Final Project

Mapreduce program for cluster logs sorting

Yu Yang, Baylor University Student ID 892449550

In this final report, I will discuss some advanced topics of mapreduce which lay the foundation of the project app for big cluster logs sorting. I present a naïve and the unit test driven development, then extends it to a more practical mapreduce solution with the usage of multiple mapper-reducers and dependencies. I also use the distributed cache mechanism. In the last part, I will give some future directions and some features not covered in the program.

Due to the limit of virtual machine, I only use small data set to test the app. Hadoop has a disk block size of 128MB. To get the true power of Hadoop against big data processing, every single input data should be about the size of a block and then multiple map-reducers can run in a parallel way. But the use of real size data will cause several hours of a single run. As a compromise, I use small data set to illustrate the "parallel mapreduce" way to solve this particular problem.

Contents

MAPREDUCE IN DEPTH

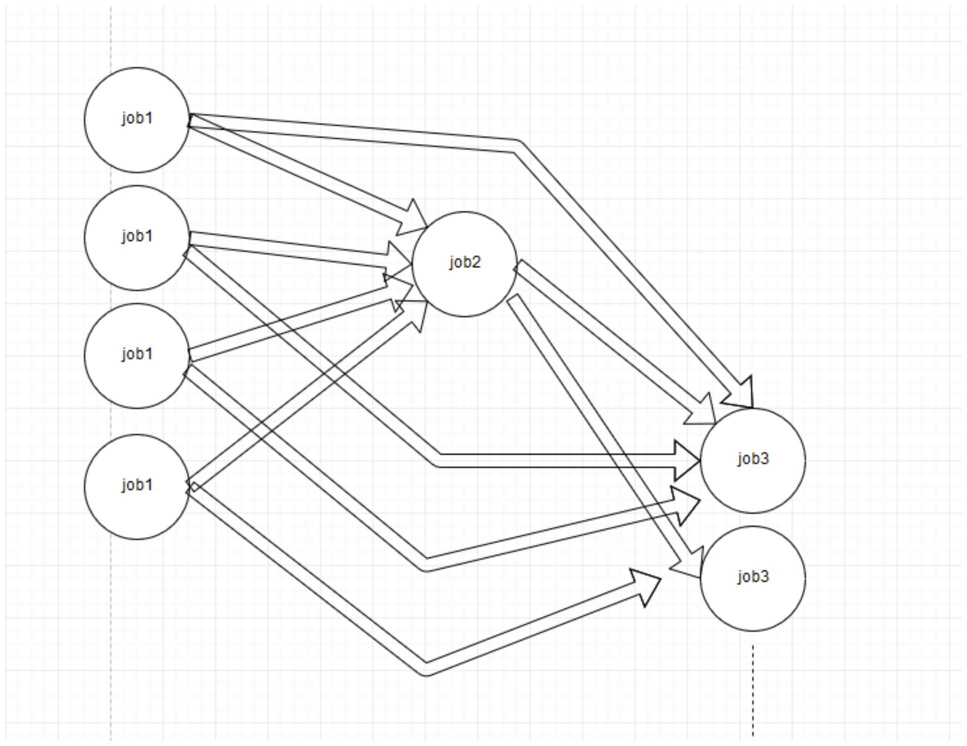
The simple run of HW1 and word count program in HW2 are faily simple. Finding or/and counting for one/every string key. They only use one mapreduce job to process the data. But when the problem becomes more complicated and data becomes bigger, one mapper-reducer is certainly not a good fit because mapreduce is best for distributed system(HDFS for hadoop). Generally, more complicated problems are solved by multiple simpler mapper-reducer rather than only one complex mapper-reducer. This brings two questions: how to control the work flow of multiple mapper-reducers, and how to decide the proper size of a single mapper-reducer(in other words, how many mapper-reducer are best for a particular problem).

Multiple mapper-reducers workflow control

For multiple mapper-reducers to work in a certain order, JobControl class is a powerful tool. I can add several mapreduce works to JobControl, and then define their dependencies. Take a code snippet in the final project program for example.

```
 Job job = Job.getInstance(conf);
 job.setNumReduceTasks(4);
 //job set up code omitted here
 Job job2 = Job.getInstance(conf);
//job set up code omitted here
 //Sort with meta info
 Job job3 = Job.getInstance(conf);
 //no reducer is used in this job
 //job set up code omitted here

 //using controlledjob to build dependencies between jobs
 ControlledJob cjob1 = new ControlledJob(job.getConfiguration());
 ControlledJob cjob2 = new ControlledJob(job2.getConfiguration());
 ControlledJob cjob3 = new ControlledJob(job3.getConfiguration());
 JobControl jobController = new JobControl("jc");
 cjob1.setJob(job);
 cjob2.setJob(job2);
 cjob3.setJob(job3);
 jobController.addJob(cjob1);
 jobController.addJob(cjob2);
 jobController.addJob(cjob3);
 cjob2.addDependingJob(cjob1);
cjob3.addDependingJob(cjob2);
```

The code defines 3 jobs as job1, job2 and job3. They work like a job-chain, as the figure shows:



There are 4 mr (mapreuce) job1 instances working as the level, mr job2 waits for mr job1s' result as input. And multiple job3 instances are dependent on both job2 and job1. In a more complicated problem, JobControl can be used to define a workflow in a DAG (directed acyclic graph) way. Every job instance is like a vertex in the DAG, and dependencies are directed edges.

How to decide the number of mapper-reducer needed for soving a problem

One mapper-reducer setting is only for some "toy" problems. For real problems in real world, the mapper-reducer number is usually large, because speed will be very slow if all the inter-process (shuffle, partition, sort, which I will cover later) happen in a cluster node.

Quoting from OReilly's Hadoop guide, "choosing the number is more of an art than a science". Increasing the number makes every mapper-reducer shorter, so that more parallel processing is achieved at the cost of overhead of smaller intermediate files. For a specific problem, one should always calibrate the number so that most intermediate files fit into one HDFS block size.

