Es: 1

Input: (unstructured) textual file

Output: number of occurrences of each word appearing at least one time in the input file

MAPPER:

```
String[] words= value.toString().split("\\s+");
for (String word : words){
   String cleanedWord= word.toLowerCase();
   context.write(new Text(cleanedWord), new IntWritable(value:1));
}
```

REDUCER

```
int occurrences = 0;

// Iterate over the set of values and sum them
//The group by ker is done by the framework
for (IntWritable value : values) {
    occurrences = occurrences + value.get();
}

context.write(key, new IntWritable(occurrences));
```

Es: 3

Input: a (structured) textual file containing the daily value of PM10 for a set of sensors. Each line of the file has the following format **sensorId,date\tPM10 value (µg/m3)\n**

Output: report for each sensor the number of days with PM10 above a specific threshold (set to 50 µg/m3). Select only the sensors that are associated at least one time with a PM10 above the threshold.

DRIVER: **job.setInputFormatClass(KeyValueTextInputFormat.class);** to memorize **sensorId,date** as key and **PM10 value (µg/m3)** as value because \t default

MAPPER

```
String[] sensorId=  key.toString().split(",");
float pm10 = Float.parseFloat(value.toString());
System.out.println(pm10);
if (pm10>50){
    context.write(new Text( sensorId[0]),new IntWritable(value:1));
}
```

Reducer equal to Es2

Es:4

Input: a (structured) textual file containing the daily value of PM10 for a set of city zones. Each line of the file has the following format **zoneId,date\tPM10 value (µg/m3)\n**

Output: report for each zone the list of dates associated with a PM10 value above a specific threshold (set to 50 µg/m3). Report only the zones with at least one date with PM10 above the threshold.

DRIVER: **job.setInputFormatClass(KeyValueTextInputFormat.class);**

MAPPER

```
String[] zoneDate = key.toString().split(",");
float pm10;
pm10 = Float.parseFloat(value.toString());
if (pm10 < 50) {
    context.write(new Text(zoneDate[0]), new Text(zoneDate[1]));
}
```

REDUCER

```
protected void reduce(
    Text key,  Iterable<Text> values,
    Context context) throws IOException, InterruptedException {
    List<String> stringList = new ArrayList<>();
    for (Text value : values) {
        stringList.add(value.toString()); }
    // Unisce la lista in una stringa separata da virgole
    String result = String.join(",", stringList);
    context.write(key, new Text(result));
```

Es:5

Input: a collection of (structured) textual csv files containing the daily value of PM10 for a set of sensors. Each line of the files has the following format **sensorId,date,PM10 value (µg/m3)\n**

Output: report for each sensor the average value of PM10

MAPPER

```
String[] values= value.toString().split(",");
context.write(new Text(values[0]), new FloatWritable (Float.parseFloat(values[2])));
}
```

REDUCER

```
int occurrences = 0;
float sum=0;
for (FloatWritable value : values) {
    occurrences = occurrences + 1;
    sum = sum + value.get();}
float avg= sum/occurrences;
context.write(key, new FloatWritable(avg));
```

Es:6

Input: a collection of (structured) textual csv files containing the daily value of PM10 for a set of sensors. Each line of the files has the following format **sensorId,date,PM10 value (µg/m3)\n**

Output: report for each sensor the maximum and the minimum value of PM10

MAPPER

```
String[] values=value.toString().split(",");
context.write(new Text(values[0]),new FloatWritable(Float.parseFloat(values[2])));
```

REDUCER

```
float min = Float.POSITIVE_INFINITY;
float max = Float.NEGATIVE_INFINITY;
for (FloatWritable value : values) {
    float pm10 = value.get();
    if (pm10 < min) {
        min = pm10;}
    if (pm10 > max) {
        max = pm10;}}
String result = max + "-" + min;
context.write(key, new Text(result));
```

Es:7

Input: a textual file containing a set of sentences. Each line of the file has the following format **sentenceId\tsentence\n**

Output: report for each word w the list of sentenceIds of the sentences containing w. Do not consider the words “and”, “or”, “not”

MAPPER

```
String[] words= value.toString().split(" ");
List<String> lista = Arrays.asList("and", "or", "not");
for (String word : words){
      String cleanedWord= word.toLowerCase();
      if (!lista.contains(cleanedWord)) {
           context.write(new Text(cleanedWord), new Text(key));
```

REDUCER

```
@Override
protected void reduce(
    Text key, // Input key type
    Iterable<Text> values, // Input value type
    Context context) throws IOException, InterruptedException {
    List<String> stringList=new ArrayList<>();
    for (Text value: values){
        stringList.add(value.toString());}
    String result=String.join(",", stringList);
    context.write(key, new Text(result));
```

Es:8

Input: a (structured) textual csv files containing the daily income of a company. Each line of the files has the following format **date\tdaily income\n**

Output: Total income for each month of the year. Average monthly income for each year considering only the months with a total income greater than 0.

DRIVER CONFIGURATION:

```
// Parse the parameters
int numberOfReducers = Integer.parseInt(args[0]);
Path inputPath1 = new Path(args[1]);
Path outputDir1 = new Path(args[2]);
Path inputPath2 = new Path(args[2]);
Path outputDir2 = new Path(args[3]);
```

JOB1

```
// Job 1: Setup and configuration
Job job1 = Job.getInstance(conf);
job1.setJobName(name:"MapReduce Job 1");

FileInputFormat.addInputPath(job1, inputPath1);
FileOutputFormat.setOutputPath(job1, outputDir1);

job1.setJarByClass(cls:DriverBigData.class);
job1.setInputFormatClass(cls:KeyValueTextInputFormat
job1.setOutputFormatClass(cls:TextOutputFormat.class

job1.setMapperClass(cls:MapperBigData.class);
job1.setMapOutputKeyClass(theClass:Text.class);
job1.setMapOutputValueClass(theClass:IntWritable.cla

job1.setReducerClass(cls:ReducerBigData.class);
job1.setOutputKeyClass(theClass:Text.class);
job1.setOutputValueClass(theClass:IntWritable.class)

job1.setNumReduceTasks(numberOfReducers);

// Execute Job 1
if (!job1.waitForCompletion(verbose:true)) {
    return 1; // If Job 1 fails, exit with error cod
}
```

JOB2

```
// Job 2: Setup and configuration
Job job2 = Job.getInstance(conf);
job2.setJobName(name:"MapReduce Job 2");

FileInputFormat.addInputPath(job2, inputPath2);
FileOutputFormat.setOutputPath(job2, outputDir2);

job2.setJarByClass(cls:DriverBigData.class);
job2.setInputFormatClass(cls:KeyValueTextInputFormat.class);
job2.setOutputFormatClass(cls:TextOutputFormat.class);

job2.setMapperClass(cls:MapperBigData2.class);
job2.setMapOutputKeyClass(theClass:Text.class);
job2.setMapOutputValueClass(theClass:IntWritable.class);

job2.setReducerClass(cls:ReducerBigData2.class);
job2.setOutputKeyClass(theClass:Text.class);
job2.setOutputValueClass(theClass:IntWritable.class);

job2.setNumReduceTasks(numberOfReducers);

// Execute Job 2
if (job2.waitForCompletion(verbose:true)) {
    exitCode = 0; // Job 2 completed successfully
} else {
    exitCode = 1; // Job 2 failed
}

return exitCode;
```

MAP1

```
String[] keys=key.toString().split("-");
String monthYear= keys[0]+"-"+keys[1];
int val=Integer.parseInt(value.toString());
context.write(new Text(monthYear), new IntWritable(val));
```

MAP2

```
//key=2015-11   value=1305
String[] keys=key.toString().split("-");
int val=Integer.parseInt(value.toString());
context.write(new Text(keys[0]), new IntWritable(val));
```

REDUCER1

```
int sum = 0;
int occurences=0;
for (IntWritable value : values) {
    sum = sum + value.get();
    occurences= occurences+1; }
int avg= sum/occurences;
context.write(key, new IntWritable(avg));
```

REDUCER2

```
int sum = 0;
for (IntWritable value : values) {
    sum = sum + value.get();}
context.write(key, new IntWritable(sum));
```

Es: 9: using an in-mapper combiners

Input: (unstructured) textual file

Output: number of occurrences of each word appearing at least one time in the input file

DRIVER: (REDUCER AND MAPPER EQUAL TO ES1)

```
// Set map class
job.setMapperClass(cls:MapperBigData.class);

// Set map output key and value classes
job.setMapOutputKeyClass(theClass:Text.class);
job.setMapOutputValueClass(theClass:IntWritable.class);

job.setCombinerClass(cls:CombinerBigData.class);
// Set reduce class
job.setReducerClass(cls:ReducerBigData.class);
```

COMBINER

```
class CombinerBigData extends Reducer<
                Text,           // Input key type
                IntWritable,    // Input value type
                Text,           // Output key type
                IntWritable> {  // Output value type
    @Override
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
        int occurrences = 0;
        for (IntWritable value : values) {
            occurrences = occurrences + value.get();
        }
        context.write(key, new IntWritable(occurrences));
```

Es:10

Input: a collection of (structured) textual csv files containing the daily value of PM10 for a set of sensors. Each line of the files has the following format **sensorId,date,PM10 value (µg/m3)\n**

Output: Print on the standard output the total number of records.

DRIVER:

import org.apache.hadoop.mapreduce.Counter;

```
public class DriverBigData extends Configured implements Tool {

  static enum COUNTERS{
    RECORD_COUNT
  }

  @Override
  public int run(String[] args) throws Exception {

// Execute the job and wait for completion
if (job.waitForCompletion(verbose:true)==true){
  Counter countRecordCounter = job.getCounters().findCounter(COUNTERS.RECORD_COUNT);
  System.out.println("Record count: " + countRecordCounter.getValue());
  exitCode = 0;
}else
  exitCode=1;

return exitCode;
```

MAPPER: Map-only job

```
protected void map(
        LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
        context.getCounter(COUNTERS.RECORD_COUNT).increment(incr:1);
        }
```

Es:11 = Es:5

Es:12

Input: a collection of (structured) textual files containing the daily value of PM10 for a set of sensors. Each line of the files has the following format **sensorId,date\tPM10 value (µg/m3)\n**.

Output: the records with a PM10 value below a user provided threshold (the threshold is an argument of the program).

DRIVER: Map-only job, remember **job.setNumReduceTasks(0);**

```
String threshold;
//Parse the parameters
inputPath = new Path(args[1]);
outputDir = new Path(args[2]);
threshold= args[0];
Configuration conf = this.getConf();
conf.set(name:"threshold", threshold);
```

MAPPER

```
String soglia = context.getConfiguration().get(name:"threshold");
double sogliaValue= Float.parseFloat(soglia);
String[] sensordDate= key.toString().split(",");
float valore=Float.parseFloat(value.toString());
if (valore<sogliaValue){
    System.out.println(valore);
   context.write(new Text(sensordDate[0]), new FloatWritable(valore));
}
```

Es:13: CONTROLLA SE E' GIUSTO AVERE UN FILE

Input: a (structured) textual csv files containing the daily income of a company. Each line of the files has the following format **date\tdaily income\n**

Output: Select the date and income of the top 1 most profitable date. In case of tie, select the first date.

MAPPER

```
private TreeMap<Text, Integer> topRecordMap = new TreeMap<>();

@Override
protected void map(Text key, Text value, Context context) throws IOException, InterruptedException {
        int intValue = Integer.parseInt(value.toString());
        topRecordMap.put(new Text(key), intValue);
}
protected Map.Entry<Text, Integer> best(TreeMap<Text, Integer> map) {
        Map.Entry<Text, Integer> maxEntry = null;
        for (Map.Entry<Text, Integer> entry : map.entrySet()) {
        if (maxEntry == null || entry.getValue() > maxEntry.getValue()) {
                maxEntry = entry;
        }}
        return maxEntry;}


@Override
protected void cleanup(Context context) throws IOException, InterruptedException {
        if (!topRecordMap.isEmpty()) {
                Map.Entry<Text, Integer> topper = best(topRecordMap);
                for (Map.Entry<Text, Integer> entry : topRecordMap.entrySet()) {
                if (entry.getValue().equals(topper.getValue())) {
                        if (entry.getKey().compareTo(topper.getKey()) > 0) {
                        topper = entry;
                        }}}
                String result = topper.getKey() + "," + topper.getValue();
                context.write(NullWritable.get(), new Text(result));
```

REDUCER:

```
int max = Integer.MIN_VALUE;
String bestDate="";
for (Text value : values) {
    //2015-11-02,1305
    String[] dateValue= value.toString().split(",");
    int val= Integer.parseInt(dateValue[1]);
    String date=  (dateValue[0]);
    //date.compareTo(bestDate) > 0 vuol dire: "date è più recente di bestDate
    if (val > max || (val == max && date.compareTo(bestDate) > 0)) {
        max = val;
        bestDate = date;  }}
context.write(new Text(bestDate), new IntWritable(max));
```

Then select the first 2 dates among the ones associated with the highest income. **RIVEDERE**

Es:14= Es:1

Es:15

Input: a collection of news (textual files).

Output: List of distinct words occurring in the collection associated with a set of unique integers. Each word is associated with a unique integer (and viceversa).

MAPPER: equal to Es 1 will value NullWritable to be more efficient

REDUCER

```
protected void reduce(
    Text key, // Input key type
    Iterable<IntWritable> values, // Input value type
    Context context) throws IOException, InterruptedException {
    context.write(key, new IntWritable(wordId));
    wordId++;
    //we do not have to use a list because a key cannot be processed by
    //different reducer so a reducer can receive (example,[1,1,1]) and
    //it will emit (example,wordId)
```

Es: 17 (no 16)

Input: two structured textual files containing the temperatures gathered by a set of sensors. Each line of the first file has the following format **sensorID,date,hour,temperature\n**. Each line of the second file has the following format **date,hour,temperature,sensorID\n**

Output: the maximum temperature for each date (considering the data of both input files)

DRIVER: **import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;**

```
//Parse the parameters
numberOfReducers = Integer.parseInt(args[0]);
inputPath1 = new Path(args[1]);
inputPath2 = new Path(args[2]);
outputDir = new Path(args[3]);


Configuration conf = this.getConf();

// Define a new job
Job job = Job.getInstance(conf);

// Assign a name to the job
job.setJobName(name:"Lab - Skeleton");
```

Delete set map part !!DELETE job.setInputFormatClass(TextInputFormat.class);

```
MultipleInputs.addInputPath(job,
  inputPath1,
  inputFormatClass:TextInputFormat.class,
  mapperClass:MapperBigData.class);

MultipleInputs.addInputPath(job,
    inputPath2,
    inputFormatClass:TextInputFormat.class,
    mapperClass:MapperBigData2.class);

// Set path of the output folder for this job
FileOutputFormat.setOutputPath(job, outputDir);
```

MAPPER 1 =MAPPER 2 only change the index of values

```
protected void map(
        LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {

        String[] values=value.toString().split(",");
        String date=values[1];
        Float temperature=Float.parseFloat(values[3]);
        context.write(new Text(date), new FloatWritable(temperature));
```

REDUCER

```
Float max = (float) 0.0;
for (FloatWritable value : values) {
    if (value.get()>max){
        max=value.get();
    } }
context.write(key, new FloatWritable(max));
```

Es: 18/19

Input: a set of textual files containing the temperatures gathered by a set of sensors. Each line of the files has the following format **sensorID,date,hour,temperature\n**

Output: The lines of the input files associated with a temperature value less than or equal to 30.0.

MAPPER: Map-only job

```
protected void map(
        LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
        //key=0 , value=s2,2016-01-02,14:10,30.2
        String[] fields= value.toString().split(",");
        Float temperature= Float.parseFloat(fields[3]);
        if (temperature > 30.0){
           context.write(NullWritable.get(), new Text(value));
        }
}
```

Es: 20 NO MULTIPLE OUTPUT

Input: a set of textual files containing the temperatures gathered by a set of sensors. Each line of the files has the following format **sensorID,date,hour,temperature\n**.

Output: a set of files with the prefix "high-temp-" containing the lines of the input files with a temperature value greater than 30.0 and a set of files with the prefix "normal-temp-" containing the lines of the input files with a temperature value less than or equal to 30.0. **CHECK**

Es: 21

Input: A large textual file containing one sentence per line. A small file containing a set of stopwords with one stopword per line

Output: A textual file containing the same sentences of the large input file without the words appearing in the small file. The order of the sentences in the output file can be different from the order of the sentences in the input file.

DRIVER: **job.addCacheFile(new Path("stopwords.txt").toUri());**

MAPPER: **import java.net.URI;** Map-only job

```
private ArrayList<String> stopWords;
protected void setup(Context context) throws IOException, InterruptedException{
        String nextLine;
        stopWords=new ArrayList<String>();
        URI[] urisCachedFiles = context.getCacheFiles();
        BufferedReader fileStopWords = new BufferedReader(new
                        FileReader(new File(urisCachedFiles[0].getPath())));
        while ((nextLine = fileStopWords.readLine()) != null) {
                stopWords.add(nextLine);
        }

        fileStopWords.close();
}


boolean stopword;
String[] words = value.toString().split("\\s+");
String sentenceWithoutStopwords=new String("");
for(String word : words) {
        if (stopWords.contains(word)==true)
                stopword=true;
        else
            stopword=false;
    if (stopword==false)
    {
            sentenceWithoutStopwords=sentenceWithoutStopwords.concat(word+" ");
    } }
context.write(NullWritable.get(),
                new Text(sentenceWithoutStopwords));
```

Es: 22

Input: A textual file containing pairs of users (one pair per line). Each line has the format **Username1,Username2**. Each pair represents the fact that Username1 is friend of Username2 (and vice versa). One username specified as parameter by means of the command line

Output: The friends of the specified username stored in a textual file. One single line with the list of friends.

DRIVER:

```
user= new String(args[3]);


Configuration conf = this.getConf();
conf.set(name:"utente", user);
```

MAPPER

```
String user= context.getConfiguration().get(name:"utente");
String[] users= value.toString().split(",");
System.out.println(users[0]);
System.out.println(user);
System.out.println(users);
if (users[0].equals(user)){
   context.write(new Text(users[1]),NullWritable.get());
}
if (users[1].equals(user)){
    context.write(new Text(users[0]),NullWritable.get());
}
```

Es: 23

Input: A textual file containing pairs of users (one pair per line). Each line has the format **Username1,Username2**. Each pair represents the fact that Username1 is friend of Username2 (and vice versa). One username specified as parameter by means of the command line

Output: The potential friends of the specified username stored in a textual file. One single line with the list of potential friends. User1 is a potential friend of User2 if they have at least one friend in common.

MAPPER: generate all the pair for which the key is different from the specified user (user1, user2), (user4, user2). It's not useful have the specified user as key, because the values will be direct friends

```
protected void setup(Context context) throws IOException, InterruptedException {
   specifiedUser = context.getConfiguration().get(name:"utente");
}
protected void map(LongWritable key,
        Text value,
        Context context) throws IOException, InterruptedException {

    String[] users = value.toString().split(",");
    if (specifiedUser.compareTo(users[0]) != 0)
        context.write(new Text(users[0]), new Text(users[1]));
    if (specifiedUser.compareTo(users[1]) != 0)
        context.write(new Text(users[1]), new Text(users[0]));
}
```

REDUCER: check if in values there is the specified user and memorize the other values.

```
@Override
protected void reduce(Text key, // Input key type
        Iterable<Text> values, // Input value type
        Context context) throws IOException, InterruptedException {
    boolean containsSpecifiedUser;
    // Partial list of potential friends
    HashSet<String> partialListOfPotentialFriends = new HashSet<String>();
    containsSpecifiedUser = false;
    //key=User1 values=[User2,User3,User4]
    for (Text value : values) {
        if (specifiedUser.compareTo(value.toString()) == 0)
            containsSpecifiedUser = true; //means that can be useful
        else {
            // Store the pontential friends
            partialListOfPotentialFriends.add(value.toString());}}

    if (containsSpecifiedUser == true && partialListOfPotentialFriends.size() > 0) {
        //means that I store the potential friends of the right user
        for (String user : partialListOfPotentialFriends) {
            finalListPotentialFriends.add(user);  //because if specifiedUser=false,
                                                  // partialListOfPotentialFriends will be ignored
        }}}

protected void cleanup(Context context) throws IOException, InterruptedException {
    String globalPotFriends = new String("");
    for (String potFriend : finalListPotentialFriends) {
        globalPotFriends = globalPotFriends.concat(potFriend + " ");
    }
    context.write(new Text(globalPotFriends), NullWritable.get());
}
```

Es: 23 bis

Solve problem #23 by removing the friends of the specified user from the list of its potential friends.

DRIVER: 2 mapper and 2 reducer

```
@Override
public int run(String[] args) throws Exception {

    Path inputPath;
    Path outputDir;
    Path outputDir2;
    int exitCode;
    // Parse the parameters
    inputPath = new Path(args[0]);
    outputDir = new Path(pathString:"ex23Bis_temp");
    outputDir2 = new Path(args[1]);
```

...

```
// Execute the job and wait for completion
if (job.waitForCompletion(verbose:true) == true) {
    // Execute the second job to find the potential friends of the user of interest
    // based on the list of its friends (that is the output of the previous job and
    // it is shared by using the distributed cache)
    // The shared file is also used to remove the users who are already friends
    // of the user of interest
    Configuration conf2 = this.getConf();

    // Define a new job
    Job job2 = Job.getInstance(conf2);

    // Add the hdfs file created by the first job (outputDir/part-r-00000)
    // in the distributed cache.
    // It contains the list of friends of the user we are interest in.
    job2.addCacheFile(
        new Path(outputDir + "/part-r-00000").toUri());

    // Assign a name to the job
    job2.setJobName(name:"Exercise #23 Bis - Job 1 - Find potential friends");

    // Set path of the input file/folder
    // The input path is the same input path of the first job
    FileInputFormat.addInputPath(job2, inputPath);
```

MAPPER 1:

```
protected void map(LongWritable key, // Input key type
        Text value, // Input value type
        Context context) throws IOException, InterruptedException {
    String[] users = value.toString().split(",");
    if (specifiedUser.compareTo(users[0]) == 0) {
        context.write(NullWritable.get(), new Text(users[1]));
    }
    if (specifiedUser.compareTo(users[1]) == 0) {
        context.write(NullWritable.get(), new Text(users[0]));
    }
```

REDUCER 1:

```
@Override
protected void reduce(NullWritable key, // Input key type
        Iterable<Text> values, // Input value type
        Context context) throws IOException, InterruptedException {
    // Iterate over the set of values and emit one line for each of friend of the
    // user of interest
    for (Text value : values) {
        context.write(new Text(value.toString()), NullWritable.get());
    }
}
```

MAPPER 2:

```
String specifiedUser;


ArrayList<String> friends;

protected void setup(Context context) throws IOException, InterruptedException {
    String line;
    // Store the information about the user of interest
    specifiedUser = context.getConfiguration().get(name:"username");
    // Store in the ArraList friends the list of friends available in the
    // shared file
    friends = new ArrayList<String>();
    // This application has one single single cached file.

    BufferedReader fileFriends = new BufferedReader(new FileReader(new File("part-r-00000")));
    // There is one friend per line
    while ((line = fileFriends.readLine()) != null) {
        friends.add(line);
    }
    fileFriends.close();
}

protected void map(LongWritable key, // Input key type
        Text value, // Input value type
        Context context) throws IOException, InterruptedException {

    // Extract username1 and username2
    String[] users = value.toString().split(",");
    // Check if one of the two users is friend of the user of interest.
    // If it is true, the the other user of the current pair is a potential friend
    // of the user of interest
    if (friends.contains(users[0]) == true &&
            users[1].compareTo(specifiedUser) != 0) {
        // users[0] is a friend of specifiedUser
        // users[1] is a potential friend of specifiedUser
        // emit the pair (null, users[1]) only if users[1] is not already a friend of specifiedUser
        if (friends.contains(users[1]) == false)
            context.write(NullWritable.get(), new Text(users[1]));
    }
    if (friends.contains(users[1]) == true && users[0].compareTo(specifiedUser) != 0) {
        // users[1] is a friend of specifiedUser
        // users[0] is a potential friend of specifiedUser
        // emit the pair (null, users[0]) only if users[0] is not already a friend of specifiedUser
        if (friends.contains(users[0]) == false)
            context.write(NullWritable.get(), new Text(users[0]));
    }
```

REDUCER 2:

```
@Override
protected void reduce(NullWritable key, // Input key type
        Iterable<Text> values, // Input value type
        Context context) throws IOException, InterruptedException {

    ArrayList<String> potFriends = new ArrayList<String>();
    String listOfPotFriends = new String("");
    // Iterate over the set of values and include them in the ArrayList of
    // potential friends
    for (Text value : values) {
        if (potFriends.contains(value.toString()) == false)
            potFriends.add(value.toString());
    }
    // Concatenate the list of potential friends
    for (String potFriend : potFriends) {
        listOfPotFriends = listOfPotFriends.concat(potFriend + " ");    }
    context.write(new Text(listOfPotFriends), NullWritable.get());
}
```

Es: 24

Input: A textual file containing pairs of users (one pair per line). Each line has the format **Username1,Username2**. Each pair represents the fact that Username1 is friend of Username2 (and vice versa).

Output: A textual file containing one line for each user. Each line contains a user and the list of its friends

MAPPER: generate all the possible pairs

```
protected void map(LongWritable key,
        Text value,
        Context context) throws IOException, InterruptedException {
    String[] users = value.toString().split(",");
    context.write(new Text(users[0]), new Text(users[1]));
    context.write(new Text(users[1]), new Text(users[0]));

}
```

REDUCER: create the string

```
protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
    String keyString=key.toString()+"-";
        for (Text value : values) {
            String user = value.toString();
            keyString=keyString.concat(user);
            keyString=keyString.concat(",");
            }
        context.write(new Text(keyString), NullWritable.get());
    }}
```

Es: 25

Input: A textual file containing pairs of users (one pair per line). Each line has the format **Username1,Username2**. Each pair represents the fact that Username1 is friend of Username2 (and vice versa).

Output: A textual file containing one line for each user with at least one potential friend. Each line contains a user and the list of its potential friends. User1 is a potential friend of User2 if they have at least one friend in common.

DRIVER: 2 job

MAPPER 1:

```
protected void map(
        LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
        // Extract username1 and username2
        String[] users = value.toString().split(",");
        // Emit two key-value pairs
        // (username1,username2)
        // (username2,username1)
        context.write(new Text(users[0]), new Text(users[1]));
        context.write(new Text(users[1]), new Text(users[0]));
```

MAPPER 2:

```
protected void map(
        Text key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
        // Emit one key-value pair of each user in value.
    // Key is equal to the key of the input key-value pair
        String[] users = value.toString().split(" ");
        for (String user: users)
        {
        context.write(new Text(key.toString()), new Text(user));
        }
```

REDUCER 1:

```
protected void reduce(
    Text key,       // Input key type
    Iterable<Text> values,  // Input value type
    Context context) throws IOException, InterruptedException {
    HashSet<String> users;
    // Each user in values is potential friend of the other users in values
    // because they have the user "key" in common.
    // Hence, the users in values are potential friends of each others.
    // Since it is not possible to iterate more than one time on values
    // we need to create a local copy of it. However, the
    // size of values is at most equal to the friend of user "key". Hence,
    // it is a small list
    users=new HashSet<String>();
    for (Text value : values) {
        users.add(value.toString());}
           // Compute the list of potential friends for each user in users
        for (String currentUser: users)
    {
        String listOfPotentialFriends=new String("");
                for (String potFriend: users)
        {   // If potFriend is not currentUser then include him/her in the
            // potential friends of currentUser
            if (currentUser.compareTo(potFriend)!=0)
                listOfPotentialFriends=listOfPotentialFriends.concat(potFriend+" ");}
            // Check if currentUser has at least one friend
        if (listOfPotentialFriends.compareTo("")!=0)
            context.write(new Text(currentUser), new Text(listOfPotentialFriends));
```

REDUCER 2:

```
String listOfPotentialFriends;
HashSet<String> potentialFriends;
potentialFriends=new HashSet<String>();
// Iterate over the values and include the users in the final set
for (Text user: values)
{
    // If the user is new then it is inserted in the set
    // Otherwise, it is already in the set, it is ignored
    potentialFriends.add(user.toString());
}
listOfPotentialFriends=new String("");
for (String user: potentialFriends){
    listOfPotentialFriends=listOfPotentialFriends.concat(user+" ");
}
context.write(new Text(key), new Text(listOfPotentialFriends));
```

Es: 26

Input: A large textual file containing a list of words per line. The small file dictionary.txt containing the mapping of each possible word appearing in the first file with an integer. Each line contain the mapping of a word with an integer and it has the following format **Word\tInteger\n**

Output: A textual file containing the content of the large file where the appearing words are substituted by the corresponding integers

DRIVER: es21

MAPPER: Map Only job

```
private Map<String, String> dict = new HashMap<>();
protected void setup(Context context) throws IOException, InterruptedException{
        String line;
        dict=new HashMap<>();
        URI[] urisCachedFiles = context.getCacheFiles();
        BufferedReader file = new BufferedReader(new
                        FileReader(new File(urisCachedFiles[0].getPath())));
        while ((line=file.readLine()) != null) {
                String[] valueWord=line.split("\\s+");
                dict.put(valueWord[1].toLowerCase(), valueWord[0]);
        }
        file.close();
}
```

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

        String newSentence = new String("");
        String[] words = value.toString().split("\\s+");

        for (String word : words) {
        String lowerWord = word.toLowerCase();
        if (dict.containsKey(lowerWord)) {
                newSentence=newSentence.concat(dict.get(lowerWord));
        } else {
                newSentence=newSentence.concat(word);
        }
        newSentence=newSentence.concat(" "); // aggiunge lo spazio dopo ogni parola
        }

        context.write(NullWritable.get(), new Text(newSentence.toString().trim()));
}
```

Es: 27

Input: A large textual file containing a set of records. Each line contains the information about one single user. Each line has the format **UserId,Name,Surname,Gender,YearOfBirth,City,Education**. A small file with a set of business rules that are used to assign each user to a category. Each line contains a business rule with the format **Gender=<value> and YearOfBirth=<value> -> Category**. Rules are mutually exclusive

Output: One record for each user with the following format. The original information about the user plus the category assigned to the user by means of the business rules. Since the rules are mutually exclusive, there is only one rule applicable for each user. If no rules is applicable/satisfied by a user, assign the user to the “Unknown” category

DRIVER AND MAPPER SETUP equal to ES:21 – Map-only job

```
protected void map(
   LongWritable key,   // Input key type
   Text value,         // Input value type
   Context context) throws IOException, InterruptedException {
       boolean flag=false;
       String cat=" ";
       String result;
       String[] fields= value.toString().split(",");
       String gender= fields[3].toString();
       String yearOfBirth=fields[4].toString();
       for (String line : rules) {
               String[] dividedLine= line.split("\\s+");
               String genderRules=dividedLine[0].split("=")[1];
               String yearRule=dividedLine[2].split("=")[1];
               String category=dividedLine[4];
               if (gender.equals(genderRules) && (yearOfBirth.equals(yearRule))){
                       flag=true;
                       cat=category;}  }
       if (flag){
                       result=value+" "+cat;}
       else{
                       result=value+" unknown";}
       context.write(new Text(result),  NullWritable.get());
       }
```

Es: 28

Input: A large textual file containing a set of questions. Each line contains one question. Each line has the format **QuestionId,Timestamp,TextOfTheQuestion**. A large textual file containing a set of answers. Each line contains one answer. Each line has the format **AnswerId,QuestionId,Timestamp,TextOfTheAnswer**

Output: One line for each pair (question,answer) with the following format **QuestionId,TextOfTheQuestion, AnswerId,TextOfTheAnswer**

DRIVER equal to es17

MAPPER 1

```
String[] fields=value.toString().split(",");
String questionId=fields[0];
String questionText=fields[2];
// Key = questionId
// Value = Q:+questionId,questionText
context.write(new Text(questionId), new Text("Q:"+questionId+","+questionText));
```

MAPPER2

```
String[] fields=value.toString().split(",");
String  answerId=fields[0];
String  questionId=fields[1];
String answerText=fields[3];

context.write(new Text(questionId), new Text("A:"+answerId+","+answerText));
```

REDUCER

```
@Override
protected void reduce(
    Text key, // Input key type
    Iterable<Text> values, // Input value type
    Context context) throws IOException, InterruptedException {

    String record;
    ArrayList<String> answers=new ArrayList<String>();
    String question=null;
    for (Text value : values) {
        String table_record=value.toString();
        if (table_record.startsWith("Q:")==true){
            record=table_record.replaceFirst("Q:", "");
            question=record;}
        else{
            record=table_record.replaceFirst("A:", "");
            answers.add(record);}}
    for (String answer:answers){
        context.write(NullWritable.get(), new Text(question+","+answer));
```

Es: 29

Input: A large textual file containing a set of records. Each line contains the information about one single user. Each line has the format **UserId,Name,Surname,Gender,YearOfBirth,City,Education.** A large textual file with pairs (Userid, MovieGenre). Each line contains pair Userid, MovieGenre with the format **Userid,MovieGenre**. It means that UserId likes movies of genre MovieGenre.

Output: One record for each user that likes both Comedy and Adventure movies. Each output record contains only Gender and YearOfBirth of a selected user **Gender,YearOfBirth**. Duplicate pairs must not be removed

DRIVER: same of ES:17

MAPPER1:

```
protected void map(
    LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
        String[] fields=value.toString().split(",");
            String userId=fields[0];
            String gender=fields[3];
            String year=fields[4];
            String info=gender+","+year;
            context.write(new Text(userId), new Text("U:"+ info));
```

MAPPER2:

```
protected void map(
        LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
        String[] fields=value.toString().split(",");
            String  user=fields[0];
            String  genre=fields[1];
            context.write(new Text(user), new Text("G:"+genre));
```

REDUCER:

```
protected void reduce(
    Text key,
    Iterable<Text> values,
    Context context) throws IOException, InterruptedException {
    String infoUser = null;
    boolean flagComedy=false;
    boolean flagAdventure=false;
    ArrayList<String> genres = new ArrayList<>();

    for (Text value : values) {
        String val = value.toString();
        if (val.startsWith("U:")) {
            infoUser = val.substring(2);
        } else if (val.startsWith("G:")) {
            genres.add(val.substring(2));
        }
    }
    if (infoUser != null) {
        for (String genre : genres) {
            if (genre.equals("Comedy")){
                flagComedy=true;
            }
            if (genre.equals("Adventure")){
                flagAdventure=true;
            }
        }
        if (flagAdventure && flagComedy){
            context.write(NullWritable.get(), new Text(infoUser));
        }
```