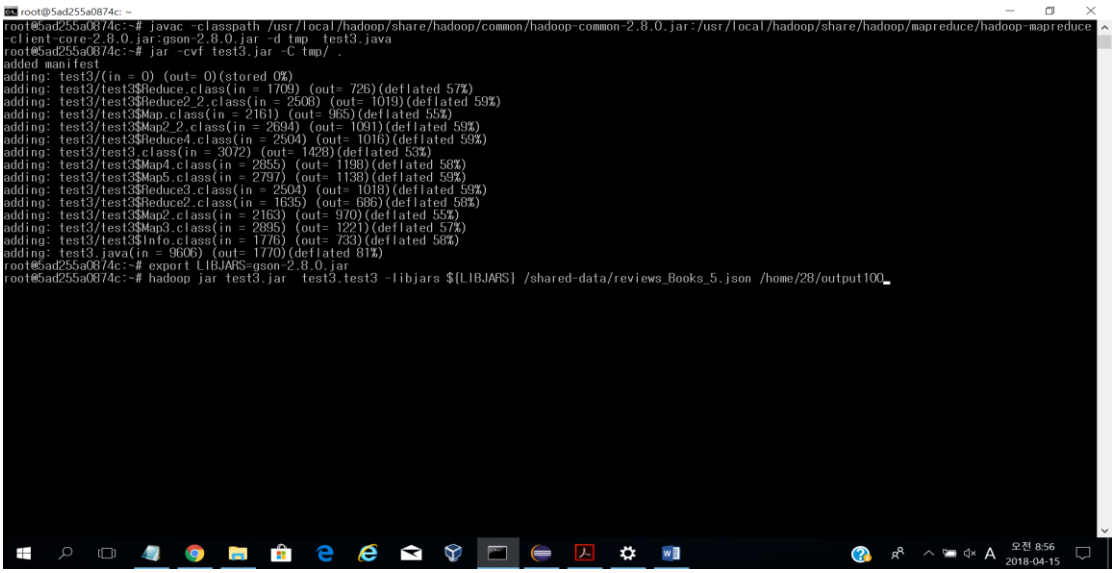
빅데이터 최신과제 프로젝트 1

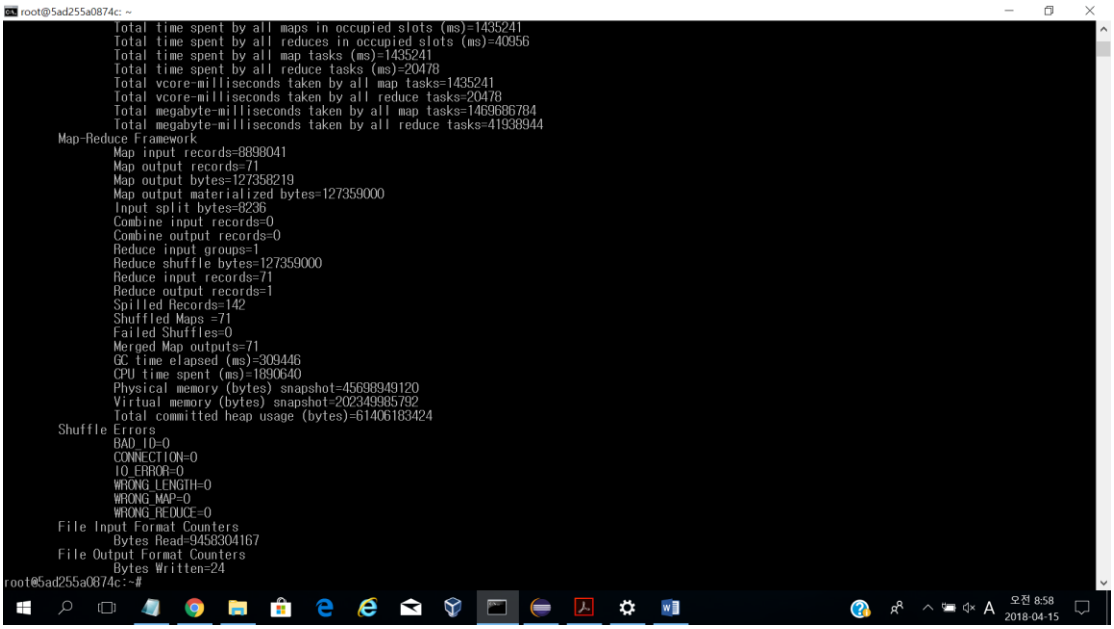
-HADOOP 실습 과제

장용훈 |빅데이터 최신기술 | 2018.04.15

자바 파일을 컴파일 및 실행한 명령어

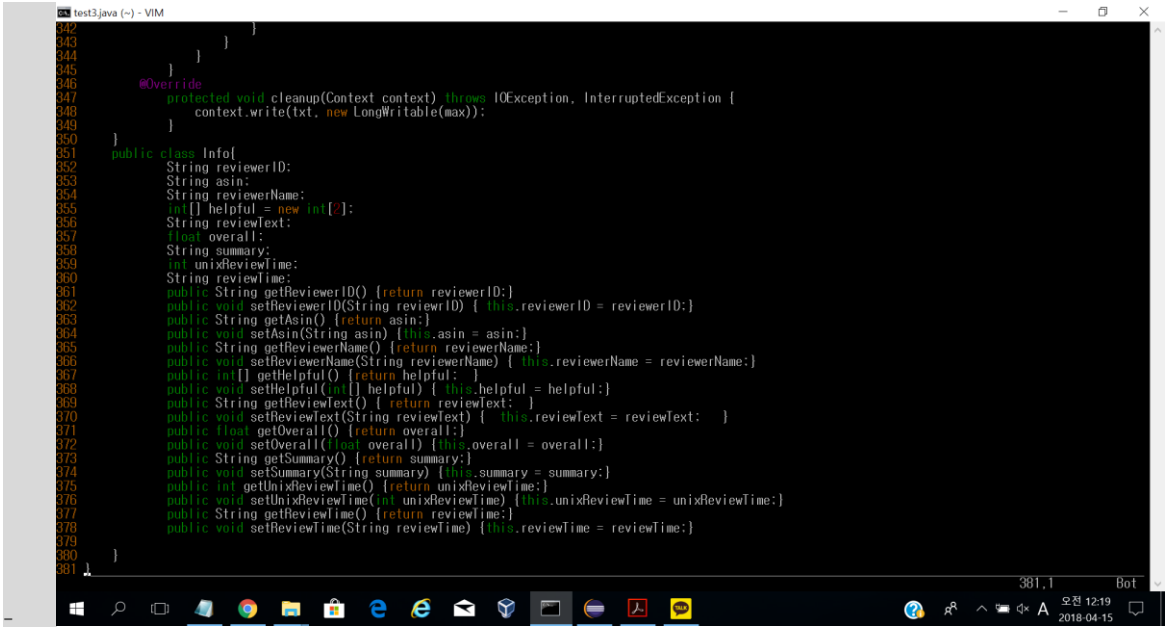
- javac -classpath /usr/local/hadoop/share/hadoop/common/hadoop-common-2.8.0.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.8.0.jar:gson-2.8.0.jar -d tmp test3.java
- 컴파일은 위의 명령어를 사용했다. Wordcount 실습 때 사용한 명령어에 gson 라이브러리를 추가하였다.
- jar -cvf test3.jar -C tmp/ .
- 그 후 컴파일한 클래스들을 test3.jar 이라는 하나의 jar 파일로 만들어 주었다.
- export LIBJARS=gson-2.8.0.jar
- 하둡에서 파싱 작업을 수행할 때, gson 라이브러리가 필요 함으로, 위와 같은 작업을 해주었다.
- hadoop jar test3.jar test3.test3 -libjars ${LIBJARS} /shared-data/reviews_Books_5.json /home/28/output2
- 하둡 작업에서 사용할 jar 파일과 외부 라이브러리, 그리고 패키지 및 클래스 이름을 명시하고, 인풋(args[0]), 아웃풋(args[1]) 위치를 명시해 주어 실행 하였다. 그 결과는 아래와 같다.