SHUFFLE PROCESS

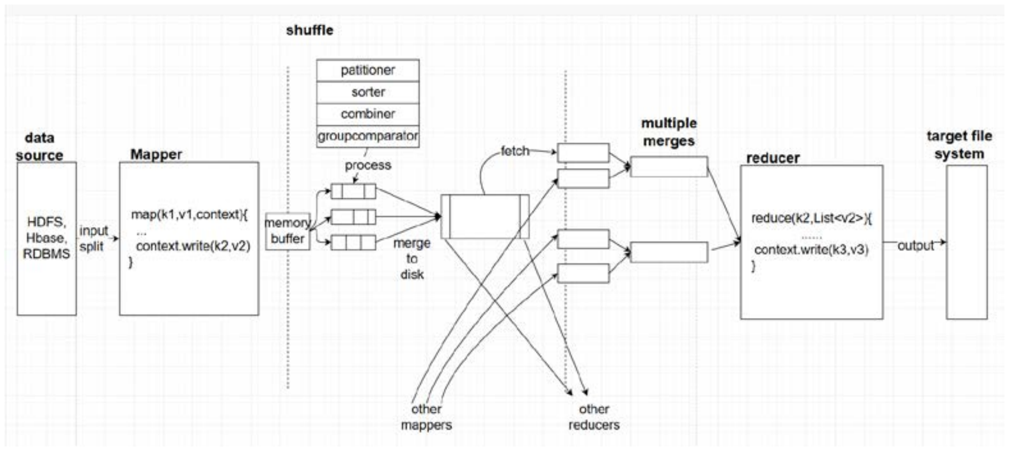
In the top level, mapreduce has a general form as:

Mapper: (K1,V1) -> (K2, V2)

Reducer: (K2, List<V2>) -> List (K3, V3) (results)

The signature and map() reduce() functions are already discussed in report2. In addition to those self-explaining Math mappings, there are more things happen in a mapreduce job.

A more detailed mapreduce process is shown as follows:



The input of every reducer is sorted and grouped by k2. There is a process called shuffle happening between mapper and reducer. In fact shuffle is the heart of mapreduce, many features like custom petitioner and secondary sort happen in the shuffle process.

Mapper Side

Every mapper writes its result into a memory buffer. Before buffer is written to disk, shuffler partitions the buffer data based on the number of reducer. Note that in this partition process, a memory sort is performed based on the raw comparator or custom practitioner comparator defined by the user. Combiner, if provided, also happens here to reduce the size of data pass to the reducers.

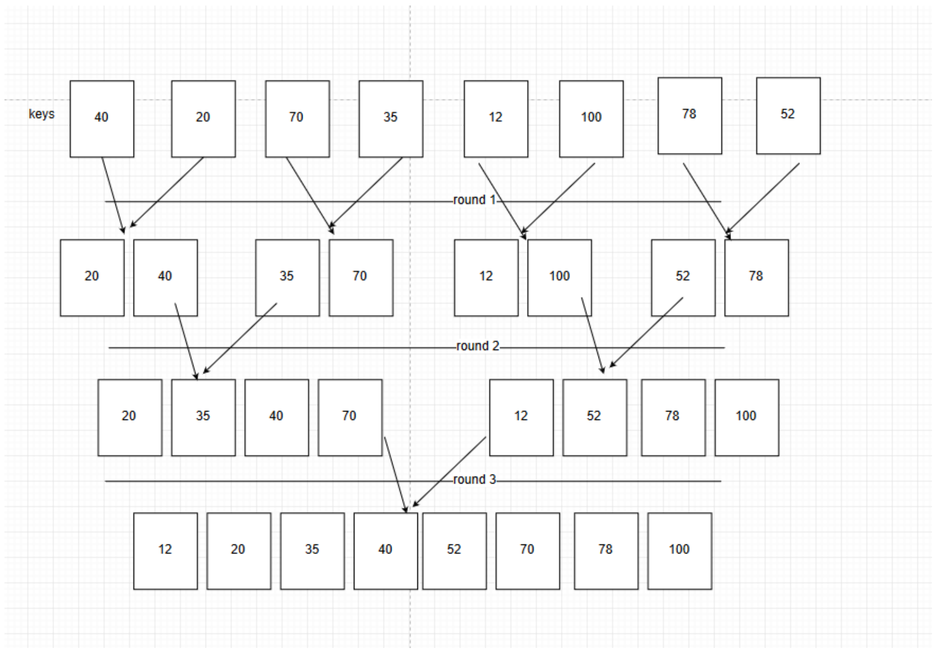
Because disk write and network communication (with reducer) are heavy tasks, it is common to optimize the serialization and writable class, so that the amount of heavy tasks is kept as small as possible. This is the in-depth reason of custom writable class and custom comparator I discussed in report 2.

Note that there can also be multiple mappers, so for one reducer, it needs to fetch data from different mappers and merge them based on keys.

Reducer Side

The reducer needs to wait different mappers' partition result on disks across the cluster. It copies the result once a mapper's output is ready, which means all the fetches are processed in parallel (which is very important for multi mapper-reducer structure used in this project program).

As those copies arrive reducer's disk, multiple merges happen and those copies form a larger sorted file. The multiple merge process in the reducer is like a bottom-up mergesort (although mergesort is a sort algorithm, but it is exactly the purpose of multiple merge here), as shown below.



Smaller files keep merge into bigger files in a recursive way.

Also note that for tuning the configuration of a mr job, a rule of thumb is to give shuffler as much memory as possible. Because both map() function in mapper and reduce() function in reducer is O(n), because they iterate through the data once in memory. Shuffle involves a lot of “mergesort” on **disks** (O(nlogn)) and network transmission, which means it needs more memory buffer to speed up.

MRUNIT TEST

Test driven development is an effective methodology and I use it through the project. In this section, I will discuss the unit test of this project along with the problem analysis.

The problem is quite straightforward. Suppose I have a computer cluster. Each node generates a log with timestamp for every event happens there, such as:

Node 1 log

Time 1: user 1 logs in

```
        Time 7: a mr job starts
        Time 18: phase 1 mapper 2 finishes
        Time 48: phase 5 mapper 3 finishes
        Time 70: user 1 log out
….
Node 2 log:
        Time 1: user 2 logs out
        Time 4: phase 1 mapper 1 starts
         Time 18: phase 7 mapper 2 starts
         Time 70: phase 1 reducer starts
…
More node logs…
```

Suppose there are thousands or even more nodes on this cluster and each has a very large log file. An administrator needs to get a whole report of each event happens in the cluster, sorted by time. For the log example mentioned above, the admin user wants a report like this:

```
Group 1 Time 1:
        user 1 log in
        user 2 logs out
Group 2 time 4:
        phase 1 mapper 1 starts
Group 3 time 7:
        a mr job starts
```

Group 4 Time 18:

phase 7 mapper 2 starts

phase 1 mapper 2 finishes

Group 5 Time 48:

phase 5 mapper 3 finishes

Group 6 Time 70:

user 1 log out

phase 1 reducer starts

…….

