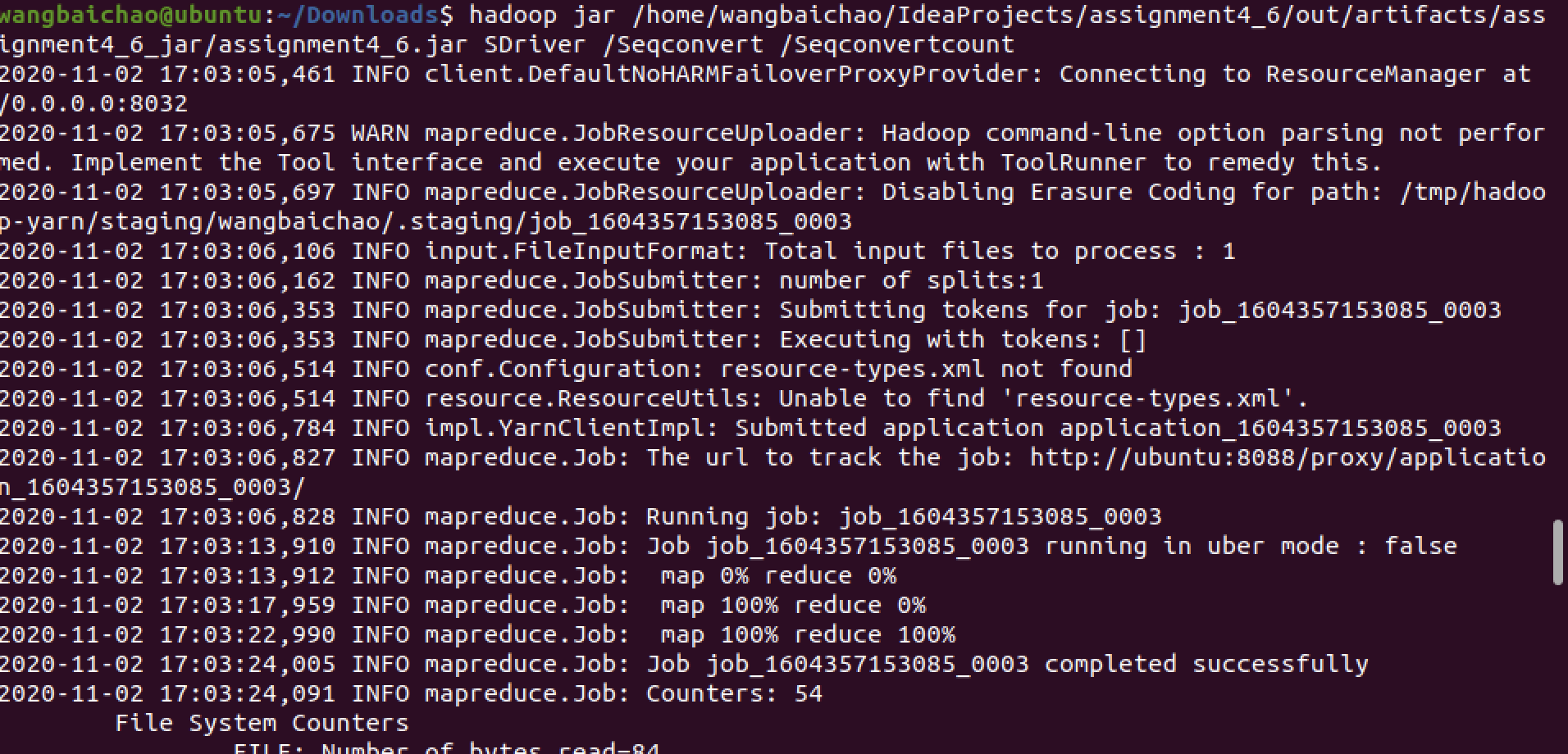
Reducer:

1. **import** org.apache.hadoop.io.IntWritable;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Reducer;
5. **import** java.io.IOException;
6. **public** **class** SReducer **extends** Reducer<Text, IntWritable, Text, IntWritable>{
8. **protected** **void** reduce(Text key, Iterable<IntWritable> values, Context context)
9. **throws** IOException,
10. InterruptedException {
11. **int** count = 0;
12. **for**(IntWritable val : values)
13. {
14. count+=val.get();
15. }
16. context.write(key, **new** IntWritable(count));
17. }
18. }

