Exercise #1

- Word count problem
  - Input: (unstructured) textual file
  - Output: number of occurrences of each word appearing at least one time in the input file

Exercise #1 - Example

- Input file

Toy example
file for Hadoop.
Hadoop running
example.

- Output pairs (toy, 1)
(example, 2)
(file, 1)
(for, 1)
(hadoop, 2)
(running, 1)

```
/**
 * Exercise 1 - Mapper
 */
class MapperBigData extends Mapper<
                    LongWritable, // Input key type
                    Text,         // Input value type
                    Text,         // Output key type
                    IntWritable> {// Output value type

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

            // Split each sentence in words. Use whitespace(s) as delimiter
             // (=a space, a tab, a line break, or a form feed)
             // The split method returns an array of strings
            String[] words = value.toString().split("\\s+");

            // Iterate over the set of words
            for(String word : words) {
             // Transform word case
                String cleanedWord = word.toLowerCase();

                // emit the pair (word, 1)
                context.write(new Text(cleanedWord), new IntWritable(1));
            }
    }
}


/**
```

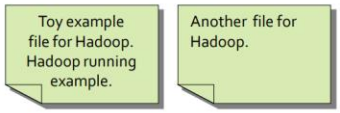
```
 * Exercise 1 - Reducer
 */
class ReducerBigData extends Reducer<
                Text,           // Input key type
                IntWritable,    // Input value type
                Text,           // Output key type
                IntWritable> {  // Output value type

    @Override

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

        int occurrences = 0;

        // Iterate over the set of values and sum them
        for (IntWritable value : values) {
            occurrences = occurrences + value.get();
        }

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

Exercise #2

- Word count problem
  - Input: a HDFS folder containing textual files
  - Output: number of occurrences of each word appearing in at least one file of the collection (i.e., files of the input directory)
- The only difference with respect to exercise #1 is given by the input
  - Now the input is a collection of textual files

Exercise #2 - Example

- Input files



- Output pairs (another, 1) (example, 2) (file, 2) (for, 2) (hadoop, 3) (running, 1) (toy, 1)

```
/**
 * Exercise 1 - Mapper
 */
class MapperBigData extends Mapper<
                    LongWritable, // Input key type
                    Text,         // Input value type
                    Text,         // Output key type
                    IntWritable> {// Output value type

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

/**

```
            // Split each sentence in words. Use whitespace(s) as delimiter (=a
space, a tab, a line break, or a form feed)
             // The split method returns an array of strings
            String[] words = value.toString().split("\\s+");

            // Iterate over the set of words
            for(String word : words) {
             // Transform word case
                String cleanedWord = word.toLowerCase();

                // emit the pair (word, 1)
                context.write(new Text(cleanedWord), new IntWritable(1));
            }
    }
}

/**
 * Exercise 1 - Reducer
 */
class ReducerBigData 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 occurrances = 0;

        // Iterate over the set of values and sum them
        for (IntWritable value : values) {
            occurrances = occurrances + value.get();
        }
        context.write(key, new IntWritable(occurrances));
    }
}
```

Exercise #3

- PM10 pollution analysis
  - 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
    - Suppose to set threshold = 50 µg/m³
    - Select only the sensors that are associated at least one time with a PM10 above the threshold

Exercise #3 - Example

- Input file

```
s1,2016-01-01   20.5
s2,2016-01-01   30.1
s1,2016-01-02   60.2
s2,2016-01-02   20.4
s1,2016-01-03   55.5
s2,2016-01-03   52.5
```

- Output pairs (s1, 2)
  (s2, 1)

```java
/**
 * Exercise 3 - Mapper
 */
class MapperBigData extends Mapper<
                    Text,          // Input key type
                    Text,          // Input value type
                    Text,         // Output key type
                    IntWritable> {// Output value type

       private static Double PM10Threshold = new Double(50);

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

            // Extract sensor and date from the key
            String[] fields = key.toString().split(",");

            String sensor_id=fields[0];
            Double PM10Level=new Double(value.toString());

            // Compare the value of PM10 with the threshold value
            if (PM10Level.compareTo(PM10Threshold)>0)
            {
                // emit the pair (sensor_id, 1)
                context.write(new Text(sensor_id), new IntWritable(1));
            }
    }
}

/**
 * Exercise 3 - Reducer
 */
class ReducerBigData 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 numDays = 0;

             // Iterate over the set of values and sum them
             for (IntWritable value : values) {
                    numDays = numDays + value.get();
             }

             context.write(new Text(key), new IntWritable(numDays));
       }
}
```