At the first glance, it is a very simple sorting problem. Reading all the logs in and using any in-memory sort against the whole data can solve the problem. But the concern is that the log file is too large that it is impossible to fit the data into memory.

In fact, a very simple naïve mapreduce app can solve this problem (I will cover the naïve solution later). To implement a test-driven method, I will start from unit test.

In fact, it is a very straightforward idea that for this problem, only the time stamp and the index matter. Because the event description associated with the time key will not be modified for the whole mr job process. So I can make a simplification here.

Suppose there are multiple large logs file, each one with the format as:

Node 1 log

1,

7,

18,

48,

```
70
….
Node 2 log:
1,
4,
18,
70,
…
More node logs…
```

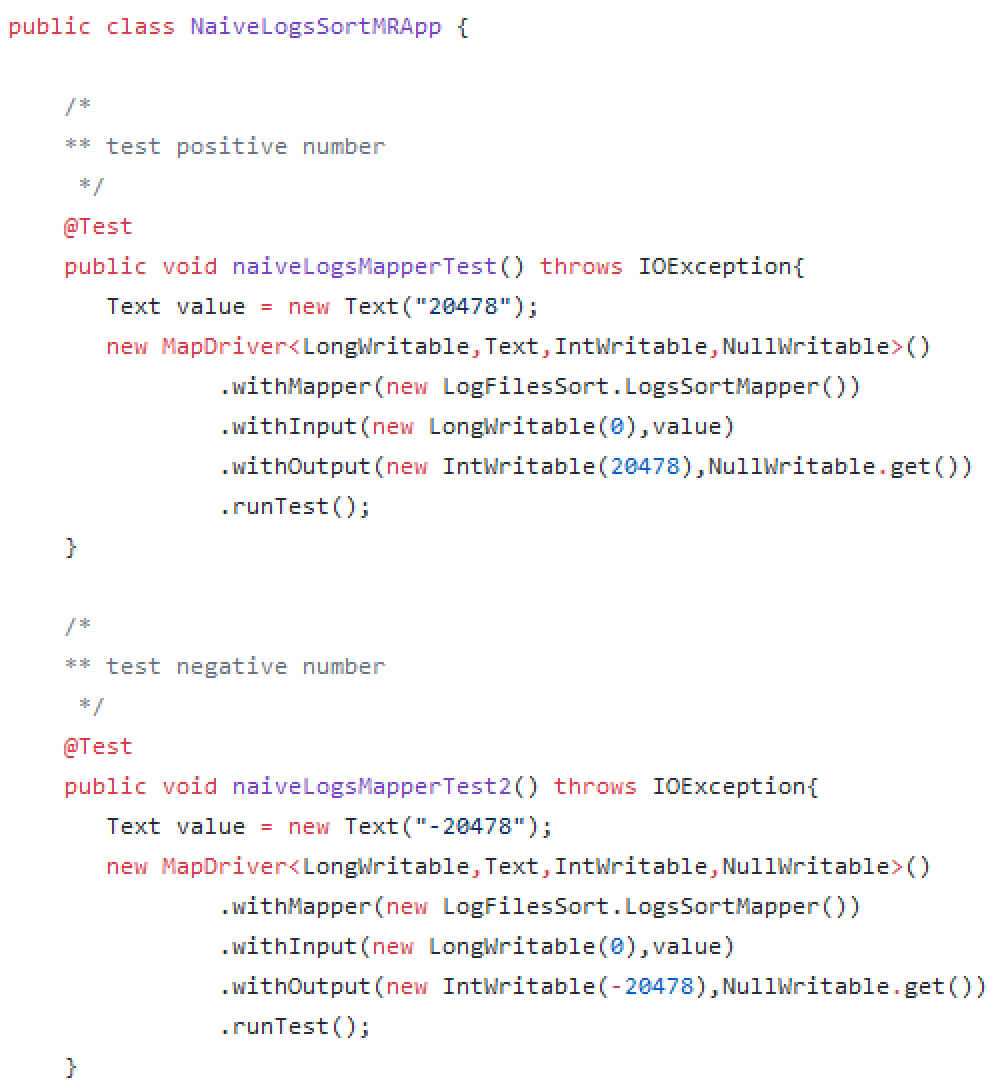
I want to get a total sorted log like this, note there could be same timestamp **in the same or different logs**:

```
1:
        1,
        1,
2:
        4
3:
        7,
4:
        18,
        18,
5:
        48,
6:
        70,
        70
…
```

The mapreduce frame work are a good fit for unit test, because they have separate in/out and processes. MRUnit is the testing library used for mapreduce programming, it is usually used with Junit. I use maven in this project for dependency management, parts of the Unit testing POM is as follows, for complete POM file, I have uploaded it in the project zip file:

```
<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-core</artifactId>
        <version>1.2.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.8.5</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-collections4</artifactId>
        <version>4.1</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.mrunit</groupId>
        <artifactId>mrunit</artifactId>
        <version>1.1.0</version>
        <classifier>hadoop1</classifier>
        <scope>test</scope>
    </dependency>
</dependencies>
```

For a naïve solution, I can use one mapper to process each record and generate one <IntWritable,NullWritable> pair for each record. The V2 type is NullWritable here because I simplify the problem by eliminating the event description. The MRUnit test's idea is very simple, pass a record as input to mapper and check the output is the one we want. The MRUnit test code snippet is as follows:



```
public class NaiveLogsSortMRApp {

    /*
    ** test positive number
     */
    @Test
    public void naiveLogsMapperTest() throws IOException{
       Text value = new Text("20478");
       new MapDriver<LongWritable,Text,IntWritable,NullWritable>()
               .withMapper(new LogFilesSort.LogsSortMapper())
               .withInput(new LongWritable(0),value)
               .withOutput(new IntWritable(20478),NullWritable.get())
               .runTest();
    }

    /*
    ** test negative number
     */
    @Test
    public void naiveLogsMapperTest2() throws IOException{
       Text value = new Text("-20478");
       new MapDriver<LongWritable,Text,IntWritable,NullWritable>()
               .withMapper(new LogFilesSort.LogsSortMapper())
               .withInput(new LongWritable(0),value)
               .withOutput(new IntWritable(-20478),NullWritable.get())
               .runTest();
    }
```

withInput() method gets the test data and withOutput() method specifies the expected output. Note even though there might not be negative timestamp in the system, it is still a good idea to have a separate test case for negative number.