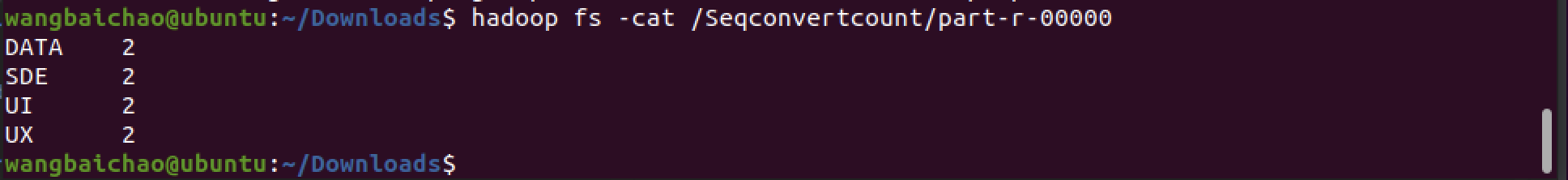
Driver:

1. **import** org.apache.hadoop.io.IntWritable;
2. **import** org.apache.hadoop.io.Text;
3. **import** org.apache.hadoop.mapreduce.Reducer;
5. **import** java.io.IOException;
6. **public** **class** SReducer **extends** Reducer<Text, IntWritable, Text, IntWritable>{
8. **protected** **void** reduce(Text key, Iterable<IntWritable> values, Context context)
9. **throws** IOException,
10. InterruptedException {
11. **int** count = 0;
12. **for**(IntWritable val : values)
13. {
14. count+=val.get();
15. }
16. context.write(key, **new** IntWritable(count));
17. }
18. }

Run job:



Result:



6) TextInputFormat

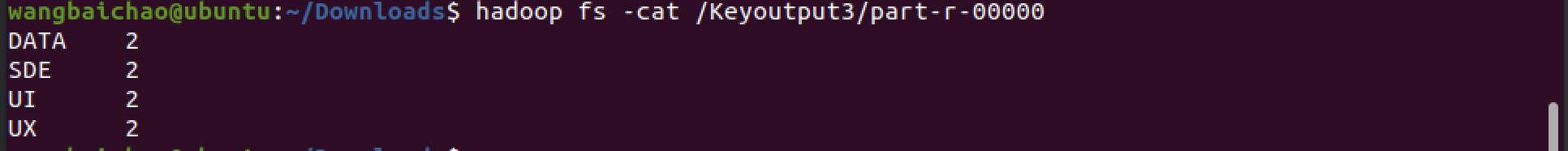
This is the Hadoop's default input format.

Use this command to set input :

1. job.setInputFormatClass(TextInputFormat.**class**);

The input, mapper and reducer is same as Par4 3)

Output：



**PART 5 – Programming Assignment**

Create a Writable object that stores some fields from the the NYSE dataset to find  
- the date of the max stock\_volume  
- the date of the min stock\_volume  
- the max stock\_price\_adj\_close  
This will be a custom writable class with the above fields.  
Mapper will use this writable object as a value, and Reducer will use this writable object as a value.