Exercise #4

- PM10 pollution analysis per city zone
- 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
    - Suppose to set threshold = 50 µg/m³
    - Report only the zones with at least one date with PM10 above the threshold

Exercise #4 - Example

- Input file

```
zone1,2016-01-01    20.5
zone2,2016-01-01    30.1
zone1,2016-01-02    60.2
zone2,2016-01-02    20.4
zone1,2016-01-03    55.5
zone2,2016-01-03    52.5
```

- Output pairs (zone1, [2016-01-03, 2016-01-02])
  (zone2, [2016-01-01])

```
/**
 * Exercise 4 - Mapper
 */
class MapperBigData extends
            Mapper<Text, // Input key type
                   Text, // Input value type
                   Text, // Output key type
                   Text> { // Output value type

      private static Double PM10Threshold = new Double(50);

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

            // Extract zone and date from the key
            String[] fields = key.toString().split(",");

            String zone = fields[0];
            String date = fields[1];
            Double PM10Level = new Double(value.toString());

            // Compare the value of PM10 with the threshold value
            if (PM10Level > PM10Threshold) {
                  // emit the pair (zoneID, date)
                  context.write(new Text(zone), new Text(date));
            }
      }
}


/**
 * Exercise 4 - Reducer
 */
class ReducerBigData extends
            Reducer<Text, // Input key type
                    Text, // Input value type
                    Text, // Output key type
                    Text> { // Output value type
```

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

                String aboveThresholdDates = new String();

                // Iterate over the set of values and concatenate them
                for (Text date : values) {
                        if (aboveThresholdDates.length() == 0)
                                aboveThresholdDates = new String(date.toString());
                        else
                                aboveThresholdDates = aboveThresholdDates.concat(","
                        + date.toString());
                }

                context.write(new Text(key), new Text(aboveThresholdDates));
        }
}
```

Exercise #5

- Average
  - 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

Exercise #5 - Example

- Input file

```
s1,2016-01-01,20.5
s2,2016-01-01,30.1
s1,2016-01-02,60.2
s2,2016-01-02,20.4
s1,2016-01-03,55.5
s2,2016-01-03,52.5
```

- Output pairs (s1, 45.4)
  (s2, 34.3)

```
/**
 * Average Mapper
 */
class MapperBigData extends
                Mapper<LongWritable, // Input key type
                        Text, // Input value type
                        Text, // Output key type
                        FloatWritable> {// Output value type

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

                // Split each record by using the field separator
                // fields[0]= first attribute - sensor id
                // fields[1]= second attribute - date
```

```
            // fields[2]= third attribute - PM10 value
            String[] fields = value.toString().split(",");
            String sensorId = fields[0];
            float PM10value = Float.parseFloat(fields[2]);

            // emit the pair (sensor_id, reading value)
            context.write(new Text(sensorId), new FloatWritable(new
Float(PM10value)));
      }
}


/**
 * WordCount Reducer
 */
class ReducerBigData extends
            Reducer<Text, // Input key type
                        FloatWritable, // Input value type
                        Text, // Output key type
                        FloatWritable> { // Output value type

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

            int count = 0;
            double sum = 0;

            // Iterate over the set of values and sum them.
            // Count also the number of values
            for (FloatWritable value : values) {
                  sum = sum + value.get();

                  count = count + 1;
            }

            // Compute average value
            // Emits pair (sensor_id, average)
            context.write(new Text(key), new FloatWritable((float) sum / count));
      }
}
```

SOLUZIONE CON COMBINER

```
package it.polito.bigdata.hadoop.exercise5withcombiner;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class StatisticsWritable implements org.apache.hadoop.io.Writable {
```

```
	private float sum = 0;
	private int count = 0;

	public float getSum() {
		return sum;
	}

	public void setSum(float sumValue) {
		sum = sumValue;
	}

	public int getCount() {
		return count;
	}

	public void setCount(int countValue) {
		count = countValue;
	}

	@Override
	public void readFields(DataInput in) throws IOException {
		sum = in.readFloat();
		count = in.readInt();
	}

	@Override
	public void write(DataOutput out) throws IOException {
		out.writeFloat(sum);
		out.writeInt(count);
	}

	public String toString() {
		String formattedString = new String("" + (float) sum / count);

		return formattedString;
	}

}


/**
 * Mapper
 */
class MapperBigData extends
		Mapper<LongWritable, // Input key type
			Text, // Input value type
			Text, // Output key type
			StatisticsWritable> {// Output value type