Reducer MRUnit test has similar structure, with a difference that the input for reducer is <k2, list<v2>>, so I need to feed the reducer a collection of v2. The code snippet is as follows:

```
/*
** reducer test
 */
@Test
public void naiveLogsReducerTest() throws IOException{
   new ReduceDriver<IntWritable,NullWritable,Text,NullWritable>()
           .withReducer(new LogFilesSort.LogsSortReducer())
           .withInput(new IntWritable(7),
                   Arrays.asList(NullWritable.get(),NullWritable.get())
           )
           .withOutput(
                   new Text("0 : at time 7 2 item(s) recorded."),
                   NullWritable.get()
           )
           .runTest();
}
```

In fact, with those 3 test cases, a naïve mapreduce solution to this problem is almost there. Before I jump to the solution, I need to construct a larger test case.

For a larger test case, I want to generate a test file contains more timestamp, feed it to mapreduce job and compare the output with the expected output.

In fact, I write a separate class InputDataGenerator. It has a function

```
public int[] InputDataGenerator(int lo, int hi, int k){
}
```

Parameter lo specifies the lower bound of time range which hi specifies the upper bound. K is the number of data records I want in that range. It will generate an interger array which contains the timestamps. Note every record in that range needs to have **equal possibility** of showing up in the array, which means every number in that range have possibility of K/(hi-lo+1) of appearing in the final array.

Because log files are very large and the InputDataGenerator() function **will be called multiple times** to generate log files. It is very important that I optimize the space and time complexity of this function.

One naïve thought is to have a array int[] result with size K. Every time selecting a random number from the range [lo, hi]. If the selected item is not previously added, then adding it to the result array. Repeat until K item are put into the array. The check process will require iterate through the exited result once, so the time complexity of this naïve solution is O(K^2). Because K is big for the log file, this solution is too slow. The check process could be made faster by the use of hashset, but the cost is additional space and heavy calculation of hash function.

My idea is to use reservoir sampling to solve it in O(n) time and O(1) space(O(1) despite the necessary K size output array). The algorithm of this solution is as follows:

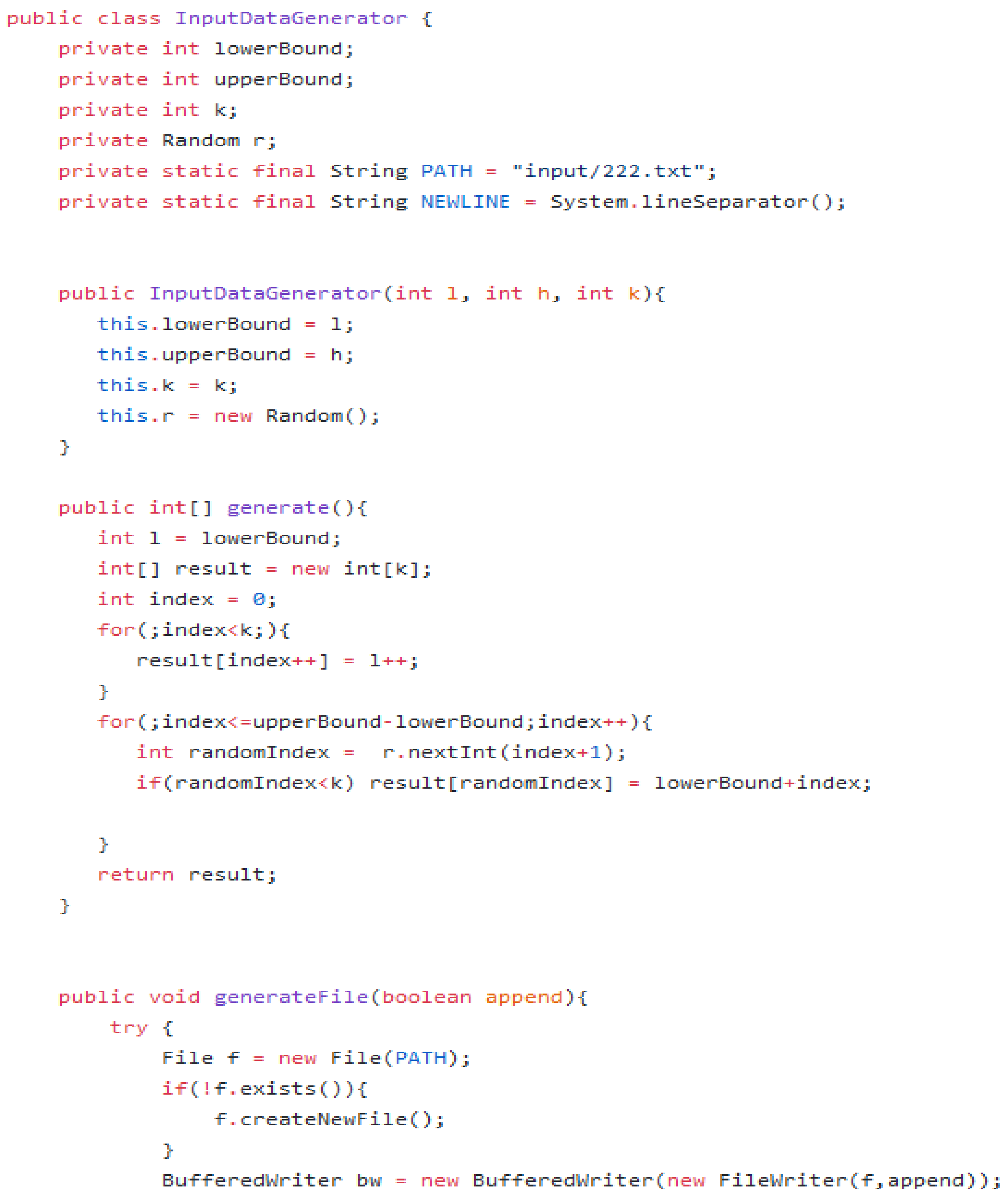
1. Copy first K items into the array.(index 0 to K-1)
2. One by one processing the following number in this way(index K, to hi-lo):

    Generate a random number in range [0, current index+1). If the random number is less than K, then put the corresponding number, which is lo+current , to [random number] position of the array.

After one round of iteration, the result array is what I want.

