TABLE OF CONTENTS

Part 3 : NYSE (Two Writables) *3*

Step 1: Implement MapperOutputWritable 3

Step 2: Implement ReducerOutputWritable. 4

Step 3: Implement Mapper 4

Step 4: Implement Reducer 5

Step 5: Driver class 5

Step 6: Run and get Results 6

Part 3_1 : NYSE (One Writable) *6*

Step 1: Implement OutputWritable 6

Step 2: Implement Mapper 7

Step 3: Driver Class 8

Part 4 : NYSE *9*

Step 1: Implement Mapper 9

Step 2: Implement Reducer 10

Step 3: Driver Class 10

Step 4: Run and get Results 11

Comparation Result 11

Part 5 : Average *12*

Step 1: Implement AverageWritable. 12

Step 2: Implement Mapper 12

Step 3: Implement Reducer 13

Step 4: Driver Class 13

Step 5: Run and get Results 14

Part 6. (Two Jobs) *14*

Step 1: Implement CountMapper 15

Step 2: Implement CountReducer 15

Step 3: Implement CompositeKeyWritable 15

Step 4: Implement GroupComparator 16

Step 5: Implement SecondarySortComparator 16

Step 6: Implement NaturalPartitioner 16

Step 7: Implement TopMapper 17

Step 8: Implement TopReducer 17

Step 9: Driver class 17

Step 10: Run and get Result 18

Part 6: (chaining) 18

Step 1: Implement CountMapper 18

Step 2: Implement CountReducer 19

Step 3: Implement TopMapper 19

Step 4: Driver Class 19

Step 5: Run and Result 20

PART 3 : NYSE (TWO WRITABLES)

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.

STEP 1: IMPLEMENT MAPPEROUTPUTWRITABLE

Which is an implementation of Writable and WritableComparable< MapperOutputWritable >.
MapperOutputWritable will be used as a value, so we don't need to implement WritableComparable.
It has three attributes: date, stock_volume, and stock_price_adj_close. All the value of these three attributes will get from the input file

```java
public class MapperOutputWritable implements Writable, WritableComparable<MapperOutputWritable>{

    private String date;
    private int stock_volume;
    private double stock_price_adj_close;

    public MapperOutputWritable(){
        super();
    }

    public MapperOutputWritable(String date, int stock_volume, double stock_price_adj_close) {
        super();
        this.date = date;
        this.stock_volume = stock_volume;
        this.stock_price_adj_close = stock_price_adj_close;
    }

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public int getStock_volume() {
        return stock_volume;
    }

    public void setStock_volume(int stock_volume) {
        this.stock_volume = stock_volume;
    }

    public double getStock_price_adj_close() {
        return stock_price_adj_close;
    }

    public void setStock_price_adj_close(double stock_price_adj_close) {
        this.stock_price_adj_close = stock_price_adj_close;
    }
```

Implement 3 unimplemented methods:

readField() and write() function is implemented by the sequence of date, stock_volume, stock_price_adj_colse.
compareTo() method is based on the stock price firstly and then sort according to the stock_volume.

```java
public void readFields(DataInput input) throws IOException {
    // TODO Auto-generated method stub
    date = input.readUTF();
    stock_volume = input.readInt();
    stock_price_adj_close = input.readDouble();
}

public void write(DataOutput output) throws IOException {
    // TODO Auto-generated method stub
    output.writeUTF(date);
    output.writeInt(stock_volume);
    output.writeDouble(stock_price_adj_close);
}

public int compareTo(MapperOutputWritable that) {
    // TODO Auto-generated method stub

    double result = this.getStock_price_adj_close() - that.getStock_price_adj_close();
    if(result == 0.0){
        result = this.getStock_volume() - that.getStock_volume();
    }
    return (result < 0.0 ? -1 : (result == 0.0 ? 0 : 1));
}
```

STEP 2: IMPLEMENT REDUCEROUTPUTWRITABLE.

That's because of the different output of mapper and reducer, I implement another Writable to write the necessary fields to an output file. maxDate, minDate, and maxPrice are required fields in homework description. (Same as MapperOutputWritable, no need to implements WritableComparable)

```
public class ReducerOutputWritable implements Writable, WritableComparable<ReducerOutputWritable>{
    private String maxDate;
    private String minDate;
    private double maxPrice;

    public ReducerOutputWritable() {
        super();
    }

    public ReducerOutputWritable(String maxDate, String minDate, double maxPrice) {
        super();
        this.maxDate = maxDate;
        this.minDate = minDate;
        this.maxPrice = maxPrice;
    }

    public String getMaxDate() {
        return maxDate;
    }

    public void setMaxDate(String maxDate) {
        this.maxDate = maxDate;
    }

    public String getMinDate() {
        return minDate;
    }

    public void setMinDate(String minDate) {
        this.minDate = minDate;
    }

    public double getMaxPrice() {
        return maxPrice;
    }

    public void setMaxPrice(double maxPrice) {
        this.maxPrice = maxPrice;
    }
```

Write unimplemented methods: readFields(), write(), and compareTo(). A sort is based on the order of maxDate firstly and then based on minDate. Besides, toString() is important. It will be the final output format, so I must override toString() to make this class readable. The output format is : max date is : [maxDate], min date is : [minDate], max price is : [maxPrice].

```
public void readFields(DataInput input) throws IOException {
    // TODO Auto-generated method stub
    maxDate = input.readUTF();
    minDate = input.readUTF();
    maxPrice = input.readDouble();
}

public void write(DataOutput output) throws IOException {
    // TODO Auto-generated method stub
    output.writeUTF(maxDate);
    output.writeUTF(minDate);
    output.writeDouble(maxPrice);
}

public int compareTo(ReducerOutputWritable that) {
    // TODO Auto-generated method stub
    int result = this.maxDate.compareTo(that.maxDate);
    if(result == 0) {
        result = this.minDate.compareTo(that.minDate);
    }
    return (result < 0 ? -1 : (result == 0 ? 0 : 1));
}

public String toString() {
    String result = "max date is : " + this.getMaxDate() + ", min date is : " + this.getMinDate() + ", max price is : " + this.getMaxP
    return result;
}
```

STEP 3: IMPLEMENT MAPPER

I used TextInputFormat, the input key is LongWritable, and the input value is Text. In mapper, I split each line by ",”; then, get the symbol, date, stock_volume, and stock_price_adj_close; finally save each field in a MapperOutputWritable instance and write to context.

```java
public class TripleMapper extends Mapper<LongWritable, Text, Text, MapperOutputWritable>{

    @Override
    protected void map(LongWritable key, Text value,
            Mapper<LongWritable, Text, Text, MapperOutputWritable>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        if(key.get() == 0)
            return;
        String[] tokens = value.toString().split(",");
        String symbol = tokens[1];
        String date = tokens[2];
        int stock_volume = Integer.parseInt(tokens[7]);
        double stock_price_adj_close = Double.parseDouble(tokens[8]);
        MapperOutputWritable m = new MapperOutputWritable(date, stock_volume,stock_price_adj_close);
        context.write(new Text(symbol), m);
    }
}
```