문제 해결과정에서 쓰인 공통적인 코드 설명

- Json 을 파싱하기 위해, GSON 라이브러리를 사용하였고, 이때 JSON 을 파싱하기 위한 클래스가 필요하다. 그 클래스는 다음과 같다.



- JSON 파일은 여러 오브젝트들로 이루어져 있는데, 오브젝트 안에 있는 모든 자료들은 Info 클래스에 있는 변수들로 변환되어 이를 이용 할 수 있게 해준다.

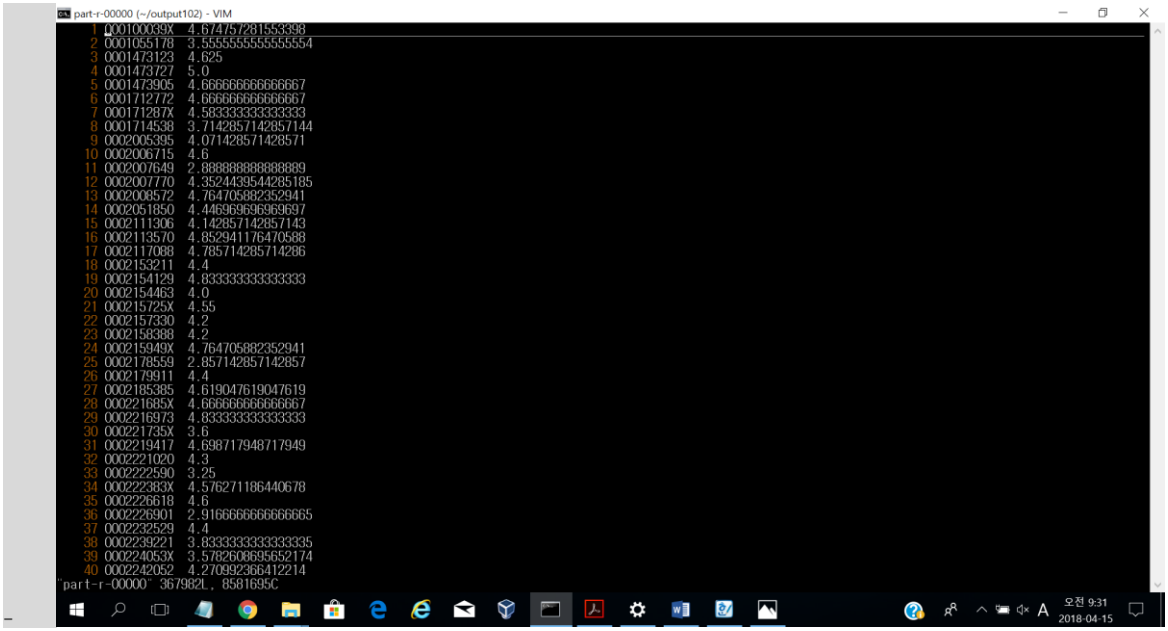
- 각 과제에 쓰인 map 과 reduce 그리고 driver class 는 매우 유사합니다.

- 구분을 위해 map 과 reduce 클래스에 문제에 해당하는 번호를 기입했습니다.

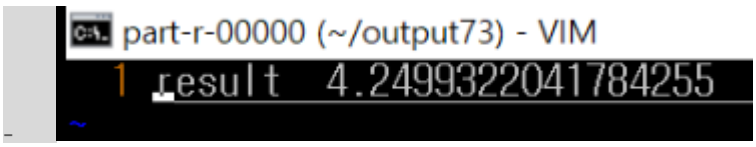
전체 상품의 평균 overall 점수는?

- 두가지 방법으로 문제를 해결해 보았다 첫번째로 전체 평균에 대한 overall 을 구해 보았다. . 두번째는 asin(상품코드)별로 평균 overall 을 구하였다. 결과 값은 아래와 같은 답을 도출 하였다.

- 상품 별 평균 값의 결과이다.



- 전체 상품에 대한 평균이다.



- 첫번째 방법에 사용한 코드는 다음과 같다.