This method is correct because. For the first k elements, the possibility of it appearing in the final array is equal to all the subsequent numbers starting from index K don't fall into its slot, P = 1*(K/K+1)*(K+1/K+2)….*((hi-lo)/(hi-lo+1)) = K/ (hi-lo+1). For the elements after first K, say its index in n, the possibility is the random number for it falls into one slot between 0 and K, and all the subsequent number don’t replace it, P = (K/(N+1))*((N+1)/(N+2))…*)….*((hi-lo)/(hi-lo+1)) = K/ (hi-lo+1).

I can call this function multiple times to generate all the log files. The code is actually very simple, shown as follows. Also note the class also provides utility functions to write files:

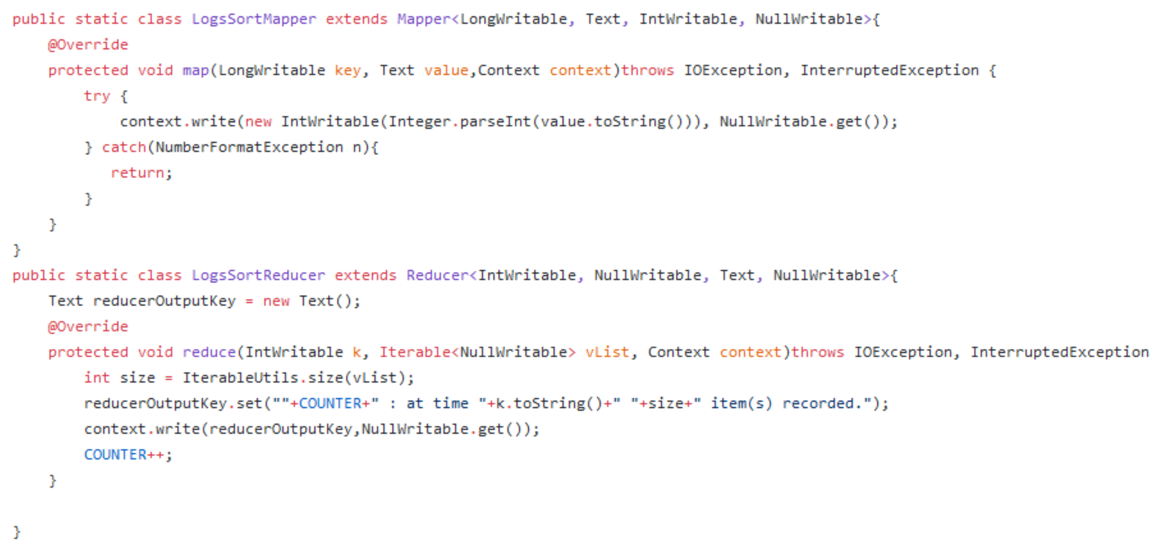


```
public class InputDataGenerator {
    private int lowerBound;
    private int upperBound;
    private int k;
    private Random r;
    private static final String PATH = "input/222.txt";
    private static final String NEWLINE = System.lineSeparator();


    public InputDataGenerator(int l, int h, int k){
       this.lowerBound = l;
       this.upperBound = h;
       this.k = k;
       this.r = new Random();
    }

    public int[] generate(){
       int l = lowerBound;
       int[] result = new int[k];
       int index = 0;
       for(;index<k;){
          result[index++] = l++;
       }
       for(;index<=upperBound-lowerBound;index++){
          int randomIndex =  r.nextInt(index+1);
          if(randomIndex<k) result[randomIndex] = lowerBound+index;

       }
       return result;
    }


    public void generateFile(boolean append){
        try {
            File f = new File(PATH);
            if(!f.exists()){
                f.createNewFile();
            }
            BufferedWriter bw = new BufferedWriter(new FileWriter(f,append));
```

With this utility class, I create a larger test case for the map reduce work. In the test case, I generate the input file in setup phase and also generate an expected output file. In test phase, I feed it to mapreduce app and compare the outputs. The code is in test/java/SortWithMultipleMRTest.java file.

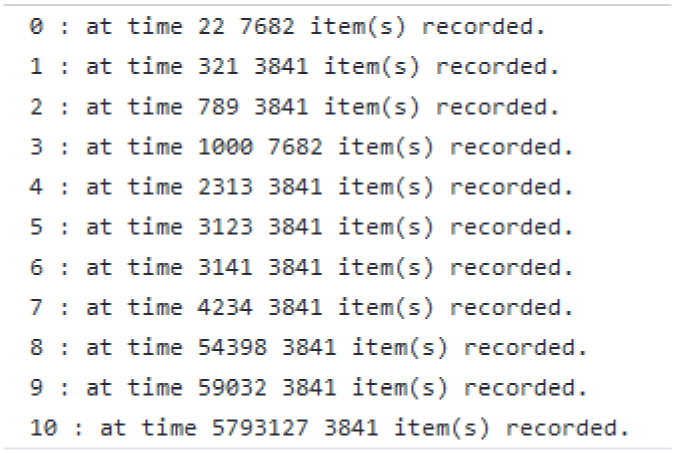
MULTIPLE MR SOLUTION

In fact, a naïve mr app is very simple. Just use feed every record into a mapper, let default shuffler do the partition and sort work, and then use a reducer to group all the log records with the same timestamp. The code snippet for mapper and reducer is shown as follows:



```
public static class LogsSortMapper extends Mapper<LongWritable, Text, IntWritable, NullWritable>{
    @Override
    protected void map(LongWritable key, Text value,Context context)throws IOException, InterruptedException {
        try {
            context.write(new IntWritable(Integer.parseInt(value.toString())), NullWritable.get());
        } catch(NumberFormatException n){
           return;
        }
    }
}
public static class LogsSortReducer extends Reducer<IntWritable, NullWritable, Text, NullWritable>{
    Text reducerOutputKey = new Text();
    @Override
    protected void reduce(IntWritable k, Iterable<NullWritable> vList, Context context)throws IOException, InterruptedException
        int size = IterableUtils.size(vList);
        reducerOutputKey.set(""+COUNTER+" : at time "+k.toString()+" "+size+" item(s) recorded.");
        context.write(reducerOutputKey,NullWritable.get());
        COUNTER++;
    }

}
```