STEP 4: IMPLEMENT REDUCER

In this step, I found the max_date, min_date, and max_price for each symbol. It's an easy comparison process. Finally, I write all of these three fields to a ReducerOutputWritable instance and write to context. The result will output according to its toString() method.

```java
public class TripleReducer extends Reducer<Text, MapperOutputWritable, Text, ReducerOutputWritable>{

    @Override
    protected void reduce(Text key, Iterable<MapperOutputWritable> values,
            Reducer<Text, MapperOutputWritable, Text, ReducerOutputWritable>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        int max_stock_volume = 0;
        int min_stock_volume = Integer.MAX_VALUE;
        String max_volume_date = "";
        String min_volume_date = "";
        double max_price = 0.0;
        for(MapperOutputWritable mow : values) {
//          String date = mow.getDate();
            int stock_volume = mow.getStock_volume();
            double price = mow.getStock_price_adj_close();
            if(stock_volume > max_stock_volume) {
                max_stock_volume = stock_volume;
                max_volume_date = mow.getDate();
            }

            if(stock_volume < min_stock_volume) {
                min_stock_volume = stock_volume;
                min_volume_date = mow.getDate();
            }

            max_price = Math.max(max_price, price);
        }

        ReducerOutputWritable row = new ReducerOutputWritable(max_volume_date, min_volume_date, max_price);
        context.write(key, row);
    }
}
```

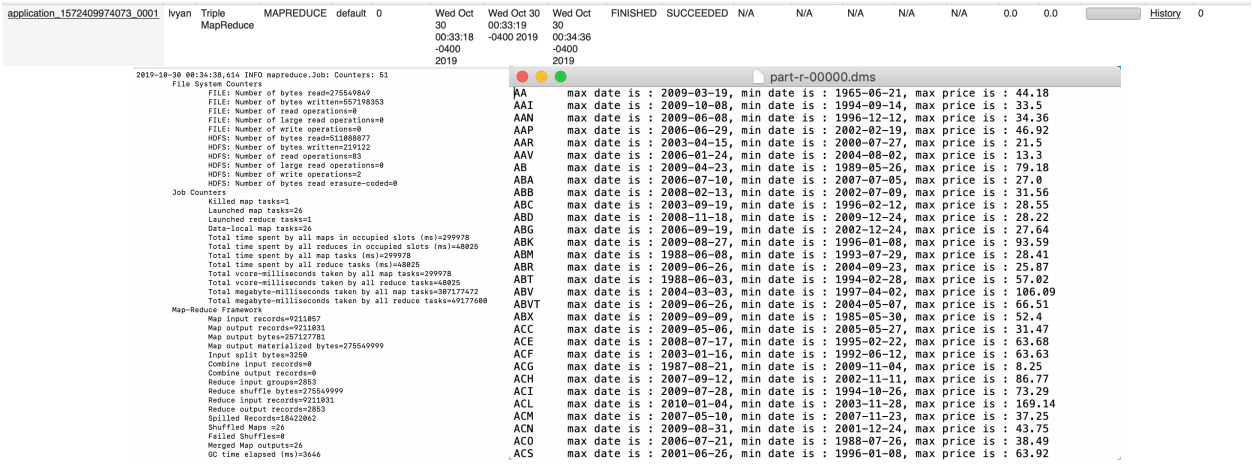
STEP 5: DRIVER CLASS

Different from previous drivers, I set the MapOutputValueClass as MapperOutputWritable.class and set the OutputValueClass as ReduceOutputWritable.class.

```java
public class App {
    public static void main( String[] args ) throws IOException, ClassNotFoundException, InterruptedException{
//        System.out.println( "Hello World!" );
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Triple MapReduce");

        //set driver class
        job.setJarByClass(App.class);

        //set mapper output
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(MapperOutputWritable.class);


        //set input output
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        //output key and value
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(ReducerOutputWritable.class);

        //set map reduce
        job.setMapperClass(TripleMapper.class);
        job.setReducerClass(TripleReducer.class);

        Path input = new Path(args[0]);
        Path output = new Path(args[1]);
//        Path input = new Path("/HW4/Part2/NYSE/NYSE_daily_prices_A.csv");
//        Path output = new Path("/HW4/Part2/Result");

        FileInputFormat.addInputPath(job, input);
        FileOutputFormat.setOutputPath(job, output);

        FileSystem hdfs = FileSystem.get(conf);
        if(hdfs.exists(output)){
            hdfs.delete(output, true);
        }

        job.waitForCompletion(true);
    }
}
```

STEP 6: RUN AND GET RESULTS

Input path is : /HW4/Part2/NYSE
output path is : /HW4/Part2/Result
command is : hadoop jar /Users/lvyan/Desktop/HW4.jar HW4.Part3.App /HW4/Part2/NYSE /HW4/Part2/Result
Running time: 1’44’’



application_1572409974073_0001 | lvyan | Triple MapReduce | MAPREDUCE | default | 0 | Wed Oct 30 00:33:18 -0400 2019 | Wed Oct 30 00:33:19 -0400 2019 | Wed Oct 30 00:34:36 -0400 2019 | FINISHED | SUCCEEDED | N/A | N/A | N/A | N/A | N/A | 0.0 | 0.0 | | History | 0

```
2019-10-30 00:34:38,614 INFO mapreduce.Job: Counters: 51
        File System Counters
                FILE: Number of bytes read=275549849
                FILE: Number of bytes written=557198353
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=511088877
                HDFS: Number of bytes written=219122
                HDFS: Number of read operations=83
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Killed map tasks=1
                Launched map tasks=26
                Launched reduce tasks=1
                Data-local map tasks=26
                Total time spent by all maps in occupied slots (ms)=299978
                Total time spent by all reduces in occupied slots (ms)=48025
                Total time spent by all map tasks (ms)=299978
                Total time spent by all reduce tasks (ms)=48025
                Total vcore-milliseconds taken by all map tasks=299978
                Total vcore-milliseconds taken by all reduce tasks=48025
                Total megabyte-milliseconds taken by all map tasks=307177472
                Total megabyte-milliseconds taken by all reduce tasks=49177600
        Map-Reduce Framework
                Map input records=9211057
                Map output records=9211031
                Map output bytes=257127781
                Map output materialized bytes=275549999
                Input split bytes=3250
                Combine input records=0
                Combine output records=0
                Reduce input groups=2853
                Reduce shuffle bytes=275549999
                Reduce input records=9211031
                Reduce output records=2853
                Spilled Records=18422062
                Shuffled Maps =26
                Failed Shuffles=0
                Merged Map outputs=26
                GC time elapsed (ms)=3646
```