- map 에 쓰인 코드이다. 파싱을 위해 처음에 map 함수에 들어온 value 를 파싱해준다. 이때, 여러 문장들을 split 메서드를 사용하여, 한줄씩으로 값을 변경해주고, 이 변경한 값을(하나의 json object)를 formJson 매서드를 이용하여, 원하는 값을 추출 할 수 있게 해준다. 이때 Info 라는 사용자 정의 클래스를 이용하여 파싱을 원할 하게 해준다. 그 후 info 클래스에

getOverall 이라는 함수를 이용하여 값을 구하면 이값은 output value 가 된다. Output key 값은 하나의 reducer 로 모으기 위해 new Text("result")라는 값을 이용하여 모든 key 를 "result"로 설정해준다. 따라서 reducer 로 가는 것은 ("result",num)형식이다.

```
public static class Map extends Mapper<LongWritable, Text, Text, LongWritable> {
    private final static LongWritable ONE = new LongWritable(1);
    private Text word = new Text("result");

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {


            String line = value.toString();
            String[] tuple = line.split("₩n");
            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {

                Info info = gson.fromJson(tuple[i], Info.class);
                long num = (long)info.getOverall();

                context.write(word, new LongWritable(num));
                }
        //      context.write(new Text("result"), new LongWritable(1));
            }
            catch(Exception e){e.printStackTrace();}


        }
    }
```

- 다음은 reducer 부분이다. Reduce 과정은 하나의 키를 사용했기에 모든 자료가 하나의 reduce 에서 수행이 된다. 전체에 대한 평균 값을 구하는 것임으로, Iterable 을 한 values 를 for 를 이용하여 하나의 value 로 변환 시켜 준다. 그후 이 값들을 sum 에 더해준다. 이 때 for 루프가 한번씩 수행할 때마다, total 이라는 값에 1 을 더해주어, 상품의 총 개수를 구해준다. 그후 (double)로 형변화를 진행하여 나누어 준다. 이때 아웃풋은 (Text key, DoubleWritable value)이다. 결과 값은 4.24993 이다.

```
public static class Reduce extends Reducer<Text, LongWritable, Text, DoubleWritable> {
    @Override
    public void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {
        double sum = 0;
        long total = 0;
        for (LongWritable val : values) {
            sum += val.get();
            total +=1;
        }
        sum = (double)sum/total;
        context.write(key, new DoubleWritable(sum));
    }
}
```

- 두 번째 방식의 map 이다. 처음 방식과 다른 점은 map 의 output key 로 상품별 id(asinId)라는 것이다. 이는 Info 클래스의 getAsin()을 이용하여 word 에 set 해주었다.

```
public static class Map extends Mapper<LongWritable, Text, Text, LongWritable> {
    private final static LongWritable ONE = new LongWritable(1);
    private Text word = new Text("result");

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {


            String line = value.toString();
            String[] tuple = line.split("₩₩n");
            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {

                Info info = gson.fromJson(tuple[i], Info.class);
                long num = (long)info.getOverall();
                word.set(info.getAsin());
                context.write(word, new LongWritable(num));
               }
        //        context.write(new Text("result"), new LongWritable(1));
            }
            catch(Exception e){e.printStackTrace();}


        }
    }
```

- 두번째 방식의 reduce 는 첫번째 방식에 사용한 reduce 와 동일하다. 다만 다른 점이 있다면, 하나의 reduce 가 아닌 복수개의 reduce 작업이 이루어진다. 상품들의 ,key 가 다르기 때문에 각 상품별로 reduce 작업이 이루어진다.

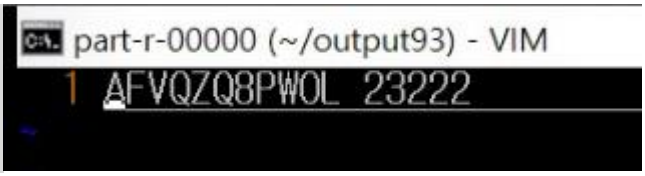
- 1 번과제의 드라이버 코드이다. Job 을 설정해주고 이 job 에 쓰일 jarClass, outputValueClass, OutputKeyClass, Mapper class, Reduce class 를 설정해주었다. 그리고 마지막으로 input 과 output path 를 설정해주었다. 이 부분은 2 번을 제외하고는 3,4,5 과제에서 거의 동일하다.

```
public class test3 extends Configured implements Tool {
    public static void main(String[] args) throws Exception {

        System.out.println(Arrays.toString(args));
        int res = ToolRunner.run(new Configuration(), new test3(), args);

        System.exit(res);
    }

    @Override
    public int run(String[] args) throws Exception {
        System.out.println(Arrays.toString(args));

        Configuration conf1=new Configuration();
        Job job = Job.getInstance(conf1);
        job.setJarByClass(test3.class);
        job.setOutputKeyClass(Text.class);
    //    job.setOutputValueClass(DoubleWritable.class);

        job.setOutputValueClass(DoubleWritable.class);
        job.setMapOutputValueClass(LongWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);


         FileInputFormat.addInputPath(job,new Path(args[0]));
         FileOutputFormat.setOutputPath(job,new Path(args[1]));
         job.waitForCompletion(true);
```

- 1 번 과제에서 쓰이는 Map Reduce 방식은 다음과 같다. 첫번째 방법에서는 인풋 값을 여러 개의 블록조각으로 나누어, "result"라는 key 값과 overall value 를 가진 쌍으로 reduce 에게 보내주게 된다. 이때 key 값이 동일 하기 때문에, 하나의 reduce 작업으로 수행된다. 두번째 방법에서는 key 값이 동일 하지 않기 때문에, 각각 key 값에 따른 reduce 작업이 이루어지고, 결과만 한파일에 쓰이게 된다.

가장 많은 리뷰를 남긴 사용자의 아이디 및 리뷰 횟수는?

- 1 번과제와 다르게 한번의 map/reduce 작업이 아닌 두번의 map/reduce 작업을 수행하여 결과를 얻었다. 결과는 다음과 같은 결과를 얻었다.



- 1 번에서 설명한 map 방식과 크게 다른 점이 없다. 다만 reviewerId 를 얻어 key 값으로 하고, value 값은 모두 동일한 1 이다. 이는 id 가 한번 나왔다는 뜻이다. 이를 reduce 로 보내준다.

```
public static class Map2 extends Mapper<LongWritable, Text, Text, LongWritable> {
    private final static LongWritable ONE = new LongWritable(1);
    private Text word = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

            String line = value.toString();
            String[] tuple = line.split("\n");
            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {

                Info info = gson.fromJson(tuple[i], Info.class);
                String id = info.getReviewerID();
                word.set(id);
                context.write(word, ONE);
               }
            }
            catch(Exception e){e.printStackTrace();}

            }
}
```

- Reduce 작업은 1 번의 2 번째 방법에서 쓰인 것과 거의 유사하다. reviewerId 별로 data 를 모으고 value 값을 더해준다. 그리고 이 value 들을 reviewerId 별로 각각 적어준다. 그리고 이를 두번째 맵 과정으로 보내준다.

```
public static class Reduce2 extends Reducer<Text, LongWritable, Text, LongWritable> {
    @Override
    public void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {
        long sum = 0;
        for (LongWritable val : values) {
            sum += val.get();
        }

        context.write(key, new LongWritable(sum));
    }
}
```

- 두 번째 map 작업은 첫 번째의 reducer 의 값, (id,횟수)를 (동일한 키값, mapWritable) 형식으로 map 하여 reduce 로 보내주게 된다. Map 함수에 id, 횟수 가 한 줄에 한쌍씩 적혀있는 값이 들어온다. 이를 파싱해 주어야 한다. Value 값을 toString 하여 전체의 String 으로 만들어 준 다음에 이를 한 줄씩으로 split 해주고 그 줄 마다 split 작업을 해준다. 이를 data[]에 저장해준다. data[0]에는 id 가 data[1]에는 횟수가 저장되게 되고, 이를 minfo 라는 MapWritable 형식에 하나씩 put 해준다.(모든 값들이 들어가게 된디) 블록에서 모든 작업이 수행된 후 cleanup 메서드가 수행된다. Cleanup

메서드에서 최종적인 결과를 작성하게 된다. 이는 (Text("result"), MapWritable minfo)형식이다.

```
public static class Map2_2 extends Mapper<LongWritable, Text, Text, MapWritable> {
    private final static LongWritable ONE = new LongWritable(1);
    private Text word = new Text("result");
    private Text word2 = new Text();
    MapWritable minfo = new MapWritable();
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            long num =0;

            String line = value.toString();
            String[] tuple = line.split("₩₩n+");

            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {
                    String []data = tuple[i].split("₩₩s+");
                    minfo.put(new Text(data[0]),new LongWritable(Long.parseLong(data[1])));
                }
            }

            catch(Exception e){e.printStackTrace();}
        }
    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(word,minfo);
        }
}
```

- 두 번째 reduce 과정은 두 번째 맵의 output 인 동일한 키값("result") 와 MapWritable 을 받게된다. 키값이 동일 하기에 하나의 reducer 로 수행하게 된다. 이때 MapWritable 을 for 문을 이용하여 하나씩 얻은 후(이것을 val 이라 하자) 각각의 val 마다 keySet 을 얻어주고 이 keySet 에 for 루플 이용하여 각각의 id 를 얻게 된다. 이 id(key)를 이용하여 값을 얻고(이를 temp 라 하자) 이를 max 와 비교 하게 된다. 만약 max<temp 의 조건을 만족하게 된다면 max = temp 가 되고 txt 에 id 값을 써주게 된다. 이러한 job 이 모두 수행되고나서 cleanup 메서드가 수행된다. 이곳에서 context 에 txt 와 그때의 max 를 적어주게 된다.

```
public static class Reduce2_2 extends Reducer<Text, MapWritable, Text, LongWritable> {
    private long max =0;
    long temp =0;
    private Text txt = new Text();
    @Override
    public void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {
                for(MapWritable val : values){
                Set<Writable> keyset = val.keySet();
                for(Writable id : keyset){
                    temp = (long)((LongWritable)val.get(id)).get();
                    if(max<temp){
                        max = temp;
                        txt.set(id.toString());

                    }
                }
            }
    }

    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(txt, new LongWritable(max));

        }
}
```

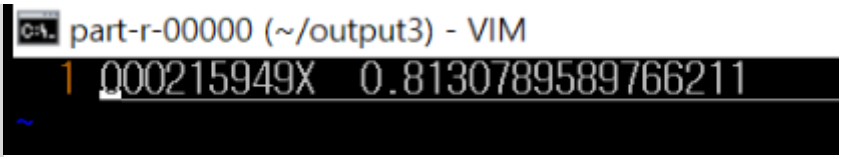
- Driver 클래스에서 1 번과 다른 점은, map/reduce 가 1 번이 아닌 두번이 수행 되었다는 것이다. Map1-reduce1-map2-reduce2 와 같은 순서로 작업이 이루어진다. 첫 번째 작업의 outpath 를 두번째 작업의 input path 로 설정해주고, 첫번째 작업이 끝날 때 까지 기다리게 된다.(wait) 첫 번째 작업이 끝난다면, 첫번째 작업을 전송 후 지워준다. 두 번째 작업은 첫번째 작업의 아웃풋을 인풋으로 받아 처리해준다. 첫 번째 작업에서 사용한 map/reduce 와 다른 별도의 map/reduce 의 class 를 mapper class, output class 로 설정해 준다. 그리고 outputvalue, outputkey class 또한 설정해 준다. 다른 점이 있다면, 추가로 mapOutputValue class 라는 첫 번째 과정에서는 쓰이지 않은 설정을 추가로 해주게 된다. 왜냐하면 map 에서 outputvalue 의 형식이 mapWritable 이기에 이와 대응이 되어야 하기 때문이다. 그 후 사용자가 정의해준 path(args[1]) 작업 수행의 결과를 써주게 된다.

```
@Override
public int run(String[] args) throws Exception {
    System.out.println(Arrays.toString(args));

    Configuration conf1=new Configuration();
    Job job = Job.getInstance(conf1);
    job.setJarByClass(test3.class);
    job.setOutputKeyClass(Text.class);
//     job.setOutputValueClass(DoubleWritable.class);

    job.setOutputValueClass(DoubleWritable.class);
    job.setMapOutputValueClass(LongWritable.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    Path outputPath=new Path("FirstMapper");
     FileInputFormat.addInputPath(job,new Path(args[0]));
     FileOutputFormat.setOutputPath(job,outputPath);
    outputPath.getFileSystem(conf1).delete(outputPath);
    job.waitForCompletion(true);
    Configuration conf2=new Configuration();
    Job j2=Job.getInstance(conf2);
    j2.setJarByClass(test3.class);
    j2.setMapperClass(Map2_2.class);
    j2.setReducerClass(Reduce2_2.class);
    j2.setOutputKeyClass(Text.class);
    j2.setOutputValueClass(LongWritable.class);
    j2.setMapOutputValueClass(MapWritable.class);
    Path outputPath1=new Path(args[1]);
    FileInputFormat.addInputPath(j2, outputPath);
    FileOutputFormat.setOutputPath(j2, outputPath1);
    outputPath1.getFileSystem(conf2).delete(outputPath1, true);
    j2.waitForCompletion(true);
    return 0;
SERT --
```

- 두 번째 문제의 map/reduce 의 원리는 다음과 같다. 연속적인 map/reduce 를 통하여, 한번으로 처리하기 어려운 일을 해주게 된다. 첫번째 map/reduce 에서 사용자 아이디 별로 횟수를 집계 해주고, 이 데이터를 바탕으로 두 번째 map/reduce 에서는 가장 많은 횟수를 가진 아이디를 찾아 이것을 결과 값으로 보여주게 된다.