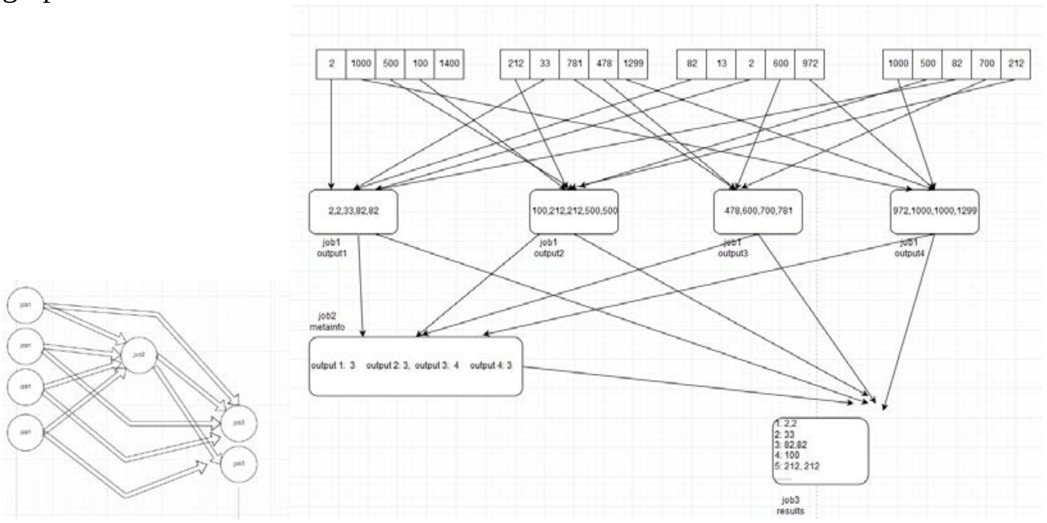
A test case run is performed against the larger input log files stored in LogSortInput/ folder, and the run result of naïve solution is in LogSortOutput/ folder. Parts of the test run result is as follows:



```
0 : at time 22 7682 item(s) recorded.
1 : at time 321 3841 item(s) recorded.
2 : at time 789 3841 item(s) recorded.
3 : at time 1000 7682 item(s) recorded.
4 : at time 2313 3841 item(s) recorded.
5 : at time 3123 3841 item(s) recorded.
6 : at time 3141 3841 item(s) recorded.
7 : at time 4234 3841 item(s) recorded.
8 : at time 54398 3841 item(s) recorded.
9 : at time 59032 3841 item(s) recorded.
10 : at time 5793127 3841 item(s) recorded.
```

This naïve solution does work, in a way totally against the key thoughts of mapreduce: multiple mapper-reducer for complicated system. As mentioned ealier, more complicated problems are solved by multiple simpler mapper-reducer rather than only one complex mapper-reducer.

I use multiple mapper-reudcer as the optimized solution for this problem. I use jobcontrol to build dependencies between 3 mapper-reducers, the code is already discuess in the previous section, the dependencies and the work flow are shown in the following graphs



I set 4 mr job1 instances to work on the input simultaneously. Every job1 intance work on parts of the input, hadoop tries to evenly split the input based on the configuration, in which case contains 4 reducers. Every reducer will write its output into a seperate intermediate output file. This is the first level of parallel processing.

Job2 is waiting for job1s' output. Note there are multiple job1 instances and job2 doesn't need to wait every job1 instance's finish to start its mapper-reducer. In fact job2 uses a first come first serve policy so anther parallel processing is achieved here.

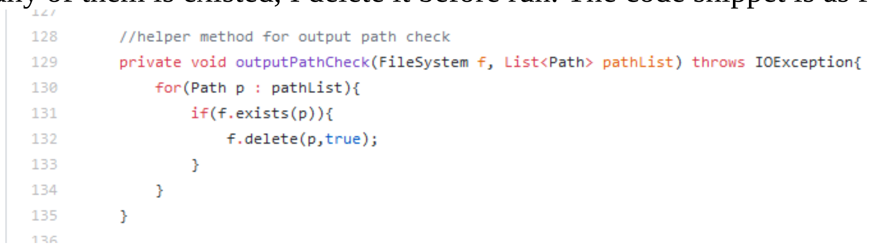
Job2 gets the start offset of every intermediate file and store it as metainfo for job3. Note the metainfo is needed because the index of every timestamp is needed. As shown in the picture, Metainfo is only a small sized data so I use distributed cache to store it so that job3 can access it from memory to speed up the whole process. The code snippet for job1 and job2 is shown as follows:

```
Path job1TempPath = new Path(OUTPUT1_PATH);
Path job2MetaInfo = new Path(OUTPUT2_PATH);
outputPathCheck(f, Arrays.asList(job1TempPath,job2MetaInfo,sortOutput));

//use multiple mappers reducers to process big files
//in a parallel way
Job job = Job.getInstance(conf);
job.setNumReduceTasks(4);
// more configurations omitted

//use addition mr to calculate the meta info of every intermediate file
//and put it into distributed cache
Job job2 = Job.getInstance(conf);
// more configurations omitted

FileInputFormat.setInputPaths(job2, job1TempPath);
FileOutputFormat.setOutputPath(job2, job2MetaInfo);

//Sort with meta info
Job job3 = Job.getInstance(conf);
job3.setJarByClass(LogsSortWithMultipleMR.class);
job3.setMapperClass(SortWithMetaInfoMapper.class);
//no reducer is used in this job
job3.setNumReduceTasks(0);

//put meta info into cache
DistributedCache.addCacheFile(new URI(OUTPUT2_PATH+"/part-r-00000"),conf);
```

FileInputFormat.setInputPaths(job3, job1TempPath);

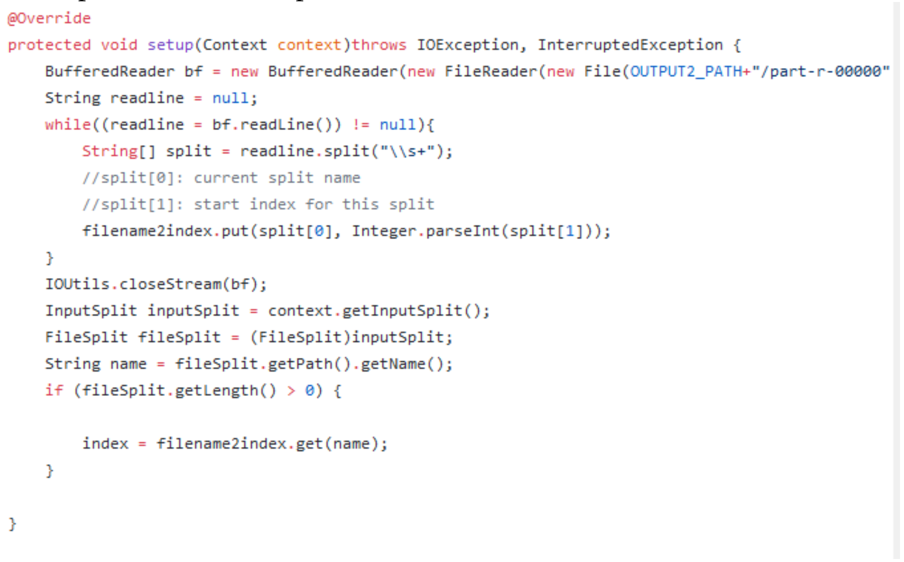
FileOutputFormat.setOutputPath(job3, sortOutput);