part-r-00000.dms

```
AA      max date is : 2009-03-19, min date is : 1965-06-21, max price is : 44.18
AAI     max date is : 2009-10-08, min date is : 1994-09-14, max price is : 33.5
AAN     max date is : 2009-06-08, min date is : 1996-12-12, max price is : 34.36
AAP     max date is : 2006-06-29, min date is : 2002-02-19, max price is : 46.92
AAR     max date is : 2003-04-15, min date is : 2000-07-27, max price is : 21.5
AAV     max date is : 2006-01-24, min date is : 2004-08-02, max price is : 13.3
AB      max date is : 2009-04-23, min date is : 1989-05-26, max price is : 79.18
ABA     max date is : 2006-07-10, min date is : 2007-07-05, max price is : 27.0
ABB     max date is : 2008-02-13, min date is : 2002-07-09, max price is : 31.56
ABC     max date is : 2003-09-19, min date is : 1996-02-12, max price is : 28.55
ABD     max date is : 2008-11-18, min date is : 2009-12-24, max price is : 28.22
ABG     max date is : 2006-09-19, min date is : 2002-12-24, max price is : 27.64
ABK     max date is : 2009-08-27, min date is : 1996-01-08, max price is : 93.59
ABM     max date is : 1988-06-08, min date is : 1993-07-29, max price is : 28.41
ABR     max date is : 2009-06-26, min date is : 2004-09-23, max price is : 25.87
ABT     max date is : 1988-06-03, min date is : 1994-02-28, max price is : 57.02
ABV     max date is : 2004-03-03, min date is : 1997-04-02, max price is : 106.09
ABVT    max date is : 2009-06-26, min date is : 2004-05-07, max price is : 66.51
ABX     max date is : 2009-09-09, min date is : 1985-05-30, max price is : 52.4
ACC     max date is : 2009-05-06, min date is : 2005-05-27, max price is : 31.47
ACE     max date is : 2008-07-17, min date is : 1995-02-22, max price is : 63.68
ACF     max date is : 2003-01-16, min date is : 1992-06-12, max price is : 63.63
ACG     max date is : 1987-08-21, min date is : 2009-11-04, max price is : 8.25
ACH     max date is : 2007-09-12, min date is : 2002-11-11, max price is : 86.77
ACI     max date is : 2009-07-28, min date is : 1994-10-26, max price is : 73.29
ACL     max date is : 2010-01-04, min date is : 2003-11-28, max price is : 169.14
ACM     max date is : 2007-05-10, min date is : 2007-11-23, max price is : 37.25
ACN     max date is : 2009-08-31, min date is : 2001-12-24, max price is : 43.75
ACO     max date is : 2006-07-21, min date is : 1988-07-26, max price is : 38.49
ACS     max date is : 2001-06-26, min date is : 1996-01-08, max price is : 63.92
```

PART 3_1 : NYSE (ONE WRITABLE)

Use one Writable to implement something similar to part 3.

STEP 1: IMPLEMENT OUTPUTWRITABLE

It has five fields: maxDate, minDate, max_stock_volume, min_stock_volume, stock_price_adj_close.

```java
public class OutputWritable implements Writable, WritableComparable<OutputWritable>{

    private String maxDate;
    private String minDate;
    private int max_stock_volume;
    private int min_stock_volume;
    private double stock_price_adj_close;

    public OutputWritable(){
        super();
    }

    public OutputWritable(String maxDate, int max_stock_volume, String minDate, int min_stock_volume, double stock_price_adj_close) {
        super();
        this.maxDate = maxDate;
        this.max_stock_volume = max_stock_volume;
        this.minDate = minDate;
        this.min_stock_volume = min_stock_volume;
        this.stock_price_adj_close = stock_price_adj_close;
    }
```

Implement 3 unimplemented methods. We will use toString() to display result.

```java
    public void readFields(DataInput input) throws IOException {
        // TODO Auto-generated method stub
        maxDate = input.readUTF();
        max_stock_volume = input.readInt();
        minDate = input.readUTF();
        min_stock_volume = input.readInt();
        stock_price_adj_close = input.readDouble();
    }

    public void write(DataOutput output) throws IOException {
        // TODO Auto-generated method stub
        output.writeUTF(maxDate);
        output.writeInt(max_stock_volume);
        output.writeUTF(minDate);
        output.writeInt(min_stock_volume);
        output.writeDouble(stock_price_adj_close);
    }

    public int compareTo(OutputWritable that) {
        // TODO Auto-generated method stub

        double result = this.getStock_price_adj_close() - that.getStock_price_adj_close();
        if(result == 0.0){
            result = this.getMax_stock_volume() - that.getMax_stock_volume();
        }
        return (result < 0.0 ? -1 : (result == 0.0 ? 0 : 1));
    }

    public String toString() {
        return this.getDate() + ", " +this.getStock_volume() + ", " + this.getStock_price_adj_close();
        return "max date is: " + this.getMaxDate() + " , min date is : " + this.getMinDate() + " , max price is : " + this.getStock_price_adj_close();
    }
```

STEP 2: IMPLEMENT MAPPER

The output type is OutputWritable. Firstly, extract symbol, date, stock_volume, and stock_price_adj_close from input line. Secondly, set all the fields of OutputWritable. Finally, write {symbol, OutputWritable instance} to context.

```java
public class TripleMapper extends Mapper<LongWritable, Text, Text, OutputWritable>{

    @Override
    protected void map(LongWritable key, Text value,
            Mapper<LongWritable, Text, Text, OutputWritable>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        if(key.get() == 0)
            return;
        String[] tokens = value.toString().split(",");
        String symbol = tokens[1];
        String date = tokens[2];
        int stock_volume = Integer.parseInt(tokens[7]);
        double stock_price_adj_close = Double.parseDouble(tokens[8]);
        OutputWritable m = new OutputWritable(date, stock_volume,date, stock_volume, stock_price_adj_close);
        context.write(new Text(symbol), m);
    }
}
```

Step 3: Implement Reducer Class

Find the maxDate, maxVolume, minDate, minVolume, and maxPrice for each symbol. Then wirte OutputWrtiable to context.

```java
public class TripleReducer extends Reducer<Text, OutputWritable, Text, OutputWritable>{

    @Override
    protected void reduce(Text key, Iterable<OutputWritable> values,
            Reducer<Text, OutputWritable, Text, OutputWritable>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        int max_stock_volume = 0;
        int min_stock_volume = Integer.MAX_VALUE;
        String max_volume_date = "";
        String min_volume_date = "";
        double max_price = 0.0;
        for(OutputWritable mow : values) {
//          String date = mow.getDate();
            int stock_volume1 = mow.getMax_stock_volume();
            int stock_volume2 = mow.getMin_stock_volume();
            double price = mow.getStock_price_adj_close();
            if(stock_volume1 > max_stock_volume) {
                max_stock_volume = stock_volume1;
                max_volume_date = mow.getMaxDate();
            }

            if(stock_volume2 < min_stock_volume) {
                min_stock_volume = stock_volume2;
                min_volume_date = mow.getMinDate();
            }

            max_price = Math.max(max_price, price);
        }

        OutputWritable row = new OutputWritable(max_volume_date,
                                                max_stock_volume,
                                                min_volume_date,
                                                min_stock_volume,
                                                max_price);
        context.write(key, row);
    }
}
```

STEP 3: DRIVER CLASS

Driver class is the same as before except setting the MapOutputValueClass and the OutputValueClass as OutputWritable.class.

```java
public static void main( String[] args ) throws IOException, ClassNotFoundExcep
    System.out.println( "Hello World!" );
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Triple MapReduce By One Writable");

    //set driver class
    job.setJarByClass(App.class);

    //set mapper output
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(OutputWritable.class);


    //set input output
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    //output key and value
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(OutputWritable.class);

    //set map reduce
    job.setMapperClass(TripleMapper.class);
    job.setReducerClass(TripleReducer.class);

    Path input = new Path(args[0]);
    Path output = new Path(args[1]);
    Path input = new Path("/HW4/Part2/NYSE/NYSE_daily_prices_A.csv");
    Path output = new Path("/HW4/Part2/Result");

    FileInputFormat.addInputPath(job, input);
    FileOutputFormat.setOutputPath(job, output);

    FileSystem hdfs = FileSystem.get(conf);
    if(hdfs.exists(output)){
        hdfs.delete(output, true);
    }

    job.waitForCompletion(true);
}
```

Step 5: Run and get Result
input path: /HW4/Part2/NYSE
output path: /HW4/Part3/Result
command: hadoop jar /Users/lvyan/Desktop/HW4.jar HW4.Part3_2.App /HW4/Part2/NYSE /HW4/Part3/Result
running time: 1‘50’‘