1. Create writable:
2. **import** java.io.DataInput;
3. **import** java.io.DataOutput;
4. **import** java.io.IOException;
5. **import** java.util.Date;
6. **import** java.text.SimpleDateFormat;
8. **import** org.apache.commons.math3.analysis.function.Min;
9. **import** org.apache.hadoop.io.Writable;
11. **public** **class** MinMaxWritable **implements** Writable {
12. **private** String symbol;
13. **private** Date maxVolumeDate;
14. **private** Date minVolumeDate;
15. **private** Double maxClose;
16. **private** **int** volume;
17. **public** **final** **static** SimpleDateFormat dateFormatter = **new** SimpleDateFormat("yyyy-MM-dd");
19. MinMaxWritable(){
20. **this**.maxVolumeDate=**null**;
21. **this**.minVolumeDate=**null**;
22. **this**.maxClose=0.0;
23. **this**.volume=-1;
24. }
26. MinMaxWritable(MinMaxWritable stock){
27. **this**.maxVolumeDate=**new** Date(stock.getMaxVolumeDate().getTime());
28. **this**.minVolumeDate=**new** Date(stock.getMinVolumeDate().getTime());
29. **this**.maxClose=stock.getMaxClose();
30. **this**.volume=stock.getVolume();
31. }
33. MinMaxWritable(String inputLine) {
35. String[] fields = inputLine.split(",");
36. **if** (fields == **null** || fields.length != 9) {
37. System.err.println("Bad data. The expected format is 9 columns seperated by comma.");
38. **return**; // skip the bad data
39. }
41. **try** {
42. **this**.symbol = fields[1];
43. **this**.volume=Integer.parseInt(fields[7]);
44. **this**.maxVolumeDate = dateFormatter.parse(fields[2]);
45. **this**.minVolumeDate = dateFormatter.parse(fields[2]);
46. **this**.maxClose = Double.parseDouble(fields[8]);
47. } **catch** (Exception e) {
48. System.err.println("Bad data. Cannot parse the symbol, date, volume and price in column 2, 3, 8 and 9.");
49. e.printStackTrace();
50. **return**;
51. }
52. }
54. **public** **int** getVolume() {
55. **return** volume;
56. }
58. **public** **void** setVolume(**int** volume) {
59. **this**.volume = volume;
60. }
62. **public** String getSymbol() {
63. **return** symbol;
64. }
66. **public** **void** setSymbol(String symbol) {
67. **this**.symbol = symbol;
68. }
70. **public** Date getMaxVolumeDate() {
71. **return** maxVolumeDate;
72. }
74. **public** **void** setMaxVolumeDate(Date maxVolumeDate) {
75. **this**.maxVolumeDate = **new** Date(maxVolumeDate.getTime());
76. }
78. **public** Date getMinVolumeDate() {
79. **return** minVolumeDate;
80. }
82. **public** **void** setMinVolumeDate(Date minVolumeDate) {
83. **this**.minVolumeDate = **new** Date(minVolumeDate.getTime());
84. }
86. **public** Double getMaxClose() {
87. **return** maxClose;
88. }
90. **public** **void** setMaxClose(Double maxClose) {
91. **this**.maxClose = maxClose;
92. }
94. @Override
95. **public** **void** write(DataOutput out) **throws** IOException{
96. out.writeLong(**this**.maxVolumeDate.getTime());
97. out.writeLong(**this**.minVolumeDate.getTime());
98. out.writeDouble(**this**.maxClose);
99. out.writeInt(**this**.volume);

102. }
103. @Override
104. **public** **void** readFields(DataInput in) **throws** IOException {
105. **this**.maxVolumeDate = **new** Date(in.readLong());
106. **this**.minVolumeDate = **new** Date(in.readLong());
107. **this**.maxClose = in.readDouble();
108. **this**.volume=in.readInt();
109. }
111. @Override
112. **public** String toString() {
113. **return** dateFormatter.format(maxVolumeDate) + "\t" + dateFormatter.format(minVolumeDate) + "\t" + maxClose;
114. }
115. }

2) Mapper:

1. **import** java.io.IOException;
3. **import** org.apache.hadoop.io.Text;
4. **import** org.apache.hadoop.mapreduce.Mapper;
5. **import** java.text.SimpleDateFormat;
6. **public** **class** MinMaxMapper **extends** Mapper<Object, Text, Text, MinMaxWritable>{
7. //MinMaxWritable stock = null;
8. **public** **final** **static** SimpleDateFormat dateFormatter = **new** SimpleDateFormat("yyyy-MM-dd");

11. @Override
12. **public** **void** map(Object key, Text value, Context context) **throws** IOException, InterruptedException {
13. MinMaxWritable stock = **new** MinMaxWritable(value.toString());
15. **if** (stock != **null**) {
16. context.write(**new** Text(stock.getSymbol()), stock);
17. } **else** {
18. System.out.println("Map: One MinMaxWritable is null");
19. }
20. }
21. }

3) Reducer:

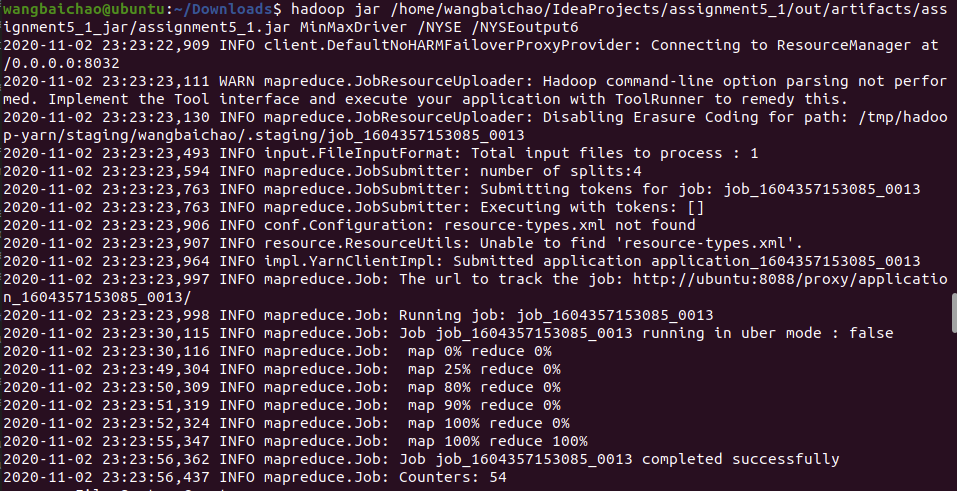
1. **import** java.io.IOException;
2. **import** java.util.Date;
3. **import** org.apache.hadoop.io.Text;
4. **import** org.apache.hadoop.mapreduce.Reducer;
6. **public** **class** MinMaxReducer **extends** Reducer<Text, MinMaxWritable, Text, MinMaxWritable>{
7. @Override
8. **public** **void** reduce(Text key, Iterable<MinMaxWritable> values, Context context)
9. **throws** IOException, InterruptedException {
11. //            move outside of reduce() to get the global max/min
12. //MinMaxWritable max = new MinMaxWritable();
13. //MinMaxWritable min = new MinMaxWritable();
14. **int** maxv = -1;
15. **int** minv= -1;
16. Double maxclose=0.0;
17. Date maxDate=**new** Date();
18. Date minDate=**new** Date();
19. MinMaxWritable output = **new** MinMaxWritable();
21. **for**(MinMaxWritable stock: values) {
23. **int** volume = stock.getVolume();
24. Double close = stock.getMaxClose();
26. **if** (maxv == -1 || maxv < volume) {
27. maxv=volume;
28. //output.setMaxVolumeDate(stock.getMaxVolumeDate());
29. maxDate=**new** Date(stock.getMaxVolumeDate().getTime());
31. }
32. **if** (minv == -1 || minv > volume) {
33. minv=volume;
34. //output.setMinVolumeDate(stock.getMinVolumeDate());
35. minDate=**new** Date(stock.getMinVolumeDate().getTime());
36. }
37. **if** (maxclose == 0.0 || maxclose < close) {
38. maxclose=close;

41. }
43. }
44. output.setMaxVolumeDate(maxDate);
45. output.setMinVolumeDate(minDate);
46. output.setMaxClose(maxclose);
48. context.write(key, output);
49. }
51. }

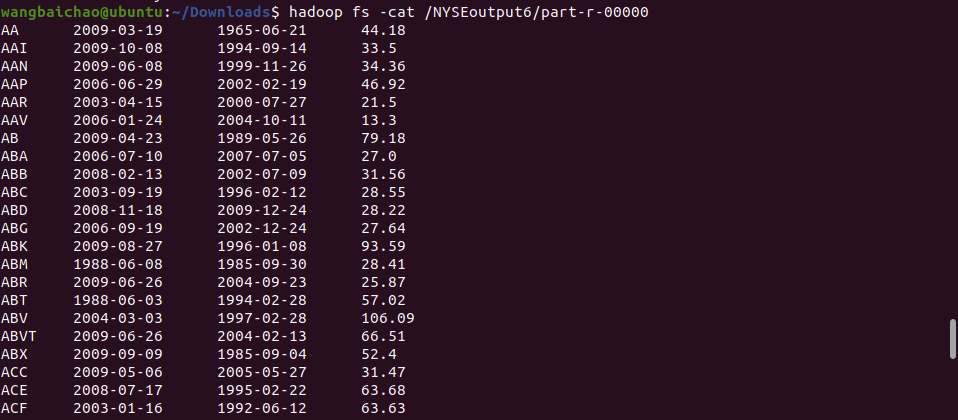
4) Driver:

1. **import** java.io.IOException;
3. **import** org.apache.hadoop.conf.Configuration;
4. **import** org.apache.hadoop.conf.Configured;
5. **import** org.apache.hadoop.fs.FileSystem;
6. **import** org.apache.hadoop.fs.Path;
7. **import** org.apache.hadoop.io.IntWritable;
8. **import** org.apache.hadoop.io.Text;
9. **import** org.apache.hadoop.mapreduce.Job;
10. **import** org.apache.hadoop.mapreduce.Mapper;
11. **import** org.apache.hadoop.mapreduce.Reducer;
12. **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13. **import** org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
14. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
15. **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
16. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
17. **import** org.apache.hadoop.util.Tool;
18. **import** org.apache.hadoop.util.ToolRunner;
19. **public** **class** MinMaxDriver {
20. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
21. Configuration conf = **new** Configuration();
22. FileSystem fs =FileSystem.get(conf);
23. **if**(fs.exists(**new** Path(args[1])))
24. {
25. fs.delete(**new** Path(args[1]), **true**);
26. }
28. Job job = Job.getInstance(conf);
29. job.setInputFormatClass(TextInputFormat.**class**);
31. job.setMapperClass(MinMaxMapper.**class**);
32. job.setReducerClass(MinMaxReducer.**class**);
33. job.setJarByClass(MinMaxDriver.**class**);
35. TextInputFormat.addInputPath(job, **new** Path(args[0]));
36. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
38. job.setMapOutputKeyClass(Text.**class**);
39. job.setMapOutputValueClass(MinMaxWritable.**class**);
41. job.setOutputKeyClass(Text.**class**);
42. job.setOutputValueClass(MinMaxWritable.**class**);
44. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
45. }
46. }

Run job:



Result:



**PART 6 – Programming Assignment**

Redo Part5 of this assignment, but cram multiple values (max stock\_volume,  min stock\_volume, max stock\_price\_adj\_close) into a Text object with some delimiter. Use a Combiner. Compare the running time of Part 2 to Part 3. You could measure the running time programmatically, or use your smartphone’s timer.

Writable:

1. **import** java.io.DataInput;
2. **import** java.io.DataOutput;
3. **import** java.io.IOException;
4. **import** java.util.Date;
5. **import** java.text.SimpleDateFormat;
7. **import** org.apache.commons.math3.analysis.function.Min;
8. **import** org.apache.hadoop.io.Writable;
10. **public** **class** MinMaxWritable **implements** Writable {
11. **private** String symbol;
12. **private** **int** maxVolume;
13. **private** **int** minVolume;
14. **private** Double maxClose;
16. MinMaxWritable(){
17. **this**.maxVolume=-1;
18. **this**.minVolume=-1;
19. **this**.maxClose=0.0;
20. }
22. MinMaxWritable(MinMaxWritable stock){
23. **this**.maxVolume=stock.getMaxVolume();
24. **this**.minVolume= stock.getMinVolume();
25. **this**.maxClose=stock.getMaxClose();
26. }
28. MinMaxWritable(String inputLine) {
30. String[] fields = inputLine.split(",");
31. **if** (fields == **null** || fields.length != 9) {
32. System.err.println("Bad data. The expected format is 9 columns seperated by comma.");
33. **return**; // skip the bad data
34. }
36. **try** {
37. **this**.symbol = fields[1];
38. **this**.maxVolume=Integer.parseInt(fields[7]);
39. **this**.minVolume=Integer.parseInt(fields[7]);
40. **this**.maxClose = Double.parseDouble(fields[8]);
41. } **catch** (Exception e) {
42. System.err.println("Bad data. Cannot parse the symbol, date, volume and price in column 2, 3, 8 and 9.");
43. e.printStackTrace();
44. **return**;
45. }
46. }

49. **public** String getSymbol() {
50. **return** symbol;
51. }
53. **public** **void** setSymbol(String symbol) {
54. **this**.symbol = symbol;
55. }
57. **public** **int** getMaxVolume() {
58. **return** maxVolume;
59. }
61. **public** **void** setMaxVolume(**int** maxVolume) {
62. **this**.maxVolume = maxVolume;
63. }
65. **public** **int** getMinVolume() {
66. **return** minVolume;
67. }
69. **public** **void** setMinVolume(**int** minVolume) {
70. **this**.minVolume = minVolume;
71. }
73. **public** Double getMaxClose() {
74. **return** maxClose;
75. }
77. **public** **void** setMaxClose(Double maxClose) {
78. **this**.maxClose = maxClose;
79. }
81. @Override
82. **public** **void** write(DataOutput out) **throws** IOException{
83. out.writeInt(**this**.maxVolume);
84. out.writeInt(**this**.minVolume);
85. out.writeDouble(**this**.maxClose);
86. }
87. @Override
88. **public** **void** readFields(DataInput in) **throws** IOException {
89. **this**.maxVolume = in.readInt();
90. **this**.minVolume = in.readInt();
91. **this**.maxClose = in.readDouble();
92. }
94. @Override
95. **public** String toString() {
96. **return** maxVolume+","+minVolume+","+maxClose;
97. }
98. }

Mapper:

1. **import** java.io.IOException;
3. **import** org.apache.hadoop.io.Text;
4. **import** org.apache.hadoop.mapreduce.Mapper;
5. **public** **class** MinMaxMapper **extends** Mapper<Object, Text, Text, MinMaxWritable>{
6. //MinMaxWritable stock = null;
8. @Override
9. **public** **void** map(Object key, Text value, Context context) **throws** IOException, InterruptedException {
10. MinMaxWritable stock = **new** MinMaxWritable(value.toString());
12. **if** (stock != **null**) {
13. context.write(**new** Text(stock.getSymbol()), stock);
14. } **else** {
15. System.out.println("Map: One MinMaxWritable is null");
16. }
17. }
18. }

Reducer:

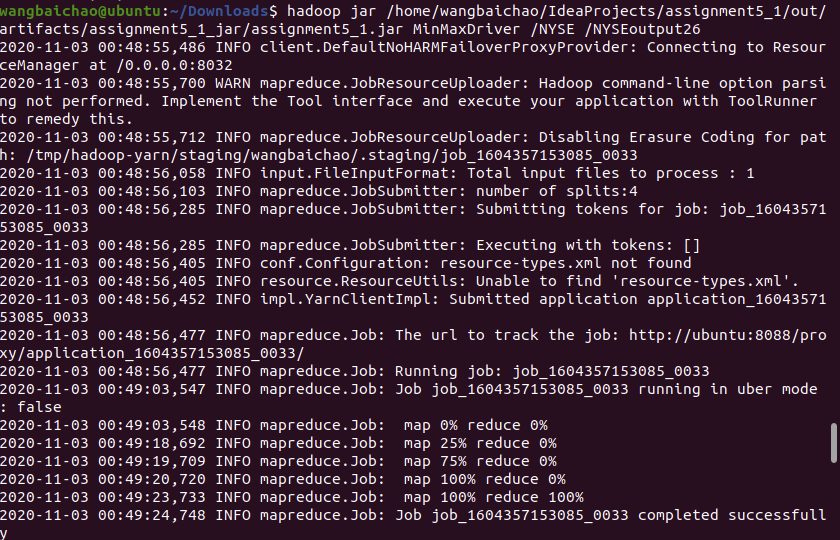
1. **import** java.io.IOException;
2. **import** java.util.Date;
3. **import** org.apache.hadoop.io.Text;
4. **import** org.apache.hadoop.mapreduce.Reducer;
6. **public** **class** MinMaxReducer **extends** Reducer<Text, MinMaxWritable, Text, MinMaxWritable>{
7. @Override
8. **public** **void** reduce(Text key, Iterable<MinMaxWritable> values, Context context)
9. **throws** IOException, InterruptedException {
11. //            move outside of reduce() to get the global max/min
12. //MinMaxWritable max = new MinMaxWritable();
13. //MinMaxWritable min = new MinMaxWritable();
14. **int** maxv = -1;
15. **int** minv= -1;
16. Double maxclose=0.0;
17. MinMaxWritable output = **new** MinMaxWritable();
19. **for**(MinMaxWritable stock: values) {
21. **int** volume1 = stock.getMaxVolume();
22. **int** volume2 = stock.getMinVolume();
23. Double close = stock.getMaxClose();
25. **if** (maxv == -1 || maxv < volume1) {
26. maxv=volume1;
28. }
29. **if** (minv == -1 || minv > volume2) {
30. minv=volume2;
31. }
32. **if** (maxclose == 0.0 || maxclose < close) {
33. maxclose=close;

36. }
38. }
40. output.setMaxVolume(maxv);
41. output.setMinVolume(minv);
42. output.setMaxClose(maxclose);
44. context.write(key, output);
45. }
47. }

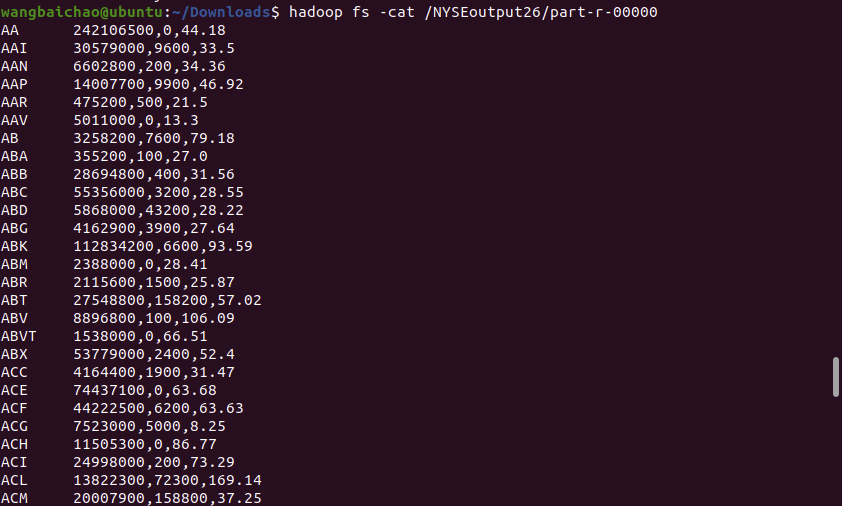
Driver:

1. **import** java.io.IOException;
3. **import** org.apache.hadoop.conf.Configuration;
4. **import** org.apache.hadoop.fs.FileSystem;
5. **import** org.apache.hadoop.fs.Path;
6. **import** org.apache.hadoop.io.Text;
7. **import** org.apache.hadoop.mapreduce.Job;
8. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
9. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
11. **public** **class** MinMaxDriver {
12. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
13. Configuration conf = **new** Configuration();
14. FileSystem fs =FileSystem.get(conf);
15. **if**(fs.exists(**new** Path(args[1])))
16. {
17. fs.delete(**new** Path(args[1]), **true**);
18. }
20. Job job = Job.getInstance(conf);
21. job.setInputFormatClass(TextInputFormat.**class**);
23. job.setMapperClass(MinMaxMapper.**class**);
25. job.setCombinerClass(MinMaxReducer.**class**);
27. job.setReducerClass(MinMaxReducer.**class**);
28. job.setJarByClass(MinMaxDriver.**class**);
30. TextInputFormat.addInputPath(job, **new** Path(args[0]));
31. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
33. job.setMapOutputKeyClass(Text.**class**);
34. job.setMapOutputValueClass(MinMaxWritable.**class**);
36. job.setOutputKeyClass(Text.**class**);
37. job.setOutputValueClass(MinMaxWritable.**class**);
39. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
40. }
41. }

Run job :



Result:



Runtime:

Start: 2020-11-03 00:48:55,486

End: 2020-11-03 00:49:24,837

It takes 29.351 seconds

It is faster than Part3

**PART 7 – Programming Assignment**

Determine the average stock\_price\_adj\_close value by the year.  
Use a Writable object to pass count and local average in which a Reducer could be used as a Combiner.

Writable:

1. **import** java.io.DataInput;
2. **import** java.io.DataOutput;
3. **import** java.io.IOException;
5. **import** org.apache.hadoop.io.Writable;
7. **public** **class** AverageTuple **implements** Writable {
8. **private** **int** count = 0;
9. **private** **float** average;
11. **public** **void** readFields(DataInput in) **throws** IOException {
12. count = in.readInt();
13. average = in.readFloat();
14. }
16. **public** **void** write(DataOutput out) **throws** IOException {
17. out.writeInt(count);
18. out.writeFloat(average);
19. }
21. **public** **int** getCount() {
22. **return** count;
23. }
25. **public** **void** setCount(**int** count) {
26. **this**.count = count;
27. }
29. **public** **float** getAverage() {
30. **return** average;
31. }
33. **public** **void** setAverage(**float** average) {
34. **this**.average = average;
35. }
37. @Override
38. **public** String toString() {
39. **return** count + "\t" + average;
40. }


44. }

Mapper:

1. **import** java.io.IOException;
2. **import** org.apache.hadoop.io.IntWritable;
3. **import** java.text.SimpleDateFormat;
4. **import** org.apache.hadoop.io.Text;
5. **import** org.apache.hadoop.mapreduce.Mapper;
7. **public** **class** AverageMapper **extends** Mapper<Object, Text, Text, AverageTuple>{
9. @Override
10. **public** **void** map(Object key, Text value, Context context) **throws** IOException, InterruptedException {
11. String[] fields = value.toString().split(",");
12. Text year = **new** Text();
13. AverageTuple tuple= **new** AverageTuple();
14. **if** (fields == **null** || fields.length != 9) {
15. System.err.println("Bad data. The expected format is 9 columns seperated by comma.");
16. **return**; // skip the bad data
17. }
19. **try** {
20. year.set(fields[2].substring(0,4));
21. **float** price = Float.parseFloat(fields[8]);
22. tuple.setAverage(price);
23. tuple.setCount(1);
24. } **catch** (Exception e) {
25. System.err.println("Bad data. Cannot parse the symbol, date, volume and price in column 2, 3, 8 and 9.");
26. e.printStackTrace();
27. **return**;
28. }
30. context.write(year, tuple);
31. }
32. }