Note I configure the distributed cache location in job configuration using addCacheFile() method. Also note because Hadoop throws exception if output path is existed, for multiple mapper-reducers I write a helper method to check the output folders, if any of them is existed, I delete it before run. The code snippet is as follows:



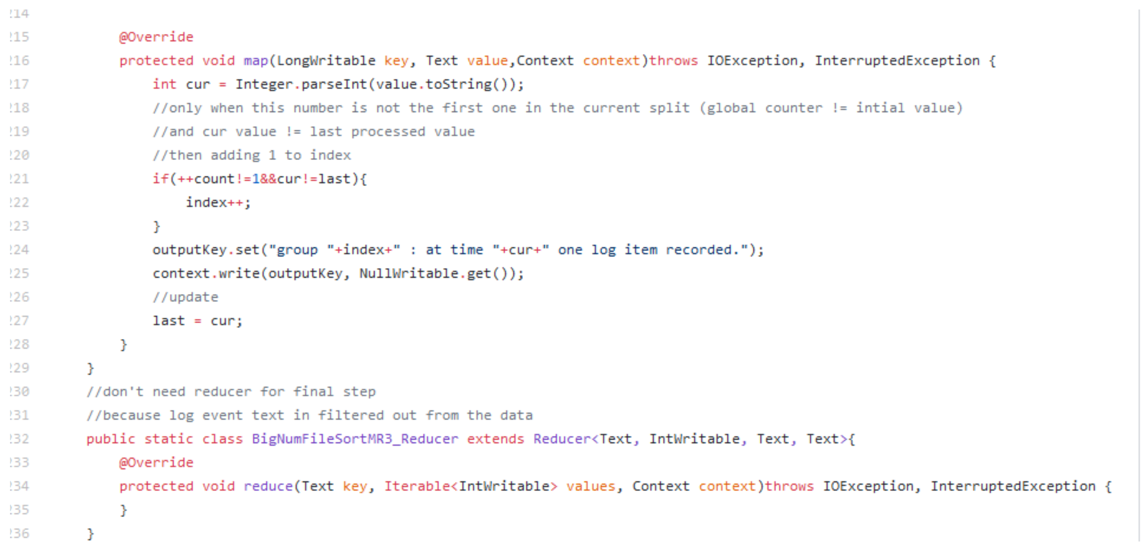
```java
    //helper method for output path check
    private void outputPathCheck(FileSystem f, List<Path> pathList) throws IOException{
        for(Path p : pathList){
            if(f.exists(p)){
                f.delete(p,true);
            }
        }
    }
```

Job3 reads the meta info from cache in the setup phase before mapper-reducer starts. This process is in setup method, as follows:



```java
@Override
protected void setup(Context context)throws IOException, InterruptedException {
    BufferedReader bf = new BufferedReader(new FileReader(new File(OUTPUT2_PATH+"/part-r-00000"
    String readline = null;
    while((readline = bf.readLine()) != null){
        String[] split = readline.split("\\s+");
        //split[0]: current split name
        //split[1]: start index for this split
        filename2index.put(split[0], Integer.parseInt(split[1]));
    }
    IOUtils.closeStream(bf);
    InputSplit inputSplit = context.getInputSplit();
    FileSplit fileSplit = (FileSplit)inputSplit;
    String name = fileSplit.getPath().getName();
    if (fileSplit.getLength() > 0) {

        index = filename2index.get(name);
    }

}
```

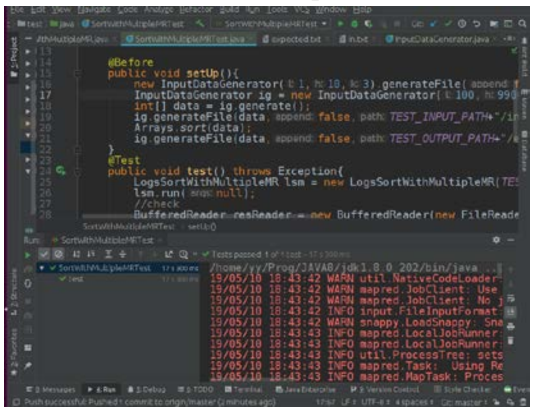
With metainfo, job3 can maintain the correct index for current input split by using a simple counter. Note the edge case of index incrementing in the first if-block. Also note to make unit test easier I modify the ouput format so that the assertion code can work more efficiently, the code snippet is shown as follows:



```
        @Override
        protected void map(LongWritable key, Text value,Context context)throws IOException, InterruptedException {
            int cur = Integer.parseInt(value.toString());
            //only when this number is not the first one in the current split (global counter != intial value)
            //and cur value != last processed value
            //then adding 1 to index
            if(++count!=1&&cur!=last){
                index++;
            }
            outputKey.set("group "+index+" : at time "+cur+" one log item recorded.");
            context.write(outputKey, NullWritable.get());
            //update
            last = cur;
        }
    }
    //don't need reducer for final step
    //because log event text in filtered out from the data
    public static class BigNumFileSortMR3_Reducer extends Reducer<Text, IntWritable, Text, Text>{
        @Override
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)throws IOException, InterruptedException {
        }
    }
```

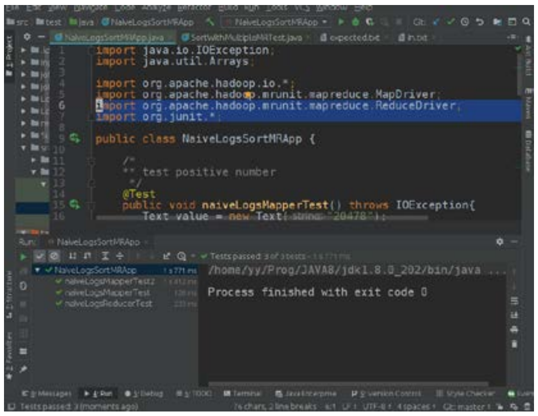
Also note I overwrite the reduce() method because log events description is removed for simplification.