Helpful[a,b] 에서 b 값이 10 보다 큰 사용자 중에서 도움이된다고 하는 사람들의 비율이 가장 높은 아이템(asin) 의 아이디는?

- 두번의 map/reduce 를 통하여 답을 얻는다. 첫번째 map/reduce 에서는 집계한 아이템별 총비율을 구할 수 있고 이를 바탕으로 두번째 map/reduce 를 수행하여 최종적으로 가장 큰 비율의 값과 그에 따른 아이디를 얻게된다. 결과는 아래와 같다.



- Map 은 앞의 문제의 과정과 동일하게, 우선 파싱을 해준다. 그 후 asinId 와 이 helpful[a,b]에 대한 정보를 얻는다. 그 후 if 문을 통하여 helpful[1]=b 가 10 보다 큰지 여부를 확인 한다. 만약 크다면 이정보를 mapWritable minfo 변수에 put 해준다. 이때 helpful 의 값들은 double 로 변환하여 각각 num1, num2 에 할당한다. 그리고 minfo.put(new DoubleWritable(num1), new DoubleWritable(num2))를 해준다. 그리고 asinId 는 word2 라는 text 에 set 해준다. 루프가 한번 진행될 때 마다 (asinId,MapWritable(helpful)) 짝으로 write 이 수행되어 reduce 의 Input 으로 가게된다.

```
public static class Map3 extends Mapper<LongWritable, Text, Text, MapWritable> {
    private Text word2 = new Text();
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            double num =0;

            String line = value.toString();
            String[] tuple = line.split("₩n");
            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {

                 MapWritable minfo = new MapWritable();
                Info info = gson.fromJson(tuple[i], Info.class);
                int [] help = info.getHelpful();
                if(help[1]>10){
                word2.set(info.getAsin());
                double num1 = (double)help[0];
                double num2 = (double)help[1];
                minfo.put(new DoubleWritable(help[0]), new DoubleWritable(help[1]));
                context.write(word2,minfo);
                }
            }

        }

            catch(Exception e){e.printStackTrace();}
        }

}
```

- reduce 에서는 앞의 map 에서 전달받은 각기 다른 key 인 asinId 와 value(mapWritable)을 가지고 작업을 수행한다. 복수개의 key 를 가지기 때문에 다수의 reducer 로 작업이 이루어진다. 이때 value 는 mapWritable 형식인데, (doubleWritable,doubleWritable) 쌍으로 구성되어 있다. 우선 Iterable 한 values 에 접근하기 위해 for 를 이용하여 개별적인 value(MapWritable)에 접근한다. 그 후 val 의 keyset 을 얻고, 이 keyset 을 바탕으로 한번더 for 를 이용하여 key(id)를 얻는다. 이것을 실질적으로 doublewritable 형식이고 helpful[0]이다. 이를 (DoubleWritable)id.get()을 이용하여 double 값으로 구하여 a 라는 변수에 계속해서 더해준다. Helpful[1]에 해당하는 값은 val 을 이용한다. ((DoubleWritable)val.get(id)).get()으로 구하게 된다. 이는 val 이라는 mapwritable 객체에서 주어진 키를 이용하여 값을 얻고, 이값을 doubleWritable 형식으로 형변화 해준 후 double 타입으로 얻은 것이다. 이 얻은 helpful[1]은 b 에 계속하여 더해준다. 이때 a 와 b 의 덧셈회수는 동일하다. Values 의 for 문이 모두 동작 한다음, b/a 를 구할 수 있다. 이를 num 에 할당해주고, 이때의 key(asinId)와 value(a/b)를 write 한다.

```
public static class Reduce3 extends Reducer<Text, MapWritable, Text, DoubleWritable> {
    private double a =0;
    double b =0;
    String temp;
    private Text txt = new Text();
    @Override
    public void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {
                for(MapWritable val : values){
                Set<Writable> keyset = val.keySet();
                for(Writable id : keyset){
                    b += (double)((DoubleWritable)val.get(id)).get();
                    temp = id.toString();
                    a += (double)((DoubleWritable)id).get();
                }
            }
            Double num = a/b;
            context.write(key, new DoubleWritable(num));
    }
}
```

- 두번째 map 과정이다. 인풋으로 key(asinId)와 value(a/b)를 받아서 처리를 해주게 된다. 이 부분은 2 번 문제의 2 번째 map 과정과 데이터 타입만 다르고 동일하다. 먼저 받은 Input 을 파싱 해준다. Value 를 toString()을 통하여 String 형식으로 만들어 주고 그 다음 한줄씩 처리를 위해 split("₩₩n+")하여 배열에 할당해 준다. 그리고 이 배열(tuple)만큼 for 문이 반복이 된다. 각각의 배열의 String 마다 split("₩₩s+")(이것은 공백이

기준이다)를 하여 data 배열을 생성해준다. 이 data 를 MapWritable minfo 에 data[0]은 asinId 임으로 text 로 , data[1] 은 앞에서 구해준 a/b 임으로 double 로 형 변환 후 DoubleWritable 로 put 해준다. 모든 map 작업이 끝난 후에 cleanup 이 실행되어 ("result",minfo)형식으로 써주게 된다. 이 수행한 map 들은 하나의 reduce 로 가게된다.

```
public static class Map3_2 extends Mapper<LongWritable, Text, Text, MapWritable> {
    private final static LongWritable ONE = new LongWritable(1);
    private Text word = new Text("result");
    private Text word2 = new Text();
    MapWritable minfo = new MapWritable();
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            long num =0;

            String line = value.toString();
            String[] tuple = line.split("₩₩n+");

            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {
                    String []data = tuple[i].split("₩₩s+");
                    minfo.put(new Text(data[0]),new DoubleWritable(Double.parseDouble(data[1])));
                }
            }

            catch(Exception e){e.printStackTrace();}
        }
    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(word,minfo);
        }
}
```

- 아래의 설명은 2 번 문제의 두번째 reduce 설명과 동일한 설명이다. 2 번 문제와 다른점이 있다면, longwritable 이 doublewritable 이고, long 타입이었던 변수들이 double 이라는 것이다.