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

		// Split each record by using the field separator
		// fields[0]= first attribute - sensor id
```

```
            // fields[1]= second attribute - date
            // fields[2]= third attribute - PM10 value
            String[] fields = value.toString().split(",");
            String sensorId = fields[0];
            float PM10value = Float.parseFloat(fields[2]);

            StatisticsWritable localSumAndCount = new StatisticsWritable();
            localSumAndCount.setSum(PM10value);
            localSumAndCount.setCount(1);

            // emit the pair (sensor_id, value - 1)
            context.write(new Text(sensorId), localSumAndCount);
      }
}


/**
 * Reducer
 */
class ReducerBigData extends Reducer<Text, // Input key type
            StatisticsWritable, // Input value type
            Text, // Output key type
            StatisticsWritable> { // Output value type

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

            int localCount = 0;
            float localSum = 0;

            // Iterate over the set of values and sum them.
            // Sum also the "number of values"
            for (StatisticsWritable value : values) {
                  localSum = localSum + value.getSum();
                  localCount = localCount + value.getCount();
            }

            StatisticsWritable localSumAndCount = new StatisticsWritable();
            localSumAndCount.setCount(localCount);
            localSumAndCount.setSum(localSum);

            // Emits pair (sensor_id, sum values - sum counts)
            context.write(new Text(key), localSumAndCount);
      }
}


/**
 * Combiner Reducer
 */
class CombinerBigData extends
            Reducer<Text, // Input key type
                        StatisticsWritable, // Input value type
```

```
                        Text, // Output key type
                        StatisticsWritable> { // Output value type

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

        int localCount = 0;
        float localSum = 0;

        // Iterate over the set of values and sum them.
        // Sum also the "number of values"
        for (StatisticsWritable value : values) {
            localSum = localSum + value.getSum();
            localCount = localCount + value.getCount();
        }

        StatisticsWritable localSumAndCount = new StatisticsWritable();
        localSumAndCount.setCount(localCount);
        localSumAndCount.setSum(localSum);

        // Emits pair (sensor_id, sum values - sum counts)
        context.write(new Text(key), localSumAndCount);
    }
}
```

Exercise #6

- Max and Min
  - 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

Exercise #6 - Example

- Input file

```
s1,2016-01-01,20.5
s2,2016-01-01,30.1
s1,2016-01-02,60.2
s2,2016-01-02,20.4
s1,2016-01-03,55.5
s2,2016-01-03,52.5
```

- Output pairs (s1, max=60.2_min=20.5)
  (s2, max=52.5_min=20.4)

SOLUZIONE BASIC

```
/**
 * Average Mapper
 */
class MapperBigData extends
            Mapper<LongWritable, // Input key type
                        Text, // Input value type
```

```
                        Text, // Output key type
                        FloatWritable> {// Output value type

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

                String record = value.toString();

                // Split each record by using the field separator
                // fields[0]= first attribute - sensor id
                // fields[1]= second attribute - date
                // fields[2]= third attribute - reading
                String[] fields = record.split(",");

                // emit the pair (sensor_id, reading value)
                context.write(new Text(fields[0]), new FloatWritable(new
Float(fields[2])));
        }
}


/**
 * WordCount Reducer
 */
class ReducerBigData extends
                Reducer<Text, // Input key type
                        FloatWritable, // Input value type
                        Text, // Output key type
                        Text> { // Output value type

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

                double min = Double.MAX_VALUE;
                double max = Double.MIN_VALUE;

                // Iterate over the set of values and update min and max.
                for (FloatWritable value : values) {
                        if (value.get() > max) {
                                max = value.get();
                        }

                        if (value.get() < min) {
                                min = value.get();
                        }
                }

                // Emits pair (sensor_id, max_min)
                context.write(new Text(key), new Text("max=" + max + "_min=" + min));
        }
}
```

SOLUZIONE CON COMBINER

```
/**
 * Mapper
 */
class MapperBigData extends
            Mapper<LongWritable, // Input key type
                        Text, // Input value type
                        Text, // Output key type
                        Text> {// Output value type

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

            String record = value.toString();

            // Split each record by using the field separator
            // fields[0]= first attribute - sensor id
            // fields[1]= second attribute - date
            // fields[2]= third attribute - reading
            String[] fields = record.split(",");