In fact, because I adopt the junit test driven method. After each code modification, I just need to run junit test cases (by one click on the IDE), it is very easy for verification and error locating. The junit test result is shown as follows. Mutilple MR solution outputs can be found in SortOutput/ folder. The unit test results are shown below:



```
    @Before
    public void setUp(){
        new InputDataGenerator( l: 1, h: 10, k: 3).generateFile( append: f
        InputDataGenerator ig = new InputDataGenerator( l: 100, h: 990
        int[] data = ig.generate();
        ig.generateFile(data, append: false, path: TEST_INPUT_PATH+"/i
        Arrays.sort(data);
        ig.generateFile(data, append: false, path: TEST_OUTPUT_PATH+"/
    }
    @Test
    public void test() throws Exception{
        LogsSortWithMultipleMR lsm = new LogsSortWithMultipleMR(TE
        lsm.run( args: null);
        //check
        BufferedReader resReader = new BufferedReader(new FileReade
```

```
/home/yy/Prog/JAVA8/jdk1.8.0_202/bin/java ...
19/05/10 18:43:42 WARN util.NativeCodeLoader
19/05/10 18:43:42 WARN mapred.JobClient: Use
19/05/10 18:43:42 WARN mapred.JobClient: No j
19/05/10 18:43:42 INFO input.FileInputFormat
19/05/10 18:43:42 WARN snappy.LoadSnappy: Sna
19/05/10 18:43:43 INFO mapred.LocalJobRunner
19/05/10 18:43:43 INFO mapred.LocalJobRunner
19/05/10 18:43:43 INFO util.ProcessTree: sets
19/05/10 18:43:43 INFO mapred.Task:  Using Re
19/05/10 18:43:43 INFO mapred.MapTask: Proces
```



```
import java.io.IOException;
import java.util.Arrays;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.apache.hadoop.mrunit.mapreduce.ReduceDriver;
import org.junit.*;

public class NaiveLogsSortMRApp {

    /*
     ** test positive number
     */
    @Test
    public void naiveLogsMapperTest() throws IOException{
        Text value = new Text( string: "20478");
```

```
/home/yy/Prog/JAVA8/jdk1.8.0_202/bin/java ...

Process finished with exit code 0
```

FUTURE DIRECTIONS

1. There is one important feature not discussed in this project: custom partitioner and group comparator. Imagine in the future, in addition to grouping by timestamp, the admin also want to add a secondary group by natural order of node number. A simple solution is to add another mr job to further sort the output by node number. But in fact, using custom practitioner and group comparator can meet the needs without additional overhead of mr job.

For example, I have some log files as follows:

Log1

Time 1, node 1 event...

Time 7, node 1 event…

Log2

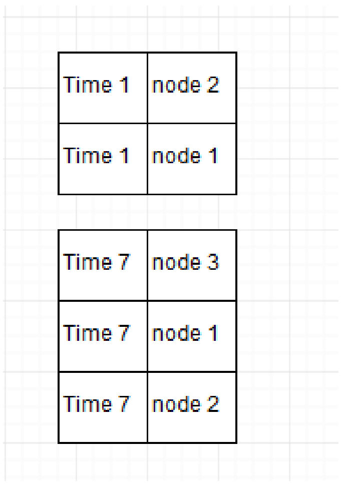
Time 1, node 2 event…

Time 7, node 2 event…

Log 3

Time 7, node 3 event…

Bacause mapreduce only guarantees to sort by key(which in this case, is time), the order of value in the output is not stable (which in this case, is node number and event description). It is possible to get a result like this:

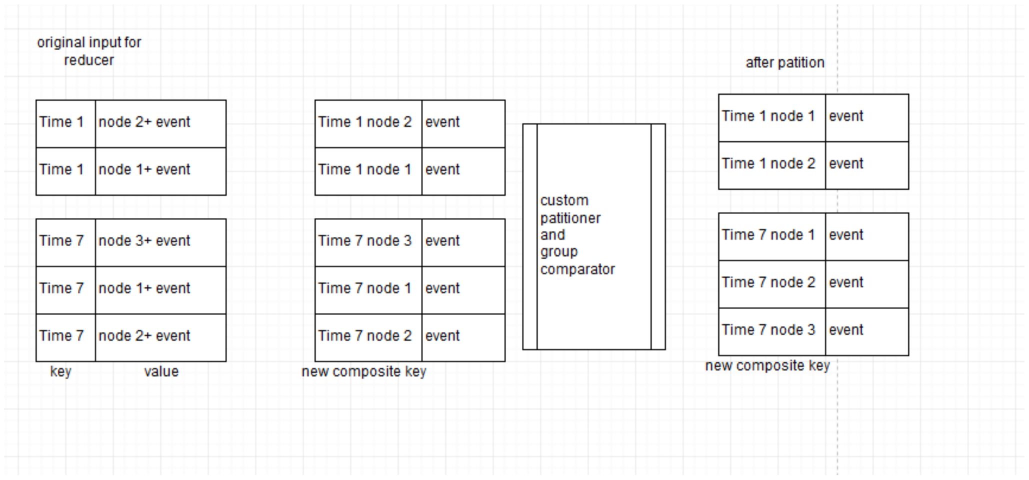


Time 1	node 2
Time 1	node 1

Time 7	node 3
Time 7	node 1
Time 7	node 2

To get a secondary sort by node number, I can add another mapper-reducer to combine the node value with time as the new key. But this will cost addition disk operation and network communication.

In fact, I can make a new key composed of the original time key and node number value. Then I can write a custom comparator which secondarily sort on node number (which is part of the new key). By doing this, shuffler will take the node number in the partition and grouping process, so that when the <k2,list<v2>> arrives the reducer, it is already secondarily sorted by node number. And this method doesn't involves additional heavy disk op and network communication. The process is shown as follows:


original input for reducer
Time 1
node 2+ event
Time 1
node 1+ event
Time 7
node 3+ event
Time 7
node 1+ event
Time 7
node 2+ event
key
value
Time 1 node 2
event
Time 1 node 1
event
Time 7 node 3
event
Time 7 node 1
event
Time 7 node 2
event
new composite key
custom patitioner and group comparator
after patition
Time 1 node 1
event
Time 1 node 2
event
Time 7 node 1
event
Time 7 node 2
event
Time 7 node 3
event
new composite key