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus
application_1572455801225_0013	lvyan	Triple MapReduce By One Writable	MAPREDUCE	default	0	Wed Oct 30 20:25:20 -0400 2019	Wed Oct 30 20:25:20 -0400 2019	Wed Oct 30 20:27:03 -0400 2019	FINISHED	SUCCEEDED

```
File System Counters
        FILE: Number of bytes read=422926351
        FILE: Number of bytes written=851951783
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=511088877
        HDFS: Number of bytes written=221975
        HDFS: Number of read operations=83
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
        HDFS: Number of bytes read erasure-coded=0
Job Counters
        Killed map tasks=1
        Launched map tasks=26
        Launched reduce tasks=1
        Data-local map tasks=26
        Total time spent by all maps in occupied slots (ms)=423911
        Total time spent by all reduces in occupied slots (ms)=49224
        Total time spent by all map tasks (ms)=423911
        Total time spent by all reduce tasks (ms)=49224
        Total vcore-milliseconds taken by all map tasks=423911
        Total vcore-milliseconds taken by all reduce tasks=49224
        Total megabyte-milliseconds taken by all map tasks=434084864
        Total megabyte-milliseconds taken by all reduce tasks=50405376
Map-Reduce Framework
        Map input records=9211057
        Map output records=9211031
        Map output bytes=404504277
        Map output materialized bytes=422926495
        Input split bytes=3250
        Combine input records=0
        Combine output records=0
        Reduce input groups=2853
        Reduce shuffle bytes=422926495
        Reduce input records=9211031
        Reduce output records=2853
        Spilled Records=18422062
        Shuffled Maps =26
        Failed Shuffles=0
        Merged Map outputs=26
        GC time elapsed (ms)=4473
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=8332509184
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
```

part-r-00000.dms

```
AA      max date is: 2009-03-19 , min date is : 1965-06-21 , max price is : 44.18
AAI     max date is: 2009-10-08 , min date is : 1994-09-14 , max price is : 33.5
AAN     max date is: 2009-06-08 , min date is : 1996-12-12 , max price is : 34.36
AAP     max date is: 2006-06-29 , min date is : 2002-02-19 , max price is : 46.92
AAR     max date is: 2003-04-15 , min date is : 2000-07-27 , max price is : 21.5
AAV     max date is: 2006-01-24 , min date is : 2004-08-02 , max price is : 13.3
AB      max date is: 2009-04-23 , min date is : 1989-05-26 , max price is : 79.18
ABA     max date is: 2006-07-10 , min date is : 2007-07-05 , max price is : 27.0
ABB     max date is: 2008-02-13 , min date is : 2002-07-09 , max price is : 31.56
ABC     max date is: 2003-09-19 , min date is : 1996-02-12 , max price is : 28.55
ABD     max date is: 2008-11-18 , min date is : 2009-12-24 , max price is : 28.22
ABG     max date is: 2006-09-19 , min date is : 2002-12-24 , max price is : 27.64
ABK     max date is: 2009-08-27 , min date is : 1996-01-08 , max price is : 93.59
ABM     max date is: 1988-06-08 , min date is : 1993-07-29 , max price is : 28.41
ABR     max date is: 2009-06-26 , min date is : 2004-09-23 , max price is : 25.87
ABT     max date is: 1988-06-03 , min date is : 1994-02-28 , max price is : 57.02
ABV     max date is: 2004-03-03 , min date is : 1997-04-02 , max price is : 106.09
ABVT    max date is: 2009-06-26 , min date is : 2004-05-07 , max price is : 66.51
ABX     max date is: 2009-09-09 , min date is : 1985-05-30 , max price is : 52.4
ACC     max date is: 2009-05-06 , min date is : 2005-05-27 , max price is : 31.47
ACE     max date is: 2008-07-17 , min date is : 1995-02-22 , max price is : 63.68
ACF     max date is: 2003-01-16 , min date is : 1992-06-12 , max price is : 63.63
ACG     max date is: 1987-08-21 , min date is : 2009-11-04 , max price is : 8.25
ACH     max date is: 2007-09-12 , min date is : 2002-11-11 , max price is : 86.77
ACI     max date is: 2009-07-28 , min date is : 1994-10-26 , max price is : 73.29
ACL     max date is: 2010-01-04 , min date is : 2003-11-28 , max price is : 169.14
ACM     max date is: 2007-05-10 , min date is : 2007-11-23 , max price is : 37.25
ACN     max date is: 2009-08-31 , min date is : 2001-12-24 , max price is : 43.75
ACO     max date is: 2006-07-21 , min date is : 1988-07-26 , max price is : 38.49
ACS     max date is: 2001-06-26 , min date is : 1996-01-08 , max price is : 63.92
```

PART 4 : NYSE

Redo Part3 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 3 to Part 4. You could measure the running time programmatically, or use your smartphone's timer.

STEP 1: IMPLEMENT MAPPER

To achieve these two requirements :
1) get the maxDate, minDate, and maxprice
2) use combiner
I must create a complex string to be the mapper output. Which includes [maxDate], [max stock_volume], [minDate], [min stock_volume], [stock_price_adj_colse]. In this step, maxDate and minDate have the same value, and max stock_volume and min stock_colume have the same value.

```java
public class DelimiterMapper extends Mapper<LongWritable, Text, Text, Text>{

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.map(key, value, context);
        if(key.get() == 0)
            return;
        String[] tokens = value.toString().split(",");
        String symbol = tokens[1];
        String date = tokens[2];
        int stock_volume = Integer.parseInt(tokens[7]);
        double stock_price_adj_close = Double.parseDouble(tokens[8]);
//      String output = date + "," + stock_volume + "," + stock_price_adj_close;
        //max date, max volume, min date, min volume, price
        String output = date + "," + stock_volume + "," + date + "," + stock_volume + "," + stock_price_adj_close;
        context.write(new Text(symbol), new Text(output));
    }
}
```

STEP 2: IMPLEMENT REDUCER

This reducer will be used as a combiner too. So, in this step, I must consider the situation of both combiner and reducer. I parse the input string, split each record by "," and find the maxDate, maxVolume, min Date, minVolume, and maxPrice. Finally, fill the value and write the result string to context.

The output of reducer is the same as the output of mapper. That's because this reducer will be used as combiner as well, the input format of reducer must be the same as the output format of combiner. When this reducer function as a combiner, it will receive the output of mapper, when this reducer function as a reducer, it will receive the output of combiner.

```java
public class DelimiterReducer extends Reducer<Text, Text, Text, Text>{

    @Override
    protected void reduce(Text key, Iterable<Text> values, Reducer<Text, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.reduce(arg0, arg1, arg2);

        //get the result arguments
        int max_stock_volume = 0;
        int min_stock_volume = Integer.MAX_VALUE;
        String max_volume_date = "";
        String min_volume_date = "";
        double max_price = 0.0;

        //max date, max volume, min date, min volume, price
        for(Text value : values) {
            String[] tokens = value.toString().split(",");
//          String date = tokens[0];
            int stock_volume1 = Integer.parseInt(tokens[1]);
            int stock_volume2 = Integer.parseInt(tokens[3]);
            double price = Double.parseDouble(tokens[4]);

            if(stock_volume1 > max_stock_volume) {
                max_stock_volume = stock_volume1;
                max_volume_date = tokens[0];
            }

            if(stock_volume2 < min_stock_volume) {
                min_stock_volume = stock_volume2;
                min_volume_date = tokens[2];
            }

            max_price = Math.max(max_price, price);
        }

//      String result = "max date is : " + max_volume_date + ", min date is : " + min_volume_date + ", max price is : " + max_price;
        String result = max_volume_date + "," + max_stock_volume + "," + min_volume_date + "," +min_stock_volume + "," + max_price;
        context.write(key, new Text(result));
    }

}
```