            // emit the pair (sensor_id, max reading value_min reading value)
            // value is composed of two parts: max and min value (they are the same
            // value in the mapper).
            context.write(new Text(fields[0]), new Text(fields[2] + "_" +
fields[2]));
      }
}


/**
 * Reducer
 */
class ReducerBigData extends
            Reducer<Text, // Input key type
                        Text, // Input value type
                        Text, // Output key type
                        Text> { // Output value type

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

            double min = Double.MAX_VALUE;
            double max = Double.MIN_VALUE;

            // Iterate over the set of values and update max and min.
            // The format of each input value is max_min
            for (Text value : values) {
                  // fields[0] = max
                  // fields[1] = min
                  String[] fields = value.toString().split("_");
```

```
                if (Double.parseDouble(fields[0]) > max) {
                        max = Double.parseDouble(fields[0]);
                }

                if (Double.parseDouble(fields[1]) < min) {
                        min = Double.parseDouble(fields[1]);
                }
        }

        // Emits pair (sensor_id, min_max)
        // emit the pair (sensor_id, max reading value_min reading value)
        context.write(new Text(key), new Text("max=" + max + "_min=" + min));
    }
}
```

SOLUZIONE CON COMBINER DATA TYPE

```
package it.polito.bigdata.hadoop.exercise6withcombineranddatatype;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class MinMaxWritable implements org.apache.hadoop.io.Writable
{
    private double min = Double.MAX_VALUE;
    private double max = Double.MIN_VALUE;

    public double getMin()
    {
        return min;
    }

    public void setMin(double minValue)
    {
        min=minValue;
    }

    public double getMax()
    {
        return max;
    }

    public void setMax(double maxValue)
    {
        max=maxValue;
    }


    @Override
    public void readFields(DataInput in) throws IOException {
        min=in.readDouble();
        max=in.readDouble();
```

```
	}

	@Override
	public void write(DataOutput out) throws IOException {
		out.writeDouble(min);
		out.writeDouble(max);
	}

	public String toString()
	{
		String formattedString=new String("max="+max+"_min="+min);

		return formattedString;
	}

}


/**
 * Average Mapper
 */
class MapperBigData extends Mapper<
		LongWritable, // Input key type
		Text,         // Input value type
		Text,         // Output key type
		MinMaxWritable> {// Output value type

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

		 String record=value.toString();

		// Split each record by using the field separator
		 // fields[0]= first attribute - sensor id
		 // fields[1]= second attribute - timestamp
			// fields[2]= third attribute - reading
			String[] fields = record.toString().split(",");

			// emit the pair (sensor_id, min reading value_max reading value)
			// value is composed of two parts: min and max value (they are
the same value in
			// the mapper).
			MinMaxWritable minMax=new MinMaxWritable();

			minMax.setMin(Double.parseDouble(fields[2]));
			minMax.setMax(Double.parseDouble(fields[2]));

			context.write(new Text(fields[0]), minMax);
    }
}


/**
```

```
 * WordCount Reducer
 */
class ReducerBigData extends Reducer<
                Text,           // Input key type
                MinMaxWritable,  // Input value type
                Text,           // Output key type
                MinMaxWritable> {  // Output value type

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

       double min=Double.MAX_VALUE;
       double max=Double.MIN_VALUE;

        // Iterate over the set of values and update min and max.
       // The format of each input value is min_max
        for (MinMaxWritable value : values) {

             if (value.getMax()>max) {
                   max=value.getMax();
             }

             if (value.getMin()<min) {
                   min=value.getMin();
             }
        }

             // emit the pair (sensor_id, min_max value)
             // value is composed of two parts: min and max.
             MinMaxWritable minMax=new MinMaxWritable();

             minMax.setMin(min);
             minMax.setMax(max);

        context.write(new Text(key), minMax);
    }
}


/**
 * WordCount Reducer
 */
class CombinerBigData extends Reducer<
                Text,           // Input key type
                MinMaxWritable,  // Input value type
                Text,           // Output key type
                MinMaxWritable> {  // Output value type

    @Override
    protected void reduce(
        Text key, // Input key type
        Iterable<MinMaxWritable> values, // Input value type
```

```
        Context context) throws IOException, InterruptedException {

    double min=Double.MAX_VALUE;
    double max=Double.MIN_VALUE;