- 두 번째 reduce 과정은 두 번째 맵의 output 인 동일한 키값("result") 와 MapWritable 을 받게된다. 키값이 동일 하기에 하나의 reducer 로 수행하게 된다. 이때 MapWritable 을 for 문을 이용하여 하나씩 얻은 후(이것을 val 이라 하자) 각각의 val 마다 keySet 을 얻어주고 이 keySet 에 for 루플 이용하여 각각의 id 를 얻게 된다. 이 id(key)를 이용하여 값을 얻고(이를 temp 라 하자) 이를 max 와 비교 하게 된다. 만약 max<temp 의 조건을 만족하게 된다면 max = temp 가 되고 txt 에 id 값을 써주게 된다. 이러한 job 이 모두 수행되고나서 cleanup 메서드가 수행된다. 이곳에서 context 에 txt 와 그때의 max 를 적어주게 된다.

```
public static class Reduce3_2 extends Reducer<Text, MapWritable, Text, DoubleWritable> {
    private double max =(double)0;
    double temp =(double)0;
    private Text txt = new Text();
    @Override
    public void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {

            for(MapWritable val : values)
            {
                Set<Writable> keyset = val.keySet();
                for(Writable id : keyset){
                    temp = ((DoubleWritable)val.get(id)).get();
                    if(max<temp){
                        max = temp;
                        txt.set(id.toString());
                    }
                }
            }
        }
    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(txt, new DoubleWritable(max));
        }
}
```

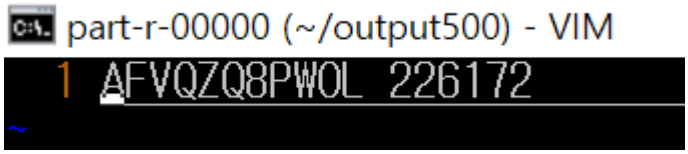
- Driver 부분은 2 번의 driver 와 작동방식과 구성이 동일하다. 다만 다른 점이 있다면, 첫번째 job 에서 mapper, reducer class 로 Map3.class, Reduce3.class 로 set 을 해준 부분과 outputValue 로 DoubleWritable.class 로 set 해주었다는 것이다. 그리고 두 번째 job 도 역시 마찬가지로 첫번째 reduce 의 결과가 두번째 map 의 input 으로 들어가게 된다. 그리고 mapper, reducer class 로 Map3_2.class, Reduce3_2.class 로 set 을 해준 부분과 outputValue 로 DoubleWritable.class 그리고 mapOutputValue 로 MapWritable.class 로 설정해준 부분의 차이만이 존재한다. 따라서 set 해준 타입의 차이만 존재하지 본질적으로 2 번 driver 와 동일하다.

```
@Override
public int run(String[] args) throws Exception {
    System.out.println(Arrays.toString(args));

    Configuration conf1=new Configuration();
    Job job = Job.getInstance(conf1);
    job.setJarByClass(test3.class);
    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(DoubleWritable.class);
    job.setMapOutputValueClass(MapWritable.class);
    job.setMapperClass(Map3.class);
    job.setReducerClass(Reduce3.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    Path outputPath=new Path("FirstMapper");
     FileInputFormat.addInputPath(job,new Path(args[0]));
     FileOutputFormat.setOutputPath(job,outputPath);
    outputPath.getFileSystem(conf1).delete(outputPath);
    job.waitForCompletion(true);
    Configuration conf2=new Configuration();
    Job j2=Job.getInstance(conf2);
    j2.setJarByClass(test3.class);
    j2.setMapperClass(Map3_2.class);
    j2.setReducerClass(Reduce3_2.class);
    j2.setOutputKeyClass(Text.class);
    j2.setOutputValueClass(DoubleWritable.class);
    j2.setMapOutputValueClass(MapWritable.class);
    Path outputPath1=new Path(args[1]);
    FileInputFormat.addInputPath(j2, outputPath);
    FileOutputFormat.setOutputPath(j2, outputPath1);
    outputPath1.getFileSystem(conf2).delete(outputPath1, true);
    j2.waitForCompletion(true);

    return 0;
}
```

- 3 번 문제의 map/reduce 원리는 다음과 같다. map 에서 parsing 을 한 후 helpful[a,b]의 값과 asin 의 아이디를 얻는다. 그 후 b>10 인 조건을 만족하는 값들을 key 에는 asinId, value 에는 mapwritable 을 이용하여 두개의 helpful 값을 하나의 형태로 만든 것을 넣어준다. 첫번째 reduce 에서 key(asinId)별로 helpful 의 값을 모두 더해준다. 모든 덧셈이 끝난 후에 a/b 를 value 로 key(asinId)로 write 해주고, 두번째 맵으로 보내준다. 이때 결과들을 임시로 저장한 후에 두번째 map 으로 들어간다. 두번째 map 에서는 동일한 키를 설정해주고 map writable 에 입력으로 받은 key-value 를 넣어준다. Map 의 수행이 끝난 후 cleanup 메서드를 이용하여 동일한 key(“result”) 그리고 mapwritable 을 write 한다. Reduce 는 키가 모두 동일하기 때문에 단일한 reduce 과정을 통해 최대값을 구 할 수 있게 된다. 마찬가지로 reduce 작업이 끝나 후 , cleanup 메서드에서 asinId 와 최대값을 결과로 적는다.

Reviewer 별로 helpful 필드 값을 모았을 때 가장 높은 a 값을 가지는 사용자의 (reviewerID) 아이디는?

- ReviewerID 는 값이 중복 됨으로 각 Id 별로 helpful 의 a,b 를 더하여 가장 높은 값을 얻었고, 이를 바탕으로 가장 큰 값을 가지는 Id 와 그때의 값을 구하였다. 이때 map-reduce 는 2 번의 작업을 필요로 한다. 결과는 아래와 같다.



- Map 의 과정은 앞의 문제와 동일한 방법으로 parsing 을 진행한다. 파싱을 info 에 해주고 info.getReviewerID 를 통하여 ID 를 얻는다. 그리고 info.getHelpfil 을 통하여 helpful 배열을 얻어, num 에 helpful[0]+helpful[1]을 더하여 할당을 해준다. 그 다음 id 는 word.set 을 통하여 id 를 word 에 저장해주고, 마지막으로 key=word, value=num 으로 하여 write 을 해준다. 따라서 (word(id), num(helpful)) 값을 가진 key-value 형식으로 reduce 로 보내게된다.

```
public static class Map4 extends Mapper<LongWritable, Text, Text, LongWritable> {
    private Text word = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

            String line = value.toString();
            String[] tuple = line.split("₩n");
            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {

                Info info = gson.fromJson(tuple[i], Info.class);
                String id = info.getReviewerID();
                int []help = info.getHelpful();
                long num = (long)help[0]+help[1];
                word.set(id);
                context.write(word, new LongWritable(num));
               }
            }
            catch(Exception e){e.printStackTrace();}

            }
}
```

- Reduce 과정은 각 reviewerID 로 이루어 진다. ID 당 여러 helpful value 값이 있는데 이를 for 를 이용하여 sum 에 모두 더해준 후, 이를 (reviewerID, sum)

형식으로 결과를 도출해낸다. 여기서 key 는 각기 다른 reviewerID 임으로 각 id 마다 결과가 나오게 된다.

```
public static class Reduce4 extends Reducer<Text, LongWritable, Text, LongWritable> {
    @Override
    public void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {
        long sum = 0;
        for (LongWritable val : values) {
            sum += val.get();
        }

        context.write(key, new LongWritable(sum));
    }
}
```

- 두 번째 map 은 첫번째 reduce 작업을 통해 얻은 결과값을 input 값으로 받게된다. 이는 (reviwerID,SUM) 형식이다. 이를 parsing 각 key-value 마다 parsing 을 해주어야 한다. Value 를 String 으로 바꾸어주고(toString) 이를 문장을 기준으로하여 split 해주어 String 배열을 만들어 준다. 이 배열에 대하여, 그 개수만큼 for 가 동작하게 된다. 그리고 각각의 문장에 관하여 공백을 기준으로 다시한번 split 을 하게 된다면, reviwerID 와 SUM 을 각각 구할 수 있다. 이를 외부에서 선언한 MapWritable minfo 에 루프가 한번씩 돌때마다 put 해준다. 그렇게 되면 하나의 블럭에 들어있는 모든 데이터가 minfo 에 들어가게 된다. Map 작업이 모두 끝나면 cleanup 메서드가 동작하는데 여기서, (Text("result"),MapWritable(mapInfo 이다)) 형식으로 써준다. 모든 작업들은 하나의 키를 가지게 되고, 다른 value(MapWritable)을 가지게 된다.

```
public static class Map4_2 extends Mapper<LongWritable, Text, Text, MapWritable> {
    private final static LongWritable ONE = new LongWritable(1);
    private Text word = new Text("result");
    private Text word2 = new Text();
    MapWritable minfo = new MapWritable();
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            long num =0;

            String line = value.toString();
            String[] tuple = line.split("₩₩n+");

            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {
                    String []data = tuple[i].split("₩₩s+");
                    minfo.put(new Text(data[0]),new LongWritable(Long.parseLong(data[1])));
                }
            }

            catch(Exception e){e.printStackTrace();}
        }
    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(word,minfo);
        }
}
```

- 두번째 reduce 작업이다. 앞의 map 에서 모두 같은 키를("result")를 가지지만, 각기 다른 value 를 가진 인풋들이 하나의 reduce 로 모이게 된다. 이때 동작 방식은 2 번문제의 2 번째 reduce 와 매우 유사하다. Iterable 한 values 를 for 를 이용하여 하나씩 접근 할 수 있게 된다. 접근 후에 keyset 메서드를 이용하여 keyset 을 얻고, 이때 의 key 는 id 를 의미한다. 그 후 id 에 따른 sum 값을 val.get(id)를 통하여 얻어 이를 temp 에 할당해준다. Max 가 temp 보다 작다면 max 값을 temp 로 변경해주고, word 에 id 를 set 해 준다. 이는 전체데이터에 대하여 조건을 만족할 때마다 이루어지게 된다. reduce 작업이 모두 끝난다면, cleanup 메서드가 호출된다. Cleanup method 가 호출되어 max 에 해당하는 id 와 그때의 max 를 write 해주어 결과를 적는다.

```
public static class Reduce4_2 extends Reducer<Text, MapWritable, Text, LongWritable> {
    private long max =0;
    long temp =0;
    private Text txt = new Text();
    @Override
    public void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {

            for(MapWritable val : values)
            {
                Set<Writable> keyset = val.keySet();
                for(Writable id : keyset){
                    temp = ((LongWritable)val.get(id)).get();
                    if(max<temp){
                        max = temp;
                        txt.set(id.toString());
                    }
                }
            }
        }
    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(txt, new LongWritable(max));
        }
}
```

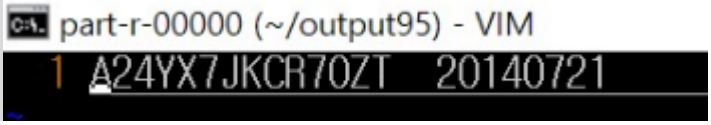
- Driver 부분은 2 번의 driver 와 작동방식과 구성이 동일하다. 다만 다른 점이 있다면, 첫번째 job 에서 mapper, reducer class 로 Map4.class, Reduce4.class 로 set 을 해준 부분과 outputValue 로 LongWritable.class 로 set 해주었다는 것이다. 그리고 두 번째 job 도 역시 마찬가지로 첫번째 reduce 의 결과가 두번째 map 의 input 으로 들어가게 된다. 그리고 mapper, reducer class 로 Map4_2.class, Reduce4_2.class 로 set 을 해준 부분과 outputValue 로 LongWritable.class 그리고 mapOutputValue 로 MapWritable.class 로 설정해준 부분의 차이만이 존재한다. 따라서 set 해준 타입의 차이만 존재하지 본질적으로 2 번 driver 와 동일하다.

```
@Override
public int run(String[] args) throws Exception {
    System.out.println(Arrays.toString(args));

    Configuration conf1=new Configuration();
    Job job = Job.getInstance(conf1);
    job.setJarByClass(test3.class);
    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(LongWritable.class);
    job.setMapperClass(Map4.class);
    job.setReducerClass(Reduce4.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);


    Path outputPath=new Path("FirstMapper");
     FileInputFormat.addInputPath(job,new Path(args[0]));
     FileOutputFormat.setOutputPath(job,outputPath);
    outputPath.getFileSystem(conf1).delete(outputPath);
    job.waitForCompletion(true);
    Configuration conf2=new Configuration();
    Job j2=Job.getInstance(conf2);
    j2.setJarByClass(test3.class);
    j2.setMapperClass(Map4_2.class);
    j2.setReducerClass(Reduce4_2.class);
    j2.setOutputKeyClass(Text.class);
    j2.setOutputValueClass(LongWritable.class);
    j2.setMapOutputValueClass(MapWritable.class);
    Path outputPath1=new Path(args[1]);
    FileInputFormat.addInputPath(j2, outputPath);
    FileOutputFormat.setOutputPath(j2, outputPath1);
    outputPath1.getFileSystem(conf2).delete(outputPath1, true);
    j2.waitForCompletion(true);

    return 0;
}
```

- 4 번과제에서 map-reduce 의 전개는 다음과 같다. 연속적인 map/reduce 를 통하여, 한번으로 처리하기 어려운 일을 해주게 된다. 첫번째 map/reduce 에서 사용자 아이디와 값을 집계 해주고, reduce 하여 중복된 사용자 아이디에 관하여 하나의 값(동일한 아이디 일 때의 값을 모두 더한 것)으로 결과물을 만들어 준다. 이를 두번째 map 의 input 으로 넣어주고, 이때 map 작업이 이루어 질때마다 하나의 MapWritable 객체에 put 해주고 모든 map 이 끝날 때 하나이 key("result")에 value 로 MapWritable 을 써주게 된다. 두번째 reduce 에서 키 값이 모두 같기 때문에 모든 데이터를 비교할 수 있게 되고, 모든 비교작업이 끝난 후 비교를 통해 얻은 key 값과 value 를 cleanup 에서 적어주고, 결국에는 결과 값으로 보여주게 된다.

가장 최근에 작성된 review 는 언제 이고 그때의 ID 는 무엇인가?

- 이 문제에서 핵심은 날짜 data 를 가공하기 위해 적절한 방법으로 parsing 해주는 것이다. 결과의 ID 와 REVIEW TIME 은 다음과 같다.



- Map 의 전반적인 과정은 앞에서 설명한 방법과 루틴이 거의 동일하다. 다른 점이 있다면, review time 에 대한 처리이다. Review time 은 String 형식이기에 이를 적절한 형태로 변환시켜야 한다. Change 함수를 이용하여 기존의 01 18, 2014 같은 데이터들을 20140118 과 같은 형태로 변환 해준다. 이때 String 이 아닌 long 이기에 이를 이용하여 max date 를 구 할 수 있게 된다. 구한 id 와 changeData 를 minfo(MapWritable)에 put 해준다. 모든 작업이 끝나면 마지막에 cleanup 메서드에서 (text("result"),minfo)형식으로 써서 reduce 에 보내주게 된다. 하나의 reduce 를 사용하기 위에서 output key 를 동일하게 해주었다.

```
public static class Map5 extends Mapper<LongWritable, Text, Text, MapWritable> {
    private Text word = new Text();
    MapWritable minfo = new MapWritable();
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String line = value.toString();
            String[] tuple = line.split("\n");
            Gson gson = new Gson();
            try{
                for(int i=0; i<tuple.length; i++)
                {

                Info info = gson.fromJson(tuple[i], Info.class);
                String id = info.getReviewerID();
                String date = info.getReviewTime();
                String changeDate = change(date);
                word.set(id);
                minfo.put(word, new LongWritable(Long.parseLong(changeDate)));
                }
            }
            catch(Exception e){e.printStackTrace();}

            }

    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(new Text("result"), minfo);
        }
}
```

```
public static String addLeadingZero (String a) {
    if(a.length() == 1) {
        a = "0" + a;
    }
    return a;
}
    public static String change(String a) {
        String[] splitStr = a.split(", ");
        String[] splitStr2 = splitStr[0].split(" ");
        String str = splitStr[1] + addLeadingZero(splitStr2[0]) + addLeadingZero(splitStr2[1]);
        return str;
    }
```

- Reduce 는 2 번 과제 2 번째 reduce 와 하는 역할이 동일하다. Iterable 한 mapwritable 에서 key 값을 뽑고 그 키에 따른 value 를 얻어 조건을 통하여 max 를 구해준다. 그리고 reduce 작업이 끝나면 cleanup 메서드를 호출해주어 그때의 id 와 날짜를 적어주게 되면 , 가장 최신에 작성된 리뷰 date 와 id 를 얻게된다.(reduce4 class 를 이용한 것이다.)

```
public static class Reduce4 extends Reducer<Text, MapWritable, Text, LongWritable> {
    private long max =0;
    long temp =0;
    private Text txt = new Text();
    @Override
    public void reduce(Text key, Iterable<MapWritable> values, Context context)
            throws IOException, InterruptedException {

            for(MapWritable val : values)
            {
                Set<Writable> keyset = val.keySet();
                for(Writable id : keyset){
                    temp = ((LongWritable)val.get(id)).get();
                    if(max<temp){
                        max = temp;
                        txt.set(id.toString());
                    }
                }
            }
        }
    @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            context.write(txt, new LongWritable(max));
        }
}
```

- 이 때의 driver class 는 앞에서 1 번 문제에서 설명한 dirver class 와 동일하다. Single map-reduce 로 작업이 발생한다. 1 번과 다른 점이 있다면, mapper class, reducer class, ouput key-value class. Mapoutvalue class 정도만 형식에 맞게 설정해주었다.

```
public class test3 extends Configured implements Tool {
    public static void main(String[] args) throws Exception {

        System.out.println(Arrays.toString(args));
        int res = ToolRunner.run(new Configuration(), new test3(), args);

        System.exit(res);
    }

    @Override
    public int run(String[] args) throws Exception {
        System.out.println(Arrays.toString(args));

        Configuration conf1=new Configuration();
        Job job = Job.getInstance(conf1);
        job.setJarByClass(test3.class);
        job.setOutputKeyClass(Text.class);
    //    job.setOutputValueClass(DoubleWritable.class);

        job.setOutputValueClass(LongWritable.class);
        job.setMapOutputValueClass(MapWritable.class);
        job.setMapperClass(Map5.class);
        job.setReducerClass(Reduce4.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);


         FileInputFormat.addInputPath(job,new Path(args[0]));
         FileOutputFormat.setOutputPath(job,new Path(args[1]));
         job.waitForCompletion(true);
        return 0;
    }
}
```

- 5 번에서 쓰인 map/reduce 방식은 앞서 사용한 방법들과 크게 다르지 않다. 다만 사용이 불가능한 date 를 사용 할 수 있는 data 로 변환시켜 이를 활용한점이 다르다. 이를 map 에서 각각 수행하여, 동일한 key, 하나의 mapwritable 로 각각 처리하여, reduce 작업을 수행해주었다. 이때 map 에서 date 를 숫자형식으로 만들어 주었기 때문에, reduce 에서 max 값을 구할 수 있었고, reduce 가 수행되고, 최종적으로 reviewerID 와 reviewTime 이 적히게 된다.