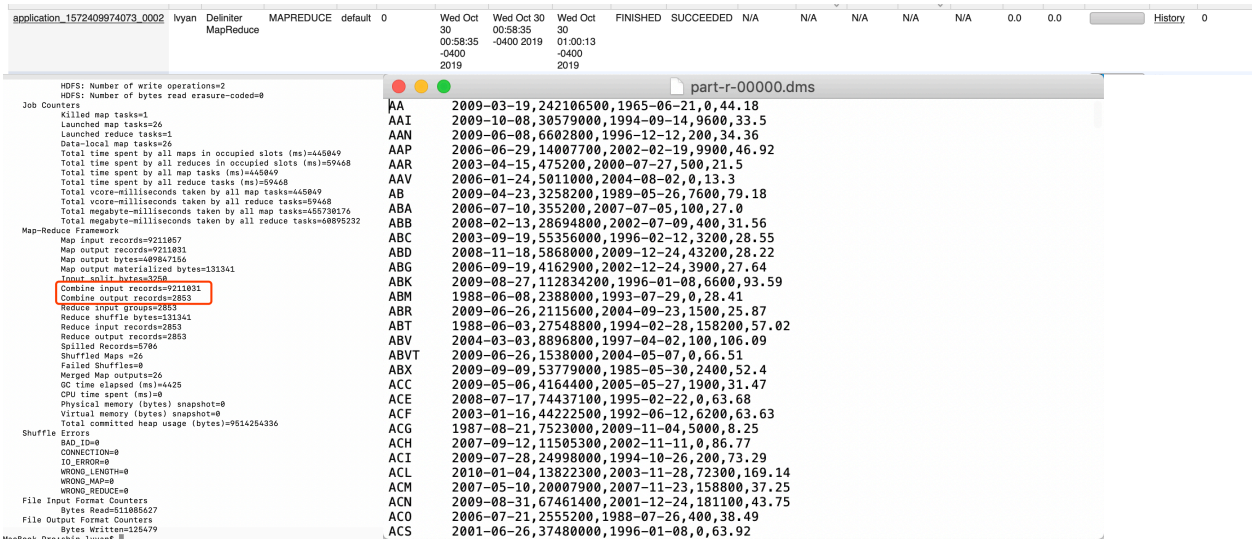
STEP 3: DRIVER CLASS

No difference with previous drives, except I set a combiner class.

```java
public class App {
    public static void main( String[] args ) throws IOException, ClassNotFoundException, InterruptedException{
//        System.out.println( "Hello World!" );
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Deliniter MapReduce");

        //set driver class
        job.setJarByClass(App.class);

        //set mapper output
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);


        //set input output
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        //output key and value
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        //set map reduce
        job.setMapperClass(DelimiterMapper.class);
        job.setReducerClass(DelimiterReducer.class);
        job.setCombinerClass(DelimiterReducer.class);

        Path input = new Path(args[0]);
        Path output = new Path(args[1]);
//      Path input = new Path("/HW4/Part2/NYSE/NYSE_daily_prices_A.csv");
//      Path output = new Path("/HW4/Part4/Result");

        FileInputFormat.addInputPath(job, input);
        FileOutputFormat.setOutputPath(job, output);

        FileSystem hdfs = FileSystem.get(conf);
        if(hdfs.exists(output)){
            hdfs.delete(output, true);
        }

        job.waitForCompletion(true);
    }
}
```

STEP 4: RUN AND GET RESULTS

Input path is : /HW4/Part2/NYSE
output path is : /HW4/Part4/Result
command is : hadoop jar /Users/lvyan/Desktop/HW4.jar HW4.Part4.App /HW4/Part2/NYSE /HW4/Part4/Result
Running time: 1’44’’



COMPARATION RESULT

There is not much difference between the run time of a job with combiner and a job without combiner. In theory, the reduce of network traffic can shorten the running time, but it is not obvious in this data set. Probably because the data set is small and only one host is used.

PART 5 : AVERAGE

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.

STEP 1: IMPLEMENT AVERAGEWRITABLE.

Which implements Writable and WritableComparable. It has two fields, average and count for calculating the overall average for each symbol.

```java
public class AverageWritable implements Writable, WritableComparable<AverageWritable>{

    //fields
    private float average;
    private float count;

    public AverageWritable() {
        super();
    }

    public AverageWritable(float average, float count) {
        super();
        this.average = average;
        this.count = count;
    }
    public float getCount() {
        return count;
    }

    public void setCount(float count) {
        this.count = count;
    }

    public float getAverage() {
        return average;
    }

    public void setAverage(float average) {
        this.average = average;
    }
```

Implement 3 unimplemented methods. ReadFields() and write() will be read or write by the sequence of average -> count. For compareTo(), compare the count firstly, and then compare the average. toString() is essential for printing the result.

```java
public void readFields(DataInput input) throws IOException {
    // TODO Auto-generated method stub
    average = input.readFloat();
    count = input.readFloat();
}

public void write(DataOutput output) throws IOException {
    // TODO Auto-generated method stub
    output.writeFloat(average);
    output.writeFloat(count);
}

public String toString() {
    return "average: " + average + ", count: " + count;
}

public int compareTo(AverageWritable that) {
    // TODO Auto-generated method stub
    double result = count - that.getCount();
    if(result == 0.0){
        result = average - that.getAverage();
    }
    return (result < 0.0 ? -1 : (result == 0.0 ? 0 : 1));
}
```

STEP 2: IMPLEMENT MAPPER

In this mapper class, the output key is year(IntWritable), and the output value is {average, count} pair(AverageWritable).

Firstly, read dateTime and stock_price_adj_close fields. Secondly, parse dateTime into date and extract year from date. Thirdly, create an AverageWritable instance and set both of the fields. Finally, write to context(Key is year, and value is an AverageWritable instance).

```java
public class AverageMapper extends Mapper<LongWritable, Text, IntWritable, AverageWritable>{

    private IntWritable year = new IntWritable();
    private final static SimpleDateFormat frmt = new SimpleDateFormat("yyyy-MM-dd", Locale.ENGLISH);
    private Calendar cal = Calendar.getInstance();
    @Override
    protected void map(LongWritable key, Text value,
            Mapper<LongWritable, Text, IntWritable, AverageWritable>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.map(key, value, context);
        //skip the first line
        if(key.get() == 0)
            return;
        //date index == 2, stock_price_adj_close  index = 8
        String[] tokens = value.toString().split(",");
        String date = tokens[2];
        try {
            Date dateType = frmt.parse(date);
            cal.setTime(dateType);
            int y = cal.get(Calendar.YEAR);
            year.set(y);
        } catch (ParseException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        float price = Float.parseFloat(tokens[8]);
        AverageWritable aw = new AverageWritable();
        aw.setAverage(price);
        aw.setCount(1);
        context.write(year, aw);
    }
```

STEP 3: IMPLEMENT REDUCER

This reducer will be used as combiner as well, so I must consider the most general situation. For sum, it should add the product of average and count rather than the value. For count, it should add the number of averages, rather than one. Finally, set the overall average and overall count in this reduce task and write to context.

```java
public class AverageReducer extends Reducer<IntWritable, AverageWritable, IntWritable, AverageWritable>{

    private AverageWritable result = new AverageWritable();
    @Override
    protected void reduce(IntWritable key, Iterable<AverageWritable> values,
            Reducer<IntWritable, AverageWritable, IntWritable, AverageWritable>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.reduce(arg0, arg1, arg2);
        float sum = 0;
        float count = 0;

        //iterate through all value for this key
        for(AverageWritable val : values) {
            sum += val.getAverage() * val.getCount();
            count += val.getCount();
        }

        result.setAverage(sum / count);
        result.setCount(count);
        context.write(key, result);
    }
}
```

STEP 4: DRIVER CLASS

This driver class is the same as before except set MapOutputKey and OutputValueClass as AverageWritable.class, and set CombinerClass as AverageReducer.

```java
public class App {
    public static void main( String[] args ) throws IOException, ClassNotFoundException, InterruptedException{
//        System.out.println( "Hello World!" );
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Average with Combiner");

        //set driver class
        job.setJarByClass(App.class);

        //set mapper output
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(AverageWritable.class);


        //set input output
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        //output key and value
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(AverageWritable.class);

        //set map reduce
        job.setMapperClass(AverageMapper.class);
        job.setReducerClass(AverageReducer.class);
        job.setCombinerClass(AverageReducer.class);

        Path input = new Path(args[0]);
        Path output = new Path(args[1]);
//      Path input = new Path("/HW4/Part2/NYSE/NYSE_daily_prices_A.csv");
//      Path input = new Path("/HW4/Part2/NYSE");
//      Path output = new Path("/HW4/Part5/Result");

        FileInputFormat.addInputPath(job, input);
        FileOutputFormat.setOutputPath(job, output);

        FileSystem hdfs = FileSystem.get(conf);
        if(hdfs.exists(output)){
            hdfs.delete(output, true);
        }

        job.waitForCompletion(true);
    }
}
```

STEP 5: RUN AND GET RESULTS

Input path is : /HW4/Part2/NYSE
output path is : /HW4/Part5/Result
command is : hadoop jar /Users/lvyan/Desktop/HW4.jar HW4.Part5.App /HW4/Part2/NYSE /HW4/Part5/Result

application_1572409974073_0004	lvyan	Average with Combiner	MAPREDUCE	default	0	Wed Oct 30 01:16:31 -0400 2019	Wed Oct 30 01:16:32 -0400 2019	Wed Oct 30 01:18:04 -0400 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0

```
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
        HDFS: Number of bytes read erasure-coded=0
    Job Counters
        Launched map tasks=26
        Launched reduce tasks=1
        Data-local map tasks=26
        Total time spent by all maps in occupied slots (ms)=424318
        Total time spent by all reduces in occupied slots (ms)=41775
        Total time spent by all map tasks (ms)=424318
        Total time spent by all reduce tasks (ms)=41775
        Total vcore-milliseconds taken by all map tasks=424318
        Total vcore-milliseconds taken by all reduce tasks=41775
        Total megabyte-milliseconds taken by all map tasks=434501632
        Total megabyte-milliseconds taken by all reduce tasks=42777600
    Map-Reduce Framework
        Map input records=9211057
        Map output records=9211031
        Map output bytes=110532372
        Map output materialized bytes=14268
        Input split bytes=3250
        Combine input records=9211031
        Combine output records=1008
        Reduce input groups=49
        Reduce shuffle bytes=14268
        Reduce input records=1008
        Reduce output records=49
        Spilled Records=2016
        Shuffled Maps =26
        Failed Shuffles=0
        Merged Map outputs=26
        GC time elapsed (ms)=4020
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=8628207616
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=511085627
    File Output Format Counters
        Bytes Written=1947
```

part-r-00000.dms

```
1962    average: 0.85143286, count: 2520.0
1963    average: 0.9509043, count: 2509.0
1964    average: 1.1544639, count: 2530.0
1965    average: 1.30946, count: 2520.0
1966    average: 1.4863434, count: 2519.0
1967    average: 1.7562312, count: 2510.0
1968    average: 1.9254028, count: 2260.0
1969    average: 1.9137762, count: 2500.0
1970    average: 4.8142977, count: 7366.0
1971    average: 6.188949, count: 7337.0
1972    average: 7.9846787, count: 7365.0
1973    average: 7.9628844, count: 7560.0
1974    average: 6.6846714, count: 7590.0
1975    average: 8.13391, count: 7590.0
1976    average: 10.459286, count: 7590.0
1977    average: 11.797204, count: 11529.0
1978    average: 12.009025, count: 12096.0
1979    average: 13.23283, count: 12144.0
1980    average: 9.588326, count: 14712.0
1981    average: 6.1370463, count: 14975.0
1982    average: 3.6106796, count: 26835.0
1983    average: 5.149617, count: 33893.0
1984    average: 4.812022, count: 44326.0
1985    average: 4.924613, count: 67063.0
1986    average: 6.2649117, count: 76588.0
1987    average: 7.210998, count: 89480.0
1988    average: 6.0039673, count: 138926.0
1989    average: 6.5829754, count: 149917.0
1990    average: 6.369107, count: 183926.0
1991    average: 7.1509476, count: 199344.0
1992    average: 7.6278534, count: 233612.0
1993    average: 8.653599, count: 254104.0
1994    average: 9.4564, count: 272928.0
1995    average: 10.286339, count: 294827.0
1996    average: 12.41703, count: 328170.0
1997    average: 15.495538, count: 344971.0
1998    average: 17.235554, count: 364971.0
1999    average: 17.467482, count: 382829.0
2000    average: 18.952648, count: 396071.0
2001    average: 19.201273, count: 416918.0
2002    average: 18.672234, count: 449501.0
2003    average: 19.323427, count: 484726.0
2004    average: 23.934397, count: 538095.0
2005    average: 26.613798, count: 576813.0
2006    average: 29.439249, count: 609136.0
2007    average: 33.30494, count: 651150.0
2008    average: 27.273092, count: 691940.0
2009    average: 22.02064, count: 700962.0
2010    average: 26.8445, count: 71287.0
```

PART 6. (TWO JOBS)

Top 10 most visiting IP Addresses (HW3-Part3). This can be achieved by 2 jobs :

A map reduce job to find how many times each item is bought.

The second job, sort items based on number of times it was bought, and get top 10 items.

The second MapReduce job swaps key and value so that we get words sorted in descending order by frequency.

Use Chaining to find the Top 10 IP addresses based on access count.

STEP 1: IMPLEMENT COUNTMAPPER

CountMapper is used for mapping the input line into {IP Address, 1}. In this progress, I extracted the IP address in each line and set the value as one. Used by the first job.

```
public class CountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    Text word = new Text();
    IntWritable one = new IntWritable(1);
    //number of times each IP accesses the website
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] tokens = line.split("\\s+");
        String ipAddress = tokens[0];
        word.set(ipAddress);
        context.write(word, one);
    }
}
```

STEP 2: IMPLEMENT COUNTREDUCER

CountReducer is used to count the frequency of each IP address. Simply add value together for each IP address. Used by the first job.

```
public class CountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
            Reducer<Text, IntWritable, Text, IntWritable>.Context context) throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        int sum = 0;
        for(IntWritable val : values) {
            sum += val.get();
        }
        IntWritable count = new IntWritable(sum);
        context.write(key, count);
    }
}
```

STEP 3: IMPLEMENT COMPOSITEKEYWRITABLE

CompositeKeyWritable will be used in the second job. CompositeKeyWritable will be used as a composite key in the second job. It has two fields, the first one is natural key, which will be set as a blank; the second one is the count, which will be set as the frequency of IP address. Natural key is used for grouping, our aim is finding the top 10 most visited IP address, which means we must group all the record together and find the top 10 records, so I used a blank as natural key to make sure all the records go to the same reducer.

```
public class CompositeKeyWritable implements Writable, WritableComparable <CompositeKeyWritable>{

    private String naturalKey;
    private int count;

    public CompositeKeyWritable() {
        super();
    }


    public CompositeKeyWritable(String naturalKey, int count) {
        super();
        this.naturalKey = naturalKey;
        this.count = count;
    }

    public String getNaturalKey() {
        return naturalKey;
    }


    public void setNaturalKey(String naturalKey) {
        this.naturalKey = naturalKey;
    }


    public int getCount() {
        return count;
    }


    public void setCount(int count) {
        this.count = count;
    }
```

Implement three unimplemented methods, and sort by count in descending order.

```
//in descending order
public int compareTo(CompositeKeyWritable that) {
    // TODO Auto-generated method stub
    int result = that.getCount() - this.getCount();
    return (result < 0 ? -1 : (result == 0 ? 0 : 1));
}

public void readFields(DataInput input) throws IOException {
    // TODO Auto-generated method stub
    naturalKey = input.readUTF();
    count = input.readInt();
}

public void write(DataOutput output) throws IOException {
    // TODO Auto-generated method stub
    output.writeUTF(naturalKey);
    output.writeInt(count);
}

public String toString() {
    String result = "count is: " + count;
    return result;
}
```

STEP 4: IMPLEMENT GROUPCOMPARATOR

GroupComparator will be used in the second job. GroupComparator will be used when group intermediate key value pair. I used natural key to group, so I used natural key to override compare() method.

```
public class GroupComparator extends WritableComparator {

    public GroupComparator() {
        super(CompositeKeyWritable.class, true);
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        CompositeKeyWritable ckw1 = (CompositeKeyWritable) a;
        CompositeKeyWritable ckw2 = (CompositeKeyWritable) b;
        int result = ckw1.getNaturalKey().compareTo(ckw2.getNaturalKey());
        return result;
    }

}
```

STEP 5: IMPLEMENT SECONDARYSORTCOMPARATOR

SecondarySortComparator will be used in the second job. SecondarySortComparator will be used for sorting records according to count. So, I override compare() method by count field in CompositeKeyWritable in descending order.

```
6  public class SecondarySortComparator extends WritableComparator{
7
8      public SecondarySortComparator() {
9          super(CompositeKeyWritable.class, true);
10     }
11
12     @Override
13     public int compare(WritableComparable a, WritableComparable b) {
14         CompositeKeyWritable c1 = (CompositeKeyWritable) a;
15         CompositeKeyWritable c2 = (CompositeKeyWritable) b;
16         int count1 = c1.getCount();
17         int count2 = c2.getCount();
18         int result = count2 - count1;
19 //      return -1 * result;
20         return (result < 0 ? -1 : (result == 0 ? 0 : 1));
21     }
22 }
```

STEP 6: IMPLEMENT NATURALPARTITIONER

NaturalPartitioner will be used in the second job. It defines how can I partition records to reducers. I used hash code here.

```
public class NaturalPartitioner extends Partitioner<CompositeKeyWritable, Text>{

    @Override
    public int getPartition(CompositeKeyWritable key, Text value, int numPartitions) {
        // TODO Auto-generated method stub

        return key.getNaturalKey().hashCode() % numPartitions;
    }
}
```

STEP 7: IMPLEMENT TOPMAPPER

TopMapper will be used in the second job. The major function is swap key and value. I used KeyValueInputFormat as InputFormatClass, so I can get the key and value directly. count is value in the input file, it will be used as a part of composite key later. I convert count to int data format and set it to CompositeKeyWritable instance. In order to make sure all the records are assigned to the same reducer, I set natural key as a blank. Finally, write this CompositeKeyWritable instance to context.

```
public class TopMapper extends Mapper<Text, Text, CompositeKeyWritable, Text>{

    @Override
    protected void map(Text key, Text value,
            Mapper<Text, Text, CompositeKeyWritable, Text>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.map(key, value, context);
//      String IPAddress = key.toString();
//      int count = value.get();
        int count = Integer.parseInt(value.toString());
        CompositeKeyWritable ckw = new CompositeKeyWritable();
        ckw.setNaturalKey(" ");
        ckw.setCount(count);
        context.write(ckw, key);
    }
}
```

STEP 8: IMPLEMENT TOPREDUCER

TopReducer will be used in the second job. Before getting into this reducer, all the records have been sort by count in descending order. So, what I need to do in this step is print the first 10 result. The output is [count] [IP address]

```
//input : ({" ", count}, ipaddress)
//output: (count, ipaddress)
public class TopReducer extends Reducer<CompositeKeyWritable, Text, IntWritable, Text >{

    @Override
    protected void reduce(CompositeKeyWritable key, Iterable<Text> values,
            Reducer<CompositeKeyWritable, Text, IntWritable, Text>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.reduce(arg0, arg1, arg2);

        for(int i = 0; i < 10; i++) {
            Text value = values.iterator().next();
            context.write(new IntWritable(key.getCount()),value);
        }
    }
}
```

STEP 9: DRIVER CLASS

For the first job, get the count result of each IP address. For the second job, get the top 10 most visited IP addresses. We must make sure the input path of the second job is correct.

```
Configuration conf = new Configuration();
Job job1 = Job.getInstance(conf, "Count Visit Times");

job1.setJarByClass(App.class);
job1.setInputFormatClass(TextInputFormat.class);
job1.setOutputFormatClass(TextOutputFormat.class);

job1.setOutputKeyClass(Text.class);
job1.setOutputValueClass(IntWritable.class);

job1.setMapperClass(CountMapper.class);
job1.setReducerClass(CountReducer.class);

  Path input = new Path("/logs/access.log");
  Path output = new Path("/HW4/Part6/CountResult");

Path input = new Path(args[0]);
Path output = new Path("/HW4/Part6/CountResult");
FileInputFormat.addInputPath(job1, input);
FileOutputFormat.setOutputPath(job1, output);

FileSystem hdfs = FileSystem.get(conf);
if(hdfs.exists(output)){
    hdfs.delete(output, true);
}

job1.waitForCompletion(true);
```

```
Job job2 = Job.getInstance(conf, "Get top 10");
job2.setJarByClass(App.class);
job2.setInputFormatClass(KeyValueTextInputFormat.class);
job2.setOutputFormatClass(TextOutputFormat.class);
job2.setMapOutputKeyClass(CompositeKeyWritable.class);
job2.setMapOutputValueClass(Text.class);
job2.setPartitionerClass(NaturalPartitioner.class);
job2.setGroupingComparatorClass(GroupComparator.class);
job2.setSortComparatorClass(SecondarySortComparator.class);
job2.setOutputKeyClass(IntWritable.class);
job2.setOutputValueClass(Text.class);
job2.setMapperClass(TopMapper.class);
job2.setReducerClass(TopReducer.class);
Path input2 = new Path("/HW4/Part6/CountResult/part-*");
Path output2 = new Path("/HW4/Part6/Result");
Path output2 = new Path(args[1]);
FileInputFormat.setInputPaths(job2, input2);
FileOutputFormat.setOutputPath(job2, output2);
```

STEP 10: RUN AND GET RESULT

input path: /logs/access.log
output path: /HW4/Part6/Result
command: hadoop jar /Users/lvyan/Desktop/HW4.jar HW4.TwoJobs.App /logs/access.log /HW4/Part6/Result

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1572455801225_0006	lvyan	Get top 10	MAPREDUCE	default	0	Wed Oct 30 13:32:37 -0400 2019	Wed Oct 30 13:32:41 -0400 2019	Wed Oct 30 13:32:56 -0400 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0
application_1572455801225_0005	lvyan	Count Visit Times	MAPREDUCE	default	0	Wed Oct 30 13:32:17 -0400 2019	Wed Oct 30 13:32:18 -0400 2019	Wed Oct 30 13:32:34 -0400 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0

top 10.dms

```
4958    155.33.18.236
3724    207.248.55.246
2812    10.15.10.129
2108    10.15.10.135
1501    129.10.65.240
1279    107.20.213.124
765     168.144.67.144
667     50.63.154.43
643     72.158.153.33
642     118.102.182.196
```

PART 6: (CHAINING)

STEP 1: IMPLEMENT COUNTMAPPER

CountMapper is used for mapping the input line into {IP Address, 1}. In this progress, I extracted the IP address in each line and set the value as one.

```java
public class CountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    Text word = new Text();
    IntWritable one = new IntWritable(1);
    //number of times each IP accesses the website
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] tokens = line.split("\\s+");
        String ipAddress = tokens[0];
        word.set(ipAddress);
        context.write(word, one);
    }
}
```

STEP 2: IMPLEMENT COUNTREDUCER

CountReducer is used to count the frequency of each IP address. Simply add value together for each IP address. Used by the first job.

```java
public class CountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
            Reducer<Text, IntWritable, Text, IntWritable>.Context context) throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        int sum = 0;
        for(IntWritable val : values) {
            sum += val.get();
        }
        IntWritable count = new IntWritable(sum);
        context.write(key, count);
    }
}
```

STEP 3: IMPLEMENT TOPMAPPER

TopMapper is used to get the top 10 most visited IP addresses. I used TreeMap and keep the size of it being 10 after each reading. After all the records are read into TreeMap (Although I just keep 10 of them), cleanup will write all of these 10 records to context one by one.

```java
public class TopMapper extends Mapper<IntWritable, Text, IntWritable, Text>{


    private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();
    @Override
    protected void map(IntWritable key, Text value, Mapper<IntWritable, Text, IntWritable, Text>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub
//      super.map(key, value, context);

        repToRecordMap.put(key.get(), value);
        if(repToRecordMap.size() > 10) {
            repToRecordMap.remove(repToRecordMap.firstKey());
        }
    }

    @Override
    protected void cleanup(Mapper<IntWritable, Text, IntWritable, Text>.Context context)
            throws IOException, InterruptedException {
        // TODO Auto-generated method stub

        while(!repToRecordMap.isEmpty()) {
            Integer k = repToRecordMap.lastKey();
            Text v = repToRecordMap.get(repToRecordMap.lastKey());
            context.write(new IntWritable(k), new Text(v));
            repToRecordMap.remove(repToRecordMap.lastKey());
        }
    }
}
```

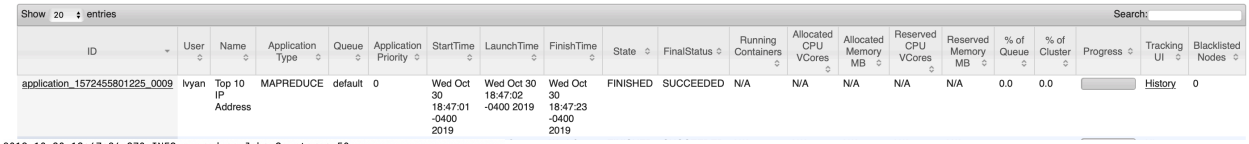
STEP 4: DRIVER CLASS

In driver class, I set one job, add one mapper to ChainMapper, add one reducer to ChainReducer, add two mappers to ChainReducer. InverseMapper is a mapper that has been defined by hadoop.
I get the frequency of each IP address -> inverse key and value -> find the top 10 most visiting IP address.

```java
public static void main( String[] args ) throws IOException, ClassNotFoundException, InterruptedException{
      System.out.println( "Hello World!" );
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Top 10 IP Address");

    //set driver class
    job.setJarByClass(App.class);

    //set input output
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    Configuration map1 = new Configuration(false);
    ChainMapper.addMapper(job,
                          CountMapper.class,
                          LongWritable.class,
                          Text.class,
                          Text.class,
                          IntWritable.class,
                          map1);

    Configuration reduce1 = new Configuration(false);
    ChainReducer.setReducer(job,
                            CountReducer.class,
                            Text.class,
                            IntWritable.class,
                            Text.class,
                            IntWritable.class,
                            reduce1);

    Configuration map2 = new Configuration(false);
    ChainReducer.addMapper(job,
                           InverseMapper.class,
                           Text.class,
                           IntWritable.class,
                           IntWritable.class,
                           Text.class,
                           map2);

    Configuration map3 = new Configuration(false);
    ChainReducer.addMapper(job,
                           TopMapper.class,
                           IntWritable.class,
                           Text.class,
                           IntWritable.class,
                           Text.class,
                           map3);
```

STEP 5: RUN AND RESULT

Input path is : /logs/access.log
output path is : /HW4/Part6/Chain/Result
command is : hadoop jar /Users/lvyan/Desktop/HW4.jar HW4.Part6.App /logs/access.log /HW4/Part6/Chain/Result



ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1572455801225_0009	lvyan	Top 10 IP Address	MAPREDUCE	default	0	Wed Oct 30 18:47:01 -0400 2019	Wed Oct 30 18:47:02 -0400 2019	Wed Oct 30 18:47:23 -0400 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0

```
2019-10-30 18:47:24,870 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=709513
                FILE: Number of bytes written=1879303
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=3497769
                HDFS: Number of bytes written=188
                HDFS: Number of read operations=8
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
                HDFS: Number of bytes read erasure-coded=0
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=3430
                Total time spent by all reduces in occupied slots (ms)=3089
                Total time spent by all map tasks (ms)=3430
                Total time spent by all reduce tasks (ms)=3089
                Total vcore-milliseconds taken by all map tasks=3430
                Total vcore-milliseconds taken by all reduce tasks=3089
                Total megabyte-milliseconds taken by all map tasks=3512320
                Total megabyte-milliseconds taken by all reduce tasks=3163136
        Map-Reduce Framework
                Map input records=35111
                Map output records=35111
                Map output bytes=639285
                Map output materialized bytes=709513
                Input split bytes=102
                Combine input records=0
                Combine output records=0
                Reduce input groups=1945
                Reduce shuffle bytes=709513
                Reduce input records=35111
                Reduce output records=10
                Spilled Records=70222
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=98
                CPU time spent (ms)=0
                Physical memory (bytes) snapshot=0
                Virtual memory (bytes) snapshot=0
                Total committed heap usage (bytes)=444071936
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
```

```
4958    155.33.18.236
3724    207.248.55.246
2812    10.15.10.129
2108    10.15.10.135
1501    129.10.65.240
1279    107.20.213.124
765     168.144.67.144
667     50.63.154.43
643     72.158.153.33
642     118.102.182.196
```