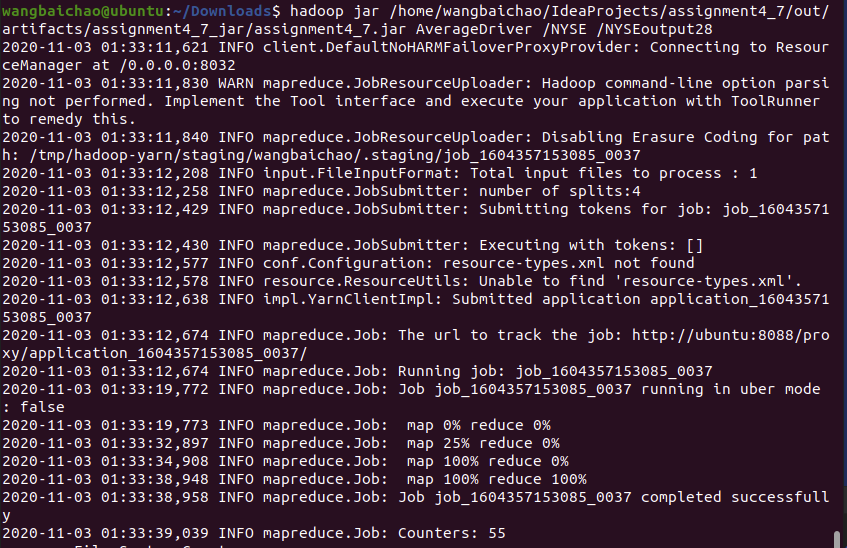
Reducer:

1. **import** java.io.IOException;
2. **import** java.util.Date;
3. **import** org.apache.hadoop.io.Text;
4. **import** org.apache.hadoop.mapreduce.Reducer;
5. **import** org.apache.hadoop.io.IntWritable;
6. **public** **class** AverageReducer **extends** Reducer<Text, AverageTuple, Text, AverageTuple> {
7. **private** AverageTuple tuple = **new** AverageTuple();
9. @Override
10. **public** **void** reduce(Text key, Iterable<AverageTuple> values, Context context)
11. **throws** IOException, InterruptedException {
12. **int** count = 0;
13. **long** sum = 0;
15. **for** (AverageTuple v: values) {
16. count += v.getCount(); // total count
17. sum += v.getCount() \* v.getAverage(); // running sum
18. }
19. tuple.setCount(count);
20. **float** averagePrice = (**float**)1.0\*sum/count;
21. tuple.setAverage(averagePrice);
22. context.write(key, tuple);
23. }
24. }

Driver:

1. **import** java.io.IOException;
3. **import** org.apache.hadoop.conf.Configuration;
4. **import** org.apache.hadoop.fs.FileSystem;
5. **import** org.apache.hadoop.fs.Path;
6. **import** org.apache.hadoop.io.Text;
7. **import** org.apache.hadoop.mapreduce.Job;
8. **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
9. **import** org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
10. **public** **class** AverageDriver {
11. **public** **static** **void** main (String[] args) **throws** IOException, ClassNotFoundException, InterruptedException  {
12. Configuration conf = **new** Configuration();
13. FileSystem fs =FileSystem.get(conf);
14. **if**(fs.exists(**new** Path(args[1])))
15. {
16. fs.delete(**new** Path(args[1]), **true**);
17. }
19. Job job = Job.getInstance(conf);
20. job.setInputFormatClass(TextInputFormat.**class**);
22. job.setMapperClass(AverageMapper.**class**);
24. job.setCombinerClass(AverageReducer.**class**);
26. job.setReducerClass(AverageReducer.**class**);
27. job.setJarByClass(AverageDriver.**class**);
29. TextInputFormat.addInputPath(job, **new** Path(args[0]));
30. TextOutputFormat.setOutputPath(job, **new** Path(args[1]));
32. job.setMapOutputKeyClass(Text.**class**);
33. job.setMapOutputValueClass(AverageTuple.**class**);
35. job.setOutputKeyClass(Text.**class**);
36. job.setOutputValueClass(AverageTuple.**class**);
38. System.exit(job.waitForCompletion(**true**) ? 0 : 1);
39. }
40. }

Run job:



Result:

