LAB: Working with HDFS and MapReduce

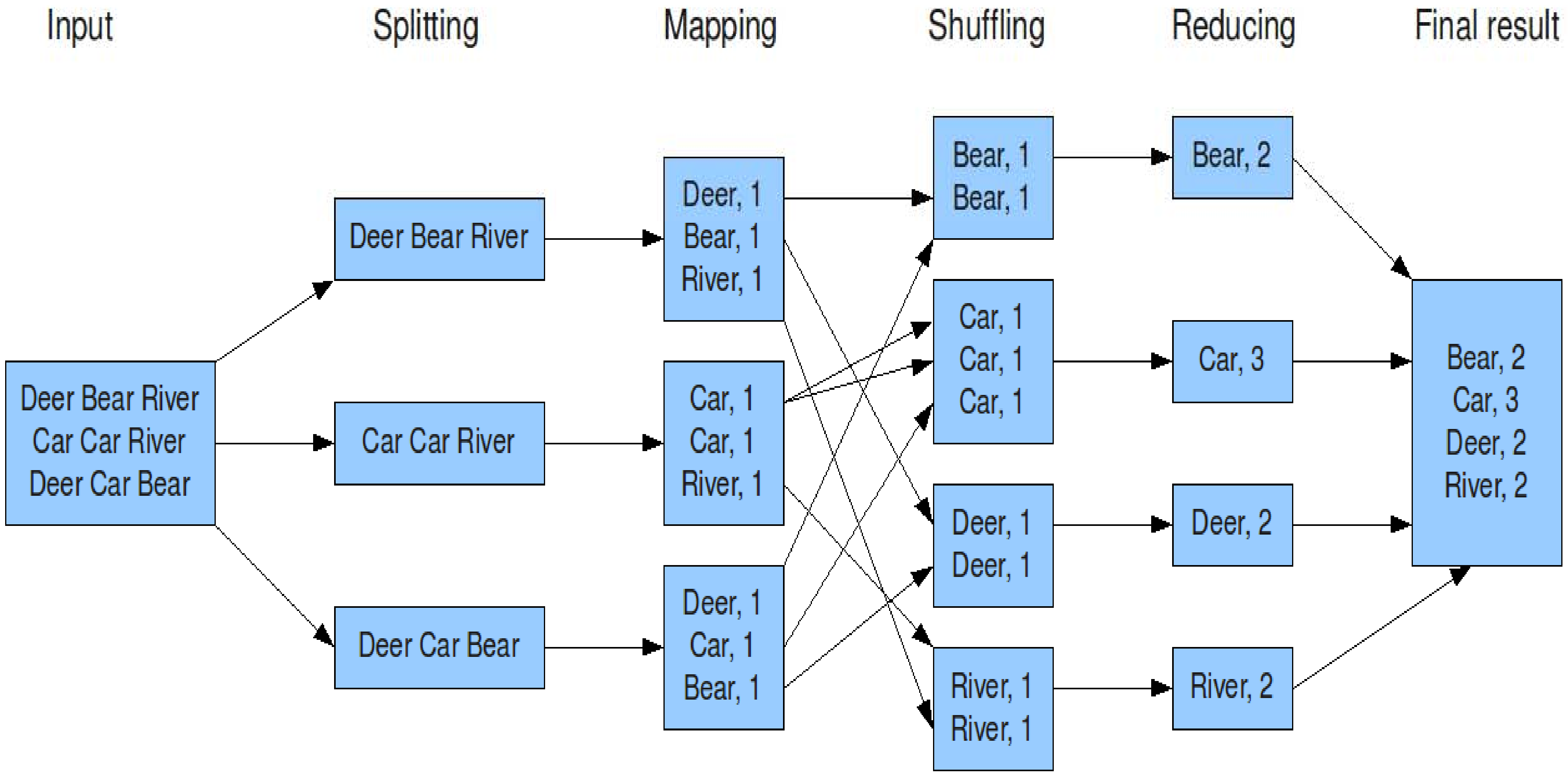
Anastasios Gounaris
Apostolos N. Papadopoulos

Outline

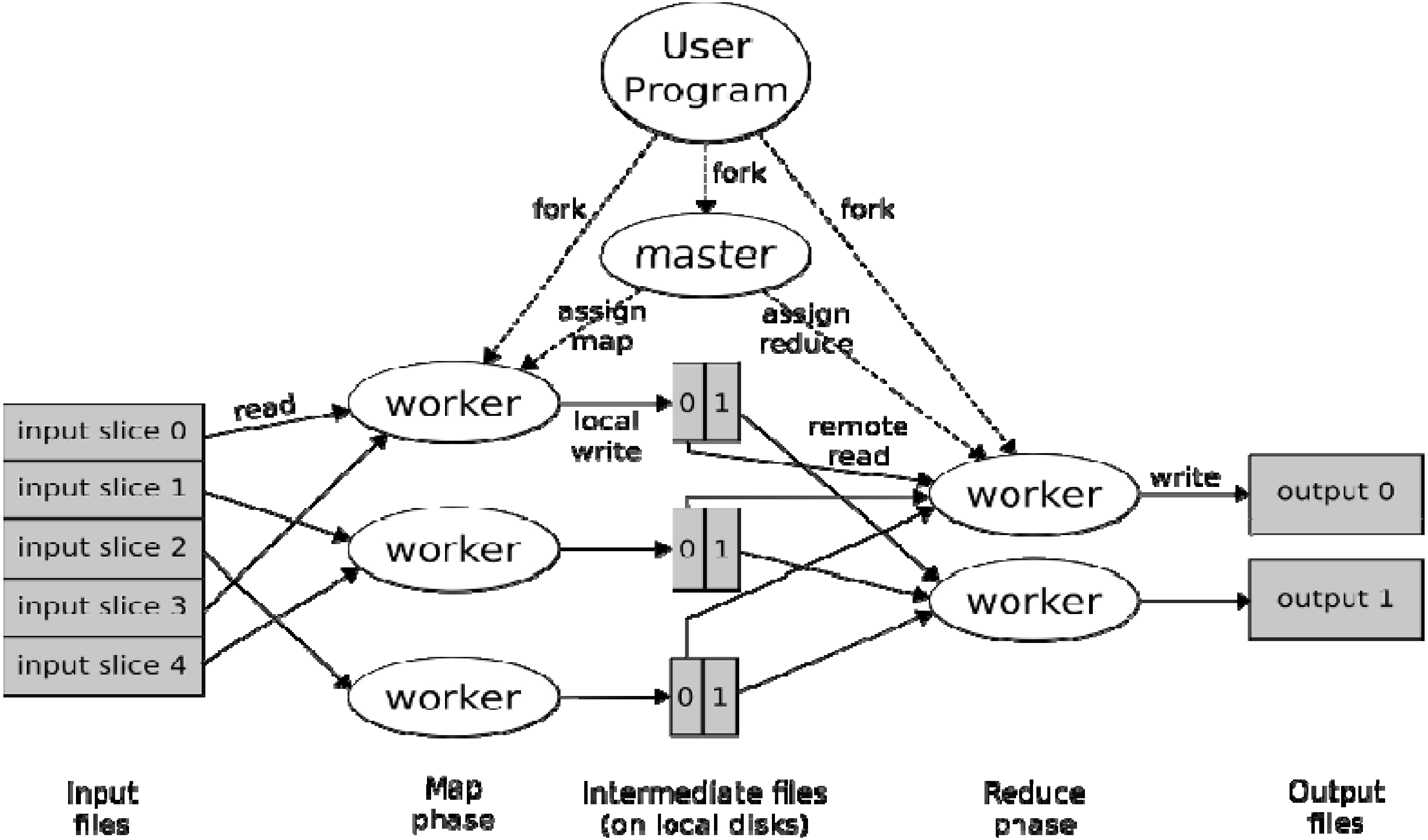
HDFS
- creating folders
- copying files
- ...

Hadoop Programming with Java
- WordCount
- MaxTemp

Reminder


Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

Reminder



HDFS

HDFS

Target

To be able to write distributed programs over a **Hadoop cluster**.

The examples are simple for illustration purposes BUT the process we will follow is the same either we have an easy or a difficult problem.

HDFS

To get a list of all available commands

```
hadoop fs -help
```

The File System (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports

HDFS

Listing files

```
hadoop fs -ls /
```

Initially the folder is empty

HDFS

Creating and deleting directories

Create:

```
hadoop fs -mkdir /input1
hadoop fs -rmdir /input1
```

Run:

```
hadoop fs -mkdir /input1
hadoop fs -mkdir /input2
```

HDFS

Putting/getting files to/from HDFS

```
hadoop fs -put fname.txt  /<hdfs_path>/input
```

```
hadoop fs -get /<hdfs_path>/fname.txt .
```

HDFS Preparation

Input data

All necessary input data files we are going to use need to be moved to hdfs:

```
hadoop fs -put leonardo.txt /input1
hadoop fs -put weather/* /input2
```

HDFS

Show file contents

```
hadoop fs -cat /input1/leonardo.txt
```

HDFS

File copy from directory1 of hdfs to directory2

```
hadoop fs -cp /directory1/leonardo.txt /directory2/leonardo.txt
```

View the file

```
hadoop fs -cat /input1/leonardo.txt
```

Delete the file

```
hadoop fs -rm /input1/leonardo.txt
```

HDFS

Delete a directory and ALL CONTENTS

```
hadoop fs -rm -r /some-directory
```

BE VERY CAREFUL WHEN YOU USE IT!

HDFS Preparation

We will create an output directory to store the output of hadoop jobs

```
hadoop fs -mkdir /output
```

Hadoop with Java

We will focus on two examples of Hadoop jobs using the Java programming language.

WordCount: given a collection of text documents, find the number of occurrences of each word in the collection.

MaxTemp: given a file containing temperature measurements, find the maximum temperature recording per year.

WordCount: the mapper

```
public static class TokenizerMapper extends Mapper<Object, Text, Text,
   IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map (Object key, Text value, Context context)
        throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set (itr.nextToken());
            context.write (word, one);
      }
    }
  }
```

WordCount: the reducer

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

WordCount: main function

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

WordCount: compiling the code

Go inside the java-wordcount folder, by executing the following command from your home folder:

```
cd <PATH>/java-wordcount
```

The relevant code is contained in the file

`WordCount.java`

WordCount: compiling the code

To compile the code run the command:

```
javac -classpath "$(yarn classpath)" WordCount.java
```

The file **`WordCount.class`** must have been produced.

WordCount: building the jar

We will create the file

`wc.jar`

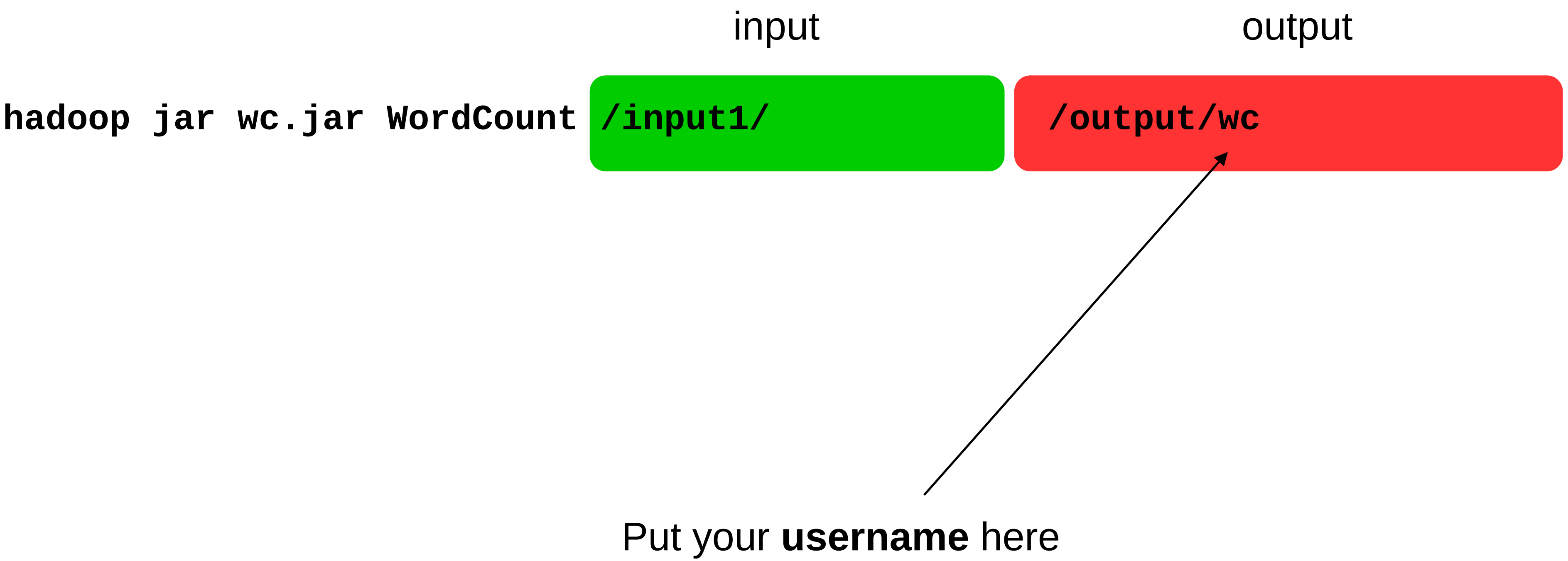
Please execute

```
jar cf wc.jar WordCount*.class
```

Everything is set! Lets run the job on the cluster.

WordCount: running the job -?

Execute the following command:



input

output

```
hadoop jar wc.jar WordCount /input1/ /output/wc
```

Put your **username** here

WordCount: exploring the results

```
hadoop fs -ls //output/wc
```

You should see something like this

```
-rw-r--r--   1 user supergroup          0 2015-10-14 18:02 /output/wc/_SUCCESS
-rw-r--r--   1 user supergroup     337639 2015-10-14 18:02 /output/wc/part-r-00000
```

WordCount: exploring the results

Examine the last lines of the output:

```
hadoop fs -tail /output/wc/part-r-00000
```

MaxTemp: the mapper

```
public class MaxTempMapper extends Mapper<LongWritable, Text, Text,
   IntWritable> {
  private static final int MISSING = 9999;

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

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus
   signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

MaxTemp: the reducer

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

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

MaxTemp: main function

```
public class MaxTemperature {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1); }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

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

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

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

    //job.setNumReduceTasks(2); // 2 reducers
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }

}
```

MaxTemp: compiling the code

Go inside the java-wordcount folder, by executing the following command from your home folder:

```
cd <PATH>/java-maxtemp
```

The relevant code is contained in the file

`MaxTemperature.java`

MaxTemp: compiling the code

o compile the code run the command:

```
javac -classpath "$(yarn classpath)" MaxTemp.java
```

The file **`MaxTemp.class`** must have been produced.

MaxTemp: building the jar

We will create the file

```
mt.jar
```

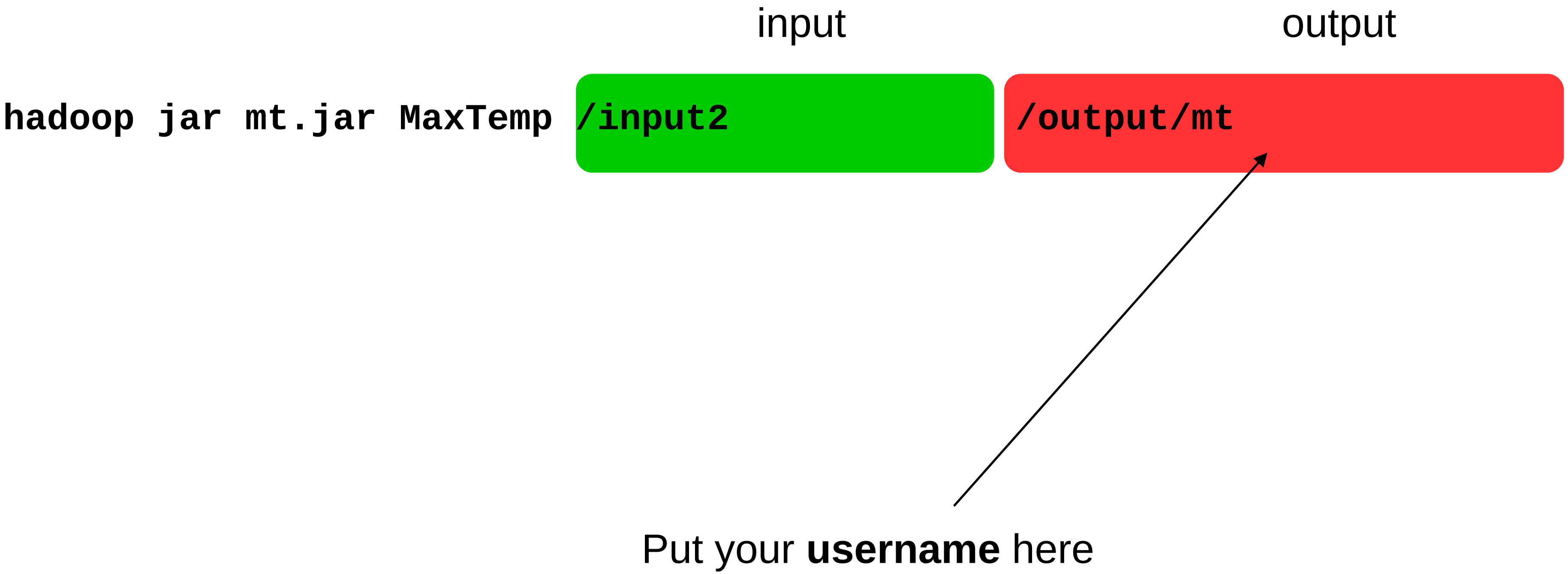
Please execute

```
jar cf mt.jar MaxTemp*.class
```

Everything is set! Lets run the job on the cluster.

MaxTemp: running the job

Execute the following command:



input

output

```
hadoop jar mt.jar MaxTemp /input2 /output/mt
```

Put your **username** here

MaxTemp: exploring the results - ?

```
hadoop fs -ls /output/mt
```

You should see something like this

```
-rw-r--r--   1 user supergroup          0 2015-10-14 18:05 /output/mt/_SUCCESS
-rw-r--r--   1 user supergroup        180 2015-10-14 18:05 /output/mt/part-r-00000
```

MaxTemp: exploring the results

Examine the last lines of the output:

```
hadoop fs -tail /output/mt/part-r-00000
```

Your turn now...

We have 2-column data from two populations, R and S, stored in text files as follows:

```
R,2,60
R,5,190
S,2,12
S,2,45
R,6,1
S,7,10
```

We want

1) **to group all the records of population S by the 1st field,**
2) **for each group to sum the values of the 2nd field**
3) **Provided that population R has a similar 1st field in one of the records**