     // Iterate over the set of values and update min and max.
    // The format of each input value is min_max
     for (MinMaxWritable value : values) {

          if (value.getMax()>max) {
               max=value.getMax();
          }

          if (value.getMin()<min) {
               min=value.getMin();
          }
     }

          // emit the pair (sensor_id, min_max value)
          // value is composed of two parts: min and max.
          MinMaxWritable minMax=new MinMaxWritable();

          minMax.setMin(min);
          minMax.setMax(max);

     context.write(new Text(key), minMax);
  }
}
```

Exercise #7

- Inverted index
  - 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"

Exercise #7 - Example

- Input file

Sentence#1	Hadoop or Spark
Sentence#2	Hadoop or Spark and Java
Sentence#3	Hadoop and Big Data

- Output pairs

(hadoop, [Sentence#1, Sentence#2, Sentence#3])
(spark, [Sentence#1, Sentence#2])
(java, [Sentence#2])
(big, [Sentence#3])
(data, [Sentence#3])

```
/**
 * Mapper
 */
class MapperBigData extends
            Mapper<Text, // Input key type
                   Text, // Input value type
                   Text, // Output key type
                   Text> {// Output value type
```

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

		// Split each sentence in words. Use whitespace(s) as delimiter (=a
		// space, a tab, a line break, or a form feed)
		// The split method returns an array of strings
		String[] words = value.toString().split("\\s+");

		// Iterate over the set of words
		for (String word : words) {
			// Transform word case
			String cleanedWord = word.toLowerCase();

			if (cleanedWord.compareTo("and") != 0 &&
cleanedWord.compareTo("or") != 0
					&& cleanedWord.compareTo("not") != 0)
				// emit the pair (word, sentenceid)
				context.write(new Text(cleanedWord), new Text(key));
		}
	}
}


/**
 * WordCount Reducer
 */
class ReducerBigData extends
		Reducer<Text, // Input key type
				Text, // Input value typeF
				Text, // Output key type
				Text> { // Output value type

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

		String invIndex = new String();

		// Iterate over the set of sentenceids and concatenate them
		for (Text value : values) {
			invIndex = invIndex.concat(value + ",");
		}

		context.write(key, new Text(invIndex));
	}
}
```

Exercise #8

- Total income for each month of the year and Average monthly income per year
  - 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

Exercise #8 - Example

- Input file

2015-11-01	1000
2015-11-02	1305
2015-12-01	500
2015-12-02	750
2016-01-01	345
2016-01-02	1145
2016-02-03	200
2016-02-04	500

- Output

(2015-11,2305)
(2015-12,1250)
(2016-01,1490)
(2016-02,700)

(2015,1777.5)

(2016,1095.0)

```
package it.polito.bigdata.hadoop.exercise8;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

/**
 * MapReduce program
 */
public class DriverBigData extends Configured implements Tool {

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

		Path inputPath;
		Path outputDir;
		Path outputDirStep2;
		int numberOfReducers;
		int exitCode;

		// Parse the parameters
		numberOfReducers = Integer.parseInt(args[0]);
		inputPath = new Path(args[1]);
		outputDir = new Path(args[2]);
		outputDirStep2 = new Path(args[3]);
```

```
Configuration conf = this.getConf();

// First job

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

// Assign a name to the job
job.setJobName("Exercise #8");

// Set path of the input file/folder (if it is a folder, the job reads
// all the files in the specified folder) for this job
FileInputFormat.addInputPath(job, inputPath);

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

// Specify the class of the Driver for this job
job.setJarByClass(DriverBigData.class);

// Set job input format
job.setInputFormatClass(KeyValueTextInputFormat.class);

// Set job output format
job.setOutputFormatClass(TextOutputFormat.class);

// Set map class
job.setMapperClass(MapperBigData.class);

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

// Set reduce class
job.setReducerClass(ReducerBigData.class);

// Set reduce output key and value classes
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);

// Set number of reducers
job.setNumReduceTasks(numberOfReducers);

// Execute the job and wait for completion
if (job.waitForCompletion(true) == true)
	exitCode = 0;
else
	exitCode = 1;

if (exitCode == 0) {

	// Second job

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

```
                // Assign a name to the job
                job2.setJobName("Exercise #8 - step 2");

                // Set path of the input file/folder (if it is a folder, the job
                // reads all the files in the specified folder) for this job
                FileInputFormat.addInputPath(job2, outputDir);

                // Set path of the output folder for this job
                FileOutputFormat.setOutputPath(job2, outputDirStep2);

                // Specify the class of the Driver for this job
                job2.setJarByClass(DriverBigData.class);

                // Set job input format
                job2.setInputFormatClass(KeyValueTextInputFormat.class);

                // Set job output format
                job2.setOutputFormatClass(TextOutputFormat.class);

                // Set map class
                job2.setMapperClass(MapperBigDataStep2.class);

                // Set map output key and value classes
                job2.setMapOutputKeyClass(Text.class);
                job2.setMapOutputValueClass(DoubleWritable.class);

                // Set reduce class
                job2.setReducerClass(ReducerBigDataStep2.class);

                // Set reduce output key and value classes
                job2.setOutputKeyClass(Text.class);
                job2.setOutputValueClass(DoubleWritable.class);

                // Set number of reducers
                job2.setNumReduceTasks(numberOfReducers);

                // Execute the job and wait for completion
                if (job2.waitForCompletion(true) == true)
                        exitCode = 0;
                else
                        exitCode = 1;

        }

        return exitCode;
}

/**
 * Main of the driver
 */

public static void main(String args[]) throws Exception {
        // Exploit the ToolRunner class to "configure" and run the Hadoop
        // application
```

```
            int res = ToolRunner.run(new Configuration(), new DriverBigData(),
args);

            System.exit(res);
      }
}


/**
 * Exercise 8 - Mapper
 */
class MapperBigData extends
            Mapper<Text, // Input key type
                        Text, // Input value type
                        Text, // Output key type
                        DoubleWritable> {// Output value type

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

            String[] date = key.toString().split("-");

            String month = new String(date[0] + "-" + date[1]);

            // emit the pair (month, value)
            context.write(new Text(month), new
DoubleWritable(Double.parseDouble(value.toString())));
      }
}


/**
 * Exercise 8 - Mapper 2
 */
class MapperBigDataStep2 extends
            Mapper<Text, // Input key type
                        Text, // Input value type
                        Text, // Output key type
                        DoubleWritable> {// Output value type

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

            String[] month = key.toString().split("-");

            String year = new String(month[0]);

            // emit the pair (month, value)
            context.write(new Text(year), new
DoubleWritable(Double.parseDouble(value.toString())));
      }
}
```

```java
/**
 * Exercise 8 - Reducer
 */
class ReducerBigData extends
		Reducer<Text, // Input key type
				DoubleWritable, // Input value type
				Text, // Output key type
				DoubleWritable> { // Output value type

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

		double totalIncome = 0;

		// Iterate over the set of values and sum them
		for (DoubleWritable value : values) {
			totalIncome = totalIncome + value.get();
		}
		context.write(new Text(key), new DoubleWritable(totalIncome));
	}
}


/**
 * Exercise 8 - Reducer 2
 */
class ReducerBigDataStep2 extends
		Reducer<Text, // Input key type
				DoubleWritable, // Input value type
				Text, // Output key type
				DoubleWritable> { // Output value type

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

		double totalIncome = 0;
		int count = 0;

		// Iterate over the set of values and sum them
		for (DoubleWritable value : values) {
			totalIncome = totalIncome + value.get();
			count++;
		}

		context.write(new Text(key), new DoubleWritable(totalIncome / count));
	}
}
```

SOLUZIONE SINGLE JOB

```java
package it.polito.bigdata.hadoop.exercise8;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class MonthIncome implements org.apache.hadoop.io.Writable {

	private String monthID;
	private double income;

	public String getMonthID() {
		return monthID;
	}

	public void setMonthID(String monthIDValue) {
		monthID = monthIDValue;
	}

	public double getIncome() {
		return income;
	}

	public void setIncome(double incomeValue) {
		income = incomeValue;
	}

	@Override
	public void readFields(DataInput in) throws IOException {
		monthID = in.readUTF();
		income = in.readDouble();
	}

	@Override
	public void write(DataOutput out) throws IOException {
		out.writeUTF(monthID);
		out.writeDouble(income);

	}

}


/**
 * Exercise 8 - Mapper
 */

class MapperBigData extends
		Mapper<Text, // Input key type
				Text, // Input value type
				Text, // Output key type
				MonthIncome> {// Output value type

	protected void map(Text key, // Input key type
			Text value, // Input value type
```

```
                Context context) throws IOException, InterruptedException {

            String[] date = key.toString().split("-");
            String year = date[0];
            String monthID = date[1];

            Double income = Double.parseDouble(value.toString());

            MonthIncome monthIncome = new MonthIncome();

            monthIncome.setMonthID(monthID);
            monthIncome.setIncome(income);

            // emit the pair (year, (month,income))
            context.write(new Text(year), monthIncome);
      }
}


/**
 * Exercise 8 - Reducer
 */
class ReducerBigData extends Reducer<Text, // Input key type
            MonthIncome, // Input value type
            Text, // Output key type
            DoubleWritable> { // Output value type

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

            // Store in the hashmap
            // monthId -> monthly income
            // for each month of the current year (=current key).
            // At most 12 => we can store it in the main memory of each reducer
            HashMap<String, Double> totalMonthIncome = new HashMap<String,
Double>();

            String year = key.toString();

            // Counters used to compute
            // - the total income for the current year (current key)
            // - the number of distinct months for this year (I consider only those
months with an associated income)
            double totalYearlyIncome = 0;
            int countMonths = 0;

            // Iterate over the set of values and compute
            // - the total income for each month
            // - the overall total income for this year
            for (MonthIncome value : values) {
                  // Retrieve the current income for the current month
                  Double income = totalMonthIncome.get(value.getMonthID());
```

```
                    if (income != null) {
                          // This month is already in the hashmap (other local
incomes for this month have been already analyzed).
                          // Update the total income for this month
                          totalMonthIncome.put(new String(value.getMonthID()), new
Double(value.getIncome() + income));
                    } else {
                          // First occurrence of this monthId
                          // Insert monthid - income in the hashmap
                          totalMonthIncome.put(new String(value.getMonthID()), new
Double(value.getIncome()));

                          // Update the number of months of the current year
                          countMonths++;
                    }

                    // Update the total income of the current year
                    totalYearlyIncome = totalYearlyIncome + value.getIncome();
              }

              // First part of the result
              // Emit the pairs (year-month, total monthly income)
              for (Entry<String, Double> pair : totalMonthIncome.entrySet()) {
                    context.write(new Text(year + "-" + pair.getKey()), new
DoubleWritable(pair.getValue()));
              }

              // Second part of the result
              // Emit the average monthly income for each year
              context.write(new Text(year), new DoubleWritable(totalYearlyIncome /
countMonths));

        }
}
```

Exercise #9

- Word count problem
  - Input: (unstructured) textual file
  - Output: number of occurrences of each word appearing in the input file
- Solve the problem by using in-mapper combiners

Exercise #9 - Example

- Input file

> Toy example
> file for Hadoop.
> Hadoop running
> example.

- Output pairs

(toy, 1)
(example, 2)
(file, 1)
(for, 1)
(hadoop, 2)
(running, 1)

```
/**
 * Exercise 9 - Mapper
 */
class MapperBigData extends
		Mapper<LongWritable, // Input key type
				Text, // Input value type
				Text, // Output key type
				IntWritable> {// Output value type

	HashMap<String, Integer> wordsCounts;

	protected void setup(Context context) {
		wordsCounts = new HashMap<String, Integer>();
	}

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

		Integer currentFreq;
		// Split each sentence in words. Use whitespace(s) as delimiter (=a
		// space, a tab, a line break, or a form feed)
		// The split method returns an array of strings
		String[] words = value.toString().split("\\s+");

		// Iterate over the set of words
		for (String word : words) {
			// Transform word case
			String cleanedWord = word.toLowerCase();

			currentFreq = wordsCounts.get(cleanedWord);

			if (currentFreq == null) { // it is the first time that the
mapper
                                                                // finds this word
				wordsCounts.put(new String(cleanedWord), new Integer(1));
			} else { // Increase the number of occurrences of the current
word
				currentFreq = currentFreq + 1;
				wordsCounts.put(new String(cleanedWord), new
Integer(currentFreq));
			}
		}
	}

	protected void cleanup(Context context) throws IOException,
InterruptedException {

		// Emit the set of (key, value) pairs of this mapper
		for (Entry<String, Integer> pair : wordsCounts.entrySet()) {
			context.write(new Text(pair.getKey()),
					new IntWritable(pair.getValue()));
		}

	}
```

```
}


/**
 * Exercise 9 - Reducer
 */
class ReducerBigData 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 occurrances = 0;

              // Iterate over the set of values and sum them
              for (IntWritable value : values) {
                    occurrances = occurrances + value.get();
              }
              context.write(key, new IntWritable(occurrances));
       }
}
```